The Python/C API

Versión 3.14.0rc3

Guido van Rossum and the Python development team

octubre 01, 2025

Python Software Foundation
Email: docs@python.org

Índice general

1 Introducción **3**
- 1.1 Language version compatibility . . . 3
- 1.2 Estándares de codificación . . . 3
- 1.3 Archivos de cabecera (*Include*) . . . 4
- 1.4 Macros útiles . . . 4
- 1.5 Objetos, tipos y conteos de referencias . . . 7
  - 1.5.1 Conteo de Referencias . . . 7
  - 1.5.2 Tipos . . . 10
- 1.6 Excepciones . . . 10
- 1.7 Integración de Python . . . 12
- 1.8 Depuración de compilaciones . . . 13
- 1.9 Recommended third party tools . . . 13

2 Estabilidad de la API en C **15**
- 2.1 Unstable C API . . . 15
- 2.2 Interfaz binaria de aplicación estable . . . 15
  - 2.2.1 Limited C API . . . 16
  - 2.2.2 Stable ABI . . . 16
  - 2.2.3 Alcance y rendimiento de la API limitada . . . 16
  - 2.2.4 Advertencias de la API limitada . . . 17
- 2.3 Consideraciones de la plataforma . . . 17
- 2.4 Contenido de la API limitada . . . 17

3 La capa de muy alto nivel **43**

4 Conteo de referencias **49**

5 Manejo de excepciones **53**
- 5.1 Impresión y limpieza . . . 53
- 5.2 Lanzando excepciones . . . 54
- 5.3 Emitir advertencias . . . 57
- 5.4 Consultando el indicador de error . . . 58
- 5.5 Manejo de señal . . . 61
- 5.6 Clases de excepción . . . 62
- 5.7 Objetos excepción . . . 62
- 5.8 Objetos unicode de excepción . . . 63
- 5.9 Control de recursión . . . 65
- 5.10 Exception and warning types . . . 65
  - 5.10.1 Exception types . . . 65
  - 5.10.2 OSError aliases . . . 70
  - 5.10.3 Warning types . . . 71

6 Defining extension modules **73**
6.1 Multiple module instances . . . 73
6.2 Initialization function . . . 74
6.3 Multi-phase initialization . . . 75
6.4 Legacy single-phase initialization . . . 75

7 Utilidades **77**
7.1 Utilidades del sistema operativo . . . 77
7.2 Funciones del Sistema . . . 80
7.3 Control de procesos . . . 82
7.4 Importando módulos . . . 83
7.5 Soporte de empaquetado (*marshalling*) de datos . . . 87
7.6 Analizando argumentos y construyendo valores . . . 88
7.6.1 Analizando argumentos . . . 88
7.6.2 Construyendo valores . . . 94
7.7 Conversión y formato de cadenas de caracteres . . . 96
7.8 PyHash API . . . 98
7.9 Reflexión . . . 100
7.10 Registro de códec y funciones de soporte . . . 101
7.10.1 API de búsqueda de códec . . . 101
7.10.2 API de registro para controladores de errores de codificación Unicode . . . 102
7.11 PyTime C API . . . 103
7.11.1 Types . . . 103
7.11.2 Clock Functions . . . 103
7.11.3 Raw Clock Functions . . . 103
7.11.4 Conversion functions . . . 104
7.12 Soporte para Mapeo Perf . . . 104

8 Capa de objetos abstractos **107**
8.1 Protocolo de objeto . . . 107
8.2 Protocolo de llamada . . . 117
8.2.1 El protocolo *tp_call* . . . 117
8.2.2 El protocolo vectorcall . . . 117
8.2.3 API para invocar objetos . . . 119
8.2.4 API de soporte de llamadas . . . 122
8.3 Protocolo de números . . . 122
8.4 Protocolo de secuencia . . . 125
8.5 Protocolo de mapeo . . . 127
8.6 Protocolo iterador . . . 128
8.7 Protocolo búfer . . . 129
8.7.1 Estructura de búfer . . . 130
8.7.2 Tipos de solicitud búfer . . . 132
8.7.3 Arreglos complejos . . . 134
8.7.4 Funciones relacionadas a búfer . . . 135

9 Capa de objetos concretos **137**
9.1 Objetos fundamentales . . . 137
9.1.1 Objetos tipo . . . 137
9.1.2 El objeto `None` . . . 144
9.2 Objetos numéricos . . . 144
9.2.1 Objetos enteros . . . 144
9.2.2 Objetos booleanos . . . 154
9.2.3 Floating-Point Objects . . . 154
9.2.4 Objetos de números complejos . . . 156
9.3 Objetos de secuencia . . . 158
9.3.1 Objetos bytes . . . 158
9.3.2 Objetos de arreglos de bytes (*bytearrays*) . . . 160
9.3.3 Objetos y códecs unicode . . . 161
9.3.4 Objetos tupla . . . 183

9.3.5 Objetos de secuencia de estructura . . . 184
9.3.6 Objetos lista . . . 186
9.4 Objetos contenedor . . . 188
9.4.1 Objetos diccionario . . . 188
9.4.2 Objetos conjunto . . . 193
9.5 Objetos de función . . . 195
9.5.1 Objetos función . . . 195
9.5.2 Objetos de método de instancia . . . 197
9.5.3 Objetos método . . . 197
9.5.4 Objetos celda . . . 198
9.5.5 Objetos código . . . 199
9.5.6 Code Object Flags . . . 201
9.5.7 Extra information . . . 203
9.6 Otros objetos . . . 204
9.6.1 Objetos archivo . . . 204
9.6.2 Objetos módulo . . . 205
9.6.3 Module definitions . . . 206
9.6.4 Creating extension modules dynamically . . . 209
9.6.5 Funciones de soporte . . . 210
9.6.6 Objetos iteradores . . . 213
9.6.7 Objetos descriptores . . . 213
9.6.8 Objetos rebanada (*slice*) . . . 214
9.6.9 Objetos de vista de memoria (*MemoryView*) . . . 216
9.6.10 Objetos de referencia débil . . . 216
9.6.11 Cápsulas . . . 218
9.6.12 Objetos frame . . . 220
9.6.13 Objetos generadores . . . 222
9.6.14 Objetos corrutina . . . 223
9.6.15 Objetos de variables de contexto . . . 223
9.6.16 Objetos *DateTime* . . . 225
9.6.17 Objetos para indicaciones de tipado . . . 229

10 Inicialización, finalización e hilos **231**
10.1 Antes de la inicialización de Python . . . 231
10.2 Variables de configuración global . . . 232
10.3 Inicializando y finalizando el intérprete . . . 235
10.4 Parámetros de todo el proceso . . . 238
10.5 Estado del hilo y el bloqueo global del intérprete . . . 242
10.5.1 Detaching the thread state from extension code . . . 242
10.5.2 Hilos creados sin Python . . . 243
10.5.3 Supporting subinterpreters in non-Python threads . . . 243
10.5.4 Precauciones sobre `fork()` . . . 244
10.5.5 Cautions regarding runtime finalization . . . 244
10.5.6 API de alto nivel . . . 245
10.5.7 API de bajo nivel . . . 247
10.6 Soporte de subinterprete . . . 250
10.6.1 A Per-Interpreter GIL . . . 252
10.6.2 Errores y advertencias . . . 253
10.7 Notificaciones asincrónicas . . . 253
10.8 Perfilado y Rastreo . . . 254
10.9 Reference tracing . . . 256
10.10 Soporte avanzado del depurador . . . 256
10.11 Soporte de almacenamiento local de hilo . . . 257
10.11.1 API de almacenamiento específico de hilo (TSS, *Thread Specific Storage*) . . . 257
10.11.2 API de almacenamiento local de hilos (TLS, *Thread Local Storage*) . . . 258
10.12 Synchronization Primitives . . . 259
10.12.1 Python Critical Section API . . . 259

11 Configuración de inicialización de Python **263**
11.1 PyInitConfig C API . . . 263
11.1.1 Ejemplo . . . 263
11.1.2 Create Config . . . 264
11.1.3 Error Handling . . . 264
11.1.4 Get Options . . . 265
11.1.5 Set Options . . . 265
11.1.6 Module . . . 266
11.1.7 Initialize Python . . . 266
11.2 Configuration Options . . . 266
11.3 Runtime Python configuration API . . . 267
11.4 PyConfig C API . . . 269
11.4.1 Ejemplo . . . 269
11.4.2 PyWideStringList . . . 270
11.4.3 PyStatus . . . 270
11.4.4 PyPreConfig . . . 271
11.4.5 Preinicialización de Python con PyPreConfig . . . 273
11.4.6 PyConfig . . . 274
11.4.7 Inicialización con PyConfig . . . 286
11.4.8 Configuración aislada . . . 288
11.4.9 Configuración de Python . . . 288
11.4.10 Configuración de la ruta de Python . . . 288
11.5 Py_GetArgcArgv() . . . 289
11.6 Delaying main module execution . . . 289

12 Gestión de la memoria **291**
12.1 Visión general . . . 291
12.2 Dominios del asignador . . . 292
12.3 Interfaz de memoria sin procesar . . . 293
12.4 Interfaz de memoria . . . 293
12.5 Asignadores de objetos . . . 295
12.6 Asignadores de memoria predeterminados . . . 296
12.7 Personalizar asignadores de memoria . . . 296
12.8 Configurar enlaces para detectar errores en las funciones del asignador de memoria de Python . . . 298
12.9 El asignador pymalloc . . . 299
12.9.1 Personalizar asignador de arena de pymalloc . . . 300
12.10 The mimalloc allocator . . . 300
12.11 tracemalloc C API . . . 300
12.12 Ejemplos . . . 300

13 Soporte de implementación de objetos **303**
13.1 Asignación de objetos en el montículo . . . 303
13.2 Object Life Cycle . . . 305
13.2.1 Life Events . . . 305
13.2.2 Cyclic Isolate Destruction . . . 308
13.2.3 Functions . . . 309
13.3 Estructuras de objetos comunes . . . 309
13.3.1 Tipos objeto base y macros . . . 309
13.3.2 Implementando funciones y métodos . . . 311
13.3.3 Acceder a atributos de tipos de extensión . . . 314
13.4 Type Object Structures . . . 318
13.4.1 Referencia rápida . . . 319
13.4.2 Definición de `PyTypeObject` . . . 323
13.4.3 Ranuras (*Slots*) `PyObject` . . . 324
13.4.4 Ranuras `PyVarObject` . . . 325
13.4.5 Ranuras `PyTypeObject` . . . 325
13.4.6 Tipos estáticos . . . 349
13.4.7 Tipos Heap . . . 349

13.4.8 Estructuras de objetos de números . . . 349
13.4.9 Estructuras de objetos mapeo . . . 352
13.4.10 Estructuras de objetos secuencia . . . 352
13.4.11 Estructuras de objetos búfer . . . 353
13.4.12 Estructuras de objetos asíncronos . . . 354
13.4.13 Tipo Ranura *typedefs* . . . 355
13.4.14 Ejemplos . . . 356
13.5 Apoyo a la recolección de basura cíclica . . . 359
13.5.1 Controlar el estado del recolector de basura . . . 362
13.5.2 Querying Garbage Collector State . . . 362

14 Versiones de API y ABI **365**
14.1 Build-time version constants . . . 365
14.2 Run-time version . . . 365
14.3 Bit-packing macros . . . 366

15 Monitoring C API **367**

16 Generating Execution Events **369**
16.1 Managing the Monitoring State . . . 370

A Glosario **375**

B About this documentation **395**
B.1 Contributors to the Python documentation . . . 395

C Historia y Licencia **397**
C.1 Historia del software . . . 397
C.2 Términos y condiciones para acceder o usar Python . . . 398
C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2 . . . 398
C.2.2 ACUERDO DE LICENCIA DE BEOPEN.COM PARA PYTHON 2.0 . . . 399
C.2.3 ACUERDO DE LICENCIA CNRI PARA PYTHON 1.6.1 . . . 400
C.2.4 ACUERDO DE LICENCIA CWI PARA PYTHON 0.9.0 HASTA 1.2 . . . 401
C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION . . . 401
C.3 Licencias y reconocimientos para software incorporado . . . 401
C.3.1 Mersenne Twister . . . 401
C.3.2 Sockets . . . 402
C.3.3 Servicios de socket asincrónicos . . . 403
C.3.4 Gestión de cookies . . . 403
C.3.5 Seguimiento de ejecución . . . 404
C.3.6 funciones UUencode y UUdecode . . . 404
C.3.7 Llamadas a procedimientos remotos XML . . . 405
C.3.8 test_epoll . . . 405
C.3.9 Seleccionar kqueue . . . 406
C.3.10 SipHash24 . . . 406
C.3.11 strtod y dtoa . . . 407
C.3.12 OpenSSL . . . 407
C.3.13 expat . . . 411
C.3.14 libffi . . . 411
C.3.15 zlib . . . 412
C.3.16 cfuhash . . . 412
C.3.17 libmpdec . . . 413
C.3.18 Conjunto de pruebas W3C C14N . . . 413
C.3.19 mimalloc . . . 414
C.3.20 asyncio . . . 414
C.3.21 Global Unbounded Sequences (GUS) . . . 415
C.3.22 Zstandard bindings . . . 415

D Derechos de autor **417**

Bibliografía **419**

Índice **421**

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.

CAPÍTULO 1

Introducción

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 Language version compatibility

Python's C API is compatible with C11 and C++11 versions of C and C++.

This is a lower limit: the C API does not require features from later C/C++ versions. You do *not* need to enable your compiler's «c11 mode».

1.2 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.

1.3 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.

The header files are typically installed with Python. On Unix, these are located in the directories *prefix*`/include/pythonversion/` and *exec_prefix*`/include/pythonversion/`, where `prefix` and `exec_prefix` are defined by the corresponding parameters to Python's **`configure`** script and *version* is `'%d.%d' % sys.version_info[:2]`. On Windows, the headers are installed in *prefix*`/include`, where `prefix` is the installation directory specified to the installer.

To include the headers, place both directories (if different) on your compiler's search path for includes. Do *not* place the parent directories on the search path and then use `#include <pythonX.Y/Python.h>`; this will break on multi-platform builds since the platform independent headers under `prefix` include the platform specific headers from `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++.

1.4 Macros útiles

Several useful macros are defined in the Python header files. Many are defined closer to where they are useful (for example, *Py_RETURN_NONE*, *PyMODINIT_FUNC*). Others of a more general utility are defined here. This is not necessarily a complete listing.

Py_ABS(x)

Retorna el valor absoluto de `x`.

Added in version 3.3.

Py_ALWAYS_INLINE

Ask the compiler to always inline a static inline function. The compiler can ignore it and decide to not inline the function.

Puede ser usado para usar inline en funciones estáticas inline de rendimiento crítico cuando se corre Python en modo de depuración con inline de funciones deshabilitado. Por ejemplo, MSC deshabilita el inline de funciones cuando se configura en modo de depuración.

Marcar ciegamente una función estática inline con Py_ALWAYS_INLINE puede resultar en peor rendimientos (debido a un aumento del tamaño del código, por ejemplo). El compilador es generalmente más inteligente que el desarrollador para el análisis costo/beneficio.

If Python is built in debug mode (if the *Py_DEBUG* macro is defined), the *Py_ALWAYS_INLINE* macro does nothing.

Debe ser especificado antes del tipo de retorno de la función. Uso:

```
static inline Py_ALWAYS_INLINE int random(void) { return 4; }
```

Added in version 3.11.

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_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.

Py_GETENV(s)

Like `getenv(s)`, but returns `NULL` if `-E` was passed on the command line (see *PyConfig.use_environment*).

Py_MAX(x, y)

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

Added in version 3.3.

Py_MEMBER_SIZE(type, member)

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

Added in version 3.6.

Py_MIN(x, y)

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

Added in version 3.3.

Py_NO_INLINE

Deshabilita el uso de inline en una función. Por ejemplo, reduce el consumo de la pila C: útil en compilaciones LTO+PGO que usan mucho inline (ver bpo-33720).

Uso:

```
Py_NO_INLINE static int random(void) { return 4; }
```

Added in version 3.11.

Py_STRINGIFY(x)

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

Added in version 3.4.

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()*.

Added in version 3.7.

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; }`.

Added in version 3.4.

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 pysqlite_row_methods[] = {
    {"keys", (PyCFunction)pysqlite_row_keys, METH_NOARGS,
        PyDoc_STR("Returns the keys of the row.")},
    {NULL, NULL}
};
```

1.5 Objetos, tipos y conteos de referencias

Most Python/C API functions have one or more arguments as well as a return value of type *PyObject**. This type is a pointer to an opaque data type representing an arbitrary Python object. Since all Python object types are treated the same way by the Python language in most situations (e.g., assignments, scope rules, and argument passing), it is only fitting that they should be represented by a single C type. Almost all Python objects live on the heap: you never declare an automatic or static variable of type *PyObject*, only pointer variables of type *PyObject** can be declared. The sole exception are the type objects; since these must never be deallocated, they are typically static *PyTypeObject* objects.

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.

1.5.1 Conteo de Referencias

The reference count is important because today's computers have a finite (and often severely limited) memory size; it counts how many different places there are that have a *strong reference* to an object. Such a place could be another object, or a global (or static) C variable, or a local variable in some C function. When the last *strong reference* to an object is released (i.e. its reference count becomes zero), the object is deallocated. If it contains references to other objects, those references are released. Those other objects may be deallocated in turn, if there are no more references to them, and so on. (There's an obvious problem with objects that reference each other here; for now, the solution is «don't do that.»)

Reference counts are always manipulated explicitly. The normal way is to use the macro *Py_INCREF()* to take a new reference to an object (i.e. increment its reference count by one), and *Py_DECREF()* to release that reference (i.e. decrement the reference count by one). The *Py_DECREF()* macro is considerably more complex than the incref one, since it must check whether the reference count becomes zero and then cause the object's deallocator to be called. The deallocator is a function pointer contained in the object's type structure. The type-specific deallocator takes care of releasing references for other objects contained in the object if this is a compound object type, such as a list, as well as performing any additional finalization that's needed. There's no chance that the reference count can overflow; at least as many bits are used to hold the reference count as there are distinct memory locations in virtual memory (assuming `sizeof(Py_ssize_t) >= sizeof(void*)`). Thus, the reference count increment is a simple operation.

It is not necessary to hold a *strong reference* (i.e. increment the reference count) for every local variable that contains a pointer to an object. In theory, the object's reference count goes up by one when the variable is made to point to it and it goes down by one when the variable goes out of scope. However, these two cancel each other out, so at the end the reference count hasn't changed. The only real reason to use the reference count is to prevent the object from being deallocated as long as our variable is pointing to it. If we know that there is at least one other reference to the object that lives at least as long as our variable, there is no need to take a new *strong reference* (i.e. increment the reference count) temporarily. An important situation where this arises is in objects that are passed as arguments to C functions in an extension module that are called from Python; the call mechanism guarantees to hold a reference to every argument for the duration of the call.

However, a common pitfall is to extract an object from a list and hold on to it for a while without taking a new reference. Some other operation might conceivably remove the object from the list, releasing that reference, and possibly deallocating it. The real danger is that innocent-looking operations may invoke arbitrary Python code which could do this; there is a code path which allows control to flow back to the user from a *Py_DECREF()*, so almost any operation is potentially dangerous.

A safe approach is to always use the generic operations (functions whose name begins with `PyObject_`, `PyNumber_`, `PySequence_` or `PyMapping_`). These operations always create a new *strong reference* (i.e. increment the reference count) of the object they return. This leaves the caller with the responsibility to call *Py_DECREF()* when they are done with the result; this soon becomes second nature.

Detalles del conteo de referencia

The reference count behavior of functions in the Python/C API is best explained in terms of *ownership of references*. Ownership pertains to references, never to objects (objects are not owned: they are always shared). «Owning a reference» means being responsible for calling Py_DECREF on it when the reference is no longer needed. Ownership can also be transferred, meaning that the code that receives ownership of the reference then becomes responsible for eventually releasing it by calling *Py_DECREF()* or *Py_XDECREF()* when it's no longer needed—or passing on this responsibility (usually to its caller). When a function passes ownership of a reference on to its caller, the caller is said to receive a *new* reference. When no ownership is transferred, the caller is said to *borrow* the reference. Nothing needs to be done for a *borrowed reference*.

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 un 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");
```

It is much more common to use *PyObject_SetItem()* and friends with items whose references you are only borrowing, like arguments that were passed in to the function you are writing. In that case, their behaviour regarding references is much saner, since you don't have to take a new reference just so you can give that reference away («have it be stolen»). For example, this function sets all items of a list (actually, any mutable sequence) to a given item:

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

    n = PyObject_Length(target);
    if (n < 0)
        return -1;
```

(continúe en la próxima página)

(proviene de la página anterior)

```
    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);
    }
    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;
```

(continúe en la próxima página)

(proviene de la página anterior)

```
    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)) {
            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;
}
```

1.5.2 Tipos

There are few other data types that play a significant role in the Python/C API; most are simple C types such as `int`, `long`, `double` and `char*`. A few structure types are used to describe static tables used to list the functions exported by a module or the data attributes of a new object type, and another is used to describe the value of a complex number. These will be discussed together with the functions that use them.

type **`Py_ssize_t`**

Part of the Stable ABI. Un tipo integral con signo tal que `sizeof(Py_ssize_t) == sizeof(size_t)`. C99 no define directamente tal cosa (size_t es un tipo integral sin signo). Vea **PEP 353** para más detalles. `PY_SSIZE_T_MAX` es el valor positivo más grande del tipo *`Py_ssize_t`*.

1.6 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.

A simple example of detecting exceptions and passing them on is shown in the `sum_sequence()` example above. It so happens that this example doesn't need to clean up any owned references when it detects an error. The following example function shows some error cleanup. First, to remind you why you like Python, we show the equivalent Python code:

```
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;
```

(continúe en la próxima página)

(proviene de la página anterior)

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

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

1.7 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 esta variable es necesaria por el código Python que se ejecutará más tarde, debe establecerse *PyConfig.argv* y *PyConfig.parse_argv*: consulte *Python Initialization Configuration*.

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`.

The embedding application can steer the search by setting *PyConfig.program_name* *before* calling *Py_InitializeFromConfig()*. Note that `PYTHONHOME` still overrides this and `PYTHONPATH` is still inserted in front of the standard path. An application that requires total control has to provide its own implementation of

Py_GetPath(), *Py_GetPrefix()*, *Py_GetExecPrefix()*, and *Py_GetProgramFullPath()* (all defined in `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.

1.8 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.

A full list of the various types of debugging builds is in the file `Misc/SpecialBuilds.txt` in the Python source distribution. Builds are available that support tracing of reference counts, debugging the memory allocator, or low-level profiling of the main interpreter loop. Only the most frequently used builds will be described in the remainder of this section.

`Py_DEBUG`

Compiling the interpreter with the `Py_DEBUG` macro defined produces what is generally meant by a debug build of Python. `Py_DEBUG` is enabled in the Unix build by adding `--with-pydebug` to the `./configure` command. It is also implied by the presence of the not-Python-specific `_DEBUG` macro. When `Py_DEBUG` is enabled in the Unix build, compiler optimization is disabled.

Además de la depuración del recuento de referencia que se describe a continuación, se realizan verificaciones adicionales, véase compilaciones de depuración.

Definiendo `Py_TRACE_REFS` habilita el rastreo de referencias (véase la opción `configure --with-trace-refs`). 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).

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

1.9 Recommended third party tools

The following third party tools offer both simpler and more sophisticated approaches to creating C, C++ and Rust extensions for Python:

- Cython
- cffi
- HPy
- nanobind (C++)
- Numba
- pybind11 (C++)
- PyO3 (Rust)
- SWIG

Using tools such as these can help avoid writing code that is tightly bound to a particular version of CPython, avoid reference counting errors, and focus more on your own code than on using the CPython API. In general, new ver-

sions of Python can be supported by updating the tool, and your code will often use newer and more efficient APIs automatically. Some tools also support compiling for other implementations of Python from a single set of sources.

These projects are not supported by the same people who maintain Python, and issues need to be raised with the projects directly. Remember to check that the project is still maintained and supported, as the list above may become outdated.

Ver también

Python Packaging User Guide: Binary Extensions
The Python Packaging User Guide not only covers several available tools that simplify the creation of binary extensions, but also discusses the various reasons why creating an extension module may be desirable in the first place.

CAPÍTULO 2

Estabilidad de la API en C

Unless documented otherwise, Python's C API is covered by the Backwards Compatibility Policy, **PEP 387**. Most changes to it are source-compatible (typically by only adding new API). Changing existing API or removing API is only done after a deprecation period or to fix serious issues.

CPython's Application Binary Interface (ABI) is forward- and backwards-compatible across a minor release (if these are compiled the same way; see *Consideraciones de la plataforma* below). So, code compiled for Python 3.10.0 will work on 3.10.8 and vice versa, but will need to be compiled separately for 3.9.x and 3.11.x.

There are two tiers of C API with different stability expectations:

- *Unstable API*, may change in minor versions without a deprecation period. It is marked by the `PyUnstable` prefix in names.
- *Limited API*, is compatible across several minor releases. When `Py_LIMITED_API` is defined, only this subset is exposed from `Python.h`.

These are discussed in more detail below.

Names prefixed by an underscore, such as `_Py_InternalState`, are private API that can change without notice even in patch releases. If you need to use this API, consider reaching out to CPython developers to discuss adding public API for your use case.

2.1 Unstable C API

Any API named with the `PyUnstable` prefix exposes CPython implementation details, and may change in every minor release (e.g. from 3.9 to 3.10) without any deprecation warnings. However, it will not change in a bugfix release (e.g. from 3.10.0 to 3.10.1).

It is generally intended for specialized, low-level tools like debuggers.

Projects that use this API are expected to follow CPython development and spend extra effort adjusting to changes.

2.2 Interfaz binaria de aplicación estable

For simplicity, this document talks about *extensions*, but the Limited API and Stable ABI work the same way for all uses of the API – for example, embedding Python.

2.2.1 Limited C API

Python 3.2 introduced the *Limited API*, a subset of Python's C API. Extensions that only use the Limited API can be compiled once and be loaded on multiple versions of Python. Contents of the Limited API are *listed below*.

Py_LIMITED_API

Se define esta macro antes de incluir `Python.h` para optar por usar sólo la API limitada y para seleccionar la versión de la API limitada.

Define `Py_LIMITED_API` to the value of *PY_VERSION_HEX* corresponding to the lowest Python version your extension supports. The extension will be ABI-compatible with all Python 3 releases from the specified one onward, and can use Limited API introduced up to that version.

En lugar de utilizar directamente la macro `PY_VERSION_HEX`, se codifica una versión menor mínima (por ejemplo, `0x030A0000` para Python 3.10) para tener estabilidad cuando se compila con futuras versiones de Python.

También se puede definir `Py_LIMITED_API` con `3`. Esto funciona igual que `0x03020000` (Python 3.2, la función que introdujo la API limitada).

2.2.2 Stable ABI

To enable this, Python provides a *Stable ABI*: a set of symbols that will remain ABI-compatible across Python 3.x versions.

> **Nota**
>
> The Stable ABI prevents ABI issues, like linker errors due to missing symbols or data corruption due to changes in structure layouts or function signatures. However, other changes in Python can change the *behavior* of extensions. See Python's Backwards Compatibility Policy (**PEP 387**) for details.

The Stable ABI contains symbols exposed in the *Limited API*, but also other ones – for example, functions necessary to support older versions of the Limited API.

En Windows, las extensiones que usan la ABI estable deben estar vinculadas con `python3.dll` en lugar de una biblioteca específica de la versión como `python39.dll`.

En algunas plataformas, Python buscará y cargará archivos de bibliotecas compartidas con el nombre de la etiqueta `abi3` (por ejemplo, `mymodule.abi3.so`). No comprueba si tales extensiones se ajustan a una ABI estable. El usuario (o sus herramientas de empaquetado) necesitan asegurarse que, por ejemplo, las extensiones que se crean con la API limitada 3.10+ no estén instaladas para versiones inferiores de Python.

Todas las funciones de la ABI estable se presentan como funciones en la biblioteca compartida de Python, no sólo como macros. Esto las hace utilizables desde lenguajes que no usan el preprocesador de C.

2.2.3 Alcance y rendimiento de la API limitada

El objetivo de la API limitada es permitir todo lo que es posible con la API completa en C, pero posiblemente con una penalización de rendimiento.

Por ejemplo, mientras *PyList_GetItem()* está disponible, su variante macro "insegura" *PyList_GET_ITEM()* no lo está. La macro puede ser más rápida porque puede confiar en los detalles de implementación específicos de la versión del objeto de lista.

Sin definirse `Py_LIMITED_API`, algunas funciones de la API en C están integradas o reemplazadas por macros. Definir `Py_LIMITED_API` desactiva esta integración, permitiendo estabilidad mientras que se mejoren las estructuras de datos de Python, pero posiblemente reduzca el rendimiento.

Al dejar fuera la definición de `Py_LIMITED_API`, es posible compilar una extensión de la API limitada con una ABI específica de la versión. Esto puede mejorar el rendimiento para esa versión de Python, pero limitará la compatibilidad. Compilar con `Py_LIMITED_API` producirá una extensión que se puede distribuir donde una versión específica no esté disponible - por ejemplo, para los prelanzamientos de una versión próxima de Python.

2.2.4 Advertencias de la API limitada

Note that compiling with `Py_LIMITED_API` is *not* a complete guarantee that code conforms to the *Limited API* or the *Stable ABI*. `Py_LIMITED_API` only covers definitions, but an API also includes other issues, such as expected semantics.

Un problema contra el que `Py_LIMITED_API` no protege es llamar una función con argumentos que son inválidos en una versión inferior de Python. Por ejemplo, se considera una función que empieza a aceptar `NULL` como un argumento. Ahora en Python 3.9, `NULL` selecciona un comportamiento predeterminado, pero en Python 3.8, el argumento se usará directamente, causando una desreferencia `NULL` y se detendrá. Un argumento similar funciona para campos de estructuras.

Otro problema es que algunos campos de estructura no se ocultan actualmente cuando se define `Py_LIMITED_API`, aunque son parte de la API limitada.

Por estas razones, recomendamos probar una extensión con *todas* las versiones menores de Python que soporte, y preferiblemente compilar con la versión *más baja*.

También recomendamos revisar la documentación de todas las API usadas para verificar si es parte explícitamente de la API limitada. Aunque se defina `Py_LIMITED_API`, algunas declaraciones privadas se exponen por razones técnicas (o incluso involuntariamente, como errores).

También tome en cuenta que la API limitada no necesariamente es estable: compilar con `Py_LIMITED_API` con Python 3.8 significa que la extensión se ejecutará con Python 3.12, pero no necesariamente *compilará* con Python 3.12. En particular, las partes de la API limitada se pueden quedar obsoletas y eliminarse, siempre que la ABI estable permanezca estable.

2.3 Consideraciones de la plataforma

ABI stability depends not only on Python, but also on the compiler used, lower-level libraries and compiler options. For the purposes of the *Stable ABI*, these details define a "platform". They usually depend on the OS type and processor architecture

Es la responsabilidad de cada distribuidor particular de Python de asegurarse de que todas las versiones de Python en una plataforma particular se compilen de una forma que no rompa la ABI estable. Este es el caso de las versiones de Windows y macOS de `python.org` y muchos distribuidores de terceros.

2.4 Contenido de la API limitada

Currently, the *Limited API* includes the following items:

- *PY_VECTORCALL_ARGUMENTS_OFFSET*
- *PyAIter_Check()*
- *PyArg_Parse()*
- *PyArg_ParseTuple()*
- *PyArg_ParseTupleAndKeywords()*
- *PyArg_UnpackTuple()*
- *PyArg_VaParse()*
- *PyArg_VaParseTupleAndKeywords()*
- *PyArg_ValidateKeywordArguments()*
- *PyBaseObject_Type*
- *PyBool_FromLong()*
- *PyBool_Type*

- *PyBuffer_FillContiguousStrides()*
- *PyBuffer_FillInfo()*
- *PyBuffer_FromContiguous()*
- *PyBuffer_GetPointer()*
- *PyBuffer_IsContiguous()*
- *PyBuffer_Release()*
- *PyBuffer_SizeFromFormat()*
- *PyBuffer_ToContiguous()*
- PyByteArrayIter_Type
- *PyByteArray_AsString()*
- *PyByteArray_Concat()*
- *PyByteArray_FromObject()*
- *PyByteArray_FromStringAndSize()*
- *PyByteArray_Resize()*
- *PyByteArray_Size()*
- *PyByteArray_Type*
- PyBytesIter_Type
- *PyBytes_AsString()*
- *PyBytes_AsStringAndSize()*
- *PyBytes_Concat()*
- *PyBytes_ConcatAndDel()*
- PyBytes_DecodeEscape()
- *PyBytes_FromFormat()*
- *PyBytes_FromFormatV()*
- *PyBytes_FromObject()*
- *PyBytes_FromString()*
- *PyBytes_FromStringAndSize()*
- PyBytes_Repr()
- *PyBytes_Size()*
- *PyBytes_Type*
- *PyCFunction*
- *PyCFunctionFast*
- *PyCFunctionFastWithKeywords*
- *PyCFunctionWithKeywords*
- PyCFunction_GetFlags()
- PyCFunction_GetFunction()
- PyCFunction_GetSelf()
- *PyCFunction_New()*
- *PyCFunction_NewEx()*

- PyCFunction_Type
- *PyCMethod_New()*
- *PyCallIter_New()*
- *PyCallIter_Type*
- *PyCallable_Check()*
- *PyCapsule_Destructor*
- *PyCapsule_GetContext()*
- *PyCapsule_GetDestructor()*
- *PyCapsule_GetName()*
- *PyCapsule_GetPointer()*
- *PyCapsule_Import()*
- *PyCapsule_IsValid()*
- *PyCapsule_New()*
- *PyCapsule_SetContext()*
- *PyCapsule_SetDestructor()*
- *PyCapsule_SetName()*
- *PyCapsule_SetPointer()*
- PyCapsule_Type
- PyClassMethodDescr_Type
- *PyCodec_BackslashReplaceErrors()*
- *PyCodec_Decode()*
- *PyCodec_Decoder()*
- *PyCodec_Encode()*
- *PyCodec_Encoder()*
- *PyCodec_IgnoreErrors()*
- *PyCodec_IncrementalDecoder()*
- *PyCodec_IncrementalEncoder()*
- *PyCodec_KnownEncoding()*
- *PyCodec_LookupError()*
- *PyCodec_NameReplaceErrors()*
- *PyCodec_Register()*
- *PyCodec_RegisterError()*
- *PyCodec_ReplaceErrors()*
- *PyCodec_StreamReader()*
- *PyCodec_StreamWriter()*
- *PyCodec_StrictErrors()*
- *PyCodec_Unregister()*
- *PyCodec_XMLCharRefReplaceErrors()*
- *PyComplex_FromDoubles()*

- *PyComplex_ImagAsDouble()*
- *PyComplex_RealAsDouble()*
- *PyComplex_Type*
- *PyDescr_NewClassMethod()*
- *PyDescr_NewGetSet()*
- *PyDescr_NewMember()*
- *PyDescr_NewMethod()*
- PyDictItems_Type
- PyDictIterItem_Type
- PyDictIterKey_Type
- PyDictIterValue_Type
- PyDictKeys_Type
- *PyDictProxy_New()*
- PyDictProxy_Type
- PyDictRevIterItem_Type
- PyDictRevIterKey_Type
- PyDictRevIterValue_Type
- PyDictValues_Type
- *PyDict_Clear()*
- *PyDict_Contains()*
- *PyDict_Copy()*
- *PyDict_DelItem()*
- *PyDict_DelItemString()*
- *PyDict_GetItem()*
- *PyDict_GetItemRef()*
- *PyDict_GetItemString()*
- *PyDict_GetItemStringRef()*
- *PyDict_GetItemWithError()*
- *PyDict_Items()*
- *PyDict_Keys()*
- *PyDict_Merge()*
- *PyDict_MergeFromSeq2()*
- *PyDict_New()*
- *PyDict_Next()*
- *PyDict_SetItem()*
- *PyDict_SetItemString()*
- *PyDict_Size()*
- *PyDict_Type*
- *PyDict_Update()*

- *PyDict_Values()*
- *PyEllipsis_Type*
- PyEnum_Type
- *PyErr_BadArgument()*
- *PyErr_BadInternalCall()*
- *PyErr_CheckSignals()*
- *PyErr_Clear()*
- PyErr_Display()
- *PyErr_DisplayException()*
- *PyErr_ExceptionMatches()*
- *PyErr_Fetch()*
- *PyErr_Format()*
- *PyErr_FormatV()*
- *PyErr_GetExcInfo()*
- *PyErr_GetHandledException()*
- *PyErr_GetRaisedException()*
- *PyErr_GivenExceptionMatches()*
- *PyErr_NewException()*
- *PyErr_NewExceptionWithDoc()*
- *PyErr_NoMemory()*
- *PyErr_NormalizeException()*
- *PyErr_Occurred()*
- *PyErr_Print()*
- *PyErr_PrintEx()*
- PyErr_ProgramText()
- *PyErr_ResourceWarning()*
- *PyErr_Restore()*
- *PyErr_SetExcFromWindowsErr()*
- *PyErr_SetExcFromWindowsErrWithFilename()*
- *PyErr_SetExcFromWindowsErrWithFilenameObject()*
- *PyErr_SetExcFromWindowsErrWithFilenameObjects()*
- *PyErr_SetExcInfo()*
- *PyErr_SetFromErrno()*
- *PyErr_SetFromErrnoWithFilename()*
- *PyErr_SetFromErrnoWithFilenameObject()*
- *PyErr_SetFromErrnoWithFilenameObjects()*
- *PyErr_SetFromWindowsErr()*
- *PyErr_SetFromWindowsErrWithFilename()*
- *PyErr_SetHandledException()*

- *PyErr_SetImportError()*
- *PyErr_SetImportErrorSubclass()*
- *PyErr_SetInterrupt()*
- *PyErr_SetInterruptEx()*
- *PyErr_SetNone()*
- *PyErr_SetObject()*
- *PyErr_SetRaisedException()*
- *PyErr_SetString()*
- *PyErr_SyntaxLocation()*
- *PyErr_SyntaxLocationEx()*
- *PyErr_WarnEx()*
- *PyErr_WarnExplicit()*
- *PyErr_WarnFormat()*
- *PyErr_WriteUnraisable()*
- *PyEval_AcquireThread()*
- *PyEval_EvalCode()*
- *PyEval_EvalCodeEx()*
- *PyEval_EvalFrame()*
- *PyEval_EvalFrameEx()*
- *PyEval_GetBuiltins()*
- *PyEval_GetFrame()*
- *PyEval_GetFrameBuiltins()*
- *PyEval_GetFrameGlobals()*
- *PyEval_GetFrameLocals()*
- *PyEval_GetFuncDesc()*
- *PyEval_GetFuncName()*
- *PyEval_GetGlobals()*
- *PyEval_GetLocals()*
- *PyEval_InitThreads()*
- *PyEval_ReleaseThread()*
- *PyEval_RestoreThread()*
- *PyEval_SaveThread()*
- *PyExc_ArithmeticError*
- *PyExc_AssertionError*
- *PyExc_AttributeError*
- *PyExc_BaseException*
- *PyExc_BaseExceptionGroup*
- *PyExc_BlockingIOError*
- *PyExc_BrokenPipeError*

- *PyExc_BufferError*
- *PyExc_BytesWarning*
- *PyExc_ChildProcessError*
- *PyExc_ConnectionAbortedError*
- *PyExc_ConnectionError*
- *PyExc_ConnectionRefusedError*
- *PyExc_ConnectionResetError*
- *PyExc_DeprecationWarning*
- *PyExc_EOFError*
- *PyExc_EncodingWarning*
- *PyExc_EnvironmentError*
- *PyExc_Exception*
- *PyExc_FileExistsError*
- *PyExc_FileNotFoundError*
- *PyExc_FloatingPointError*
- *PyExc_FutureWarning*
- *PyExc_GeneratorExit*
- *PyExc_IOError*
- *PyExc_ImportError*
- *PyExc_ImportWarning*
- *PyExc_IndentationError*
- *PyExc_IndexError*
- *PyExc_InterruptedError*
- *PyExc_IsADirectoryError*
- *PyExc_KeyError*
- *PyExc_KeyboardInterrupt*
- *PyExc_LookupError*
- *PyExc_MemoryError*
- *PyExc_ModuleNotFoundError*
- *PyExc_NameError*
- *PyExc_NotADirectoryError*
- *PyExc_NotImplementedError*
- *PyExc_OSError*
- *PyExc_OverflowError*
- *PyExc_PendingDeprecationWarning*
- *PyExc_PermissionError*
- *PyExc_ProcessLookupError*
- *PyExc_RecursionError*
- *PyExc_ReferenceError*

- *PyExc_ResourceWarning*
- *PyExc_RuntimeError*
- *PyExc_RuntimeWarning*
- *PyExc_StopAsyncIteration*
- *PyExc_StopIteration*
- *PyExc_SyntaxError*
- *PyExc_SyntaxWarning*
- *PyExc_SystemError*
- *PyExc_SystemExit*
- *PyExc_TabError*
- *PyExc_TimeoutError*
- *PyExc_TypeError*
- *PyExc_UnboundLocalError*
- *PyExc_UnicodeDecodeError*
- *PyExc_UnicodeEncodeError*
- *PyExc_UnicodeError*
- *PyExc_UnicodeTranslateError*
- *PyExc_UnicodeWarning*
- *PyExc_UserWarning*
- *PyExc_ValueError*
- *PyExc_Warning*
- *PyExc_WindowsError*
- *PyExc_ZeroDivisionError*
- *PyExceptionClass_Name()*
- *PyException_GetArgs()*
- *PyException_GetCause()*
- *PyException_GetContext()*
- *PyException_GetTraceback()*
- *PyException_SetArgs()*
- *PyException_SetCause()*
- *PyException_SetContext()*
- *PyException_SetTraceback()*
- *PyFile_FromFd()*
- *PyFile_GetLine()*
- *PyFile_WriteObject()*
- *PyFile_WriteString()*
- PyFilter_Type
- *PyFloat_AsDouble()*
- *PyFloat_FromDouble()*

- *PyFloat_FromString()*
- *PyFloat_GetInfo()*
- *PyFloat_GetMax()*
- *PyFloat_GetMin()*
- *PyFloat_Type*
- *PyFrameObject*
- *PyFrame_GetCode()*
- *PyFrame_GetLineNumber()*
- *PyFrozenSet_New()*
- *PyFrozenSet_Type*
- *PyGC_Collect()*
- *PyGC_Disable()*
- *PyGC_Enable()*
- *PyGC_IsEnabled()*
- *PyGILState_Ensure()*
- *PyGILState_GetThisThreadState()*
- *PyGILState_Release()*
- PyGILState_STATE
- *PyGetSetDef*
- PyGetSetDescr_Type
- *PyImport_AddModule()*
- *PyImport_AddModuleObject()*
- *PyImport_AddModuleRef()*
- *PyImport_AppendInittab()*
- *PyImport_ExecCodeModule()*
- *PyImport_ExecCodeModuleEx()*
- *PyImport_ExecCodeModuleObject()*
- *PyImport_ExecCodeModuleWithPathnames()*
- *PyImport_GetImporter()*
- *PyImport_GetMagicNumber()*
- *PyImport_GetMagicTag()*
- *PyImport_GetModule()*
- *PyImport_GetModuleDict()*
- *PyImport_Import()*
- *PyImport_ImportFrozenModule()*
- *PyImport_ImportFrozenModuleObject()*
- *PyImport_ImportModule()*
- *PyImport_ImportModuleLevel()*
- *PyImport_ImportModuleLevelObject()*

- *PyImport_ImportModuleNoBlock()*
- *PyImport_ReloadModule()*
- *PyIndex_Check()*
- *PyInterpreterState*
- *PyInterpreterState_Clear()*
- *PyInterpreterState_Delete()*
- *PyInterpreterState_Get()*
- *PyInterpreterState_GetDict()*
- *PyInterpreterState_GetID()*
- *PyInterpreterState_New()*
- *PyIter_Check()*
- *PyIter_Next()*
- *PyIter_NextItem()*
- *PyIter_Send()*
- PyListIter_Type
- PyListRevIter_Type
- *PyList_Append()*
- *PyList_AsTuple()*
- *PyList_GetItem()*
- *PyList_GetItemRef()*
- *PyList_GetSlice()*
- *PyList_Insert()*
- *PyList_New()*
- *PyList_Reverse()*
- *PyList_SetItem()*
- *PyList_SetSlice()*
- *PyList_Size()*
- *PyList_Sort()*
- *PyList_Type*
- *PyLongObject*
- PyLongRangeIter_Type
- *PyLong_AsDouble()*
- *PyLong_AsInt()*
- *PyLong_AsInt32()*
- *PyLong_AsInt64()*
- *PyLong_AsLong()*
- *PyLong_AsLongAndOverflow()*
- *PyLong_AsLongLong()*
- *PyLong_AsLongLongAndOverflow()*

- *PyLong_AsNativeBytes()*
- *PyLong_AsSize_t()*
- *PyLong_AsSsize_t()*
- *PyLong_AsUInt32()*
- *PyLong_AsUInt64()*
- *PyLong_AsUnsignedLong()*
- *PyLong_AsUnsignedLongLong()*
- *PyLong_AsUnsignedLongLongMask()*
- *PyLong_AsUnsignedLongMask()*
- *PyLong_AsVoidPtr()*
- *PyLong_FromDouble()*
- *PyLong_FromInt32()*
- *PyLong_FromInt64()*
- *PyLong_FromLong()*
- *PyLong_FromLongLong()*
- *PyLong_FromNativeBytes()*
- *PyLong_FromSize_t()*
- *PyLong_FromSsize_t()*
- *PyLong_FromString()*
- *PyLong_FromUInt32()*
- *PyLong_FromUInt64()*
- *PyLong_FromUnsignedLong()*
- *PyLong_FromUnsignedLongLong()*
- *PyLong_FromUnsignedNativeBytes()*
- *PyLong_FromVoidPtr()*
- *PyLong_GetInfo()*
- *PyLong_Type*
- PyMap_Type
- *PyMapping_Check()*
- *PyMapping_GetItemString()*
- *PyMapping_GetOptionalItem()*
- *PyMapping_GetOptionalItemString()*
- *PyMapping_HasKey()*
- *PyMapping_HasKeyString()*
- *PyMapping_HasKeyStringWithError()*
- *PyMapping_HasKeyWithError()*
- *PyMapping_Items()*
- *PyMapping_Keys()*
- *PyMapping_Length()*

- *PyMapping_SetItemString()*
- *PyMapping_Size()*
- *PyMapping_Values()*
- *PyMem_Calloc()*
- *PyMem_Free()*
- *PyMem_Malloc()*
- *PyMem_RawCalloc()*
- *PyMem_RawFree()*
- *PyMem_RawMalloc()*
- *PyMem_RawRealloc()*
- *PyMem_Realloc()*
- *PyMemberDef*
- PyMemberDescr_Type
- *PyMember_GetOne()*
- *PyMember_SetOne()*
- *PyMemoryView_FromBuffer()*
- *PyMemoryView_FromMemory()*
- *PyMemoryView_FromObject()*
- *PyMemoryView_GetContiguous()*
- PyMemoryView_Type
- *PyMethodDef*
- PyMethodDescr_Type
- *PyModuleDef*
- PyModuleDef_Base
- *PyModuleDef_Init()*
- PyModuleDef_Type
- *PyModule_Add()*
- *PyModule_AddFunctions()*
- *PyModule_AddIntConstant()*
- *PyModule_AddObject()*
- *PyModule_AddObjectRef()*
- *PyModule_AddStringConstant()*
- *PyModule_AddType()*
- *PyModule_Create2()*
- *PyModule_ExecDef()*
- *PyModule_FromDefAndSpec2()*
- *PyModule_GetDef()*
- *PyModule_GetDict()*
- *PyModule_GetFilename()*

- *PyModule_GetFilenameObject()*
- *PyModule_GetName()*
- *PyModule_GetNameObject()*
- *PyModule_GetState()*
- *PyModule_New()*
- *PyModule_NewObject()*
- *PyModule_SetDocString()*
- *PyModule_Type*
- *PyNumber_Absolute()*
- *PyNumber_Add()*
- *PyNumber_And()*
- *PyNumber_AsSsize_t()*
- *PyNumber_Check()*
- *PyNumber_Divmod()*
- *PyNumber_Float()*
- *PyNumber_FloorDivide()*
- *PyNumber_InPlaceAdd()*
- *PyNumber_InPlaceAnd()*
- *PyNumber_InPlaceFloorDivide()*
- *PyNumber_InPlaceLshift()*
- *PyNumber_InPlaceMatrixMultiply()*
- *PyNumber_InPlaceMultiply()*
- *PyNumber_InPlaceOr()*
- *PyNumber_InPlacePower()*
- *PyNumber_InPlaceRemainder()*
- *PyNumber_InPlaceRshift()*
- *PyNumber_InPlaceSubtract()*
- *PyNumber_InPlaceTrueDivide()*
- *PyNumber_InPlaceXor()*
- *PyNumber_Index()*
- *PyNumber_Invert()*
- *PyNumber_Long()*
- *PyNumber_Lshift()*
- *PyNumber_MatrixMultiply()*
- *PyNumber_Multiply()*
- *PyNumber_Negative()*
- *PyNumber_Or()*
- *PyNumber_Positive()*
- *PyNumber_Power()*

- *PyNumber_Remainder()*
- *PyNumber_Rshift()*
- *PyNumber_Subtract()*
- *PyNumber_ToBase()*
- *PyNumber_TrueDivide()*
- *PyNumber_Xor()*
- *PyOS_AfterFork()*
- *PyOS_AfterFork_Child()*
- *PyOS_AfterFork_Parent()*
- *PyOS_BeforeFork()*
- *PyOS_CheckStack()*
- *PyOS_FSPath()*
- *PyOS_InputHook*
- PyOS_InterruptOccurred()
- *PyOS_double_to_string()*
- *PyOS_getsig()*
- PyOS_mystricmp()
- PyOS_mystrnicmp()
- *PyOS_setsig()*
- *PyOS_sighandler_t*
- *PyOS_snprintf()*
- *PyOS_string_to_double()*
- *PyOS_strtol()*
- *PyOS_strtoul()*
- *PyOS_vsnprintf()*
- *PyObject*
- *PyObject.ob_refcnt*
- *PyObject.ob_type*
- *PyObject_ASCII()*
- *PyObject_AsFileDescriptor()*
- *PyObject_Bytes()*
- *PyObject_Call()*
- *PyObject_CallFunction()*
- *PyObject_CallFunctionObjArgs()*
- *PyObject_CallMethod()*
- *PyObject_CallMethodObjArgs()*
- *PyObject_CallNoArgs()*
- *PyObject_CallObject()*
- *PyObject_Calloc()*

- *PyObject_CheckBuffer()*
- *PyObject_ClearWeakRefs()*
- *PyObject_CopyData()*
- *PyObject_DelAttr()*
- *PyObject_DelAttrString()*
- *PyObject_DelItem()*
- *PyObject_DelItemString()*
- *PyObject_Dir()*
- *PyObject_Format()*
- *PyObject_Free()*
- *PyObject_GC_Del()*
- *PyObject_GC_IsFinalized()*
- *PyObject_GC_IsTracked()*
- *PyObject_GC_Track()*
- *PyObject_GC_UnTrack()*
- *PyObject_GenericGetAttr()*
- *PyObject_GenericGetDict()*
- *PyObject_GenericSetAttr()*
- *PyObject_GenericSetDict()*
- *PyObject_GetAIter()*
- *PyObject_GetAttr()*
- *PyObject_GetAttrString()*
- *PyObject_GetBuffer()*
- *PyObject_GetItem()*
- *PyObject_GetIter()*
- *PyObject_GetOptionalAttr()*
- *PyObject_GetOptionalAttrString()*
- *PyObject_GetTypeData()*
- *PyObject_HasAttr()*
- *PyObject_HasAttrString()*
- *PyObject_HasAttrStringWithError()*
- *PyObject_HasAttrWithError()*
- *PyObject_Hash()*
- *PyObject_HashNotImplemented()*
- *PyObject_Init()*
- *PyObject_InitVar()*
- *PyObject_IsInstance()*
- *PyObject_IsSubclass()*
- *PyObject_IsTrue()*

- *PyObject_Length()*
- *PyObject_Malloc()*
- *PyObject_Not()*
- *PyObject_Realloc()*
- *PyObject_Repr()*
- *PyObject_RichCompare()*
- *PyObject_RichCompareBool()*
- *PyObject_SelfIter()*
- *PyObject_SetAttr()*
- *PyObject_SetAttrString()*
- *PyObject_SetItem()*
- *PyObject_Size()*
- *PyObject_Str()*
- *PyObject_Type()*
- *PyObject_Vectorcall()*
- *PyObject_VectorcallMethod()*
- *PyProperty_Type*
- PyRangeIter_Type
- PyRange_Type
- PyReversed_Type
- *PySeqIter_New()*
- *PySeqIter_Type*
- *PySequence_Check()*
- *PySequence_Concat()*
- *PySequence_Contains()*
- *PySequence_Count()*
- *PySequence_DelItem()*
- *PySequence_DelSlice()*
- *PySequence_Fast()*
- *PySequence_GetItem()*
- *PySequence_GetSlice()*
- *PySequence_In()*
- *PySequence_InPlaceConcat()*
- *PySequence_InPlaceRepeat()*
- *PySequence_Index()*
- *PySequence_Length()*
- *PySequence_List()*
- *PySequence_Repeat()*
- *PySequence_SetItem()*

- *PySequence_SetSlice()*
- *PySequence_Size()*
- *PySequence_Tuple()*
- PySetIter_Type
- *PySet_Add()*
- *PySet_Clear()*
- *PySet_Contains()*
- *PySet_Discard()*
- *PySet_New()*
- *PySet_Pop()*
- *PySet_Size()*
- *PySet_Type*
- *PySlice_AdjustIndices()*
- *PySlice_GetIndices()*
- *PySlice_GetIndicesEx()*
- *PySlice_New()*
- *PySlice_Type*
- *PySlice_Unpack()*
- *PyState_AddModule()*
- *PyState_FindModule()*
- *PyState_RemoveModule()*
- *PyStructSequence_Desc*
- *PyStructSequence_Field*
- *PyStructSequence_GetItem()*
- *PyStructSequence_New()*
- *PyStructSequence_NewType()*
- *PyStructSequence_SetItem()*
- *PyStructSequence_UnnamedField*
- PySuper_Type
- *PySys_Audit()*
- *PySys_AuditTuple()*
- *PySys_FormatStderr()*
- *PySys_FormatStdout()*
- *PySys_GetObject()*
- *PySys_GetXOptions()*
- *PySys_ResetWarnOptions()*
- *PySys_SetArgv()*
- *PySys_SetArgvEx()*
- *PySys_SetObject()*

- *PySys_WriteStderr()*
- *PySys_WriteStdout()*
- *PyThreadState*
- *PyThreadState_Clear()*
- *PyThreadState_Delete()*
- *PyThreadState_Get()*
- *PyThreadState_GetDict()*
- *PyThreadState_GetFrame()*
- *PyThreadState_GetID()*
- *PyThreadState_GetInterpreter()*
- *PyThreadState_New()*
- *PyThreadState_SetAsyncExc()*
- *PyThreadState_Swap()*
- PyThread_GetInfo()
- *PyThread_ReInitTLS()*
- PyThread_acquire_lock()
- PyThread_acquire_lock_timed()
- PyThread_allocate_lock()
- *PyThread_create_key()*
- *PyThread_delete_key()*
- *PyThread_delete_key_value()*
- PyThread_exit_thread()
- PyThread_free_lock()
- *PyThread_get_key_value()*
- PyThread_get_stacksize()
- PyThread_get_thread_ident()
- PyThread_get_thread_native_id()
- PyThread_init_thread()
- PyThread_release_lock()
- *PyThread_set_key_value()*
- PyThread_set_stacksize()
- PyThread_start_new_thread()
- *PyThread_tss_alloc()*
- *PyThread_tss_create()*
- *PyThread_tss_delete()*
- *PyThread_tss_free()*
- *PyThread_tss_get()*
- *PyThread_tss_is_created()*
- *PyThread_tss_set()*

- PyTraceBack_Here()
- PyTraceBack_Print()
- PyTraceBack_Type
- PyTupleIter_Type
- *PyTuple_GetItem()*
- *PyTuple_GetSlice()*
- *PyTuple_New()*
- *PyTuple_Pack()*
- *PyTuple_SetItem()*
- *PyTuple_Size()*
- *PyTuple_Type*
- *PyTypeObject*
- *PyType_ClearCache()*
- *PyType_Freeze()*
- *PyType_FromMetaclass()*
- *PyType_FromModuleAndSpec()*
- *PyType_FromSpec()*
- *PyType_FromSpecWithBases()*
- *PyType_GenericAlloc()*
- *PyType_GenericNew()*
- *PyType_GetBaseByToken()*
- *PyType_GetFlags()*
- *PyType_GetFullyQualifiedName()*
- *PyType_GetModule()*
- *PyType_GetModuleByDef()*
- *PyType_GetModuleName()*
- *PyType_GetModuleState()*
- *PyType_GetName()*
- *PyType_GetQualName()*
- *PyType_GetSlot()*
- *PyType_GetTypeDataSize()*
- *PyType_IsSubtype()*
- *PyType_Modified()*
- *PyType_Ready()*
- *PyType_Slot*
- *PyType_Spec*
- *PyType_Type*
- *PyUnicodeDecodeError_Create()*
- *PyUnicodeDecodeError_GetEncoding()*

- *PyUnicodeDecodeError_GetEnd()*
- *PyUnicodeDecodeError_GetObject()*
- *PyUnicodeDecodeError_GetReason()*
- *PyUnicodeDecodeError_GetStart()*
- *PyUnicodeDecodeError_SetEnd()*
- *PyUnicodeDecodeError_SetReason()*
- *PyUnicodeDecodeError_SetStart()*
- *PyUnicodeEncodeError_GetEncoding()*
- *PyUnicodeEncodeError_GetEnd()*
- *PyUnicodeEncodeError_GetObject()*
- *PyUnicodeEncodeError_GetReason()*
- *PyUnicodeEncodeError_GetStart()*
- *PyUnicodeEncodeError_SetEnd()*
- *PyUnicodeEncodeError_SetReason()*
- *PyUnicodeEncodeError_SetStart()*
- *PyUnicodeIter_Type*
- *PyUnicodeTranslateError_GetEnd()*
- *PyUnicodeTranslateError_GetObject()*
- *PyUnicodeTranslateError_GetReason()*
- *PyUnicodeTranslateError_GetStart()*
- *PyUnicodeTranslateError_SetEnd()*
- *PyUnicodeTranslateError_SetReason()*
- *PyUnicodeTranslateError_SetStart()*
- *PyUnicode_Append()*
- *PyUnicode_AppendAndDel()*
- *PyUnicode_AsASCIIString()*
- *PyUnicode_AsCharmapString()*
- PyUnicode_AsDecodedObject()
- PyUnicode_AsDecodedUnicode()
- PyUnicode_AsEncodedObject()
- *PyUnicode_AsEncodedString()*
- PyUnicode_AsEncodedUnicode()
- *PyUnicode_AsLatin1String()*
- *PyUnicode_AsMBCSString()*
- *PyUnicode_AsRawUnicodeEscapeString()*
- *PyUnicode_AsUCS4()*
- *PyUnicode_AsUCS4Copy()*
- *PyUnicode_AsUTF16String()*
- *PyUnicode_AsUTF32String()*

- *PyUnicode_AsUTF8AndSize()*
- *PyUnicode_AsUTF8String()*
- *PyUnicode_AsUnicodeEscapeString()*
- *PyUnicode_AsWideChar()*
- *PyUnicode_AsWideCharString()*
- *PyUnicode_BuildEncodingMap()*
- *PyUnicode_Compare()*
- *PyUnicode_CompareWithASCIIString()*
- *PyUnicode_Concat()*
- *PyUnicode_Contains()*
- *PyUnicode_Count()*
- *PyUnicode_Decode()*
- *PyUnicode_DecodeASCII()*
- *PyUnicode_DecodeCharmap()*
- *PyUnicode_DecodeCodePageStateful()*
- *PyUnicode_DecodeFSDefault()*
- *PyUnicode_DecodeFSDefaultAndSize()*
- *PyUnicode_DecodeLatin1()*
- *PyUnicode_DecodeLocale()*
- *PyUnicode_DecodeLocaleAndSize()*
- *PyUnicode_DecodeMBCS()*
- *PyUnicode_DecodeMBCSStateful()*
- *PyUnicode_DecodeRawUnicodeEscape()*
- *PyUnicode_DecodeUTF16()*
- *PyUnicode_DecodeUTF16Stateful()*
- *PyUnicode_DecodeUTF32()*
- *PyUnicode_DecodeUTF32Stateful()*
- *PyUnicode_DecodeUTF7()*
- *PyUnicode_DecodeUTF7Stateful()*
- *PyUnicode_DecodeUTF8()*
- *PyUnicode_DecodeUTF8Stateful()*
- *PyUnicode_DecodeUnicodeEscape()*
- *PyUnicode_EncodeCodePage()*
- *PyUnicode_EncodeFSDefault()*
- *PyUnicode_EncodeLocale()*
- *PyUnicode_Equal()*
- *PyUnicode_EqualToUTF8()*
- *PyUnicode_EqualToUTF8AndSize()*
- *PyUnicode_FSConverter()*

- *PyUnicode_FSDecoder()*
- *PyUnicode_Find()*
- *PyUnicode_FindChar()*
- *PyUnicode_Format()*
- *PyUnicode_FromEncodedObject()*
- *PyUnicode_FromFormat()*
- *PyUnicode_FromFormatV()*
- *PyUnicode_FromObject()*
- *PyUnicode_FromOrdinal()*
- *PyUnicode_FromString()*
- *PyUnicode_FromStringAndSize()*
- *PyUnicode_FromWideChar()*
- *PyUnicode_GetDefaultEncoding()*
- *PyUnicode_GetLength()*
- *PyUnicode_InternFromString()*
- *PyUnicode_InternInPlace()*
- *PyUnicode_IsIdentifier()*
- *PyUnicode_Join()*
- *PyUnicode_Partition()*
- *PyUnicode_RPartition()*
- *PyUnicode_RSplit()*
- *PyUnicode_ReadChar()*
- *PyUnicode_Replace()*
- *PyUnicode_Resize()*
- *PyUnicode_RichCompare()*
- *PyUnicode_Split()*
- *PyUnicode_Splitlines()*
- *PyUnicode_Substring()*
- *PyUnicode_Tailmatch()*
- *PyUnicode_Translate()*
- *PyUnicode_Type*
- *PyUnicode_WriteChar()*
- *PyVarObject*
- PyVarObject.ob_base
- *PyVarObject.ob_size*
- *PyVectorcall_Call()*
- *PyVectorcall_NARGS()*
- PyWeakReference
- *PyWeakref_GetObject()*

- *PyWeakref_GetRef()*
- *PyWeakref_NewProxy()*
- *PyWeakref_NewRef()*
- PyWrapperDescr_Type
- *PyWrapper_New()*
- PyZip_Type
- *Py_AddPendingCall()*
- *Py_AtExit()*
- *Py_BEGIN_ALLOW_THREADS*
- *Py_BLOCK_THREADS*
- *Py_BuildValue()*
- *Py_BytesMain()*
- *Py_CompileString()*
- *Py_DecRef()*
- *Py_DecodeLocale()*
- *Py_END_ALLOW_THREADS*
- *Py_EncodeLocale()*
- *Py_EndInterpreter()*
- *Py_EnterRecursiveCall()*
- *Py_Exit()*
- *Py_FatalError()*
- Py_FileSystemDefaultEncodeErrors
- Py_FileSystemDefaultEncoding
- *Py_Finalize()*
- *Py_FinalizeEx()*
- *Py_GenericAlias()*
- *Py_GenericAliasType*
- *Py_GetBuildInfo()*
- *Py_GetCompiler()*
- *Py_GetConstant()*
- *Py_GetConstantBorrowed()*
- *Py_GetCopyright()*
- *Py_GetExecPrefix()*
- *Py_GetPath()*
- *Py_GetPlatform()*
- *Py_GetPrefix()*
- *Py_GetProgramFullPath()*
- *Py_GetProgramName()*
- *Py_GetPythonHome()*

- Py_GetRecursionLimit()
- *Py_GetVersion()*
- Py_HasFileSystemDefaultEncoding
- *Py_IncRef()*
- *Py_Initialize()*
- *Py_InitializeEx()*
- *Py_Is()*
- *Py_IsFalse()*
- *Py_IsFinalizing()*
- *Py_IsInitialized()*
- *Py_IsNone()*
- *Py_IsTrue()*
- *Py_LeaveRecursiveCall()*
- *Py_Main()*
- Py_MakePendingCalls()
- *Py_NewInterpreter()*
- *Py_NewRef()*
- *Py_PACK_FULL_VERSION()*
- *Py_PACK_VERSION()*
- *Py_REFCNT()*
- *Py_ReprEnter()*
- *Py_ReprLeave()*
- *Py_SetProgramName()*
- *Py_SetPythonHome()*
- Py_SetRecursionLimit()
- *Py_TYPE()*
- *Py_UCS4*
- *Py_UNBLOCK_THREADS*
- Py_UTF8Mode
- *Py_VaBuildValue()*
- *Py_Version*
- *Py_XNewRef()*
- *Py_buffer*
- Py_intptr_t
- *Py_ssize_t*
- Py_uintptr_t
- *allocfunc*
- *binaryfunc*
- *descrgetfunc*

- *descrsetfunc*
- *destructor*
- *getattrfunc*
- *getattrofunc*
- *getbufferproc*
- *getiterfunc*
- *getter*
- *hashfunc*
- *initproc*
- *inquiry*
- *iternextfunc*
- *lenfunc*
- *newfunc*
- *objobjargproc*
- *objobjproc*
- *releasebufferproc*
- *reprfunc*
- *richcmpfunc*
- *setattrfunc*
- *setattrofunc*
- *setter*
- *ssizeargfunc*
- *ssizeobjargproc*
- ssizessizeargfunc
- ssizessizeobjargproc
- symtable
- *ternaryfunc*
- *traverseproc*
- *unaryfunc*
- *vectorcallfunc*
- *visitproc*

CAPÍTULO 3

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.

Several of these functions accept a start symbol from the grammar as a parameter. The available start symbols are `Py_eval_input`, `Py_file_input`, and `Py_single_input`. These are described following the functions which accept them as parameters.

Note also that several of these functions take `FILE*` parameters. One particular issue which needs to be handled carefully is that the `FILE` structure for different C libraries can be different and incompatible. Under Windows (at least), it is possible for dynamically linked extensions to actually use different libraries, so care should be taken that `FILE*` parameters are only passed to these functions if it is certain that they were created by the same library that the Python runtime is using.

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)

This is a simplified interface to `PyRun_SimpleStringFlags()` below, leaving the `PyCompilerFlags`* argument set to `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.

Note that if an otherwise unhandled `SystemExit` is raised, this function will not return `-1`, but exit the process, as long as *`PyConfig.inspect`* is zero.

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 fichero, se decodifica desde *filesystem encoding and error handler*. Si *closeit* es verdadero, el fichero 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 del *manejador de codificación y errores del sistema de archivos*.

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 del *manejador de codificación y errores del sistema de archivos*. Retorna `0` en EOF o un número negativo en caso de falla.

int (***`PyOS_InputHook`**)(void)

Part of the Stable ABI. 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.

Distinto en la versión 3.12: This function is only called from the *main interpreter*.

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()*.

Distinto en la versión 3.12: This function is only called from the *main interpreter*.

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 de *fp* en lugar de una cadena de caracteres en memoria. *filename* debe ser el nombre del fichero, es decodificado desde el *filesystem encoding and error handler*. Si *closeit* es verdadero, el fichero se cierra antes de que *PyRun_FileExFlags()* retorne.

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

Return value: New reference. Part of the Stable ABI. 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. Parse and compile the Python source code in *str*, returning the resulting code object. The start token is given by *start*; this can be used to constrain the code which can be compiled and should be *Py_eval_input*, *Py_file_input*, or *Py_single_input*. The filename specified by *filename* is used to construct the code object and may appear in tracebacks or `SyntaxError` exception messages. This returns `NULL` if the code cannot be parsed or compiled.

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))

Added in version 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 bytes decodificada a partir del *manejador de codificación y errores del sistema de archivos*.

Added in version 3.2.

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

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. 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.

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

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. 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 lanza 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 leídos 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*.

Whenever PyCompilerFlags *flags is NULL, *cf_flags* is treated as equal to 0, and any modification due to from __future__ import is discarded.

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`.

The field is ignored by default, it is used if and only if `PyCF_ONLY_AST` flag is set in *cf_flags*.

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

The available compiler flags are accessible as macros:

PyCF_ALLOW_TOP_LEVEL_AWAIT

PyCF_ONLY_AST

PyCF_OPTIMIZED_AST

PyCF_TYPE_COMMENTS

See compiler flags in documentation of the `ast` Python module, which exports these constants under the same names.

The «`PyCF`» flags above can be combined with «`CO_FUTURE`» flags such as *CO_FUTURE_ANNOTATIONS* to enable features normally selectable using future statements. See *Code Object Flags* for a complete list.

CAPÍTULO 4

Conteo de referencias

Las funciones y macros de esta sección se utilizan para administrar conteos de referencia de objetos en Python.

Py_ssize_t **Py_REFCNT**(*PyObject* *o)

Part of the Stable ABI *since version 3.14.* Obtiene el recuento de referencias para el objeto de Python *o*.

Ten cuenta que el valor devuelto puede que no reflejar cuantas referencias al objecto existen realmente. Por ejemplo, algunos objetos son *immortal* y tienen un refcount muy alto que no refleja el número real de referencias. Por lo tanto, no confíes en que el valor devuelto sea preciso, salvo cuando sea 0 o 1.

Usa la función `Py_SET_REFCNT()` para establecer la cuenta de referencias de un objeto.

> **Nota**
>
> On *free threaded* builds of Python, returning 1 isn't sufficient to determine if it's safe to treat *o* as having no access by other threads. Use `PyUnstable_Object_IsUniquelyReferenced()` for that instead.
>
> See also the function `PyUnstable_Object_IsUniqueReferencedTemporary()`.

Distinto en la versión 3.10: `Py_REFCNT()` se convierte en una función estática en línea.

Distinto en la versión 3.11: El tipo de parámetro ya no es `const PyObject*`.

void **Py_SET_REFCNT**(*PyObject* *o, *Py_ssize_t* refcnt)

Establece la cuenta de referencias del objeto *o* al valor *refcnt*.

En compilación de Python con Free Threading, si *refcnt* es mas mayor que `UINT32_MAX`, el objeto se convierte en *immortal*.

Esta función no afecta a los objetos *immortal*.

Added in version 3.9.

Distinto en la versión 3.12: Los objetos inmortales no se modifican.

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

Indica tomar una nueva *strong reference* al objeto *o*, lo que indica que está en uso y no debe ser destruido.

Esta función no afecta a los objetos *immortal*.

Esta función se usa generalmente para convertir un *borrowed reference* en un *strong reference* en su lugar. La función `Py_NewRef()` se puede utilizar para crear un nuevo *strong reference*.

Cuando se termine de usar el objeto, se libera llamando a *Py_DECREF()*.

El objeto no debe ser NULL; si no está seguro de que no sea NULL, use *Py_XINCREF()*.

No esperes que esta función modifique realmente *o* de ninguna manera. Al menos para **algunos objetos**, esta función no tiene ningún efecto.

Distinto en la versión 3.12: Los objetos inmortales no se modifican.

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

Similar a *Py_INCREF()*, pero el objeto *o* puede ser NULL, en cuyo caso esto no tiene efecto.

Ver también *Py_XNewRef()*.

PyObject ***Py_NewRef**(*PyObject* *o)

Part of the Stable ABI *since version 3.10.* Crea una nueva *strong reference* a un objeto: llama a *Py_INCREF()* sobre *o* y devuelve el objeto *o*.

Cuando la *strong reference* ya no sea necesaria, se debe llamar a *Py_DECREF()* para disminuir el recuento de referencias del objeto.

El objeto *o* no debe ser NULL; use *Py_XNewRef()* si *o* puede ser NULL.

Por ejemplo:

```
Py_INCREF(obj);
self->attr = obj;
```

puede ser escrito como:

```
self->attr = Py_NewRef(obj);
```

Ver también *Py_INCREF()*.

Added in version 3.10.

PyObject ***Py_XNewRef**(*PyObject* *o)

Part of the Stable ABI *since version 3.10.* Similar a *Py_NewRef()*, pero el objeto *o* puede ser NULL.

Si el objeto *o* es NULL, la función solo retorna NULL.

Added in version 3.10.

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

Libera una *strong reference* al objeto *o*, indicando que la referencia ya no se usa.

Esta función no afecta a los objetos *immortal*.

Una vez que la última *strong reference* sea liberada (por ejemplo, cuando la cuenta de referencias del objeto llegue a 0), se invoca la función de desasignación del tipo de objeto (la cual no debe ser NULL).

Esta función se usa generalmente para eliminar un *strong reference* antes de salir de su alcance.

El objeto no debe ser NULL; si no está seguro de que no sea NULL, use *Py_XINCREF()*.

No esperes que esta función modifique realmente *o* de ninguna manera. Al menos para **algunos objetos**, esta función no tiene ningún efecto.

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 el método `__del__()`). Mientras las excepciones en dicho código no sean propagadas, el código ejecutado tendrá acceso libre a todas las variables globales de Python. Esto significa que cualquier objeto al que se pueda acceder desde una variable global debería estar en un estado coherente antes de invocar a *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.

Distinto en la versión 3.12: Los objetos inmortales no se modifican.

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

Similar a `Py_DECREF()`, pero el objeto *o* puede ser `NULL`, en cuyo caso esto no tendría efecto alguno. El mismo aviso de `Py_DECREF()` aplica aquí también.

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

Libera una *strong reference* del objeto *o*. El objeto puede ser `NULL`, en cuyo caso el macro no tiene efecto; de lo contrario, el efecto es el mismo que el de `Py_DECREF()`, excepto que el argumento también se establece en `NULL`. La advertencia de `Py_DECREF()` no se aplica en este caso, ya que el macro usa cuidadosamente una variable temporal y asigna `NULL` al argumento antes de liberar la referencia.

Es buena idea usar este macro al liberar una referencia de un objeto que podría ser recorrido durante la recolección de basura.

Distinto en la versión 3.12: Ahora, el macro argumento solo se evalúa una vez. Si el argumento tiene efectos secundarios, estos ya no se duplican.

void **Py_IncRef**(*PyObject* *o)

Part of the Stable ABI. Indica la toma de una nueva *strong reference* al objeto *o*. Es una versión en forma de función de `Py_XINCREF()`. Puede utilizarse para la integración dinámica de Python en tiempo de ejecución.

void **Py_DecRef**(*PyObject* *o)

Part of the Stable ABI. Libera una *strong reference* al objeto *o*. Una versión en forma de función de `Py_XDECREF()`. Puede utilizarse para la integración dinámica de Python en tiempo de ejecución.

Py_SETREF(dst, src)

Un macro que libera de forma segura un *strong reference* al objeto *dst* y establece *dst* al valor *src*.

Como en el caso de `Py_CLEAR()`, el código «obvio» puede ser mortal:

```
Py_DECREF(dst);
dst = src;
```

La forma segura es:

```
Py_SETREF(dst, src);
```

That arranges to set *dst* to *src before* releasing the reference to the old value of *dst*, so that any code triggered as a side-effect of *dst* getting torn down no longer believes *dst* points to a valid object.

Added in version 3.6.

Distinto en la versión 3.12: Los macro argumentos ahora solo se evalúan una vez. Si algún argumento tiene efectos secundarios, estos ya no se duplican.

Py_XSETREF(dst, src)

Un variante del macro `Py_SETREF` que usa `Py_XDECREF()` en lugar de `Py_DECREF()`.

Added in version 3.6.

Distinto en la versión 3.12: Los macro argumentos ahora solo se evalúan una vez. Si algún argumento tiene efectos secundarios, estos ya no se duplican.

CAPÍTULO 5

Manejo de excepciones

Las funciones descritas en este capítulo le permitirán manejar y lanzar 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 hilo) del último error que ocurrió. La mayoría de las funciones de C API no borran esto en caso de éxito, pero lo configurarán 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

The error indicator is **not** the result of `sys.exc_info()`. The former corresponds to an exception that is not yet caught (and is therefore still propagating), while the latter returns an exception after it is caught (and has therefore stopped propagating).

5.1 Impresión y limpieza

void **`PyErr_Clear`**()

Part of the Stable ABI. Borra el indicador de error. Si el indicador de error no está configurado, no hay efecto.

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

Part of the Stable ABI. 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!

If *set_sys_last_vars* is nonzero, the variable `sys.last_exc` is set to the printed exception. For backwards compatibility, the deprecated variables `sys.last_type`, `sys.last_value` and `sys.last_traceback` are also set to the type, value and traceback of this exception, respectively.

Distinto en la versión 3.12: The setting of `sys.last_exc` was added.

void **PyErr_Print**()

Part of the Stable ABI. Alias para `PyErr_PrintEx(1)`.

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

Part of the Stable ABI. Llama `sys.unraisablehook()` utilizando la excepción actual y el argumento *obj*.

This utility function prints a warning message to `sys.stderr` when an exception has been set but it is impossible for the interpreter to actually raise the exception. It is used, for example, when an exception occurs in an `__del__()` method.

The function is called with a single argument *obj* that identifies the context in which the unraisable exception occurred. If possible, the repr of *obj* will be printed in the warning message. If *obj* is `NULL`, only the traceback is printed.

Se debe establecer una excepción al llamar a esta función.

Distinto en la versión 3.4: Print a traceback. Print only traceback if *obj* is `NULL`.

Distinto en la versión 3.8: Use `sys.unraisablehook()`.

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

Similar to *PyErr_WriteUnraisable()*, but the *format* and subsequent parameters help format the warning message; they have the same meaning and values as in *PyUnicode_FromFormat()*. `PyErr_WriteUnraisable(obj)` is roughly equivalent to `PyErr_FormatUnraisable("Exception ignored in: %R", obj)`. If *format* is `NULL`, only the traceback is printed.

Added in version 3.13.

void **PyErr_DisplayException**(*PyObject* *exc)

Part of the Stable ABI *since version 3.12*. Print the standard traceback display of `exc` to `sys.stderr`, including chained exceptions and notes.

Added in version 3.12.

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)

Part of the Stable ABI. 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 create a new *strong reference* to it (e.g. with *Py_INCREF()*). The second argument is an error message; it is decoded from `'utf-8'`.

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

Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI *since version 3.5.* Igual que `PyErr_Format()`, pero tomando un argumento `va_list` en lugar de un número variable de argumentos.

Added in version 3.5.

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

Part of the Stable ABI. Esta es una abreviatura de `PyErr_SetObject(type, Py_None)`.

int **PyErr_BadArgument**()

Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. This is a convenience function to raise an exception when a C library function has returned an error and set the C variable `errno`. It constructs a tuple object whose first item is the integer `errno` value and whose second item is the corresponding error message (gotten from `strerror()`), and then calls `PyErr_SetObject(type, object)`. On Unix, when the `errno` value is `EINTR`, indicating an interrupted system call, this calls `PyErr_CheckSignals()`, and if that set the error indicator, leaves it set to that. The function always returns `NULL`, so a wrapper function around a system call can write `return PyErr_SetFromErrno(type);` when the system call returns an error.

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

Return value: Always NULL. Part of the Stable ABI. Similar to `PyErr_SetFromErrno()`, with the additional behavior that if *filenameObject* is not `NULL`, it is passed to the constructor of *type* as a third parameter. In the case of `OSError` exception, this is used to define the `filename` attribute of the exception instance.

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

Return value: Always NULL. Part of the Stable ABI *since version 3.7.* Similar a `PyErr_SetFromErrnoWithFilenameObject()`, pero toma un segundo objeto de nombre de archivo, para lanzar errores cuando falla una función que toma dos nombres de archivo.

Added in version 3.4.

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

Return value: Always NULL. Part of the Stable ABI. 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 de *filesystem encoding and error handler*.

PyObject ***PyErr_SetFromWindowsErr**(int ierr)

Return value: Always NULL. Part of the Stable ABI *on Windows since version 3.7.* This is a convenience function to raise `OSError`. If called with *ierr* of `0`, the error code returned by a call to `GetLastError()` is used instead. It calls the Win32 function `FormatMessage()` to retrieve the Windows description of error code given by *ierr* or `GetLastError()`, then it constructs a `OSError` object with the `winerror` attribute set to the error code, the `strerror` attribute set to the corresponding error message (gotten from `FormatMessage()`), and then calls `PyErr_SetObject(PyExc_OSError, object)`. This function always returns `NULL`.

Availability: Windows.

PyObject ***PyErr_SetExcFromWindowsErr**(*PyObject* *type, int ierr)

Return value: Always NULL. Part of the Stable ABI *on Windows since version 3.7.* Similar a `PyErr_SetFromWindowsErr()`, con un parámetro adicional que especifica el tipo de excepción que se lanzará.

Availability: Windows.

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

Return value: Always NULL. Part of the Stable ABI *on Windows since version 3.7.* Similar to `PyErr_SetFromWindowsErr()`, with the additional behavior that if *filename* is not `NULL`, it is decoded from the filesystem encoding (`os.fsdecode()`) and passed to the constructor of `OSError` as a third parameter to be used to define the `filename` attribute of the exception instance.

Availability: Windows.

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

Return value: Always NULL. Part of the Stable ABI *on Windows since version 3.7.* Similar to `PyErr_SetExcFromWindowsErr()`, with the additional behavior that if *filename* is not `NULL`, it is passed to the constructor of `OSError` as a third parameter to be used to define the `filename` attribute of the exception instance.

Availability: Windows.

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

Return value: Always NULL. Part of the Stable ABI *on Windows since version 3.7.* Similar a `PyErr_SetExcFromWindowsErrWithFilenameObject()`, pero acepta un segundo objeto de nombre de archivo.

Availability: Windows.

Added in version 3.4.

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

Return value: Always NULL. Part of the Stable ABI *on Windows since version 3.7.* Similar a `PyErr_SetFromWindowsErrWithFilename()`, con un parámetro adicional que especifica el tipo de excepción que se lanzará.

Availability: Windows.

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

Return value: Always NULL. Part of the Stable ABI *since version 3.7.* 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`.

Added in version 3.3.

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

Return value: Always NULL. Part of the Stable ABI *since version 3.6.* Al igual que `PyErr_SetImportError()` pero esta función permite especificar una subclase de `ImportError` para aumentar.

Added in version 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`.

Added in version 3.4.

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

Part of the Stable ABI *since version 3.7.* Como `PyErr_SyntaxLocationObject()`, pero *filename* es una cadena de bytes decodificada a partir de *filesystem encoding and error handler*.

Added in version 3.2.

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

Part of the Stable ABI. Como `PyErr_SyntaxLocationEx()`, pero se omite el parámetro *col_offset*.

void **PyErr_BadInternalCall**()

Part of the Stable ABI. 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 lanzarán 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 lanza una excepción, o `-1` si se lanza 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)

Part of the Stable ABI. 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 *Warning types*.

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)

Emite un mensaje de advertencia con control explícito sobre todos los atributos de advertencia. Este es un contenedor sencillo alrededor de la función Python `warnings.warn_explicit()`; consulte allí para obtener más información. Los argumentos *module* y *registry* pueden establecerse en `NULL` para obtener el efecto predeterminado que se describe allí.

Added in version 3.4.

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

Part of the Stable ABI. Similar a *PyErr_WarnExplicitObject()* excepto que *message* y *module* son cadenas codificadas UTF-8, y *filename* se decodifica de *filesystem encoding and error handler*.

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

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

Added in version 3.2.

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

Part of the Stable ABI *since version 3.6.* Function similar to *PyErr_WarnFormat()*, but *category* is `ResourceWarning` and it passes *source* to `warnings.WarningMessage`.

Added in version 3.6.

5.4 Consultando el indicador de error

PyObject ***PyErr_Occurred**()

Return value: Borrowed reference. Part of the Stable ABI. 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()`*.

The caller must have an *attached thread state*.

> **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)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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).

PyObject ***PyErr_GetRaisedException**(void)

Return value: New reference. Part of the Stable ABI *since version 3.12.* Return the exception currently being raised, clearing the error indicator at the same time. Return `NULL` if the error indicator is not set.

This function is used by code that needs to catch exceptions, or code that needs to save and restore the error indicator temporarily.

For example:

```
{
   PyObject *exc = PyErr_GetRaisedException();

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

   PyErr_SetRaisedException(exc);
}
```

> **Ver también**
>
> *`PyErr_GetHandledException()`*, to save the exception currently being handled.

Added in version 3.12.

void **PyErr_SetRaisedException**(*PyObject* *exc)

Part of the Stable ABI *since version 3.12.* Set *exc* as the exception currently being raised, clearing the existing exception if one is set.

Advertencia

This call steals a reference to *exc*, which must be a valid exception.

Added in version 3.12.

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

Part of the Stable ABI. Obsoleto desde la versión 3.12: Use `PyErr_GetRaisedException()` instead.

Recupera 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

This function is normally only used by legacy code that needs to catch exceptions or save and restore the error indicator temporarily.

For example:

```
{
   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)

Part of the Stable ABI. Obsoleto desde la versión 3.12: Use `PyErr_SetRaisedException()` instead.

Set the error indicator from the three objects, *type*, *value*, and *traceback*, clearing the existing exception if one is set. If the objects are `NULL`, the error indicator is cleared. Do not pass a `NULL` type and non-`NULL` value or traceback. The exception type should be a class. Do not pass an invalid exception type or value. (Violating these rules will cause subtle problems later.) This call takes away a reference to each object: you must own a reference to each object before the call and after the call you no longer own these references. (If you don't understand this, don't use this function. I warned you.)

Nota

This function is normally only used by legacy code that needs to save and restore the error indicator temporarily. Use `PyErr_Fetch()` to save the current error indicator.

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

Part of the Stable ABI. Obsoleto desde la versión 3.12: Use `PyErr_GetRaisedException()` instead, to avoid any possible de-normalization.

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

This function *does not* implicitly set the `__traceback__` attribute on the exception value. If setting the traceback appropriately is desired, the following additional snippet is needed:

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

PyObject ***PyErr_GetHandledException**(void)

Part of the Stable ABI *since version 3.11.* Recupera la instancia de excepción activa, como la que devolvería `sys.exception()`. Esto se refiere a una excepción que *ya fue capturada*, no a una excepción recién lanzada. Retorna una nueva referencia a la excepción o `NULL`. No modifica el estado de excepción del intérprete.

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_SetHandledException()`* para restaurar o borrar el estado de excepción.

Added in version 3.11.

void **PyErr_SetHandledException**(*PyObject* *exc)

Part of the Stable ABI *since version 3.11.* Establece la excepción activa, como se conoce de `sys.exception()`. Esto se refiere a la excepción que **ya fue capturada**, no a una excepción que fue lanzada recientemente. Para borrar el estado de la excepción, pasa `NULL`.

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_GetHandledException()`* para leer el estado de excepción.

Added in version 3.11.

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

Part of the Stable ABI *since version 3.7.* Recupera 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 fue lanzada recientemente. Retorna nuevas referencias para los tres objetos, cualquiera de los cuales puede ser `NULL`. No modifica el estado de información de excepción. Esta función se mantiene por retro-compatibilidad. Es preferible usar *`PyErr_GetHandledException()`*.

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.

Added in version 3.3.

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

Part of the Stable ABI *since version 3.7.* Establece 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 fue lanzada 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_SetHandledException()`*.

> **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.

Added in version 3.3.

Distinto en la versión 3.11: Los argumentos `type` y `traceback` ya no se utilizan y pueden ser NULL. El intérprete los deriva ahora de la instancia de la excepción (el argumento `value`). La función sigue robando referencias de los tres argumentos.

5.5 Manejo de señal

int **PyErr_CheckSignals**()

Part of the Stable ABI. Esta función interactúa con el manejo de señales de Python.

Si la función se llama desde el hilo principal y bajo el intérprete principal de Python, verifica si se ha enviado una señal a los procesos y, de ser así, invoca el manejador de señales correspondiente. Si el módulo `signal` es compatible, esto puede invocar un manejador de señales escrito en Python.

La función intenta manejar todas las señales pendientes y luego devuelve `0`. Sin embargo, si un manejador de señales de Python lanza una excepción, el indicador de error se establece y la función devuelve `-1` inmediatamente (de modo que es posible que otras señales pendientes no se hayan manejado todavía: estarán en la siguiente invocación de *PyErr_CheckSignals()*).

Si la función se llama desde un hilo no principal, o bajo un intérprete de Python no principal, no hace nada y devuelve `0`.

Esta función se puede llamar mediante un código C de ejecución prolongada que quiere ser interrumpible por las peticiones del usuario (como presionar Ctrl-C).

> **Nota**
>
> The default Python signal handler for `SIGINT` raises the `KeyboardInterrupt` exception.

void **PyErr_SetInterrupt**()

Part of the Stable ABI. Simulate the effect of a `SIGINT` signal arriving. This is equivalent to `PyErr_SetInterruptEx(SIGINT)`.

> **Nota**
>
> This function is async-signal-safe. It can be called without an *attached thread state* and from a C signal handler.

int **PyErr_SetInterruptEx**(int signum)

Part of the Stable ABI *since version 3.10.* Simula el efecto de la llegada de una señal. La próxima vez que sea llamado *PyErr_CheckSignals()*, se llamará al manejador de señal de Python para el número de señal dado.

Esta función puede ser llamada por código C que configura su propio manejo de señales y quiere que los manejadores de señales de Python sean invocados como se espera cuando se solicita una interrupción (por ejemplo, cuando el usuario presiona Ctrl-C para interrumpir una operación).

If the given signal isn't handled by Python (it was set to `signal.SIG_DFL` or `signal.SIG_IGN`), it will be ignored.

Si *signum* está fuera del rango permitido de números de señal, se devuelve `-1`. De lo contrario, se devuelve `0`. Esta función nunca cambia el indicador de error.

> **Nota**
>
> This function is async-signal-safe. It can be called without an *attached thread state* and from a C signal handler.

Added in version 3.10.

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. Part of the Stable ABI. 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`).

The `__module__` attribute of the new class is set to the first part (up to the last dot) of the *name* argument, and the class name is set to the last part (after the last dot). The *base* argument can be used to specify alternate base classes; it can either be only one class or a tuple of classes. The *dict* argument can be used to specify a dictionary of class variables and methods.

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

Return value: New reference. Part of the Stable ABI. 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.

Added in version 3.2.

int **PyExceptionClass_Check**(*PyObject* *ob)

Return non-zero if *ob* is an exception class, zero otherwise. This function always succeeds.

const char ***PyExceptionClass_Name**(*PyObject* *ob)

Part of the Stable ABI *since version 3.8.* Return `tp_name` of the exception class *ob*.

5.7 Objetos excepción

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

Return value: New reference. Part of the Stable ABI. Return the traceback associated with the exception as a new reference, as accessible from Python through the `__traceback__` attribute. If there is no traceback associated, this returns `NULL`.

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

Part of the Stable ABI. Establezca el rastreo asociado con la excepción a *tb*. Use `Py_None` para borrarlo.

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

Return value: New reference. Part of the Stable ABI. Return the context (another exception instance during whose handling *ex* was raised) associated with the exception as a new reference, as accessible from Python through the `__context__` attribute. If there is no context associated, this returns `NULL`.

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

Part of the Stable ABI. 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. Part of the Stable ABI. Return the cause (either an exception instance, or `None`, set by `raise ... from ...`) associated with the exception as a new reference, as accessible from Python through the `__cause__` attribute.

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

Part of the Stable ABI. Set the cause associated with the exception to *cause*. Use `NULL` to clear it. There is no type check to make sure that *cause* is either an exception instance or `None`. This steals a reference to *cause*.

The `__suppress_context__` attribute is implicitly set to `True` by this function.

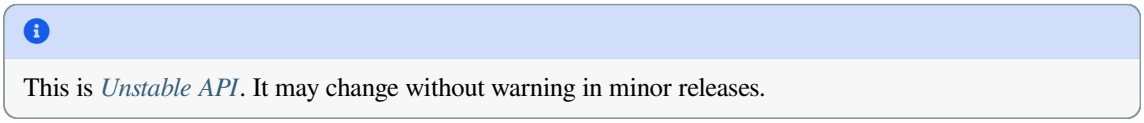
PyObject ***PyException_GetArgs**(*PyObject* *ex)

Return value: New reference. Part of the Stable ABI *since version 3.12*. Return `args` of exception *ex*.

void **PyException_SetArgs**(*PyObject* *ex, *PyObject* *args)

Part of the Stable ABI *since version 3.12*. Set `args` of exception *ex* to *args*.

PyObject ***PyUnstable_Exc_PrepReraiseStar**(*PyObject* *orig, *PyObject* *excs)



This is *Unstable API*. It may change without warning in minor releases.

Implement part of the interpreter's implementation of `except*`. *orig* is the original exception that was caught, and *excs* is the list of the exceptions that need to be raised. This list contains the unhandled part of *orig*, if any, as well as the exceptions that were raised from the `except*` clauses (so they have a different traceback from *orig*) and those that were reraised (and have the same traceback as *orig*). Return the `ExceptionGroup` that needs to be reraised in the end, or `None` if there is nothing to reraise.

Added in version 3.12.

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. Part of the Stable ABI. Crea un objeto `UnicodeDecodeError` con los atributos *encoding*, *object*, *length*, *start*, *end* y *reason*. *encoding* y *reason* son cadenas codificadas UTF-8.

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

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

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. 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)

Part of the Stable ABI. 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.

If the `UnicodeError.object` is an empty sequence, the resulting *start* is `0`. Otherwise, it is clipped to `[0, len(object) - 1]`.

Ver también

`UnicodeError.start`

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)

Part of the Stable ABI. Set the *start* attribute of the given exception object to *start*. Return `0` on success, `-1` on failure.

Nota

While passing a negative *start* does not raise an exception, the corresponding getters will not consider it as a relative offset.

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)

Part of the Stable ABI. 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.

If the `UnicodeError.object` is an empty sequence, the resulting *end* is `0`. Otherwise, it is clipped to `[1, len(object)]`.

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)

Part of the Stable ABI. Establece el atributo *end* del objeto de excepción dado en *end*. Retorna `0` en caso de éxito, `-1` en caso de error.

Ver también

`UnicodeError.end`

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

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

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

Return value: New reference. Part of the Stable ABI. 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)

Part of the Stable ABI. 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)

Part of the Stable ABI *since version 3.9.* Marca un punto donde una llamada recursiva de nivel C está a punto de realizarse.

The function then checks if the stack limit is reached. If this is the case, a `RecursionError` is set and a nonzero value is returned. Otherwise, zero is returned.

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: This function is now also available in the *limited API*.

void **Py_LeaveRecursiveCall**(void)

Part of the Stable ABI *since version 3.9.* Termina una *Py_EnterRecursiveCall()*. Se debe llamar una vez por cada invocación *exitosa* de *Py_EnterRecursiveCall()*.

Distinto en la versión 3.9: This function is now also available in the *limited API*.

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)

Part of the Stable ABI. 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)

Part of the Stable ABI. Termina a *Py_ReprEnter()*. Se debe llamar una vez por cada invocación de *Py_ReprEnter()* que retorna cero.

5.10 Exception and warning types

All standard Python exceptions and warning categories are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type *PyObject**; they are all class objects.

For completeness, here are all the variables:

5.10.1 Exception types

C name	Python name
PyObject ***PyExc_BaseException** *Part of the* Stable ABI.	`BaseException`

continúe en la próxima página

Tabla 1 – proviene de la página anterior

C name	Python name
PyObject ***PyExc_BaseExceptionGroup** *Part of the Stable ABI since version 3.11.*	`BaseExceptionGroup`
PyObject ***PyExc_Exception** *Part of the Stable ABI.*	`Exception`
PyObject ***PyExc_ArithmeticError** *Part of the Stable ABI.*	`ArithmeticError`
PyObject ***PyExc_AssertionError** *Part of the Stable ABI.*	`AssertionError`
PyObject ***PyExc_AttributeError** *Part of the Stable ABI.*	`AttributeError`
PyObject ***PyExc_BlockingIOError** *Part of the Stable ABI since version 3.7.*	`BlockingIOError`
PyObject ***PyExc_BrokenPipeError** *Part of the Stable ABI since version 3.7.*	`BrokenPipeError`
PyObject ***PyExc_BufferError** *Part of the Stable ABI.*	`BufferError`
PyObject ***PyExc_ChildProcessError** *Part of the Stable ABI since version 3.7.*	`ChildProcessError`
PyObject ***PyExc_ConnectionAbortedError** *Part of the Stable ABI since version 3.7.*	`ConnectionAbortedError`
PyObject ***PyExc_ConnectionError** *Part of the Stable ABI since version 3.7.*	`ConnectionError`
PyObject ***PyExc_ConnectionRefusedError** *Part of the Stable ABI since version 3.7.*	`ConnectionRefusedError`
PyObject ***PyExc_ConnectionResetError** *Part of the Stable ABI since version 3.7.*	`ConnectionResetError`

continúe en la próxima página

Tabla 1 – proviene de la página anterior

C name	Python name
PyObject ***PyExc_EOFError** *Part of the* Stable ABI.	`EOFError`
PyObject ***PyExc_FileExistsError** *Part of the* Stable ABI *since version 3.7.*	`FileExistsError`
PyObject ***PyExc_FileNotFoundError** *Part of the* Stable ABI *since version 3.7.*	`FileNotFoundError`
PyObject ***PyExc_FloatingPointError** *Part of the* Stable ABI.	`FloatingPointError`
PyObject ***PyExc_GeneratorExit** *Part of the* Stable ABI.	`GeneratorExit`
PyObject ***PyExc_ImportError** *Part of the* Stable ABI.	`ImportError`
PyObject ***PyExc_IndentationError** *Part of the* Stable ABI.	`IndentationError`
PyObject ***PyExc_IndexError** *Part of the* Stable ABI.	`IndexError`
PyObject ***PyExc_InterruptedError** *Part of the* Stable ABI *since version 3.7.*	`InterruptedError`
PyObject ***PyExc_IsADirectoryError** *Part of the* Stable ABI *since version 3.7.*	`IsADirectoryError`
PyObject ***PyExc_KeyError** *Part of the* Stable ABI.	`KeyError`
PyObject ***PyExc_KeyboardInterrupt** *Part of the* Stable ABI.	`KeyboardInterrupt`
PyObject ***PyExc_LookupError** *Part of the* Stable ABI.	`LookupError`

continúe en la próxima página

Tabla 1 – proviene de la página anterior

C name	Python name
PyObject ***PyExc_MemoryError** *Part of the* Stable ABI.	`MemoryError`
PyObject ***PyExc_ModuleNotFoundError** *Part of the* Stable ABI *since version 3.6.*	`ModuleNotFoundError`
PyObject ***PyExc_NameError** *Part of the* Stable ABI.	`NameError`
PyObject ***PyExc_NotADirectoryError** *Part of the* Stable ABI *since version 3.7.*	`NotADirectoryError`
PyObject ***PyExc_NotImplementedError** *Part of the* Stable ABI.	`NotImplementedError`
PyObject ***PyExc_OSError** *Part of the* Stable ABI.	`OSError`
PyObject ***PyExc_OverflowError** *Part of the* Stable ABI.	`OverflowError`
PyObject ***PyExc_PermissionError** *Part of the* Stable ABI *since version 3.7.*	`PermissionError`
PyObject ***PyExc_ProcessLookupError** *Part of the* Stable ABI *since version 3.7.*	`ProcessLookupError`
PyObject ***PyExc_PythonFinalizationError**	`PythonFinalizationError`
PyObject ***PyExc_RecursionError** *Part of the* Stable ABI *since version 3.7.*	`RecursionError`
PyObject ***PyExc_ReferenceError** *Part of the* Stable ABI.	`ReferenceError`
PyObject ***PyExc_RuntimeError** *Part of the* Stable ABI.	`RuntimeError`
PyObject ***PyExc_StopAsyncIteration** *Part of the* Stable ABI *since version 3.7.*	`StopAsyncIteration`

continúe en la próxima página

Tabla 1 – proviene de la página anterior

C name	Python name
PyObject ***PyExc_StopIteration** *Part of the* Stable ABI.	`StopIteration`
PyObject ***PyExc_SyntaxError** *Part of the* Stable ABI.	`SyntaxError`
PyObject ***PyExc_SystemError** *Part of the* Stable ABI.	`SystemError`
PyObject ***PyExc_SystemExit** *Part of the* Stable ABI.	`SystemExit`
PyObject ***PyExc_TabError** *Part of the* Stable ABI.	`TabError`
PyObject ***PyExc_TimeoutError** *Part of the* Stable ABI *since version 3.7.*	`TimeoutError`
PyObject ***PyExc_TypeError** *Part of the* Stable ABI.	`TypeError`
PyObject ***PyExc_UnboundLocalError** *Part of the* Stable ABI.	`UnboundLocalError`
PyObject ***PyExc_UnicodeDecodeError** *Part of the* Stable ABI.	`UnicodeDecodeError`
PyObject ***PyExc_UnicodeEncodeError** *Part of the* Stable ABI.	`UnicodeEncodeError`
PyObject ***PyExc_UnicodeError** *Part of the* Stable ABI.	`UnicodeError`
PyObject ***PyExc_UnicodeTranslateError** *Part of the* Stable ABI.	`UnicodeTranslateError`
PyObject ***PyExc_ValueError** *Part of the* Stable ABI.	`ValueError`

continúe en la próxima página

Tabla 1 – proviene de la página anterior

C name	Python name
PyObject ***PyExc_ZeroDivisionError** *Part of the* Stable ABI.	`ZeroDivisionError`

Added in version 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**.

Added in version 3.5: *PyExc_StopAsyncIteration* y *PyExc_RecursionError*.

Added in version 3.6: *PyExc_ModuleNotFoundError*.

Added in version 3.11: *PyExc_BaseExceptionGroup*.

5.10.2 OSError aliases

The following are a compatibility aliases to *PyExc_OSError*.

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

C name	Python name	Notas
PyObject ***PyExc_EnvironmentError** *Part of the* Stable ABI.	`OSError`	
PyObject ***PyExc_IOError** *Part of the* Stable ABI.	`OSError`	
PyObject ***PyExc_WindowsError** *Part of the* Stable ABI *on Windows since version 3.7.*	`OSError`	[win]

Notas:

5.10.3 Warning types

C name	Python name
PyObject ***PyExc_Warning** *Part of the* Stable ABI.	`Warning`
PyObject ***PyExc_BytesWarning** *Part of the* Stable ABI.	`BytesWarning`
PyObject ***PyExc_DeprecationWarning** *Part of the* Stable ABI.	`DeprecationWarning`
PyObject ***PyExc_EncodingWarning** *Part of the* Stable ABI *since version 3.10.*	`EncodingWarning`
PyObject ***PyExc_FutureWarning** *Part of the* Stable ABI.	`FutureWarning`
PyObject ***PyExc_ImportWarning** *Part of the* Stable ABI.	`ImportWarning`
PyObject ***PyExc_PendingDeprecationWarning** *Part of the* Stable ABI.	`PendingDeprecationWarning`
PyObject ***PyExc_ResourceWarning** *Part of the* Stable ABI *since version 3.7.*	`ResourceWarning`
PyObject ***PyExc_RuntimeWarning** *Part of the* Stable ABI.	`RuntimeWarning`
PyObject ***PyExc_SyntaxWarning** *Part of the* Stable ABI.	`SyntaxWarning`
PyObject ***PyExc_UnicodeWarning** *Part of the* Stable ABI.	`UnicodeWarning`
PyObject ***PyExc_UserWarning** *Part of the* Stable ABI.	`UserWarning`

Added in version 3.2: *PyExc_ResourceWarning*.

Added in version 3.10: *PyExc_EncodingWarning*.

CAPÍTULO 6

Defining extension modules

A C extension for CPython is a shared library (for example, a `.so` file on Linux, `.pyd` DLL on Windows), which is loadable into the Python process (for example, it is compiled with compatible compiler settings), and which exports an *initialization function*.

To be importable by default (that is, by `importlib.machinery.ExtensionFileLoader`), the shared library must be available on `sys.path`, and must be named after the module name plus an extension listed in `importlib.machinery.EXTENSION_SUFFIXES`.

Nota

Building, packaging and distributing extension modules is best done with third-party tools, and is out of scope of this document. One suitable tool is Setuptools, whose documentation can be found at https://setuptools.pypa.io/en/latest/setuptools.html.

Normally, the initialization function returns a module definition initialized using `PyModuleDef_Init()`. This allows splitting the creation process into several phases:

- Before any substantial code is executed, Python can determine which capabilities the module supports, and it can adjust the environment or refuse loading an incompatible extension.
- By default, Python itself creates the module object – that is, it does the equivalent of `object.__new__()` for classes. It also sets initial attributes like `__package__` and `__loader__`.
- Afterwards, the module object is initialized using extension-specific code – the equivalent of `__init__()` on classes.

This is called *multi-phase initialization* to distinguish it from the legacy (but still supported) *single-phase initialization* scheme, where the initialization function returns a fully constructed module. See the *single-phase-initialization section below* for details.

Distinto en la versión 3.5: Added support for multi-phase initialization (**PEP 489**).

6.1 Multiple module instances

By default, extension modules are not singletons. For example, if the `sys.modules` entry is removed and the module is re-imported, a new module object is created, and typically populated with fresh method and type objects. The old module is subject to normal garbage collection. This mirrors the behavior of pure-Python modules.

Additional module instances may be created in *sub-interpreters* or after Python runtime reinitialization (`Py_Finalize()` and `Py_Initialize()`). In these cases, sharing Python objects between module instances would likely cause crashes or undefined behavior.

To avoid such issues, each instance of an extension module should be *isolated*: changes to one instance should not implicitly affect the others, and all state owned by the module, including references to Python objects, should be specific to a particular module instance. See isolating-extensions-howto for more details and a practical guide.

A simpler way to avoid these issues is raising an error on repeated initialization.

All modules are expected to support *sub-interpreters*, or otherwise explicitly signal a lack of support. This is usually achieved by isolation or blocking repeated initialization, as above. A module may also be limited to the main interpreter using the `Py_mod_multiple_interpreters` slot.

6.2 Initialization function

The initialization function defined by an extension module has the following signature:

PyObject ***PyInit_modulename**(void)

Its name should be `PyInit_<name>`, with `<name>` replaced by the name of the module.

For modules with ASCII-only names, the function must instead be named `PyInit_<name>`, with `<name>` replaced by the name of the module. When using *Multi-phase initialization*, non-ASCII module names are allowed. In this case, the initialization function name is `PyInitU_<name>`, with `<name>` encoded using Python's *punycode* encoding with hyphens replaced by underscores. In Python:

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

It is recommended to define the initialization function using a helper macro:

PyMODINIT_FUNC

Declare an extension module initialization function. This macro:

- specifies the `PyObject*` return type,
- adds any special linkage declarations required by the platform, and
- for C++, declares the function as `extern "C"`.

For example, a module called `spam` would be defined like this:

```
static struct PyModuleDef spam_module = {
    .m_base = PyModuleDef_HEAD_INIT,
    .m_name = "spam",
    ...
};

PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModuleDef_Init(&spam_module);
}
```

It is possible to export multiple modules from a single shared library by defining multiple initialization functions. However, importing them requires using symbolic links or a custom importer, because by default only the function corresponding to the filename is found. See the Multiple modules in one library section in **PEP 489** for details.

The initialization function is typically the only non-`static` item defined in the module's C source.

6.3 Multi-phase initialization

Normally, the *initialization function* (`PyInit_modulename`) returns a *`PyModuleDef`* instance with non-`NULL` *`m_slots`*. Before it is returned, the `PyModuleDef` instance must be initialized using the following function:

PyObject ***PyModuleDef_Init**(*PyModuleDef* *def)

Part of the Stable ABI *since version 3.5.* Ensure a module definition is a properly initialized Python object that correctly reports its type and a reference count.

Return *def* cast to `PyObject*`, or `NULL` if an error occurred.

Calling this function is required for *Multi-phase initialization*. It should not be used in other contexts.

Note that Python assumes that `PyModuleDef` structures are statically allocated. This function may return either a new reference or a borrowed one; this reference must not be released.

Added in version 3.5.

6.4 Legacy single-phase initialization

Atención

Single-phase initialization is a legacy mechanism to initialize extension modules, with known drawbacks and design flaws. Extension module authors are encouraged to use multi-phase initialization instead.

In single-phase initialization, the *initialization function* (`PyInit_modulename`) should create, populate and return a module object. This is typically done using *`PyModule_Create()`* and functions like *`PyModule_AddObjectRef()`*.

Single-phase initialization differs from the *default* in the following ways:

- Single-phase modules are, or rather *contain*, "singletons".

  When the module is first initialized, Python saves the contents of the module's `__dict__` (that is, typically, the module's functions and types).

  For subsequent imports, Python does not call the initialization function again. Instead, it creates a new module object with a new `__dict__`, and copies the saved contents to it. For example, given a single-phase module `_testsinglephase`[1] that defines a function `sum` and an exception class `error`:

  ```
  >>> import sys
  >>> import _testsinglephase as one
  >>> del sys.modules['_testsinglephase']
  >>> import _testsinglephase as two
  >>> one is two
  False
  >>> one.__dict__ is two.__dict__
  False
  >>> one.sum is two.sum
  True
  >>> one.error is two.error
  True
  ```

  The exact behavior should be considered a CPython implementation detail.

- To work around the fact that `PyInit_modulename` does not take a *spec* argument, some state of the import machinery is saved and applied to the first suitable module created during the `PyInit_modulename` call. Specifically, when a sub-module is imported, this mechanism prepends the parent package name to the name of the module.

[1] `_testsinglephase` is an internal module used in CPython's self-test suite; your installation may or may not include it.

A single-phase `PyInit_modulename` function should create "its" module object as soon as possible, before any other module objects can be created.

- Non-ASCII module names (`PyInitU_modulename`) are not supported.
- Single-phase modules support module lookup functions like *PyState_FindModule()*.

CAPÍTULO 7

Utilidades

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.

7.1 Utilidades del sistema operativo

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

Return value: New reference. Part of the Stable ABI *since version 3.6.* Return the file system representation for *path*. If the object is a `str` or `bytes` object, then a new *strong reference* is returned. If the object implements the `os.PathLike` interface, then `__fspath__()` is returned as long as it is a `str` or `bytes` object. Otherwise `TypeError` is raised and `NULL` is returned.

Added in version 3.6.

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

Return true (nonzero) if the standard I/O file *fp* with name *filename* is deemed interactive. This is the case for files for which `isatty(fileno(fp))` is true. If the *`PyConfig.interactive`* is non-zero, this function also returns true if the *filename* pointer is `NULL` or if the name is equal to one of the strings `'<stdin>'` or `'???'`.

This function must not be called before Python is initialized.

void **PyOS_BeforeFork**()

Part of the Stable ABI *on platforms with fork() since version 3.7.* 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()`.

Added in version 3.7.

void **PyOS_AfterFork_Parent**()

Part of the Stable ABI *on platforms with fork() since version 3.7.* 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()`.

Added in version 3.7.

void **PyOS_AfterFork_Child**()

Part of the Stable ABI *on platforms with fork() since version 3.7.* 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()`.

Added in version 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**()

Part of the Stable ABI *on platforms with fork().* 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**()

Part of the Stable ABI *on platforms with USE_STACKCHECK since version 3.7.* Return true when the interpreter runs out of stack space. This is a reliable check, but is only available when `USE_STACKCHECK` is defined (currently on certain versions of Windows using the Microsoft Visual C++ compiler). `USE_STACKCHECK` will be defined automatically; you should never change the definition in your own code.

typedef void (***PyOS_sighandler_t**)(int)

Part of the Stable ABI.

PyOS_sighandler_t **PyOS_getsig**(int i)

Part of the Stable ABI. Return the current signal handler for signal *i*. This is a thin wrapper around either `sigaction()` or `signal()`. Do not call those functions directly!

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

Part of the Stable ABI. Set the signal handler for signal *i* to be *h*; return the old signal handler. This is a thin wrapper around either `sigaction()` or `signal()`. Do not call those functions directly!

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

Part of the Stable ABI *since version 3.7.*

> **Advertencia**
>
> Esta función no debe llamarse directamente: utilice la API `PyConfig` con la función `PyConfig_SetBytesString()` que asegura que *Python está preinicializado*.
>
> Esta función no debe llamarse antes de que *Python esté preinicializado* y para que la configuración local LC_CTYPE esté correctamente configurada: véase la función `Py_PreInitialize()`.

Decodifica una cadena de bytes a partir del *manejador de codificación y errores del sistema de archivos*. Si el controlador de error es el controlador de error surrogateescape, los bytes no codificables se decodifican como caracteres en el rango U+DC80..U+DCFF; y si una secuencia de bytes se puede decodificar como un carácter sustituto, escape los bytes usando el controlador de error surrogateescape en lugar de decodificarlos.

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.

El *filesystem encoding and error handler* son seleccionados por `PyConfig_Read()`: ver `filesystem_encoding` y `filesystem_errors` que pertenecen a `PyConfig`.

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()`.

Added in version 3.5.

Distinto en la versión 3.7: La función ahora utiliza la codificación UTF-8 en el Modo Python UTF-8.

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

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

Part of the Stable ABI *since version 3.7.* Codifica una cadena de caracteres amplios según el término *filesystem encoding and error handler*. Si el gestor de errores es surrogateescape error handler, los caracteres sustituidos en el rango U+DC80..U+DCFF se convierten en bytes 0x80..0xFF.

Retorna un puntero a una cadena de bytes recién asignada, usa `PyMem_Free()` para liberar la memoria. Retorna `NULL` si se genera un error de codificación o error de asignación de memoria.

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.

El *filesystem encoding and error handler* son seleccionados por `PyConfig_Read()`: ver `filesystem_encoding` y `filesystem_errors` que pertenecen a `PyConfig`.

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

> **Advertencia**

Esta función no debe llamarse antes de que *Python esté preinicializado* y para que la configuración local LC_CTYPE esté correctamente configurada: véase la función `Py_PreInitialize()`.

Ver también

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

Added in version 3.5.

Distinto en la versión 3.7: La función ahora utiliza la codificación UTF-8 en el Modo Python UTF-8.

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

FILE ***Py_fopen**(*PyObject* *path, const char *mode)

Similar to `fopen()`, but *path* is a Python object and an exception is set on error.

path must be a `str` object, a `bytes` object, or a *path-like object*.

On success, return the new file pointer. On error, set an exception and return `NULL`.

The file must be closed by `Py_fclose()` rather than calling directly `fclose()`.

The file descriptor is created non-inheritable (**PEP 446**).

The caller must have an *attached thread state*.

Added in version 3.14.

int **Py_fclose**(FILE *file)

Close a file that was opened by `Py_fopen()`.

On success, return `0`. On error, return `EOF` and `errno` is set to indicate the error. In either case, any further access (including another call to `Py_fclose()`) to the stream results in undefined behavior.

Added in version 3.14.

7.2 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 subproceso actual del intérprete, que está contenido en la estructura interna del estado del subproceso.

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

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

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

Part of the Stable ABI. 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**()

Part of the Stable ABI. Restablece `sys.warnoptions` a una lista vacía. Esta función puede llamarse antes de `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Clear `sys.warnoptions` and `warnings.filters` instead.

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

Part of the Stable ABI. 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, ...)

Part of the Stable ABI. Como *PySys_WriteStdout()*, pero escribe a `sys.stderr` o *stderr* en su lugar.

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

Part of the Stable ABI. Función similar a `PySys_WriteStdout()` pero formatea el mensaje usando *PyUnicode_FromFormatV()* y no trunca el mensaje a una longitud arbitraria.

Added in version 3.2.

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

Part of the Stable ABI. Como *PySys_FormatStdout()*, pero escribe a `sys.stderr` o *stderr* en su lugar.

Added in version 3.2.

PyObject ***PySys_GetXOptions**()

Return value: Borrowed reference. Part of the Stable ABI *since version 3.7.* 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.

Added in version 3.2.

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

Part of the Stable ABI *since version 3.13.* 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.

The *event* string argument must not be *NULL*.

If any hooks have been added, *format* and other arguments will be used to construct a tuple to pass. Apart from `N`, the same format characters as used in *Py_BuildValue()* are available. If the built value is not a tuple, it will be added into a single-element tuple.

The `N` format option must not be used. It consumes a reference, but since there is no way to know whether arguments to this function will be consumed, using it may cause reference leaks.

Tenga en cuenta que los caracteres de formato `#` deben tratarse como *Py_ssize_t*, independientemente de si se definió `PY_SSIZE_T_CLEAN`.

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

See also *PySys_AuditTuple()*.

Added in version 3.8.

Distinto en la versión 3.8.2: Requiere *Py_ssize_t* para los caracteres de formato `#`. Anteriormente, se lanzaba una advertencia de deprecación inevitable.

int **PySys_AuditTuple**(const char *event, *PyObject* *args)

Part of the Stable ABI *since version 3.13.* Similar to *PySys_Audit()*, but pass arguments as a Python object. *args* must be a `tuple`. To pass no arguments, *args* can be *NULL*.

Added in version 3.13.

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

Agrega el *hook* invocable a la lista de hooks de auditoría activos. Retorna cero para el éxito y no cero en caso de error. Si el tiempo de ejecución se ha inicializado, también configura un error en caso de fallo. Los hooks agregados a través de esta API se llaman para todos los intérpretes creados por el tiempo de ejecución.

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 subclasificado de `Excepción` (otros errores no se silenciarán).

The hook function is always called with an *attached thread state* by the Python interpreter that raised the event.

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 auditoria. Los detalles se encuentran en la documentación de cada función.

If the interpreter is initialized, this function raises an auditing event `sys.addaudithook` with no arguments. If any existing hooks raise an exception derived from `Exception`, the new hook will not be added and the exception is cleared. As a result, callers cannot assume that their hook has been added unless they control all existing hooks.

typedef int (***Py_AuditHookFunction**)(const char *event, *PyObject* *args, void *userData)

The type of the hook function. *event* is the C string event argument passed to *PySys_Audit()* or *PySys_AuditTuple()*. *args* is guaranteed to be a *PyTupleObject*. *userData* is the argument passed to PySys_AddAuditHook().

Added in version 3.8.

7.3 Control de procesos

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

Part of the Stable ABI. Print a fatal error message and kill the process. No cleanup is performed. This function should only be invoked when a condition is detected that would make it dangerous to continue using the Python interpreter; e.g., when the object administration appears to be corrupted. On Unix, the standard C library function `abort()` is called which will attempt to produce a `core` file.

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)

Part of the Stable ABI. 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)())

Part of the Stable ABI. 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.

Ver también

PyUnstable_AtExit() for passing a `void *data` argument.

7.4 Importando módulos

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

Return value: New reference. Part of the Stable ABI. This is a wrapper around `PyImport_Import()` which takes a `const char*` as an argument instead of a `PyObject*`.

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

Return value: New reference. Part of the Stable ABI. 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.

Deprecated since version 3.13, will be removed in version 3.15: Use `PyImport_ImportModule()` instead.

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. Part of the Stable ABI *since version 3.7.* 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.

Added in version 3.3.

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

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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_AddModuleRef**(const char *name)

Return value: New reference. Part of the Stable ABI *since version 3.13.* Return the module object corresponding to a module name.

The *name* argument may be of the form `package.module`. First check the modules dictionary if there's one there, and if not, create a new one and insert it in the modules dictionary.

Return a *strong reference* to the module on success. Return `NULL` with an exception set on failure.

The module name *name* is decoded from UTF-8.

This function does not load or import the module; if the module wasn't already loaded, you will get an empty module object. Use *PyImport_ImportModule()* or one of its variants to import a module. Package structures implied by a dotted name for *name* are not created if not already present.

Added in version 3.13.

PyObject ***PyImport_AddModuleObject**(*PyObject* *name)

Return value: Borrowed reference. Part of the Stable ABI *since version 3.7.* Similar to *PyImport_AddModuleRef()*, but return a *borrowed reference* and *name* is a Python `str` object.

Added in version 3.3.

PyObject ***PyImport_AddModule**(const char *name)

Return value: Borrowed reference. Part of the Stable ABI. Similar to *PyImport_AddModuleRef()*, but return a *borrowed reference*.

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

Return value: New reference. Part of the Stable ABI. Given a module name (possibly of the form `package.module`) and a code object read from a Python bytecode file or obtained from the built-in function `compile()`, load the module. Return a new reference to the module object, or `NULL` with an exception set if an error occurred. *name* is removed from `sys.modules` in error cases, even if *name* was already in `sys.modules` on entry to *PyImport_ExecCodeModule()*. Leaving incompletely initialized modules in `sys.modules` is dangerous, as imports of such modules have no way to know that the module object is an unknown (and probably damaged with respect to the module author's intents) state.

The module's `__spec__` and `__loader__` will be set, if not set already, with the appropriate values. The spec's loader will be set to the module's `__loader__` (if set) and to an instance of `SourceFileLoader` otherwise.

The module's `__file__` attribute will be set to the code object's `co_filename`. If applicable, `__cached__` will also be set.

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()*.

Distinto en la versión 3.12: The setting of `__cached__` and `__loader__` is deprecated. See `ModuleSpec` for alternatives.

PyObject ***PyImport_ExecCodeModuleEx**(const char *name, *PyObject* *co, const char *pathname)

Return value: New reference. Part of the Stable ABI. Like *PyImport_ExecCodeModule()*, but the `__file__` attribute of the module object is set to *pathname* if it is non-`NULL`.

Ver también *PyImport_ExecCodeModuleWithPathnames()*.

PyObject ***PyImport_ExecCodeModuleObject**(*PyObject* *name, *PyObject* *co, *PyObject* *pathname, *PyObject* *cpathname)

Return value: New reference. Part of the Stable ABI *since version 3.7.* Like *PyImport_ExecCodeModuleEx()*, but the `__cached__` attribute of the module object is set to *cpathname* if it is non-`NULL`. Of the three functions, this is the preferred one to use.

Added in version 3.3.

Distinto en la versión 3.12: Setting `__cached__` is deprecated. See `ModuleSpec` for alternatives.

PyObject ***PyImport_ExecCodeModuleWithPathnames**(const char *name, *PyObject* *co, const char *pathname, const char *cpathname)

Return value: New reference. Part of the Stable ABI. 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`.

Added in version 3.2.

Distinto en la versión 3.3: Uses `imp.source_from_cache()` in calculating the source path if only the bytecode path is provided.

Distinto en la versión 3.12: No longer uses the removed `imp` module.

long **PyImport_GetMagicNumber**()

Part of the Stable ABI. 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**()

Part of the Stable ABI. 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.

Added in version 3.2.

PyObject ***PyImport_GetModuleDict**()

Return value: Borrowed reference. Part of the Stable ABI. 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. Part of the Stable ABI *since version 3.8.* 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.

Added in version 3.7.

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

Return value: New reference. Part of the Stable ABI. Return a finder object for a `sys.path`/`pkg.__path__` item *path*, possibly by fetching it from the `sys.path_importer_cache` dict. If it wasn't yet cached, traverse `sys.path_hooks` until a hook is found that can handle the path item. Return `None` if no hook could; this tells our caller that the *path based finder* could not find a finder for this path item. Cache the result in `sys.path_importer_cache`. Return a new reference to the finder object.

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

Part of the Stable ABI *since version 3.7.* 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).

Added in version 3.3.

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

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

Part of the Stable ABI. 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;
    bool is_package;
};
```

Distinto en la versión 3.11: El nuevo campo `is_package` indica si el módulo es un paquete o no. Esto sustituye a la configuración del campo `size` con un valor negativo.

const struct *_frozen* ***PyImport_FrozenModules**

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

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

Part of the Stable ABI. 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**

Structure describing a single entry in the list of built-in modules. Programs which embed Python may use an array of these structures in conjunction with *PyImport_ExtendInittab()* to provide additional built-in modules. The structure consists of two members:

const char ***name**

The module name, as an ASCII encoded string.

PyObject *(***initfunc**)(void)

Initialization function for a module built into the interpreter.

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()*.

Si Python es inicializado múltiples veces, se debe llamar *PyImport_AppendInittab()* o *PyImport_ExtendInittab()* antes de cada inicialización de Python.

PyObject ***PyImport_ImportModuleAttr**(*PyObject* *mod_name, *PyObject* *attr_name)

Return value: New reference. Import the module *mod_name* and get its attribute *attr_name*.

Names must be Python `str` objects.

Helper function combining *PyImport_Import()* and *PyObject_GetAttr()*. For example, it can raise `ImportError` if the module is not found, and `AttributeError` if the attribute doesn't exist.

Added in version 3.14.

PyObject ***PyImport_ImportModuleAttrString**(const char *mod_name, const char *attr_name)

Return value: New reference. Similar to *PyImport_ImportModuleAttr()*, but names are UTF-8 encoded strings instead of Python `str` objects.

Added in version 3.14.

7.5 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.

The module supports several versions of the data format; see the `Python module documentation` for details.

Py_MARSHAL_VERSION

The current format version. See `marshal.version`.

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

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

Esta función puede fallar, en cuyo caso establece el indicador de error. Utiliza *PyErr_Occurred()* para comprobarlo.

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.

Esta función puede fallar, en cuyo caso establece el indicador de error. Utiliza *PyErr_Occurred()* para comprobarlo.

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 entero `long` de C desde el flujo de datos `FILE*` abierto para lectura. Solo se puede leer un valor de 32 bits con esta función, independientemente del tamaño nativo del tipo `long`.

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

int **PyMarshal_ReadShortFromFile**(FILE *file)

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

7.6 Analizando argumentos y construyendo valores

These functions are useful when creating your own extension functions and methods. Additional information and examples are available in 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.

7.6.1 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

> **Nota**
>
> On Python 3.12 and older, the macro `PY_SSIZE_T_CLEAN` must be defined before including `Python.h` to use all `#` variants of formats (`s#`, `y#`, etc.) explained below. This is not necessary on Python 3.13 and later.

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.

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

There are three ways strings and buffers can be converted to C:

- Formats such as `y*` and `s*` fill a *Py_buffer* structure. This locks the underlying buffer so that the caller can subsequently use the buffer even inside a *Py_BEGIN_ALLOW_THREADS* block without the risk of mutable data being resized or destroyed. As a result, **you have to call** *PyBuffer_Release()* after you have finished processing the data (or in any early abort case).
- The `es`, `es#`, `et` and `et#` formats allocate the result buffer. **You have to call** *PyMem_Free()* after you have finished processing the data (or in any early abort case).
- Other formats take a `str` or a read-only *bytes-like object*, such as `bytes`, and provide a `const char *` pointer to its buffer. In this case the buffer is «borrowed»: it is managed by the corresponding Python object, and shares the lifetime of this object. You won't have to release any memory yourself.

  To ensure that the underlying buffer may be safely borrowed, the object's *PyBufferProcs.bf_releasebuffer* field must be `NULL`. This disallows common mutable objects such as `bytearray`, but also some read-only objects such as `memoryview` of `bytes`.

  Besides this `bf_releasebuffer` requirement, there is no check to verify whether the input object is immutable (e.g. whether it would honor a request for a writable buffer, or whether another thread can mutate the data).

`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 incrustado 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 *, `Py_ssize_t`]
Like `s*`, except that it provides a *borrowed buffer*. The result is stored into two C variables, the first one a pointer to a C string, the second one its length. The string may contain embedded null bytes. Unicode objects are converted to C strings using `'utf-8'` encoding.

`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 *, `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 *]
This format converts a bytes-like object to a C pointer to a *borrowed* character string; it does not accept Unicode objects. The bytes buffer must not contain embedded null bytes; if it does, a `ValueError` exception is raised.

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 *, `Py_ssize_t`]
Esta variante en `s#` no acepta objetos Unicode, solo objetos similares a bytes.

`S` (`bytes`) [PyBytesObject *]
Requires that the Python object is a `bytes` object, without attempting any conversion. Raises `TypeError` if the object is not a bytes object. The C variable may also be declared as `PyObject*`.

`Y` (`bytearray`) [PyByteArrayObject *]
Requires that the Python object is a `bytearray` object, without attempting any conversion. Raises `TypeError` if the object is not a `bytearray` object. The C variable may also be declared as `PyObject*`.

`U` (`str`) [PyObject *]
Requires that the Python object is a Unicode object, without attempting any conversion. Raises `TypeError` if the object is not a Unicode object. The C variable may also be declared as `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.

This format requires two arguments. The first is only used as input, and must be a `const char*` which points to the name of an encoding as a NUL-terminated string, or `NULL`, in which case `'utf-8'` encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a `char**`; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument.

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, *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.

It requires three arguments. The first is only used as input, and must be a `const char*` which points to the name of an encoding as a NUL-terminated string, or `NULL`, in which case `'utf-8'` encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a `char**`; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument. The third argument must be a pointer to an integer; the referenced integer will be set to the number of bytes in the output buffer.

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, *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.

Distinto en la versión 3.12: `u`, `u#`, `Z`, and `Z#` are removed because they used a legacy `Py_UNICODE*` representation.

Números

These formats allow representing Python numbers or single characters as C numbers. Formats that require `int`, `float` or `complex` can also use the corresponding special methods `__index__()`, `__float__()` or `__complex__()` to convert the Python object to the required type.

For signed integer formats, `OverflowError` is raised if the value is out of range for the C type. For unsigned integer formats, no range checking is done — the most significant bits are silently truncated when the receiving field is too small to receive the value.

b (int) [unsigned char]
Convert a nonnegative Python integer to an unsigned tiny integer, stored in a C `unsigned char`.

B (int) [unsigned char]
Convert a Python integer to a tiny integer without overflow checking, stored in a C `unsigned char`.

h (int) [short int]
Convert a Python integer to a C `short int`.

H (int) [unsigned short int]
Convert a Python integer to a C `unsigned short int`, without overflow checking.

i (int) [int]
Convert a Python integer to a plain C `int`.

I (int) [unsigned int]
Convert a Python integer to a C `unsigned int`, without overflow checking.

l (int) [long int]
Convert a Python integer to a C `long int`.

k (int) [unsigned long]
Convert a Python integer to a C `unsigned long` without overflow checking.

Distinto en la versión 3.14: Use `__index__()` if available.

L (int) [long long]
Convert a Python integer to a C `long long`.

K (int) [unsigned long long]
Convert a Python integer to a C `unsigned long long` without overflow checking.

Distinto en la versión 3.14: Use `__index__()` if available.

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]
Convert a Python byte, represented as a `bytes` or `bytearray` object of length 1, to a C `char`.

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

C (str de largo 1) [int]
Convert a Python character, represented as a `str` object of length 1, to a C `int`.

f (float) [float]
Convert a Python floating-point number to a C `float`.

d (float) [double]
Convert a Python floating-point number to a C `double`.

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

Otros objetos

O (object) [PyObject *]
Store a Python object (without any conversion) in a C object pointer. The C program thus receives the actual object that was passed. A new *strong reference* to the object is not created (i.e. its reference count is not increased). The pointer stored is not `NULL`.

O! (object) [*typeobject*, PyObject *]
Store a Python object in a C object pointer. This is similar to `O`, but takes two C arguments: the first is the address of a Python type object, the second is the address of the C variable (of type *PyObject**) into which the object pointer is stored. If the Python object does not have the required type, `TypeError` is raised.

O& (object) [*converter*, *address*]
Convert a Python object to a C variable through a *converter* function. This takes two arguments: the first is a function, the second is the address of a C variable (of arbitrary type), converted to `void*`. The *converter* function in turn is called as follows:

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

where *object* is the Python object to be converted and *address* is the `void*` argument that was passed to the `PyArg_Parse*` function. The returned *status* should be `1` for a successful conversion and `0` if the conversion has failed. When the conversion fails, the *converter* function should raise an exception and leave the content

of *address* unmodified. If the *converter* returns `Py_CLEANUP_SUPPORTED`, it may get called a second time if the argument parsing eventually fails, giving the converter a chance to release any memory that it had already allocated. In this second call, the *object* parameter will be `NULL`; *address* will have the same value as in the original call.

Examples of converters: *PyUnicode_FSConverter()* and *PyUnicode_FSDecoder()*.

Distinto en la versión 3.1: `Py_CLEANUP_SUPPORTED` was added.

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.

Added in version 3.3.

(items) (sequence) [*matching-items*]
The object must be a Python sequence (except `str`, `bytes` or `bytearray`) whose length is the number of format units in *items*. The C arguments must correspond to the individual format units in *items*. Format units for sequences may be nested.

If *items* contains format units which store a *borrowed buffer* (`s`, `s#`, `z`, `z#`, `y`, or `y#`) or a *borrowed reference* (`S`, `Y`, `U`, `O`, or `O!`), the object must be a Python tuple. The *converter* for the `O&` format unit in *items* must not store a borrowed buffer or a borrowed reference.

Distinto en la versión 3.14: `str` and `bytearray` objects no longer accepted as a sequence.

Obsoleto desde la versión 3.14: Non-tuple sequences are deprecated if *items* contains format units which store a borrowed buffer or a borrowed reference.

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.

Added in version 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.

Note that any Python object references which are provided to the caller are *borrowed* references; do not release them (i.e. do not decrement their reference count)!

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.

For the conversion to succeed, the *arg* object must match the format and the format must be exhausted. On success, the `PyArg_Parse*` functions return true, otherwise they return false and raise an appropriate exception. When the `PyArg_Parse*` functions fail due to conversion failure in one of the format units, the variables at the addresses corresponding to that and the following format units are left untouched.

Funciones API

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

Part of the Stable ABI. 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 lanza la excepción apropiada.

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

Part of the Stable ABI. 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 *const *keywords, ...)

Part of the Stable ABI. Parse the parameters of a function that takes both positional and keyword parameters into local variables. The *keywords* argument is a `NULL`-terminated array of keyword parameter names specified as null-terminated ASCII or UTF-8 encoded C strings. Empty names denote *positional-only parameters*. Returns true on success; on failure, it returns false and raises the appropriate exception.

> **Nota**
>
> The *keywords* parameter declaration is `char *const*` in C and `const char *const*` in C++. This can be overridden with the `PY_CXX_CONST` macro.

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

Distinto en la versión 3.13: The *keywords* parameter has now type `char *const*` in C and `const char *const*` in C++, instead of `char**`. Added support for non-ASCII keyword parameter names.

int **PyArg_VaParseTupleAndKeywords**(*PyObject* *args, *PyObject* *kw, const char *format, char *const *keywords, va_list vargs)

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

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

Part of the Stable ABI. 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.

Added in version 3.2.

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

Part of the Stable ABI. Parse the parameter of a function that takes a single positional parameter into a local variable. Returns true on success; on failure, it returns false and raises the appropriate exception.

Example:

```
// Function using METH_O calling convention
static PyObject*
my_function(PyObject *module, PyObject *arg)
{
    int value;
    if (!PyArg_Parse(arg, "i:my_function", &value)) {
        return NULL;
    }
    // ... use value ...
}
```

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

Part of the Stable ABI. A simpler form of parameter retrieval which does not use a format string to specify the types of the arguments. Functions which use this method to retrieve their parameters should be declared as `METH_VARARGS` in function or method tables. The tuple containing the actual parameters should be passed

as *args*; it must actually be a tuple. The length of the tuple must be at least *min* and no more than *max*; *min* and *max* may be equal. Additional arguments must be passed to the function, each of which should be a pointer to a *PyObject** variable; these will be filled in with the values from *args*; they will contain *borrowed references*. The variables which correspond to optional parameters not given by *args* will not be filled in; these should be initialized by the caller. This function returns true on success and false if *args* is not a tuple or contains the wrong number of elements; an exception will be set if there was a failure.

This is an example of the use of this function, taken from the sources for the `_weakref` helper module for weak references:

```
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)
```

PY_CXX_CONST

The value to be inserted, if any, before `char *const*` in the *keywords* parameter declaration of *PyArg_ParseTupleAndKeywords()* and *PyArg_VaParseTupleAndKeywords()*. Default empty for C and `const` for C++ (`const char *const*`). To override, define it to the desired value before including `Python.h`.

Added in version 3.13.

7.6.2 Construyendo valores

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

Return value: New reference. Part of the Stable ABI. Create a new value based on a format string similar to those accepted by the `PyArg_Parse*` family of functions and a sequence of values. Returns the value or `NULL` in the case of an error; an exception will be raised if `NULL` is returned.

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 *, *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 *, *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 *, *Py_ssize_t*]
Igual que `s#`.

u (str) [const wchar_t *]
Convert a null-terminated `wchar_t` buffer of Unicode (UTF-16 or UCS-4) data to a Python Unicode object. If the Unicode buffer pointer is `NULL`, `None` is returned.

u# (str) [const wchar_t *, *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`.

z# (str o None) [const char *, *Py_ssize_t*]
Igual que `s#`.

i (int) [int]
Convert a plain C `int` to a Python integer object.

b (int) [char]
Convert a plain C `char` to a Python integer object.

h (int) [short int]
Convert a plain C `short int` to a Python integer object.

l (int) [long int]
Convert a C `long int` to a Python integer object.

B (int) [unsigned char]
Convert a C `unsigned char` to a Python integer object.

H (int) [unsigned short int]
Convert a C `unsigned short int` to a Python integer object.

I (int) [unsigned int]
Convert a C `unsigned int` to a Python integer object.

k (int) [unsigned long]
Convert a C `unsigned long` to a Python integer object.

L (int) [long long]
Convert a C `long long` to a Python integer object.

K (int) [unsigned long long]
Convert a C `unsigned long long` to a Python integer object.

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

p (bool) [int]
Convert a C `int` to a Python `bool` object.

Be aware that this format requires an `int` argument. Unlike most other contexts in C, variadic arguments are not coerced to a suitable type automatically. You can convert another type (for example, a pointer or a float) to a suitable `int` value using `(x) ? 1 : 0` or `!!x`.

Added in version 3.14.

c (bytes de largo 1) [char]
Convert a C `int` representing a byte to a Python `bytes` object of length 1.

C (str de largo 1) [int]
Convert a C `int` representing a character to Python `str` object of length 1.

d (float) [double]
Convert a C `double` to a Python floating-point number.

f (float) [float]
Convert a C `float` to a Python floating-point number.

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

O (object) [PyObject *]
Pass a Python object untouched but create a new *strong reference* to it (i.e. its reference count is incremented by one). If the object passed in is a `NULL` pointer, it is assumed that this was caused because the call producing the argument found an error and set an exception. Therefore, *Py_BuildValue()* will return `NULL` but won't raise an exception. If no exception has been raised yet, `SystemError` is set.

S (object) [PyObject *]
Igual que `O`.

N (object) [PyObject *]
Same as `O`, except it doesn't create a new *strong reference*. Useful when the object is created by a call to an object constructor in the argument list.

O& (object) [*converter*, *anything*]
Convert *anything* to a Python object through a *converter* function. The function is called with *anything* (which should be compatible with `void*`) as its argument and should return a «new» Python object, or `NULL` if an error occurred.

(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 vargs)

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

7.7 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, ...)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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`, `format != NULL` y `size < INT_MAX`. Tenga en cuenta que esto significa que no hay equivalente a la expresión `n = snprintf(NULL, 0, ...)` de C99, la cual determinaría el tamaño del búfer necesario.

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.

unsigned long **PyOS_strtoul**(const char *str, char **ptr, int base)

Part of the Stable ABI. Convierte la parte inicial de la cadena de caracteres en `str` a un valor `unsigned long` según la `base` dada, que debe estar entre `2` y `36` inclusive, o ser el valor especial `0`.

Se ignoran los espacios en blanco y las mayúsculas y minúsculas. Si `base` es cero se busca un prefijo `0b`, `0o` o `0x` para indicar la base. Si no están, por defecto es `10`. La base debe ser 0 o entre 2 y 36 (ambos inclusive). Si `ptr` no es `NULL`, contendrá un puntero al final del escaneo.

Si el valor convertido queda fuera del rango del tipo de retorno correspondiente, se produce un error de rango (`errno` se establece en `ERANGE`) y se devuelve `ULONG_MAX`. Si no se puede realizar la conversión, se devuelve `0`.

Vea también el manual Unix de *strtoul(3)*.

Added in version 3.2.

long **PyOS_strtol**(const char *str, char **ptr, int base)

Part of the Stable ABI. Convierte la parte inicial de la cadena de caracteres en `str` a un valor `long` según la `base` dada, que debe estar entre `2` y `36` inclusive, o ser el valor especial `0`.

Igual que *PyOS_strtoul()*, pero devuelve un valor `long` en su lugar y `LONG_MAX` en desbordamientos.

Vea también el manual Unix de *strtol(3)*.

Added in version 3.2.

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

Part of the Stable ABI. Convierte una cadena de caracteres `s` a un valor `double`, generando una excepción de Python en caso de fallo. 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`.

If `s` represents a value that is too large to store in a float (for example, `"1e500"` is such a string on many platforms) then if `overflow_exception` is `NULL` return `Py_INFINITY` (with an appropriate sign) and don't set any exception. Otherwise, `overflow_exception` must point to a Python exception object; raise that exception and return `-1.0`. In both cases, set `*endptr` to point to the first character after the converted value.

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`.

Added in version 3.1.

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

Part of the Stable ABI. Convierte un `double` *val* a 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_DTSF_SIGN`, `Py_DTSF_ADD_DOT_0`, o `Py_DTSF_ALT`, unidos por *or* (*or-ed*) juntos:

- `Py_DTSF_SIGN` significa preceder siempre a la cadena de caracteres retornada con un carácter de signo, incluso si *val* no es negativo.
- `Py_DTSF_ADD_DOT_0` significa asegurarse de que la cadena de caracteres retornada no se verá como un número entero.
- `Py_DTSF_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()`*.

Added in version 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 mayúsculas y minúsculas.

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 mayúsculas y minúsculas.

7.8 PyHash API

See also the *`PyTypeObject.tp_hash`* member and numeric-hash.

type **Py_hash_t**

Hash value type: signed integer.

Added in version 3.2.

type **Py_uhash_t**

Hash value type: unsigned integer.

Added in version 3.2.

PyHASH_MODULUS

The Mersenne prime `P = 2**n -1`, used for numeric hash scheme.

Added in version 3.13.

PyHASH_BITS

The exponent `n` of `P` in *PyHASH_MODULUS*.

Added in version 3.13.

PyHASH_MULTIPLIER

Prime multiplier used in string and various other hashes.

Added in version 3.13.

PyHASH_INF

The hash value returned for a positive infinity.

Added in version 3.13.

PyHASH_IMAG

The multiplier used for the imaginary part of a complex number.

Added in version 3.13.

type **PyHash_FuncDef**

Hash function definition used by *PyHash_GetFuncDef()*.

Py_hash_t (*const **hash**)(const void*, *Py_ssize_t*)

Hash function.

const char ***name**

Hash function name (UTF-8 encoded string).

const int **hash_bits**

Internal size of the hash value in bits.

const int **seed_bits**

Size of seed input in bits.

Added in version 3.4.

PyHash_FuncDef ***PyHash_GetFuncDef**(void)

Get the hash function definition.

Ver también

PEP 456 «Secure and interchangeable hash algorithm».

Added in version 3.4.

Py_hash_t **Py_HashPointer**(const void *ptr)

Hash a pointer value: process the pointer value as an integer (cast it to `uintptr_t` internally). The pointer is not dereferenced.

The function cannot fail: it cannot return `-1`.

Added in version 3.13.

Py_hash_t **Py_HashBuffer**(const void *ptr, *Py_ssize_t* len)

Compute and return the hash value of a buffer of *len* bytes starting at address *ptr*. The hash is guaranteed to match that of `bytes`, `memoryview`, and other built-in objects that implement the *buffer protocol*.

Use this function to implement hashing for immutable objects whose *tp_richcompare* function compares to another object's buffer.

len must be greater than or equal to `0`.

This function always succeeds.

Added in version 3.14.

Py_hash_t **PyObject_GenericHash**(*PyObject* *obj)

Generic hashing function that is meant to be put into a type object's `tp_hash` slot. Its result only depends on the object's identity.

Detalles de implementación de CPython: In CPython, it is equivalent to *Py_HashPointer()*.

Added in version 3.13.

7.9 Reflexión

PyObject ***PyEval_GetBuiltins**(void)

Return value: Borrowed reference. Part of the Stable ABI. Obsoleto desde la versión 3.13: Utilice *PyEval_GetFrameBuiltins()* en su lugar.

Retorna un diccionario de los builtins en el marco de ejecución actual, o el intérprete del estado del hilo si no hay ningún marco en ejecución actualmente.

PyObject ***PyEval_GetLocals**(void)

Return value: Borrowed reference. Part of the Stable ABI. Obsoleto desde la versión 3.13: Utilice *PyEval_GetFrameLocals()* para obtener el mismo comportamiento que llamando a `locals()` en código Python, o bien llama a *PyFrame_GetLocals()* sobre el resultado de *PyEval_GetFrame()* para acceder al atributo `f_locals` del frame actualmente en ejecución.

Retorna un diccionario de las variables locales en el marco de ejecución actual, o `NULL` si actualmente no se está ejecutando ningún marco.

Consulte `locals()` para obtener más información sobre el mapeo retornado en diferentes ámbitos.

Como esta función retorna una *referencia prestada*, el diccionario retornado para *ámbitos optimizados* se almacena en caché en el objeto marco y permanecerá vivo mientras lo haga el marco del objeto. A diferencia de *PyEval_GetFrameLocals()* y `locals()`, las llamadas posteriores a esta función en el mismo marco actualizarán el contenido del diccionario en caché para reflejar los cambios en el estado de las variables locales en lugar de retornar una nueva instantánea.

Distinto en la versión 3.13: Como parte de **PEP 667**, *PyFrame_GetLocals()*, `locals()`, y `FrameType.f_locals` ya no utilizan el diccionario de caché compartido. Consulte la entrada What's New para más detalles.

PyObject ***PyEval_GetGlobals**(void)

Return value: Borrowed reference. Part of the Stable ABI. Obsoleto desde la versión 3.13: Utilice *PyEval_GetFrameGlobals()* en su lugar.

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. Part of the Stable ABI. Return the *attached thread state*'s frame, which is `NULL` if no frame is currently executing.

Vea también *PyThreadState_GetFrame()*.

PyObject ***PyEval_GetFrameBuiltins**(void)

Return value: New reference. Part of the Stable ABI *since version 3.13.* Retorna un diccionario de los builtins en el marco de ejecución actual, o el intérprete del estado del hilo si no hay ningún marco en ejecución actualmente.

Added in version 3.13.

PyObject ***PyEval_GetFrameLocals**(void)

Return value: New reference. Part of the Stable ABI *since version 3.13.* Retorna un diccionario de las variables locales en el marco de ejecución actual, o `NULL` si no se está ejecutando ningún marco. Equivale a llamar a `locals()` en código Python.

Para acceder a `f_locals` en el marco actual sin hacer una captura independiente en *ámbitos optimizados*, llame a *PyFrame_GetLocals()* sobre el resultado de *PyEval_GetFrame()*.

Added in version 3.13.

PyObject ***PyEval_GetFrameGlobals**(void)

Return value: New reference. Part of the Stable ABI *since version 3.13.* Retorna un diccionario de las variables locales en el marco de ejecución actual, o `NULL` si no se está ejecutando ningún marco. Equivale a llamar a `globals()` en código Python.

Added in version 3.13.

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

Part of the Stable ABI. Retorna el nombre de *func* si es una función, clase u objeto de instancia; de lo contrario, el nombre del tipo *func*s.

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

Part of the Stable ABI. 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*.

7.10 Registro de códec y funciones de soporte

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

Part of the Stable ABI. Registra una nueva función de búsqueda de códec.

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

int **PyCodec_Unregister**(*PyObject* *search_function)

Part of the Stable ABI *since version 3.10.* Anula el registro de una función de búsqueda de códecs y borra el caché del registro. Si la función de búsqueda no está registrada, no hace nada. Retorna 0 en caso de éxito. Lanza una excepción y devuelva -1 en caso de error.

Added in version 3.10.

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

Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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.

7.10.1 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. Part of the Stable ABI. Obtiene una función de codificador para el *encoding* dado.

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

Return value: New reference. Part of the Stable ABI. Obtiene una función de decodificador para el *encoding* dado.

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

Return value: New reference. Part of the Stable ABI. Obtiene un objeto `IncrementalEncoder` para el *encoding* dada.

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

Return value: New reference. Part of the Stable ABI. Obtiene un objeto `IncrementalDecoder` para el *encoding* dado.

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

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. Obtiene una función de fábrica `StreamWriter` por el *encoding* dado.

7.10.2 API de registro para controladores de errores de codificación Unicode

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

Part of the Stable ABI. 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 lanzar 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. Part of the Stable ABI. 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. Part of the Stable ABI. Lanza *exc* como una excepción.

PyObject ***PyCodec_IgnoreErrors**(*PyObject* *exc)

Return value: New reference. Part of the Stable ABI. Ignora el error Unicode, omitiendo la entrada defectuosa.

PyObject ***PyCodec_ReplaceErrors**(*PyObject* *exc)

Return value: New reference. Part of the Stable ABI. Reemplaza el error de codificación Unicode con `?` o `U+FFFD`.

PyObject ***PyCodec_XMLCharRefReplaceErrors**(*PyObject* *exc)

Return value: New reference. Part of the Stable ABI. Reemplaza el error de codificación Unicode con referencias de caracteres XML.

PyObject ***PyCodec_BackslashReplaceErrors**(*PyObject* *exc)

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

PyObject ***PyCodec_NameReplaceErrors**(*PyObject* *exc)

Return value: New reference. Part of the Stable ABI *since version 3.7.* Reemplaza el error de codificación Unicode con escapes `\N{...}`.

Added in version 3.5.

7.11 PyTime C API

Added in version 3.13.

The clock C API provides access to system clocks. It is similar to the Python `time` module.

For C API related to the `datetime` module, see *Objetos DateTime*.

7.11.1 Types

type **PyTime_t**

A timestamp or duration in nanoseconds, represented as a signed 64-bit integer.

The reference point for timestamps depends on the clock used. For example, *PyTime_Time()* returns timestamps relative to the UNIX epoch.

The supported range is around [-292.3 years; +292.3 years]. Using the Unix epoch (January 1st, 1970) as reference, the supported date range is around [1677-09-21; 2262-04-11]. The exact limits are exposed as constants:

PyTime_t **PyTime_MIN**

Minimum value of *PyTime_t*.

PyTime_t **PyTime_MAX**

Maximum value of *PyTime_t*.

7.11.2 Clock Functions

The following functions take a pointer to a *PyTime_t* that they set to the value of a particular clock. Details of each clock are given in the documentation of the corresponding Python function.

The functions return `0` on success, or `-1` (with an exception set) on failure.

On integer overflow, they set the *PyExc_OverflowError* exception and set `*result` to the value clamped to the `[PyTime_MIN; PyTime_MAX]` range. (On current systems, integer overflows are likely caused by misconfigured system time.)

As any other C API (unless otherwise specified), the functions must be called with an *attached thread state*.

int **PyTime_Monotonic**(*PyTime_t* *result)

Read the monotonic clock. See `time.monotonic()` for important details on this clock.

int **PyTime_PerfCounter**(*PyTime_t* *result)

Read the performance counter. See `time.perf_counter()` for important details on this clock.

int **PyTime_Time**(*PyTime_t* *result)

Read the “wall clock” time. See `time.time()` for details important on this clock.

7.11.3 Raw Clock Functions

Similar to clock functions, but don’t set an exception on error and don’t require the caller to have an *attached thread state*.

On success, the functions return `0`.

On failure, they set `*result` to `0` and return `-1`, *without* setting an exception. To get the cause of the error, *attach* a *thread state*, and call the regular (non-`Raw`) function. Note that the regular function may succeed after the `Raw` one failed.

int **PyTime_MonotonicRaw**(*PyTime_t* *result)

Similar to `PyTime_Monotonic()`, but don't set an exception on error and don't require an *attached thread state*.

int **PyTime_PerfCounterRaw**(*PyTime_t* *result)

Similar to `PyTime_PerfCounter()`, but don't set an exception on error and don't require an *attached thread state*.

int **PyTime_TimeRaw**(*PyTime_t* *result)

Similar to `PyTime_Time()`, but don't set an exception on error and don't require an *attached thread state*.

7.11.4 Conversion functions

double **PyTime_AsSecondsDouble**(*PyTime_t* t)

Convert a timestamp to a number of seconds as a C `double`.

The function cannot fail, but note that `double` has limited accuracy for large values.

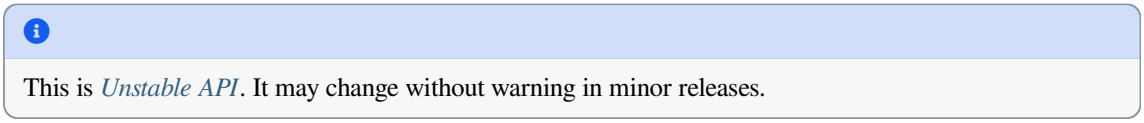
7.12 Soporte para Mapeo Perf

En las plataformas soportadas (en el momento de escribir esto, sólo Linux), el tiempo de ejecución puede aprovechar *perf map files* para hacer que las funciones de Python sean visibles para una herramienta externa de perfiles (como perf). Un proceso en ejecución puede crear un fichero en el directorio `/tmp`, que contiene entradas que pueden asignar una sección de código ejecutable a un nombre. Esta interfaz se describe en la documentación de la herramienta Perf de Linux.

En Python, estas API auxiliares pueden ser utilizadas por bibliotecas y funciones que dependen de la generación de código de máquina sobre la marcha.

Note that holding an *attached thread state* is not required for these APIs.

int **PyUnstable_PerfMapState_Init**(void)

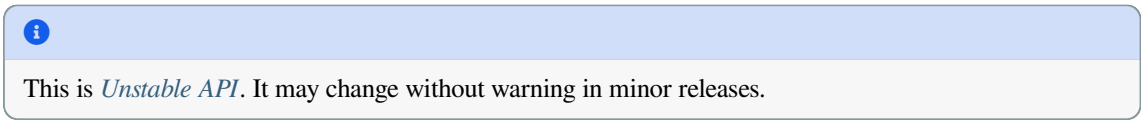


This is *Unstable API*. It may change without warning in minor releases.

Abre el archivo `/tmp/perf-$pid.map`, a menos que ya esté abierto, y crea un bloqueo para garantizar escrituras seguras para hilos en el archivo (siempre que las escrituras se realicen mediante `PyUnstable_WritePerfMapEntry()`). Normalmente, no es necesario llamar a esto explícitamente; solo use `PyUnstable_WritePerfMapEntry()` e inicializará el estado en la primera llamada.

Retorna `0` en caso de éxito, `-1` en caso de fallo al crear/abrir el fichero perf map, o `-2` en caso de fallo al crear un bloqueo. Comprueba `errno` para más información sobre la causa de un fallo.

int **PyUnstable_WritePerfMapEntry**(const void *code_addr, unsigned int code_size, const char *entry_name)



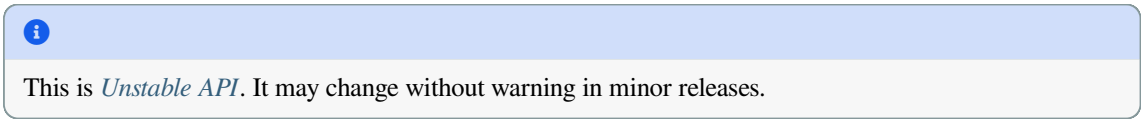
This is *Unstable API*. It may change without warning in minor releases.

Escribe una única entrada en el fichero `/tmp/perf-$pid.map`. Esta función es segura para hilos. Aquí hay un ejemplo de entrada:

```
# address      size  name
7f3529fcf759 b     py::bar:/run/t.py
```

Llamará a *PyUnstable_PerfMapState_Init()* antes de escribir la entrada, si el fichero perf map no está ya abierto. Retorna `0` en caso de éxito, o los mismos códigos de error que *PyUnstable_PerfMapState_Init()* en caso de fallo.

void **PyUnstable_PerfMapState_Fini**(void)



This is *Unstable API*. It may change without warning in minor releases.

Cierra el fichero perf map abierto por *PyUnstable_PerfMapState_Init()*. Esto es llamado por el propio tiempo de ejecución durante el cierre del intérprete. En general, no debería haber una razón para llamar explícitamente a esto, excepto para manejar escenarios específicos como la bifurcación.

CAPÍTULO 8

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, lanzarán 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.

8.1 Protocolo de objeto

PyObject ***Py_GetConstant**(unsigned int constant_id)

Part of the Stable ABI *since version 3.13.* Get a *strong reference* to a constant.

Set an exception and return `NULL` if *constant_id* is invalid.

constant_id must be one of these constant identifiers:

Constant Identifier	Value	Returned object
Py_CONSTANT_NONE	0	None
Py_CONSTANT_FALSE	1	False
Py_CONSTANT_TRUE	2	True
Py_CONSTANT_ELLIPSIS	3	Ellipsis
Py_CONSTANT_NOT_IMPLEMENTI	4	NotImplemented
Py_CONSTANT_ZERO	5	0
Py_CONSTANT_ONE	6	1
Py_CONSTANT_EMPTY_STR	7	''
Py_CONSTANT_EMPTY_BYTES	8	b''
Py_CONSTANT_EMPTY_TUPLE	9	()

Numeric values are only given for projects which cannot use the constant identifiers.

Added in version 3.13.

Detalles de implementación de CPython: In CPython, all of these constants are *immortal*.

PyObject ***Py_GetConstantBorrowed**(unsigned int constant_id)

Part of the Stable ABI *since version 3.13.* Similar to `Py_GetConstant()`, but return a *borrowed reference*.

This function is primarily intended for backwards compatibility: using `Py_GetConstant()` is recommended for new code.

The reference is borrowed from the interpreter, and is valid until the interpreter finalization.

Added in version 3.13.

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

Properly handle returning `Py_NotImplemented` from within a C function (that is, create a new *strong reference* to `NotImplemented` and return it).

Py_PRINT_RAW

Flag to be used with multiple functions that print the object (like `PyObject_Print()` and `PyFile_WriteObject()`). If passed, these function would use the `str()` of the object instead of the `repr()`.

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

Print an object *o*, on file *fp*. Returns `-1` on error. The flags argument is used to enable certain printing options. The only option currently supported is *Py_PRINT_RAW*; if given, the `str()` of the object is written instead of the `repr()`.

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

Part of the Stable ABI *since version 3.13*. Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This is equivalent to the Python expression `hasattr(o, attr_name)`. On failure, return `-1`.

Added in version 3.13.

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

Part of the Stable ABI *since version 3.13*. This is the same as *PyObject_HasAttrWithError()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

Added in version 3.13.

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

Part of the Stable ABI. Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This function always succeeds.

Nota

Exceptions that occur when this calls `__getattr__()` and `__getattribute__()` methods aren't propagated, but instead given to `sys.unraisablehook()`. For proper error handling, use *PyObject_HasAttrWithError()*, *PyObject_GetOptionalAttr()* or *PyObject_GetAttr()* instead.

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

Part of the Stable ABI. This is the same as *PyObject_HasAttr()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

Nota

Exceptions that occur when this calls `__getattr__()` and `__getattribute__()` methods or while creating the temporary `str` object are silently ignored. For proper error handling, use *PyObject_HasAttrStringWithError()*, *PyObject_GetOptionalAttrString()* or *PyObject_GetAttrString()* instead.

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

Return value: New reference. Part of the Stable ABI. 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`.

If the missing attribute should not be treated as a failure, you can use *PyObject_GetOptionalAttr()* instead.

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

Return value: New reference. Part of the Stable ABI. This is the same as *PyObject_GetAttr()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

If the missing attribute should not be treated as a failure, you can use *PyObject_GetOptionalAttrString()* instead.

int **PyObject_GetOptionalAttr**(*PyObject* *obj, *PyObject* *attr_name, *PyObject* **result);

Part of the Stable ABI *since version 3.13*. Variant of *PyObject_GetAttr()* which doesn't raise `AttributeError` if the attribute is not found.

If the attribute is found, return `1` and set **result* to a new *strong reference* to the attribute. If the attribute is not found, return `0` and set **result* to `NULL`; the `AttributeError` is silenced. If an error other than `AttributeError` is raised, return `-1` and set **result* to `NULL`.

Added in version 3.13.

int **PyObject_GetOptionalAttrString**(*PyObject* *obj, const char *attr_name, *PyObject* **result);

Part of the Stable ABI *since version 3.13.* This is the same as *PyObject_GetOptionalAttr()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

Added in version 3.13.

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

Return value: New reference. Part of the Stable ABI. 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 descriptores de datos tienen preferencia sobre los atributos de instancia, mientras que los descriptores que no son de datos no lo hacen. De lo contrario, se lanza un `AttributeError`.

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

Part of the Stable ABI. Establece el valor del atributo llamado *attr_name*, para el objeto *o*, en el valor *v*. 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.attr_name = v`.

Si *v* es `NULL`, el atributo se elimina. Este comportamiento está deprecado en favor de usar *PyObject_DelAttr()*, pero por el momento no hay planes de quitarlo.

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

Part of the Stable ABI. This is the same as *PyObject_SetAttr()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

Si *v* es `NULL`, el atributo se elimina, sin embargo, esta característica está deprecada en favor de usar *PyObject_DelAttrString()*.

The number of different attribute names passed to this function should be kept small, usually by using a statically allocated string as *attr_name*. For attribute names that aren't known at compile time, prefer calling *PyUnicode_FromString()* and *PyObject_SetAttr()* directly. For more details, see *PyUnicode_InternFromString()*, which may be used internally to create a key object.

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

Part of the Stable ABI. 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 lanza un `AttributeError` y se retorna `-1`.

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

Part of the Stable ABI *since version 3.13.* 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)

Part of the Stable ABI *since version 3.13.* This is the same as *PyObject_DelAttr()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

The number of different attribute names passed to this function should be kept small, usually by using a statically allocated string as *attr_name*. For attribute names that aren't known at compile time, prefer calling *PyUnicode_FromString()* and *PyObject_DelAttr()* directly. For more details, see *PyUnicode_InternFromString()*, which may be used internally to create a key object for lookup.

PyObject ***PyObject_GenericGetDict**(*PyObject* *o, void *context)

Return value: New reference. Part of the Stable ABI *since version 3.10.* Una implementación genérica para obtener un descriptor `__dict__`. Crea el diccionario si es necesario.

Esta función también puede ser llamada para obtener el `__dict__` del objeto *o*. Se pasa *context* igual a `NULL` cuando se lo llama. Dado que esta función puede necesitar asignar memoria para el diccionario, puede ser más eficiente llamar a `PyObject_GetAttr()` para acceder a un atributo del objeto.

En caso de fallo, retorna `NULL` con una excepción establecida.

Added in version 3.3.

int **PyObject_GenericSetDict**(*PyObject* *o, *PyObject* *value, void *context)

Part of the Stable ABI *since version 3.7.* Una implementación genérica para el creador de un descriptor `__dict__`. Esta implementación no permite que se elimine el diccionario.

Added in version 3.3.

PyObject ****_PyObject_GetDictPtr**(*PyObject* *obj)

Retorna un puntero al `__dict__` del objeto *obj*. Si no hay `__dict__`, retorna `NULL` sin establecer una excepción.

Esta función puede necesitar asignar memoria para el diccionario, por lo que puede ser más eficiente llamar a `PyObject_GetAttr()` para acceder a un atributo del objeto.

PyObject ***PyObject_RichCompare**(*PyObject* *o1, *PyObject* *o2, int opid)

Return value: New reference. Part of the Stable ABI. Compare the values of *o1* and *o2* using the operation specified by *opid*, which must be one of `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, or `Py_GE`, corresponding to `<`, `<=`, `==`, `!=`, `>`, or `>=` respectively. This is the equivalent of the Python expression `o1 op o2`, where `op` is the operator corresponding to *opid*. Returns the value of the comparison on success, or `NULL` on failure.

int **PyObject_RichCompareBool**(*PyObject* *o1, *PyObject* *o2, int opid)

Part of the Stable ABI. Compare the values of *o1* and *o2* using the operation specified by *opid*, like `PyObject_RichCompare()`, but returns `-1` on error, `0` if the result is false, `1` otherwise.

Nota

If *o1* and *o2* are the same object, `PyObject_RichCompareBool()` will always return `1` for `Py_EQ` and `0` for `Py_NE`.

PyObject ***PyObject_Format**(*PyObject* *obj, *PyObject* *format_spec)

Part of the Stable ABI. Format *obj* using *format_spec*. This is equivalent to the Python expression `format(obj, format_spec)`.

format_spec may be `NULL`. In this case the call is equivalent to `format(obj)`. Returns the formatted string on success, `NULL` on failure.

PyObject ***PyObject_Repr**(*PyObject* *o)

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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)

Part of the Stable ABI. 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`.

If *cls* has a `__subclasscheck__()` method, it will be called to determine the subclass status as described in **PEP 3119**. Otherwise, *derived* is a subclass of *cls* if it is a direct or indirect subclass, i.e. contained in `cls.__mro__`.

Normally only class objects, i.e. instances of `type` or a derived class, are considered classes. However, objects can override this by having a `__bases__` attribute (which must be a tuple of base classes).

int **PyObject_IsInstance**(*PyObject* *inst, *PyObject* *cls)

Part of the Stable ABI. 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`.

If *cls* has a `__instancecheck__()` method, it will be called to determine the subclass status as described in **PEP 3119**. Otherwise, *inst* is an instance of *cls* if its class is a subclass of *cls*.

An instance *inst* can override what is considered its class by having a `__class__` attribute.

An object *cls* can override if it is considered a class, and what its base classes are, by having a `__bases__` attribute (which must be a tuple of base classes).

Py_hash_t **PyObject_Hash**(*PyObject* *o)

Part of the Stable ABI. 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: El tipo de retorno ahora es *Py_hash_t*. Este es un entero con signo del mismo tamaño que *Py_ssize_t*.

Py_hash_t **PyObject_HashNotImplemented**(*PyObject* *o)

Part of the Stable ABI. Set a `TypeError` indicating that `type(o)` is not *hashable* and return `-1`. This function receives special treatment when stored in a `tp_hash` slot, allowing a type to explicitly indicate to the interpreter that it is not hashable.

int **PyObject_IsTrue**(*PyObject* *o)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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. Part of the Stable ABI. 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 creates a new *strong reference* to 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 a new *strong reference* is needed.

int **PyObject_TypeCheck**(*PyObject* *o, *PyTypeObject* *type)

Retorna un valor no-nulo si el objeto *o* es de tipo *type* o un subtipo de *type*, y `0` en cualquier otro caso. Ninguno de los dos parámetros debe ser `NULL`.

Py_ssize_t **PyObject_Size**(*PyObject* *o)

Py_ssize_t **PyObject_Length**(*PyObject* *o)

Part of the Stable ABI. 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)

Retorna una longitud estimada para el objeto *o*. Primero intenta retornar su longitud real, luego una estimación usando `__length_hint__()`, y finalmente retorna el valor predeterminado. En caso de error, retorna `-1`. Este es el equivalente a la expresión de Python `operator.length_hint(o, defaultvalue)`.

Added in version 3.4.

PyObject ***PyObject_GetItem**(*PyObject* *o, *PyObject* *key)

Return value: New reference. Part of the Stable ABI. 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)

Part of the Stable ABI. Asigna el objeto *key* al valor *v*. Lanza 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)

Part of the Stable ABI. 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]`.

int **PyObject_DelItemString**(*PyObject* *o, const char *key)

Part of the Stable ABI. This is the same as *`PyObject_DelItem()`*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *`PyObject*`*.

PyObject ***PyObject_Dir**(*PyObject* *o)

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. 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.

PyObject ***PyObject_SelfIter**(*PyObject* *obj)

Return value: New reference. Part of the Stable ABI. This is equivalent to the Python `__iter__(self): return self` method. It is intended for *iterator* types, to be used in the *`PyTypeObject.tp_iter`* slot.

PyObject ***PyObject_GetAIter**(*PyObject* *o)

Return value: New reference. Part of the Stable ABI *since version 3.10.* Esto es equivalente a la expresión de Python `aiter(o)`. Toma un objeto `AsyncIterable` y retorna `AsyncIterator`. Este es típicamente un nuevo iterador, pero si el argumento es `AsyncIterator`, se retornará a sí mismo. Lanza `TypeError` y retorna `NULL` si el objeto no puede ser iterado.

Added in version 3.10.

void ***PyObject_GetTypeData**(*PyObject* *o, *PyTypeObject* *cls)

Part of the Stable ABI *since version 3.12.* Get a pointer to subclass-specific data reserved for *cls*.

The object *o* must be an instance of *cls*, and *cls* must have been created using negative `PyType_Spec.basicsize`. Python does not check this.

On error, set an exception and return `NULL`.

Added in version 3.12.

Py_ssize_t **PyType_GetTypeDataSize**(*PyTypeObject* *cls)

Part of the Stable ABI *since version 3.12.* Return the size of the instance memory space reserved for *cls*, i.e. the size of the memory `PyObject_GetTypeData()` returns.

This may be larger than requested using `-PyType_Spec.basicsize`; it is safe to use this larger size (e.g. with `memset()`).

The type *cls* **must** have been created using negative `PyType_Spec.basicsize`. Python does not check this.

On error, set an exception and return a negative value.

Added in version 3.12.

void ***PyObject_GetItemData**(*PyObject* *o)

Get a pointer to per-item data for a class with `Py_TPFLAGS_ITEMS_AT_END`.

On error, set an exception and return `NULL`. `TypeError` is raised if *o* does not have `Py_TPFLAGS_ITEMS_AT_END` set.

Added in version 3.12.

int **PyObject_VisitManagedDict**(*PyObject* *obj, *visitproc* visit, void *arg)

Visit the managed dictionary of *obj*.

This function must only be called in a traverse function of the type which has the `Py_TPFLAGS_MANAGED_DICT` flag set.

Added in version 3.13.

void **PyObject_ClearManagedDict**(*PyObject* *obj)

Clear the managed dictionary of *obj*.

This function must only be called in a traverse function of the type which has the `Py_TPFLAGS_MANAGED_DICT` flag set.

Added in version 3.13.

int **PyUnstable_Object_EnableDeferredRefcount**(*PyObject* *obj)

This is *Unstable API*. It may change without warning in minor releases.

Enable deferred reference counting on *obj*, if supported by the runtime. In the *free-threaded* build, this allows the interpreter to avoid reference count adjustments to *obj*, which may improve multi-threaded performance. The tradeoff is that *obj* will only be deallocated by the tracing garbage collector, and not when the interpreter no longer has any references to it.

This function returns `1` if deferred reference counting is enabled on *obj*, and `0` if deferred reference counting is not supported or if the hint was ignored by the interpreter, such as when deferred reference counting is already enabled on *obj*. This function is thread-safe, and cannot fail.

This function does nothing on builds with the *GIL* enabled, which do not support deferred reference counting. This also does nothing if *obj* is not an object tracked by the garbage collector (see `gc.is_tracked()` and `PyObject_GC_IsTracked()`).

This function is intended to be used soon after *obj* is created, by the code that creates it, such as in the object's `tp_new` slot.

Added in version 3.14.

int **PyUnstable_Object_IsUniqueReferencedTemporary**(*PyObject* *obj)

> This is *Unstable API*. It may change without warning in minor releases.

Check if *obj* is a unique temporary object. Returns `1` if *obj* is known to be a unique temporary object, and `0` otherwise. This function cannot fail, but the check is conservative, and may return `0` in some cases even if *obj* is a unique temporary object.

If an object is a unique temporary, it is guaranteed that the current code has the only reference to the object. For arguments to C functions, this should be used instead of checking if the reference count is `1`. Starting with Python 3.14, the interpreter internally avoids some reference count modifications when loading objects onto the operands stack by *borrowing* references when possible, which means that a reference count of `1` by itself does not guarantee that a function argument uniquely referenced.

In the example below, `my_func` is called with a unique temporary object as its argument:

```
my_func([1, 2, 3])
```

In the example below, `my_func` is **not** called with a unique temporary object as its argument, even if its refcount is `1`:

```
my_list = [1, 2, 3]
my_func(my_list)
```

See also the function *Py_REFCNT()*.

Added in version 3.14.

int **PyUnstable_IsImmortal**(*PyObject* *obj)

> This is *Unstable API*. It may change without warning in minor releases.

This function returns non-zero if *obj* is *immortal*, and zero otherwise. This function cannot fail.

> **Nota**
>
> Objects that are immortal in one CPython version are not guaranteed to be immortal in another.

Added in version 3.14.

int **PyUnstable_TryIncRef**(*PyObject* *obj)

> This is *Unstable API*. It may change without warning in minor releases.

Increments the reference count of *obj* if it is not zero. Returns `1` if the object's reference count was successfully incremented. Otherwise, this function returns `0`.

PyUnstable_EnableTryIncRef() must have been called earlier on *obj* or this function may spuriously return `0` in the *free threading* build.

This function is logically equivalent to the following C code, except that it behaves atomically in the *free threading* build:

```
if (Py_REFCNT(op) > 0) {
   Py_INCREF(op);
   return 1;
}
return 0;
```

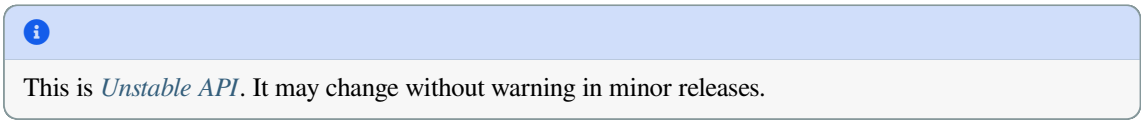
This is intended as a building block for managing weak references without the overhead of a Python *weak reference object*.

Typically, correct use of this function requires support from *obj*'s deallocator (`tp_dealloc`). For example, the following sketch could be adapted to implement a «weakmap» that works like a `WeakValueDictionary` for a specific type:

```
PyMutex mutex;

PyObject *
add_entry(weakmap_key_type *key, PyObject *value)
{
    PyUnstable_EnableTryIncRef(value);
    weakmap_type weakmap = ...;
    PyMutex_Lock(&mutex);
    weakmap_add_entry(weakmap, key, value);
    PyMutex_Unlock(&mutex);
    Py_RETURN_NONE;
}

PyObject *
get_value(weakmap_key_type *key)
{
    weakmap_type weakmap = ...;
    PyMutex_Lock(&mutex);
    PyObject *result = weakmap_find(weakmap, key);
    if (PyUnstable_TryIncRef(result)) {
        // `result` is safe to use
        PyMutex_Unlock(&mutex);
        return result;
    }
    // if we get here, `result` is starting to be garbage-collected,
    // but has not been removed from the weakmap yet
    PyMutex_Unlock(&mutex);
    return NULL;
}

// tp_dealloc function for weakmap values
void
value_dealloc(PyObject *value)
{
    weakmap_type weakmap = ...;
    PyMutex_Lock(&mutex);
    weakmap_remove_value(weakmap, value);

    ...
    PyMutex_Unlock(&mutex);
}
```

Added in version 3.14.

void **PyUnstable_EnableTryIncRef**(*PyObject* *obj)

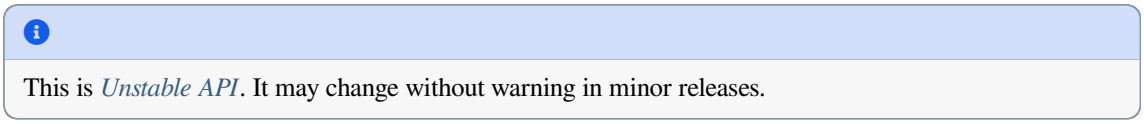


This is *Unstable API*. It may change without warning in minor releases.

Enables subsequent uses of `PyUnstable_TryIncRef()` on *obj*. The caller must hold a *strong reference* to *obj* when calling this.

Added in version 3.14.

int **PyUnstable_Object_IsUniquelyReferenced**(*PyObject* *op)



This is *Unstable API*. It may change without warning in minor releases.

Determine if *op* only has one reference.

On GIL-enabled builds, this function is equivalent to `Py_REFCNT(op) == 1`.

On a *free threaded* build, this checks if *op*'s *reference count* is equal to one and additionally checks if *op* is only used by this thread. `Py_REFCNT(op) == 1` is **not** thread-safe on free threaded builds; prefer this function.

The caller must hold an *attached thread state*, despite the fact that this function doesn't call into the Python interpreter. This function cannot fail.

Added in version 3.14.

8.2 Protocolo de llamada

CPython admite dos protocolos de llamada diferentes: *tp_call* y vectorcall.

8.2.1 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*.

8.2.2 El protocolo vectorcall

Added in version 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.

Distinto en la versión 3.12: The `Py_TPFLAGS_HAVE_VECTORCALL` flag is now removed from a class when the class's `__call__()` method is reassigned. (This internally sets `tp_call` only, and thus may make it behave differently than the vectorcall function.) In earlier Python versions, vectorcall should only be used with `immutable` or static types.

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.

Classes can implement the vectorcall protocol by enabling the `Py_TPFLAGS_HAVE_VECTORCALL` flag and setting `tp_vectorcall_offset` to the offset inside the object structure where a *vectorcallfunc* appears. This is a pointer to a function with the following signature:

typedef *PyObject* *(***vectorcallfunc**)(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

Part of the Stable ABI *since version 3.12.*

- *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** `PY_VECTORCALL_ARGUMENTS_OFFSET` flag. To get the actual number of positional arguments from *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

Part of the Stable ABI *since version 3.12.* 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.

Whenever they can do so cheaply (without additional allocation), callers are encouraged to use `PY_VECTORCALL_ARGUMENTS_OFFSET`. Doing so will allow callables such as bound methods to make their onward calls (which include a prepended *self* argument) very efficiently.

Added in version 3.8.

Para llamar a un objeto que implementa vectorcall, use una función *call API* como con cualquier otro invocable. `PyObject_Vectorcall()` normalmente será más eficiente.

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)

Part of the Stable ABI *since version 3.12.* 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.

Added in version 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`.

Added in version 3.9.

PyObject ***PyVectorcall_Call**(*PyObject* *callable, *PyObject* *tuple, *PyObject* *dict)

Part of the Stable ABI *since version 3.12.* Llama a la *vectorcallfunc* de *callable* con argumentos posicionales y de palabras clave dados en una tupla y dict, respectivamente.

This is a specialized function, intended to be put in the *tp_call* slot or be used in an implementation of `tp_call`. It does not check the *Py_TPFLAGS_HAVE_VECTORCALL* flag and it does not fall back to `tp_call`.

Added in version 3.8.

8.2.3 API para invocar objetos

Hay varias funciones disponibles para llamar a un objeto Python. Cada una convierte sus argumentos a una convención respaldada por el objeto llamado, ya sea *tp_call* o vectorcall. Para realizar la menor conversión posible, elija la que mejor se adapte al formato de datos que tiene disponible.

La siguiente tabla resume las funciones disponibles; consulte la documentación individual para obtener más detalles.

Función	invocable	args	kwargs
PyObject_Call()	`PyObject *`	tupla	dict/`NULL`
PyObject_CallNoArgs()	`PyObject *`	—	—
PyObject_CallOneArg()	`PyObject *`	1 objeto	—
PyObject_CallObject()	`PyObject *`	tuple/`NULL`	—
PyObject_CallFunction()	`PyObject *`	formato	—
PyObject_CallMethod()	obj + `char*`	formato	—
PyObject_CallFunctionObjArgs()	`PyObject *`	variadica	—
PyObject_CallMethodObjArgs()	obj + nombre	variadica	—
PyObject_CallMethodNoArgs()	obj + nombre	—	—
PyObject_CallMethodOneArg()	obj + nombre	1 objeto	—
PyObject_Vectorcall()	`PyObject *`	vectorcall	vectorcall
PyObject_VectorcallDict()	`PyObject *`	vectorcall	dict/`NULL`
PyObject_VectorcallMethod()	arg + nombre	vectorcall	vectorcall

PyObject ***PyObject_Call**(*PyObject* *callable, *PyObject* *args, *PyObject* *kwargs)

Return value: New reference. Part of the Stable ABI. 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 lanza 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)

Return value: New reference. Part of the Stable ABI *since version 3.10.* 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 lanza una excepción y retorna *NULL* en caso de error.

Added in version 3.9.

PyObject ***PyObject_CallOneArg**(*PyObject* *callable, *PyObject* *arg)

Return value: New reference. 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 lanza una excepción y retorna *NULL* en caso de error.

Added in version 3.9.

PyObject ***PyObject_CallObject**(*PyObject* *callable, *PyObject* *args)

Return value: New reference. Part of the Stable ABI. 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 lanza 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. Part of the Stable ABI. 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 lanza una excepción y retorna *NULL* en caso de error.

Este es el equivalente de la expresión de Python: `callable(*args)`.

Note that if you only pass *PyObject** args, *PyObject_CallFunctionObjArgs()* is a faster alternative.

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. Part of the Stable ABI. 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 lanza una excepción y retorna *NULL* en caso de error.

Este es el equivalente de la expresión de Python: `obj.name(arg1, arg2, ...)`.

Note that if you only pass *PyObject** args, *PyObject_CallMethodObjArgs()* is a faster alternative.

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. Part of the Stable ABI. Call a callable Python object *callable*, with a variable number of *PyObject** arguments. The arguments are provided as a variable number of parameters followed by *NULL*.

Retorna el resultado de la llamada en caso de éxito o lanza 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. Part of the Stable ABI. Call a method of the Python object *obj*, where the name of the method is given as a Python string object in *name*. It is called with a variable number of `PyObject*` arguments. The arguments are provided as a variable number of parameters followed by *NULL*.

Retorna el resultado de la llamada en caso de éxito o lanza 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 lanza una excepción y retorna *NULL* en caso de error.

Added in version 3.9.

PyObject ***PyObject_CallMethodOneArg**(*PyObject* *obj, *PyObject* *name, *PyObject* *arg)

Llame 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 lanza una excepción y retorna *NULL* en caso de error.

Added in version 3.9.

PyObject ***PyObject_Vectorcall**(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

Part of the Stable ABI *since version 3.12.* 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 lanza una excepción y retorna *NULL* en caso de error.

Added in version 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.

Added in version 3.9.

PyObject ***PyObject_VectorcallMethod**(*PyObject* *name, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

Part of the Stable ABI *since version 3.12.* Call a method using the vectorcall calling convention. The name of the method is given as a Python string *name*. The object whose method is called is *args[0]*, and the *args* array starting at *args[1]* represents the arguments of the call. There must be at least one positional argument. *nargsf* is the number of positional arguments including *args[0]*, plus `PY_VECTORCALL_ARGUMENTS_OFFSET` if the value of `args[0]` may temporarily be changed. Keyword arguments can be passed just like in `PyObject_Vectorcall()`.

If the object has the `Py_TPFLAGS_METHOD_DESCRIPTOR` feature, this will call the unbound method object with the full *args* vector as arguments.

Retorna el resultado de la llamada en caso de éxito o lanza una excepción y retorna *NULL* en caso de error.

Added in version 3.9.

8.2.4 API de soporte de llamadas

int **PyCallable_Check**(*PyObject* *o)

Part of the Stable ABI. 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.

8.3 Protocolo de números

int **PyNumber_Check**(*PyObject* *o)

Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI *since version 3.7.* 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`.

Added in version 3.5.

PyObject ***PyNumber_FloorDivide**(*PyObject* *o1, *PyObject* *o2)

Return value: New reference. Part of the Stable ABI. Retorna el cociente de la división entera a la baja entre *o1* y *o2*, o `NULL` en caso de error. Este es el equivalente de la expresión de Python `o1 // o2`.

PyObject ***PyNumber_TrueDivide**(*PyObject* *o1, *PyObject* *o2)

Return value: New reference. Part of the Stable ABI. Retorna una aproximación razonable para el valor matemático de *o1* dividido por *o2* o `NULL` en caso de falla. El valor de retorno es «aproximado» porque los números binarios de punto flotante son aproximados; No es posible representar todos los números reales en base dos. Esta función puede retornar un valor de punto flotante cuando se pasan dos enteros. Es equivalente a la expresión de Python `o1 / o2`.

PyObject ***PyNumber_Remainder**(*PyObject* *o1, *PyObject* *o2)

Return value: New reference. Part of the Stable ABI. Retorna el resto de la división entera a la baja entre *o1* y *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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI *since version 3.7.* 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`.

Added in version 3.5.

PyObject ***PyNumber_InPlaceFloorDivide**(*PyObject* *o1, *PyObject* *o2)

Return value: New reference. Part of the Stable ABI. Retorna el cociente de la división entera a la baja entre *o1* y *o2*, o `NULL` en caso de error. La operación se realiza in situ (*in-place*) cuando *o1* lo admite. Es el equivalente de la sentencia de Python `o1 //= o2`.

PyObject ***PyNumber_InPlaceTrueDivide**(*PyObject* *o1, *PyObject* *o2)

Return value: New reference. Part of the Stable ABI. Retorna una aproximación razonable para el valor matemático de *o1* dividido por *o2* o `NULL` en caso de falla. El valor de retorno es «aproximado» porque los números binarios de coma flotante son aproximados; No es posible representar todos los números reales en base dos. Esta función puede retornar un valor de punto flotante cuando se pasan dos enteros. La operación se realiza en su lugar (*in-place*) cuando *o1* lo admite.

PyObject ***PyNumber_InPlaceRemainder**(*PyObject* *o1, *PyObject* *o2)

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. Retorna el *o* convertido aun entero de Python (*int*) en caso de éxito o `NULL` con una excepción `TypeError` lanzada en caso de error.

Distinto en la versión 3.10: El resultado siempre tiene el tipo exacto `int`. Anteriormente, el resultado podía ser una instancia de una subclase de `int`.

PyObject ***PyNumber_ToBase**(*PyObject* *n, int base)

Return value: New reference. Part of the Stable ABI. 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)

Part of the Stable ABI. Retorna *o* convertido a un valor `Py_ssize_t` si *o* puede interpretarse como un entero. Si la llamada falla, se lanza una excepción y se retorna `-1`.

Si *o* puede convertirse a un entero de Python pero el intento de convertirlo a un valor `Py_ssize_t` lanzaría un `OverflowError`, entonces el argumento *exc* es el tipo de excepción que se lanzará (normalmente `IndexError` o `OverflowError`). Si *exc* es `NULL`, la excepción se borra y el valor se recorta a `PY_SSIZE_T_MIN` para un entero negativo o a `PY_SSIZE_T_MAX` para un entero positivo.

int **PyIndex_Check**(*PyObject* *o)

Part of the Stable ABI *since version 3.8.* Retorna `1` si *o* es un entero índice (tiene la ranura `nb_index` de la estructura `tp_as_number` rellenada), y `0` en caso contrario. Esta función siempre finaliza con éxito.

8.4 Protocolo de secuencia

int **PySequence_Check**(*PyObject* *o)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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`.

int **PySequence_In**(*PyObject* *o, *PyObject* *value)

Part of the Stable ABI. Alias for *PySequence_Contains()*.

Obsoleto desde la versión 3.14: The function is *soft deprecated* and should no longer be used to write new code.

Py_ssize_t **PySequence_Index**(*PyObject* *o, *PyObject* *value)

Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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.

8.5 Protocolo de mapeo

Consulte también `PyObject_GetItem()`, `PyObject_SetItem()` y `PyObject_DelItem()`.

int **PyMapping_Check**(*PyObject* *o)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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. Part of the Stable ABI. This is the same as `PyObject_GetItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

int **PyMapping_GetOptionalItem**(*PyObject* *obj, *PyObject* *key, *PyObject* **result)

Part of the Stable ABI *since version 3.13.* Variant of `PyObject_GetItem()` which doesn't raise `KeyError` if the key is not found.

If the key is found, return `1` and set **result* to a new *strong reference* to the corresponding value. If the key is not found, return `0` and set **result* to `NULL`; the `KeyError` is silenced. If an error other than `KeyError` is raised, return `-1` and set **result* to `NULL`.

Added in version 3.13.

int **PyMapping_GetOptionalItemString**(*PyObject* *obj, const char *key, *PyObject* **result)

Part of the Stable ABI *since version 3.13.* This is the same as `PyMapping_GetOptionalItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Added in version 3.13.

int **PyMapping_SetItemString**(*PyObject* *o, const char *key, *PyObject* *v)

Part of the Stable ABI. This is the same as `PyObject_SetItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

int **PyMapping_DelItem**(*PyObject* *o, *PyObject* *key)

This is an alias of `PyObject_DelItem()`.

int **PyMapping_DelItemString**(*PyObject* *o, const char *key)

This is the same as `PyObject_DelItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

int **PyMapping_HasKeyWithError**(*PyObject* *o, *PyObject* *key)

Part of the Stable ABI *since version 3.13.* Return `1` if the mapping object has the key *key* and `0` otherwise. This is equivalent to the Python expression `key in o`. On failure, return `-1`.

Added in version 3.13.

int **PyMapping_HasKeyStringWithError**(*PyObject* *o, const char *key)

Part of the Stable ABI *since version 3.13.* This is the same as *PyMapping_HasKeyWithError()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

Added in version 3.13.

int **PyMapping_HasKey**(*PyObject* *o, *PyObject* *key)

Part of the Stable ABI. 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.

> **Nota**
>
> Exceptions which occur when this calls `__getitem__()` method are silently ignored. For proper error handling, use *PyMapping_HasKeyWithError()*, *PyMapping_GetOptionalItem()* or *PyObject_GetItem()* instead.

int **PyMapping_HasKeyString**(*PyObject* *o, const char *key)

Part of the Stable ABI. This is the same as *PyMapping_HasKey()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

> **Nota**
>
> Exceptions that occur when this calls `__getitem__()` method or while creating the temporary `str` object are silently ignored. For proper error handling, use *PyMapping_HasKeyStringWithError()*, *PyMapping_GetOptionalItemString()* or *PyMapping_GetItemString()* instead.

PyObject ***PyMapping_Keys**(*PyObject* *o)

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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.

8.6 Protocolo iterador

Hay dos funciones específicas para trabajar con iteradores.

int **PyIter_Check**(*PyObject* *o)

Part of the Stable ABI *since version 3.8.* Return non-zero if the object *o* can be safely passed to *PyIter_NextItem()* and `0` otherwise. This function always succeeds.

int **PyAIter_Check**(*PyObject* *o)

Part of the Stable ABI *since version 3.10.* Retorna un valor distinto de cero si el objeto *o* proporciona el protocolo `AsyncIterator`, y `0` en caso contrario. Esta función siempre tiene éxito.

Added in version 3.10.

int **PyIter_NextItem**(*PyObject* *iter, *PyObject* **item)

Part of the Stable ABI *since version 3.14.* Return `1` and set *item* to a *strong reference* of the next value of the iterator *iter* on success. Return `0` and set *item* to `NULL` if there are no remaining values. Return `-1`, set *item* to `NULL` and set an exception on error.

Added in version 3.14.

PyObject ***PyIter_Next**(*PyObject* *o)

Return value: New reference. Part of the Stable ABI. This is an older version of `PyIter_NextItem()`, which is retained for backwards compatibility. Prefer *`PyIter_NextItem()`*.

Retorna el siguiente valor del iterador *o*. El objeto debe ser un iterador según *`PyIter_Check()`* (depende del llamador verificar esto). Si no hay valores restantes, retorna `NULL` sin establecer una excepción. Si ocurre un error al recuperar el elemento, devuelve `NULL` y envía la excepción.

type **PySendResult**

El valor de enumeración utilizado para representar diferentes resultados de *`PyIter_Send()`*.

Added in version 3.10.

PySendResult **PyIter_Send**(*PyObject* *iter, *PyObject* *arg, *PyObject* **presult)

Part of the Stable ABI *since version 3.10.* Envía el valor *arg* al iterador *iter*. Retorna:

- `PYGEN_RETURN` si el iterador regresa. El valor de retorno se retorna a través de *presult*.
- `PYGEN_NEXT` si el iterador cede. El valor cedido se retorna a través de *presult*.
- `PYGEN_ERROR` si el iterador ha lanzado una excepción. *presult* se establece en `NULL`.

Added in version 3.10.

8.7 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 provides such a facility at the C and Python level in the form of the *buffer protocol*. This protocol has two sides:

- on the producer side, a type can export a «buffer interface» which allows objects of that type to expose information about their underlying buffer. This interface is described in the section *Estructuras de objetos búfer*; for Python see python-buffer-protocol.
- on the consumer side, several means are available to obtain a pointer to the raw underlying data of an object (for example a method parameter). For Python see `memoryview`.

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.

An example consumer of the buffer interface is the `write()` method of file objects: any object that can export a series of bytes through the buffer interface can be written to a file. While `write()` only needs read-only access to the internal contents of the object passed to it, other methods such as `readinto()` need write access to the contents of

their argument. The buffer interface allows objects to selectively allow or reject exporting of read-write and read-only buffers.

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.

Added in version 3.12: The buffer protocol is now accessible in Python, see python-buffer-protocol and `memoryview`.

8.7.1 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()*.

type **Py_buffer**

Part of the Stable ABI *(including all members) since version 3.11.*

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.

PyObject ***obj**

A new reference to the exporting object. The reference is owned by the consumer and automatically released (i.e. reference count decremented) and set to `NULL` by *PyBuffer_Release()*. The field is the equivalent of the return value of any standard C-API function.

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 asume `itemsize == 1`.

char ***format**

A *NULL* terminated string in `struct` module style syntax describing the contents of a single item. If this is `NULL`, `"B"` (unsigned bytes) is assumed.

Este campo está controlado por el indicador *PyBUF_FORMAT*.

int **ndim**

The number of dimensions the memory represents as an n-dimensional array. If it is `0`, *buf* points to a single item representing a scalar. In this case, *shape*, *strides* and *suboffsets* MUST be `NULL`. The maximum number of dimensions is given by *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 enésima 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.

Constants:

PyBUF_MAX_NDIM

The maximum number of dimensions the memory represents. Exporters MUST respect this limit, consumers of multi-dimensional buffers SHOULD be able to handle up to `PyBUF_MAX_NDIM` dimensions. Currently set to 64.

8.7.2 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.

All *Py_buffer* fields are unambiguously defined by the request type.

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

Controls the *readonly* field. If set, the exporter MUST provide a writable buffer or else report failure. Otherwise, the exporter MAY provide either a read-only or writable buffer, but the choice MUST be consistent for all consumers. For example, *PyBUF_SIMPLE* | *PyBUF_WRITABLE* can be used to request a simple writable buffer.

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 |"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 must be |"d to any of the flags except *PyBUF_SIMPLE*, because the latter already implies format `B` (unsigned bytes). `PyBUF_FORMAT` cannot be used on its own.

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	*strides*	*suboffsets*
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	*strides*	*suboffsets*	contig
PyBUF_C_CONTIGUOUS	sí	sí	NULL	C
PyBUF_F_CONTIGUOUS	sí	sí	NULL	F
PyBUF_ANY_CONTIGUOUS	sí	sí	NULL	C o F
PyBUF_ND	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	*strides*	*suboffsets*	contig	sólo lectura	formato
PyBUF_FULL	sí	sí	si es necesario	U	0	sí
PyBUF_FULL_RO	sí	sí	si es necesario	U	1 o 0	sí
PyBUF_RECORDS	sí	sí	NULL	U	0	sí
PyBUF_RECORDS_RO	sí	sí	NULL	U	1 o 0	sí
PyBUF_STRIDED	sí	sí	NULL	U	0	NULL
PyBUF_STRIDED_RO	sí	sí	NULL	U	1 o 0	NULL
PyBUF_CONTIG	sí	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	sí	NULL	NULL	C	1 o 0	NULL

8.7.3 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

    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+itemsize <= 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++) {
```

(continúe en la próxima página)

(proviene de la página anterior)

```
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >=0 ) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

8.7.4 Funciones relacionadas a búfer

int **PyObject_CheckBuffer**(*PyObject* *obj)

Part of the Stable ABI *since version 3.11.* 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)

Part of the Stable ABI *since version 3.11.* Send a request to *exporter* to fill in *view* as specified by *flags*. If the exporter cannot provide a buffer of the exact type, it MUST raise `BufferError`, set `view->obj` to `NULL` and return `-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)

Part of the Stable ABI *since version 3.11.* Release the buffer *view* and release the *strong reference* (i.e. decrement the reference count) to the view's supporting object, `view->obj`. This function MUST be called when the buffer is no longer being used, otherwise reference leaks may occur.

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)

Part of the Stable ABI *since version 3.11.* Return the implied `itemsize` from `format`. On error, raise an exception and return -1.

Added in version 3.9.

int **PyBuffer_IsContiguous**(const *Py_buffer* *view, char order)

Part of the Stable ABI *since version 3.11.* 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**(const *Py_buffer* *view, const *Py_ssize_t* *indices)

Part of the Stable ABI *since version 3.11.* 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**(const *Py_buffer* *view, const void *buf, *Py_ssize_t* len, char fort)

Part of the Stable ABI *since version 3.11.* 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, const *Py_buffer* *src, *Py_ssize_t* len, char order)

Part of the Stable ABI *since version 3.11.* 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*.

int **PyObject_CopyData**(*PyObject* *dest, *PyObject* *src)

Part of the Stable ABI *since version 3.11.* Copiar datos del búfer *src* al *dest*. Puede convertir entre búferes de estilo C o Fortran.

Se retorna `0` en caso de éxito, `-1` en caso de error.

void **PyBuffer_FillContiguousStrides**(int ndims, *Py_ssize_t* *shape, *Py_ssize_t* *strides, int itemsize, char order)

Part of the Stable ABI *since version 3.11.* 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)

Part of the Stable ABI *since version 3.11.* 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*.

On success, set `view->obj` to a new reference to *exporter* and return 0. Otherwise, raise `BufferError`, set `view->obj` to `NULL` and return `-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`.

CAPÍTULO 9

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.

9.1 Objetos fundamentales

Esta sección describe los objetos de tipo Python y el objeto singleton `None`.

9.1.1 Objetos tipo

type **PyTypeObject**

Part of the Limited API *(as an opaque struct).* La estructura C de los objetos utilizados para describir los tipos incorporados.

PyTypeObject **PyType_Type**

Part of the Stable ABI. 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**()

Part of the Stable ABI. Borra la caché de búsqueda interna. Retorna la etiqueta (*tag*) de la versión actual.

unsigned long **PyType_GetFlags**(*PyTypeObject* *type)

Part of the Stable ABI. Return the `tp_flags` member of *type*. This function is primarily meant for use with `Py_LIMITED_API`; the individual flag bits are guaranteed to be stable across Python releases, but access to `tp_flags` itself is not part of the *limited API*.

Added in version 3.2.

Distinto en la versión 3.4: El tipo de retorno es ahora `unsigned long` en vez de `long`.

PyObject ***PyType_GetDict**(*PyTypeObject* *type)

Return the type object's internal namespace, which is otherwise only exposed via a read-only proxy (`cls.__dict__`). This is a replacement for accessing `tp_dict` directly. The returned dictionary must be treated as read-only.

This function is meant for specific embedding and language-binding cases, where direct access to the dict is necessary and indirect access (e.g. via the proxy or `PyObject_GetAttr()`) isn't adequate.

Extension modules should continue to use `tp_dict`, directly or indirectly, when setting up their own types.

Added in version 3.12.

void **PyType_Modified**(*PyTypeObject* *type)

Part of the Stable ABI. 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_AddWatcher**(*PyType_WatchCallback* callback)

Register *callback* as a type watcher. Return a non-negative integer ID which must be passed to future calls to `PyType_Watch()`. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

In free-threaded builds, `PyType_AddWatcher()` is not thread-safe, so it must be called at start up (before spawning the first thread).

Added in version 3.12.

int **PyType_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* (previously returned from `PyType_AddWatcher()`). Return `0` on success, `-1` on error (e.g. if *watcher_id* was never registered.)

An extension should never call `PyType_ClearWatcher` with a *watcher_id* that was not returned to it by a previous call to `PyType_AddWatcher()`.

Added in version 3.12.

int **PyType_Watch**(int watcher_id, *PyObject* *type)

Mark *type* as watched. The callback granted *watcher_id* by `PyType_AddWatcher()` will be called whenever `PyType_Modified()` reports a change to *type*. (The callback may be called only once for a series of consecutive modifications to *type*, if `_PyType_Lookup()` is not called on *type* between the modifications; this is an implementation detail and subject to change.)

An extension should never call `PyType_Watch` with a *watcher_id* that was not returned to it by a previous call to `PyType_AddWatcher()`.

Added in version 3.12.

typedef int (***PyType_WatchCallback**)(*PyObject* *type)

Type of a type-watcher callback function.

The callback must not modify *type* or cause `PyType_Modified()` to be called on *type* or any type in its MRO; violating this rule could cause infinite recursion.

Added in version 3.12.

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)

Return true if the type object includes support for the cycle detector; this tests the type flag *Py_TPFLAGS_HAVE_GC*.

int **PyType_IsSubtype**(*PyTypeObject* *a, *PyTypeObject* *b)

Part of the Stable ABI. Retorna verdadero si *a* es un subtipo de *b*.

This function only checks for actual subtypes, which means that `__subclasscheck__()` is not called on *b*. Call *PyObject_IsSubclass()* to do the same check that `issubclass()` would do.

PyObject ***PyType_GenericAlloc**(*PyTypeObject* *type, *Py_ssize_t* nitems)

Return value: New reference. Part of the Stable ABI. Generic handler for the *tp_alloc* slot of a type object. Uses Python's default memory allocation mechanism to allocate memory for a new instance, zeros the memory, then initializes the memory as if by calling *PyObject_Init()* or *PyObject_InitVar()*.

Do not call this directly to allocate memory for an object; call the type's *tp_alloc* slot instead.

For types that support garbage collection (i.e., the *Py_TPFLAGS_HAVE_GC* flag is set), this function behaves like *PyObject_GC_New* or *PyObject_GC_NewVar* (except the memory is guaranteed to be zeroed before initialization), and should be paired with *PyObject_GC_Del()* in *tp_free*. Otherwise, it behaves like *PyObject_New* or *PyObject_NewVar* (except the memory is guaranteed to be zeroed before initialization) and should be paired with *PyObject_Free()* in *tp_free*.

PyObject ***PyType_GenericNew**(*PyTypeObject* *type, *PyObject* *args, *PyObject* *kwds)

Return value: New reference. Part of the Stable ABI. Generic handler for the *tp_new* slot of a type object. Creates a new instance using the type's *tp_alloc* slot and returns the resulting object.

int **PyType_Ready**(*PyTypeObject* *type)

Part of the Stable ABI. 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.

PyObject ***PyType_GetName**(*PyTypeObject* *type)

Return value: New reference. Part of the Stable ABI *since version 3.11.* Return the type's name. Equivalent to getting the type's `__name__` attribute.

Added in version 3.11.

PyObject ***PyType_GetQualName**(*PyTypeObject* *type)

Return value: New reference. Part of the Stable ABI *since version 3.11.* Return the type's qualified name. Equivalent to getting the type's `__qualname__` attribute.

Added in version 3.11.

PyObject ***PyType_GetFullyQualifiedName**(*PyTypeObject* *type)

Part of the Stable ABI *since version 3.13.* Return the type's fully qualified name. Equivalent to `f"{type.__module__}.{type.__qualname__}"`, or `type.__qualname__` if `type.__module__` is not a string or is equal to `"builtins"`.

Added in version 3.13.

PyObject ***PyType_GetModuleName**(*PyTypeObject* *type)

Part of the Stable ABI *since version 3.13.* Return the type's module name. Equivalent to getting the `type.__module__` attribute.

Added in version 3.13.

void ***PyType_GetSlot**(*PyTypeObject* *type, int slot)

Part of the Stable ABI *since version 3.4.* 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*.

Added in version 3.4.

Distinto en la versión 3.10: `PyType_GetSlot()` ahora puede aceptar todos los tipos. Anteriormente, estaba limitado a *heap types*.

PyObject ***PyType_GetModule**(*PyTypeObject* *type)

Part of the Stable ABI *since version 3.10.* 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`.

This function is usually used to get the module in which a method is defined. Note that in such a method, `PyType_GetModule(Py_TYPE(self))` may not return the intended result. `Py_TYPE(self)` may be a *subclass* of the intended class, and subclasses are not necessarily defined in the same module as their superclass. See `PyCMethod` to get the class that defines the method. See `PyType_GetModuleByDef()` for cases when `PyCMethod` cannot be used.

Added in version 3.9.

void ***PyType_GetModuleState**(*PyTypeObject* *type)

Part of the Stable ABI *since version 3.10.* 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.

Added in version 3.9.

PyObject ***PyType_GetModuleByDef**(*PyTypeObject* *type, struct *PyModuleDef* *def)

Return value: Borrowed reference. Part of the Stable ABI *since version 3.13.* Encuentra la primer superclase cuyo módulo fue creado a partir del `PyModuleDef` *def* dado, y retorna ese módulo.

Si no se encuentra ningún módulo, lanza `TypeError` y retorna `NULL`.

Esta función está pensada para ser utilizada junto con `PyModule_GetState()` para obtener el estado del módulo de los métodos de ranura (como `tp_init` o `nb_add`) y en otros lugares donde la clase que define a un método no se puede pasar utilizando la convención de llamada `PyCMethod`.

The returned reference is *borrowed* from *type*, and will be valid as long as you hold a reference to *type*. Do not release it with `Py_DECREF()` or similar.

Added in version 3.11.

int **PyType_GetBaseByToken**(*PyTypeObject* *type, void *token, *PyTypeObject* **result)

Part of the Stable ABI *since version 3.14.* Find the first superclass in *type*'s *method resolution order* whose `Py_tp_token` token is equal to the given one.

- If found, set **result* to a new *strong reference* to it and return `1`.
- If not found, set **result* to `NULL` and return `0`.
- On error, set **result* to `NULL` and return `-1` with an exception set.

The *result* argument may be `NULL`, in which case **result* is not set. Use this if you need only the return value.

The *token* argument may not be `NULL`.

Added in version 3.14.

int **PyUnstable_Type_AssignVersionTag**(*PyTypeObject* *type)

> This is *Unstable API*. It may change without warning in minor releases.

Attempt to assign a version tag to the given type.

Returns 1 if the type already had a valid version tag or a new one was assigned, or 0 if a new tag could not be assigned.

Added in version 3.12.

Crear tipos asignados en montículo (*heap*)

Las siguientes funciones y estructuras se utilizan para crear *heap types*.

PyObject ***PyType_FromMetaclass**(*PyTypeObject* *metaclass, *PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

Part of the Stable ABI *since version 3.12.* Create and return a *heap type* from the *spec* (see `Py_TPFLAGS_HEAPTYPE`).

The metaclass *metaclass* is used to construct the resulting type object. When *metaclass* is `NULL`, the metaclass is derived from *bases* (or *Py_tp_base[s]* slots if *bases* is `NULL`, see below).

Metaclasses that override `tp_new` are not supported, except if `tp_new` is `NULL`.

El argumento *bases* se puede utilizar para especificar clases base; puede ser solo una clase o una tupla de clases. Si *bases* es `NULL`, en su lugar se utiliza la ranura *Py_tp_bases*. Si esa también es `NULL`, se usa la ranura *Py_tp_base* en su lugar. Si también es `NULL`, el nuevo tipo se deriva de `object`.

El argumento *module* se puede utilizar para registrar el módulo en el que se define la nueva clase. Debe ser un objeto de módulo o `NULL`. Si no es `NULL`, el módulo se asocia con el nuevo tipo y luego se puede recuperar con `PyType_GetModule()`. El módulo asociado no es heredado por subclases; debe especificarse para cada clase individualmente.

Esta función llama `PyType_Ready()` en el tipo nuevo.

Note that this function does *not* fully match the behavior of calling `type()` or using the `class` statement. With user-provided base types or metaclasses, prefer *calling* `type` (or the metaclass) over `PyType_From*` functions. Specifically:

- `__new__()` is not called on the new class (and it must be set to `type.__new__`).
- `__init__()` is not called on the new class.
- `__init_subclass__()` is not called on any bases.
- `__set_name__()` is not called on new descriptors.

Added in version 3.12.

PyObject ***PyType_FromModuleAndSpec**(*PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

Return value: New reference. Part of the Stable ABI *since version 3.10.* Equivalent to `PyType_FromMetaclass(NULL, module, spec, bases)`.

Added in version 3.9.

Distinto en la versión 3.10: La función ahora acepta una sola clase como argumento *bases* y `NULL` como ranura `tp_doc`.

Distinto en la versión 3.12: The function now finds and uses a metaclass corresponding to the provided base classes. Previously, only `type` instances were returned.

The `tp_new` of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides `tp_new` is deprecated.

Distinto en la versión 3.14: Creating classes whose metaclass overrides *tp_new* is no longer allowed.

PyObject ***PyType_FromSpecWithBases**(*PyType_Spec* *spec, *PyObject* *bases)

Return value: New reference. Part of the Stable ABI *since version 3.3.* Equivalent to `PyType_FromMetaclass(NULL, NULL, spec, bases)`.

Added in version 3.3.

Distinto en la versión 3.12: The function now finds and uses a metaclass corresponding to the provided base classes. Previously, only `type` instances were returned.

The *tp_new* of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides *tp_new* is deprecated.

Distinto en la versión 3.14: Creating classes whose metaclass overrides *tp_new* is no longer allowed.

PyObject ***PyType_FromSpec**(*PyType_Spec* *spec)

Return value: New reference. Part of the Stable ABI. Equivalent to `PyType_FromMetaclass(NULL, NULL, spec, NULL)`.

Distinto en la versión 3.12: The function now finds and uses a metaclass corresponding to the base classes provided in *Py_tp_base[s]* slots. Previously, only `type` instances were returned.

The *tp_new* of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides *tp_new* is deprecated.

Distinto en la versión 3.14: Creating classes whose metaclass overrides *tp_new* is no longer allowed.

int **PyType_Freeze**(*PyTypeObject* *type)

Part of the Stable ABI *since version 3.14.* Make a type immutable: set the *Py_TPFLAGS_IMMUTABLETYPE* flag.

All base classes of *type* must be immutable.

On success, return `0`. On error, set an exception and return `-1`.

The type must not be used before it's made immutable. For example, type instances must not be created before the type is made immutable.

Added in version 3.14.

type **PyType_Spec**

Part of the Stable ABI *(including all members).* Estructura que define el comportamiento de un tipo.

const char ***name**

Nombre del tipo, utilizado para establecer *PyTypeObject.tp_name*.

int **basicsize**

If positive, specifies the size of the instance in bytes. It is used to set *PyTypeObject.tp_basicsize*.

If zero, specifies that *tp_basicsize* should be inherited.

If negative, the absolute value specifies how much space instances of the class need *in addition* to the superclass. Use *PyObject_GetTypeData()* to get a pointer to subclass-specific memory reserved this way. For negative `basicsize`, Python will insert padding when needed to meet *tp_basicsize*'s alignment requirements.

Distinto en la versión 3.12: Previously, this field could not be negative.

int **itemsize**

Size of one element of a variable-size type, in bytes. Used to set *PyTypeObject.tp_itemsize*. See `tp_itemsize` documentation for caveats.

If zero, *tp_itemsize* is inherited. Extending arbitrary variable-sized classes is dangerous, since some types use a fixed offset for variable-sized memory, which can then overlap fixed-sized memory used by a subclass. To help prevent mistakes, inheriting `itemsize` is only possible in the following situations:

- The base is not variable-sized (its *tp_itemsize*).

- The requested *PyType_Spec.basicsize* is positive, suggesting that the memory layout of the base class is known.
- The requested *PyType_Spec.basicsize* is zero, suggesting that the subclass does not access the instance's memory directly.
- With the *Py_TPFLAGS_ITEMS_AT_END* flag.

unsigned int **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 ***slots**

Arreglo de estructuras *PyType_Slot*. Terminado por el valor de ranura especial `{0, NULL}`.

Each slot ID should be specified at most once.

type **PyType_Slot**

Part of the Stable ABI *(including all members).* Estructura que define la funcionalidad opcional de un tipo, que contiene una ranura ID y un puntero de valor.

int **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*

An additional slot is supported that does not correspond to a `PyTypeObject` struct field:

- *Py_tp_token*

The following "offset" fields cannot be set using *PyType_Slot*:

- *tp_weaklistoffset* (use *Py_TPFLAGS_MANAGED_WEAKREF* instead if possible)
- *tp_dictoffset* (use *Py_TPFLAGS_MANAGED_DICT* instead if possible)
- *tp_vectorcall_offset* (use `"__vectorcalloffset__"` in *PyMemberDef*)

If it is not possible to switch to a `MANAGED` flag (for example, for vectorcall or to support Python older than 3.12), specify the offset in *Py_tp_members*. See *PyMemberDef documentation* for details.

The following internal fields cannot be set at all when creating a heap type:

- *tp_dict*, *tp_mro*, *tp_cache*, *tp_subclasses*, and *tp_weaklist*.

Setting `Py_tp_bases` or `Py_tp_base` may be problematic on some platforms. To avoid issues, use the *bases* argument of *PyType_FromSpecWithBases()* instead.

Distinto en la versión 3.9: Las ranuras en *PyBufferProcs* se pueden configurar en la API ilimitada.

Distinto en la versión 3.11: *bf_getbuffer* and *bf_releasebuffer* are now available under the *limited API*.

Distinto en la versión 3.14: The field *tp_vectorcall* can now set using `Py_tp_vectorcall`. See the field's documentation for details.

void ***pfunc**

El valor deseado de la ranura. En la mayoría de los casos, este es un puntero a una función.

pfunc values may not be `NULL`, except for the following slots:

- `Py_tp_doc`
- *`Py_tp_token`* (for clarity, prefer *`Py_TP_USE_SPEC`* rather than `NULL`)

`Py_tp_token`

A *`slot`* that records a static memory layout ID for a class.

If the *`PyType_Spec`* of the class is statically allocated, the token can be set to the spec using the special value *`Py_TP_USE_SPEC`*:

```
static PyType_Slot foo_slots[] = {
   {Py_tp_token, Py_TP_USE_SPEC},
```

It can also be set to an arbitrary pointer, but you must ensure that:

- The pointer outlives the class, so it's not reused for something else while the class exists.
- It «belongs» to the extension module where the class lives, so it will not clash with other extensions.

Use *`PyType_GetBaseByToken()`* to check if a class's superclass has a given token – that is, check whether the memory layout is compatible.

To get the token for a given class (without considering superclasses), use *`PyType_GetSlot()`* with `Py_tp_token`.

Added in version 3.14.

`Py_TP_USE_SPEC`

Used as a value with *`Py_tp_token`* to set the token to the class's *`PyType_Spec`*. Expands to `NULL`.

Added in version 3.14.

9.1.2 El objeto `None`

Note that the *`PyTypeObject`* for `None` is not directly exposed in the Python/C API. Since `None` is a singleton, testing for object identity (using == in C) is sufficient. There is no `PyNone_Check()` function for the same reason.

PyObject ***Py_None**

The Python `None` object, denoting lack of value. This object has no methods and is *immortal*.

Distinto en la versión 3.12: *`Py_None`* is *immortal*.

`Py_RETURN_NONE`

Return *`Py_None`* from a function.

9.2 Objetos numéricos

9.2.1 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.

type **`PyLongObject`**

Part of the Limited API *(as an opaque struct).* Este subtipo de *`PyObject`* representa un objeto entero de Python.

PyTypeObject **`PyLong_Type`**

Part of the Stable ABI. Esta instancia de *`PyTypeObject`* 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. Part of the Stable ABI. Retorna un objeto `PyLongObject` nuevo desde *v*, o `NULL` en caso de error.

Detalles de implementación de CPython: CPython 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. Part of the Stable ABI. Return a new `PyLongObject` object from a C `unsigned long`, or `NULL` on failure.

PyObject ***PyLong_FromSsize_t**(*Py_ssize_t* v)

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. Return a new `PyLongObject` object from a C `long long`, or `NULL` on failure.

PyObject ***PyLong_FromInt32**(int32_t value)

PyObject ***PyLong_FromInt64**(int64_t value)

Part of the Stable ABI *since version 3.14.* Return a new `PyLongObject` object from a signed C `int32_t` or `int64_t`, or `NULL` with an exception set on failure.

Added in version 3.14.

PyObject ***PyLong_FromUnsignedLongLong**(unsigned long long v)

Return value: New reference. Part of the Stable ABI. Return a new `PyLongObject` object from a C `unsigned long long`, or `NULL` on failure.

PyObject ***PyLong_FromUInt32**(uint32_t value)

PyObject ***PyLong_FromUInt64**(uint64_t value)

Part of the Stable ABI *since version 3.14.* Return a new `PyLongObject` object from an unsigned C `uint32_t` or `uint64_t`, or `NULL` with an exception set on failure.

Added in version 3.14.

PyObject ***PyLong_FromDouble**(double v)

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. Return a new `PyLongObject` based on the string value in *str*, which is interpreted according to the radix in *base*, or `NULL` on failure. If *pend* is non-`NULL`, **pend* will point to the end of *str* on success or to the first character that could not be processed on error. If *base* is `0`, *str* is interpreted using the integers definition; in this case, leading zeros in a non-zero decimal number raises a `ValueError`. If *base* is not `0`, it must be between `2` and `36`, inclusive. Leading and trailing whitespace and single underscores after a base specifier and between digits are ignored. If there are no digits or *str* is not NULL-terminated following the digits and trailing whitespace, `ValueError` will be raised.

Ver también

PyLong_AsNativeBytes() and *PyLong_FromNativeBytes()* functions can be used to convert a *PyLongObject* to/from an array of bytes in base 256.

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.

Added in version 3.3.

PyObject ***PyLong_FromVoidPtr**(void *p)

Return value: New reference. Part of the Stable ABI. Crea un entero de Python desde el puntero *p*. El valor del puntero se puede recuperar del valor resultante usando *PyLong_AsVoidPtr()*.

PyObject ***PyLong_FromNativeBytes**(const void *buffer, size_t n_bytes, int flags)

Part of the Stable ABI *since version 3.14.* Create a Python integer from the value contained in the first *n_bytes* of *buffer*, interpreted as a two's-complement signed number.

flags are as for *PyLong_AsNativeBytes()*. Passing `-1` will select the native endian that CPython was compiled with and assume that the most-significant bit is a sign bit. Passing `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` will produce the same result as calling *PyLong_FromUnsignedNativeBytes()*. Other flags are ignored.

Added in version 3.13.

PyObject ***PyLong_FromUnsignedNativeBytes**(const void *buffer, size_t n_bytes, int flags)

Part of the Stable ABI *since version 3.14.* Create a Python integer from the value contained in the first *n_bytes* of *buffer*, interpreted as an unsigned number.

flags are as for *PyLong_AsNativeBytes()*. Passing `-1` will select the native endian that CPython was compiled with and assume that the most-significant bit is not a sign bit. Flags other than endian are ignored.

Added in version 3.13.

long **PyLong_AsLong**(*PyObject* *obj)

Part of the Stable ABI. Return a C `long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

Raise `OverflowError` if the value of *obj* is out of range for a `long`.

Retorna `-1` en caso de error. Use *PyErr_Occurred()* para desambiguar.

Distinto en la versión 3.8: Use `__index__()` if available.

Distinto en la versión 3.10: This function will no longer use `__int__()`.

long **PyLong_AS_LONG**(*PyObject* *obj)

A *soft deprecated* alias. Exactly equivalent to the preferred `PyLong_AsLong`. In particular, it can fail with `OverflowError` or another exception.

Obsoleto desde la versión 3.14: The function is soft deprecated.

int **PyLong_AsInt**(*PyObject* *obj)

Part of the Stable ABI *since version 3.13.* Similar to *PyLong_AsLong()*, but store the result in a C `int` instead of a C `long`.

Added in version 3.13.

long **PyLong_AsLongAndOverflow**(*PyObject* *obj, int *overflow)

Part of the Stable ABI. Return a C `long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

If the value of *obj* is greater than `LONG_MAX` or less than `LONG_MIN`, set **overflow* to `1` or `-1`, respectively, and return `-1`; otherwise, set **overflow* to `0`. If any other exception occurs set **overflow* to `0` and return `-1` as usual.

Retorna `-1` en caso de error. Use *PyErr_Occurred()* para desambiguar.

Distinto en la versión 3.8: Use `__index__()` if available.

Distinto en la versión 3.10: This function will no longer use `__int__()`.

long long **PyLong_AsLongLong**(*PyObject* *obj)

Part of the Stable ABI. Return a C `long long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

Raise `OverflowError` if the value of *obj* is out of range for a `long long`.

Retorna `-1` en caso de error. Use *PyErr_Occurred()* para desambiguar.

Distinto en la versión 3.8: Use `__index__()` if available.

Distinto en la versión 3.10: This function will no longer use `__int__()`.

long long **PyLong_AsLongLongAndOverflow**(*PyObject* *obj, int *overflow)

Part of the Stable ABI. Return a C `long long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

If the value of *obj* is greater than `LLONG_MAX` or less than `LLONG_MIN`, set **overflow* to `1` or `-1`, respectively, and return `-1`; otherwise, set **overflow* to `0`. If any other exception occurs set **overflow* to `0` and return `-1` as usual.

Retorna `-1` en caso de error. Use *PyErr_Occurred()* para desambiguar.

Added in version 3.2.

Distinto en la versión 3.8: Use `__index__()` if available.

Distinto en la versión 3.10: This function will no longer use `__int__()`.

Py_ssize_t **PyLong_AsSsize_t**(*PyObject* *pylong)

Part of the Stable ABI. 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)

Part of the Stable ABI. Return a C `unsigned long` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a `unsigned long`.

Retorna `(unsigned long)-1` en caso de error. Use *PyErr_Occurred()* para desambiguar.

size_t **PyLong_AsSize_t**(*PyObject* *pylong)

Part of the Stable ABI. 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)

Part of the Stable ABI. Return a C `unsigned long long` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for an `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)

Part of the Stable ABI. Return a C `unsigned long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

If the value of *obj* is out of range for an `unsigned long`, return the reduction of that value modulo `ULONG_MAX + 1`.

Retorna `(unsigned long)-1` en caso de error. Use *PyErr_Occurred()* para desambiguar.

Distinto en la versión 3.8: Use `__index__()` if available.

Distinto en la versión 3.10: This function will no longer use `__int__()`.

unsigned long long **PyLong_AsUnsignedLongLongMask**(*PyObject* *obj)

Part of the Stable ABI. Return a C `unsigned long long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

If the value of *obj* is out of range for an `unsigned long long`, return the reduction of that value modulo `ULLONG_MAX + 1`.

Retorna `(unsigned long long) -1` por error. Use *PyErr_Occurred()* para desambiguar.

Distinto en la versión 3.8: Use `__index__()` if available.

Distinto en la versión 3.10: This function will no longer use `__int__()`.

int **PyLong_AsInt32**(*PyObject* *obj, int32_t *value)

int **PyLong_AsInt64**(*PyObject* *obj, int64_t *value)

Part of the Stable ABI *since version 3.14*. Set **value* to a signed C `int32_t` or `int64_t` representation of *obj*.

If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

If the *obj* value is out of range, raise an `OverflowError`.

Set **value* and return `0` on success. Set an exception and return `-1` on error.

value must not be `NULL`.

Added in version 3.14.

int **PyLong_AsUInt32**(*PyObject* *obj, uint32_t *value)

int **PyLong_AsUInt64**(*PyObject* *obj, uint64_t *value)

Part of the Stable ABI *since version 3.14*. Set **value* to an unsigned C `uint32_t` or `uint64_t` representation of *obj*.

If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

- If *obj* is negative, raise a `ValueError`.
- If the *obj* value is out of range, raise an `OverflowError`.

Set **value* and return `0` on success. Set an exception and return `-1` on error.

value must not be `NULL`.

Added in version 3.14.

double **PyLong_AsDouble**(*PyObject* *pylong)

Part of the Stable ABI. Return a C `double` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a `double`.

Retorna `-1.0` en caso de error. Use *PyErr_Occurred()* para desambiguar.

void *`PyLong_AsVoidPtr`(*PyObject* *pylong)

Part of the Stable ABI. Convert a Python integer *pylong* to a C `void` pointer. If *pylong* cannot be converted, an `OverflowError` will be raised. This is only assured to produce a usable `void` pointer for values created with *`PyLong_FromVoidPtr()`*.

Retorna `NULL` en caso de error. Use *`PyErr_Occurred()`* para desambiguar.

Py_ssize_t `PyLong_AsNativeBytes`(*PyObject* *pylong, void *buffer, *Py_ssize_t* n_bytes, int flags)

Part of the Stable ABI *since version 3.14*. Copy the Python integer value *pylong* to a native *buffer* of size *n_bytes*. The *flags* can be set to `-1` to behave similarly to a C cast, or to values documented below to control the behavior.

Returns `-1` with an exception raised on error. This may happen if *pylong* cannot be interpreted as an integer, or if *pylong* was negative and the `Py_ASNATIVEBYTES_REJECT_NEGATIVE` flag was set.

Otherwise, returns the number of bytes required to store the value. If this is equal to or less than *n_bytes*, the entire value was copied. All *n_bytes* of the buffer are written: large buffers are padded with zeroes.

If the returned value is greater than *n_bytes*, the value was truncated: as many of the lowest bits of the value as could fit are written, and the higher bits are ignored. This matches the typical behavior of a C-style downcast.

> **Nota**
>
> Overflow is not considered an error. If the returned value is larger than *n_bytes*, most significant bits were discarded.

`0` will never be returned.

Values are always copied as two's-complement.

Usage example:

```
int32_t value;
Py_ssize_t bytes = PyLong_AsNativeBytes(pylong, &value, sizeof(value), -1);
if (bytes < 0) {
    // Failed. A Python exception was set with the reason.
    return NULL;
}
else if (bytes <= (Py_ssize_t)sizeof(value)) {
    // Success!
}
else {
    // Overflow occurred, but 'value' contains the truncated
    // lowest bits of pylong.
}
```

Passing zero to *n_bytes* will return the size of a buffer that would be large enough to hold the value. This may be larger than technically necessary, but not unreasonably so. If *n_bytes=0*, *buffer* may be `NULL`.

> **Nota**
>
> Passing *n_bytes=0* to this function is not an accurate way to determine the bit length of the value.

To get at the entire Python value of an unknown size, the function can be called twice: first to determine the buffer size, then to fill it:

```
// Ask how much space we need.
Py_ssize_t expected = PyLong_AsNativeBytes(pylong, NULL, 0, -1);
if (expected < 0) {
```

(continúe en la próxima página)

(proviene de la página anterior)

```
    // Failed. A Python exception was set with the reason.
    return NULL;
}
assert(expected != 0);  // Impossible per the API definition.
uint8_t *bignum = malloc(expected);
if (!bignum) {
    PyErr_SetString(PyExc_MemoryError, "bignum malloc failed.");
    return NULL;
}
// Safely get the entire value.
Py_ssize_t bytes = PyLong_AsNativeBytes(pylong, bignum, expected, -1);
if (bytes < 0) {  // Exception has been set.
    free(bignum);
    return NULL;
}
else if (bytes > expected) {  // This should not be possible.
    PyErr_SetString(PyExc_RuntimeError,
        "Unexpected bignum truncation after a size check.");
    free(bignum);
    return NULL;
}
// The expected success given the above pre-check.
// ... use bignum ...
free(bignum);
```

flags is either `-1` (`Py_ASNATIVEBYTES_DEFAULTS`) to select defaults that behave most like a C cast, or a combination of the other flags in the table below. Note that `-1` cannot be combined with other flags.

Currently, `-1` corresponds to `Py_ASNATIVEBYTES_NATIVE_ENDIAN | Py_ASNATIVEBYTES_UNSIGNED_BUFFER`.

Flag	Value
Py_ASNATIVEBYTES_DEFAULTS	-1
Py_ASNATIVEBYTES_BIG_ENDIAN	0
Py_ASNATIVEBYTES_LITTLE_ENDIAN	1
Py_ASNATIVEBYTES_NATIVE_ENDIAN	3
Py_ASNATIVEBYTES_UNSIGNED_BUFFER	4
Py_ASNATIVEBYTES_REJECT_NEGATIVE	8
Py_ASNATIVEBYTES_ALLOW_INDEX	16

Specifying `Py_ASNATIVEBYTES_NATIVE_ENDIAN` will override any other endian flags. Passing `2` is reserved.

By default, sufficient buffer will be requested to include a sign bit. For example, when converting 128 with

n_bytes=1, the function will return 2 (or more) in order to store a zero sign bit.

If `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` is specified, a zero sign bit will be omitted from size calculations. This allows, for example, 128 to fit in a single-byte buffer. If the destination buffer is later treated as signed, a positive input value may become negative. Note that the flag does not affect handling of negative values: for those, space for a sign bit is always requested.

Specifying `Py_ASNATIVEBYTES_REJECT_NEGATIVE` causes an exception to be set if *pylong* is negative. Without this flag, negative values will be copied provided there is enough space for at least one sign bit, regardless of whether `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` was specified.

If `Py_ASNATIVEBYTES_ALLOW_INDEX` is specified and a non-integer value is passed, its `__index__()` method will be called first. This may result in Python code executing and other threads being allowed to run, which could cause changes to other objects or values in use. When *flags* is `-1`, this option is not set, and non-integer values will raise `TypeError`.

Nota

With the default *flags* (`-1`, or *UNSIGNED_BUFFER* without *REJECT_NEGATIVE*), multiple Python integers can map to a single value without overflow. For example, both `255` and `-1` fit a single-byte buffer and set all its bits. This matches typical C cast behavior.

Added in version 3.13.

int **PyLong_GetSign**(*PyObject* *obj, int *sign)

Get the sign of the integer object *obj*.

On success, set **sign* to the integer sign (0, -1 or +1 for zero, negative or positive integer, respectively) and return 0.

On failure, return -1 with an exception set. This function always succeeds if *obj* is a `PyLongObject` or its subtype.

Added in version 3.14.

int **PyLong_IsPositive**(*PyObject* *obj)

Check if the integer object *obj* is positive (`obj > 0`).

If *obj* is an instance of `PyLongObject` or its subtype, return `1` when it's positive and `0` otherwise. Else set an exception and return `-1`.

Added in version 3.14.

int **PyLong_IsNegative**(*PyObject* *obj)

Check if the integer object *obj* is negative (`obj < 0`).

If *obj* is an instance of `PyLongObject` or its subtype, return `1` when it's negative and `0` otherwise. Else set an exception and return `-1`.

Added in version 3.14.

int **PyLong_IsZero**(*PyObject* *obj)

Check if the integer object *obj* is zero.

If *obj* is an instance of `PyLongObject` or its subtype, return `1` when it's zero and `0` otherwise. Else set an exception and return `-1`.

Added in version 3.14.

PyObject ***PyLong_GetInfo**(void)

Part of the Stable ABI. On success, return a read only *named tuple*, that holds information about Python's internal representation of integers. See `sys.int_info` for description of individual fields.

On failure, return `NULL` with an exception set.

Added in version 3.1.

int **PyUnstable_Long_IsCompact**(const *PyLongObject* *op)

> This is *Unstable API*. It may change without warning in minor releases.

Return 1 if *op* is compact, 0 otherwise.

This function makes it possible for performance-critical code to implement a "fast path" for small integers. For compact values use `PyUnstable_Long_CompactValue()`; for others fall back to a `PyLong_As*` function or `PyLong_AsNativeBytes()`.

The speedup is expected to be negligible for most users.

Exactly what values are considered compact is an implementation detail and is subject to change.

Added in version 3.12.

Py_ssize_t **PyUnstable_Long_CompactValue**(const *PyLongObject* *op)

> This is *Unstable API*. It may change without warning in minor releases.

If *op* is compact, as determined by `PyUnstable_Long_IsCompact()`, return its value.

Otherwise, the return value is undefined.

Added in version 3.12.

Export API

Added in version 3.14.

struct **PyLongLayout**

Layout of an array of «digits» («limbs» in the GMP terminology), used to represent absolute value for arbitrary precision integers.

Use `PyLong_GetNativeLayout()` to get the native layout of Python `int` objects, used internally for integers with «big enough» absolute value.

See also `sys.int_info` which exposes similar information in Python.

uint8_t **bits_per_digit**

Bits per digit. For example, a 15 bit digit means that bits 0-14 contain meaningful information.

uint8_t **digit_size**

Digit size in bytes. For example, a 15 bit digit will require at least 2 bytes.

int8_t **digits_order**

Digits order:

- `1` for most significant digit first
- `-1` for least significant digit first

int8_t **digit_endianness**

Digit endianness:

- `1` for most significant byte first (big endian)
- `-1` for least significant byte first (little endian)

const *PyLongLayout* ***PyLong_GetNativeLayout**(void)

Get the native layout of Python `int` objects.

See the *PyLongLayout* structure.

The function must not be called before Python initialization nor after Python finalization. The returned layout is valid until Python is finalized. The layout is the same for all Python sub-interpreters in a process, and so it can be cached.

struct **PyLongExport**

Export of a Python `int` object.

There are two cases:

- If *digits* is `NULL`, only use the *value* member.
- If *digits* is not `NULL`, use *negative*, *ndigits* and *digits* members.

int64_t **value**

The native integer value of the exported `int` object. Only valid if *digits* is `NULL`.

uint8_t **negative**

`1` if the number is negative, `0` otherwise. Only valid if *digits* is not `NULL`.

Py_ssize_t **ndigits**

Number of digits in *digits* array. Only valid if *digits* is not `NULL`.

const void ***digits**

Read-only array of unsigned digits. Can be `NULL`.

int **PyLong_Export**(*PyObject* *obj, *PyLongExport* *export_long)

Export a Python `int` object.

export_long must point to a *PyLongExport* structure allocated by the caller. It must not be `NULL`.

On success, fill in **export_long* and return `0`. On error, set an exception and return `-1`.

PyLong_FreeExport() must be called when the export is no longer needed.

Detalles de implementación de CPython: This function always succeeds if *obj* is a Python `int` object or a subclass.

void **PyLong_FreeExport**(*PyLongExport* *export_long)

Release the export *export_long* created by *PyLong_Export()*.

Detalles de implementación de CPython: Calling *PyLong_FreeExport()* is optional if *export_long->digits* is `NULL`.

PyLongWriter API

The *PyLongWriter* API can be used to import an integer.

Added in version 3.14.

struct **PyLongWriter**

A Python `int` writer instance.

The instance must be destroyed by *PyLongWriter_Finish()* or *PyLongWriter_Discard()*.

PyLongWriter ***PyLongWriter_Create**(int negative, *Py_ssize_t* ndigits, void **digits)

Create a *PyLongWriter*.

On success, allocate **digits* and return a writer. On error, set an exception and return `NULL`.

negative is `1` if the number is negative, or `0` otherwise.

ndigits is the number of digits in the *digits* array. It must be greater than 0.

digits must not be NULL.

After a successful call to this function, the caller should fill in the array of digits *digits* and then call *PyLongWriter_Finish()* to get a Python `int`. The layout of *digits* is described by *PyLong_GetNativeLayout()*.

Digits must be in the range [`0`; `(1 << bits_per_digit) - 1`] (where the *bits_per_digit* is the number of bits per digit). Any unused most significant digits must be set to `0`.

Alternately, call *PyLongWriter_Discard()* to destroy the writer instance without creating an `int` object.

PyObject ***PyLongWriter_Finish**(*PyLongWriter* *writer)

Return value: New reference. Finish a *PyLongWriter* created by *PyLongWriter_Create()*.

On success, return a Python `int` object. On error, set an exception and return `NULL`.

The function takes care of normalizing the digits and converts the object to a compact integer if needed.

The writer instance and the *digits* array are invalid after the call.

void **PyLongWriter_Discard**(*PyLongWriter* *writer)

Discard a *PyLongWriter* created by *PyLongWriter_Create()*.

If *writer* is `NULL`, no operation is performed.

The writer instance and the *digits* array are invalid after the call.

9.2.2 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.

PyTypeObject **PyBool_Type**

Part of the Stable ABI. Esta instancia de *PyTypeObject* representa el tipo booleano de Python; es el mismo objeto que `bool` en la capa de Python.

int **PyBool_Check**(*PyObject* *o)

Retorna verdadero si *o* es de tipo *PyBool_Type*. Esta función siempre finaliza con éxito.

PyObject ***Py_False**

The Python `False` object. This object has no methods and is *immortal*.

Distinto en la versión 3.12: *Py_False* is *immortal*.

PyObject ***Py_True**

The Python `True` object. This object has no methods and is *immortal*.

Distinto en la versión 3.12: *Py_True* is *immortal*.

Py_RETURN_FALSE

Retorna *Py_False* desde una función.

Py_RETURN_TRUE

Retorna *Py_True* desde una función..

PyObject ***PyBool_FromLong**(long v)

Return value: New reference. Part of the Stable ABI. Retorna *Py_True* o *Py_False*, dependiendo del valor verdadero de *v*.

9.2.3 Floating-Point Objects

type **PyFloatObject**

This subtype of *PyObject* represents a Python floating-point object.

PyTypeObject **PyFloat_Type**

Part of the Stable ABI. This instance of *PyTypeObject* represents the Python floating-point type. This is the same object as `float` in the Python layer.

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. Part of the Stable ABI. Crea un objeto *PyFloatObject* basado en la cadena de caracteres en *str*, o `NULL` en caso de error.

PyObject ***PyFloat_FromDouble**(double v)

Return value: New reference. Part of the Stable ABI. Crea un objeto *PyFloatObject* a partir de *v*, o `NULL` en caso de error.

double **PyFloat_AsDouble**(*PyObject* *pyfloat)

Part of the Stable ABI. Return a C `double` representation of the contents of *pyfloat*. If *pyfloat* is not a Python floating-point object but has a `__float__()` method, this method will first be called to convert *pyfloat* into a float. If `__float__()` is not defined then it falls back to `__index__()`. This method returns `-1.0` upon failure, so one should call *PyErr_Occurred()* to check for errors.

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. Part of the Stable ABI. Retorna una instancia de *structseq* que contiene información sobre la precisión, los valores mínimos y máximos de un flotante. Es un contenedor reducido alrededor del archivo de encabezado `float.h`.

double **PyFloat_GetMax**()

Part of the Stable ABI. Retorna el máximo flotante finito representable *DBL_MAX* como C `double`.

double **PyFloat_GetMin**()

Part of the Stable ABI. Retorna el flotante positivo normalizado mínimo *DBL_MIN* como C `double`.

Funciones de empaquetado y desempaquetado

Las funciones de empaquetar y desempaquetar proporcionan una manera eficiente e independiente de la plataforma para almacenar valores de coma flotante como cadenas de bytes. Las rutinas Pack producen una cadena de bytes a partir de un C `double`, y las rutinas Desempaquetar producen un C `double` a partir de dicha cadena de bytes. El sufijo (2, 4 u 8) especifica el número de bytes en la cadena de bytes.

En plataformas que parecen usar formatos IEEE 754, estas funciones actúan copiando los bits. En otras plataformas, el formato 2-byte es idéntico al formato de media precision IEEE 754 binary16, el formato de 4-byte (32 bits) es idéntico al formato de precisión simple binario IEEE 754 binary32, y el formato de 8-byte al formato de doble precisión binario IEEE 754 binary64, aunque el empaquetado de INFs y NaNs (si existen en la plataforma) no se maneja correctamente, mientras que intentar desempaquetar una cadena de bytes que contenga un IEEE INF o NaN generará una excepción.

Note that NaNs type may not be preserved on IEEE platforms (silent NaN become quiet), for example on x86 systems in 32-bit mode.

En plataformas que no son IEEE con más precisión, o mayor rango dinámico, que el IEEE 754 admite, no se pueden empaquetar todos los valores; en plataformas que no son IEEE con menos precisión o con un rango dinámico más pequeño, no se pueden desempaquetar todos los valores. Lo que sucede en tales casos es en parte accidental (desafortunadamente).

Added in version 3.11.

Funciones de Empaquetado

Las rutinas de empaquetado escriben 2, 4 o 8 bytes, comenzando en *p*. *le* es un argumento `int`, distinto de cero si desea que la cadena de bytes esté en formato little-endian (exponente al final, en `p+1`, `p+3`, o `p+6` `p+7`), y cero si desea el formato big-endian (exponente primero, en *p*). La constante `PY_BIG_ENDIAN` se puede usar para emplear el endian nativo: es igual a `1` en el procesador big-endian, o `0` en el procesador little-endian.

Valor retornado: `0` si todo está bien, `-1` si hay error (y se establece una excepción, probablemente `OverflowError`).

Hay dos problemas en plataformas que no son IEEE:

- Lo que esto hace es indefinido si *x* es un NaN o infinito.
- `-0.0` and `+0.0` produce la misma cadena de bytes.

int **PyFloat_Pack2**(double x, char *p, int le)

Empaquete un C doble como el formato de media precisión IEEE 754 binary16.

int **PyFloat_Pack4**(double x, char *p, int le)

Empaque un C doble como el formato de precisión simple IEEE 754 binary32.

int **PyFloat_Pack8**(double x, char *p, int le)

Empaque un C doble como el formato de doble precisión IEEE 754 binary64.

Funciones de Desempaquetado

Las rutinas de desempaquetado leen 2, 4 u 8 bytes, comenzando en *p*. *le* es un argumento `int` , distinto de cero si la cadena bytes está en formato little-endian (exponente al final, en `p+1`, `p+3` o `p+6` y `p+7`), cero si está en formato big-endian (exponente primero, en *p*). La constante `PY_BIG_ENDIAN` se puede usar para utilizar el endian nativo: es igual a `1` en un procesador big endian, o `0` en un procesador little-endian.

Valor retornado: Doble desempaquetado. Si hay error, `-1.0` y *PyErr_Occurred()* es verdadero (y se establece una excepción, probablemente `OverflowError`).

Hay que tener en cuenta que en una plataforma que no sea IEEE, esto se negará a desempaquetar una cadena de bytes que representa un NaN o infinito.

double **PyFloat_Unpack2**(const char *p, int le)

Descomprima el formato de media precisión IEEE 754 binary16 como un doble C.

double **PyFloat_Unpack4**(const char *p, int le)

Descomprima el formato de precisión simple IEEE 754 binary32 como un doble C.

double **PyFloat_Unpack8**(const char *p, int le)

Descomprima el formato de doble precisión IEEE 754 binary64 como un doble C.

9.2.4 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.

type **Py_complex**

The C structure which corresponds to the value portion of a Python complex number object. Most of the functions for dealing with complex number objects use structures of this type as input or output values, as appropriate.

double **real**

double **imag**

The structure is defined as:

```
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)

Retorna la negación del número complejo *num*, utilizando la representación C *Py_complex*.

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*.

If *divisor* is null, this method returns zero and sets `errno` to `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*.

If *num* is null and *exp* is not a positive real number, this method returns zero and sets `errno` to `EDOM`.

Set `errno` to `ERANGE` on overflows.

Números complejos como objetos de Python

type **PyComplexObject**

Este subtipo de *PyObject* representa un objeto de número complejo de Python.

PyTypeObject **PyComplex_Type**

Part of the Stable ABI. 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. Create a new Python complex number object from a C *Py_complex* value. Return `NULL` with an exception set on error.

PyObject ***PyComplex_FromDoubles**(double real, double imag)

Return value: New reference. Part of the Stable ABI. Return a new *PyComplexObject* object from *real* and *imag*. Return `NULL` with an exception set on error.

double **PyComplex_RealAsDouble**(*PyObject* *op)

Part of the Stable ABI. Return the real part of *op* as a C `double`.

If *op* is not a Python complex number object but has a `__complex__()` method, this method will first be called to convert *op* to a Python complex number object. If `__complex__()` is not defined then it falls back to call *PyFloat_AsDouble()* and returns its result.

Upon failure, this method returns `-1.0` with an exception set, so one should call *PyErr_Occurred()* to check for errors.

Distinto en la versión 3.13: Use `__complex__()` if available.

double **PyComplex_ImagAsDouble**(*PyObject* *op)

Part of the Stable ABI. Return the imaginary part of *op* as a C `double`.

If *op* is not a Python complex number object but has a `__complex__()` method, this method will first be called to convert *op* to a Python complex number object. If `__complex__()` is not defined then it falls back to call *PyFloat_AsDouble()* and returns `0.0` on success.

Upon failure, this method returns `-1.0` with an exception set, so one should call *PyErr_Occurred()* to check for errors.

Distinto en la versión 3.13: Use `__complex__()` if available.

Py_complex **PyComplex_AsCComplex**(*PyObject* *op)

Retorna el valor *Py_complex* del número complejo *op*.

If *op* is not a Python complex number object but has a `__complex__()` method, this method will first be called to convert *op* to a Python complex number object. If `__complex__()` is not defined then it falls back to `__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

Upon failure, this method returns *Py_complex* with *real* set to `-1.0` and with an exception set, so one should call *PyErr_Occurred()* to check for errors.

Distinto en la versión 3.8: Use `__index__()` if available.

9.3 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.

9.3.1 Objetos bytes

Estas funciones lanzan `TypeError` cuando se espera un parámetro de bytes y se llama con un parámetro que no es bytes.

type **PyBytesObject**

Este subtipo de *PyObject* representa un objeto bytes de Python.

PyTypeObject **PyBytes_Type**

Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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. Part of the Stable ABI. 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
`%%`	*n/a*	El carácter literal %.
`%c`	int	Un solo byte, representado como un C int.
`%d`	int	Equivalente a `printf("%d")`.[1]
`%u`	unsigned int	Equivalente a `printf("%u")`.[1]
`%ld`	long	Equivalente a `printf("%ld")`.[1]
`%lu`	unsigned long	Equivalente a `printf("%lu")`.[1]
`%zd`	*Py_ssize_t*	Equivalente a `printf("%zd")`.[1]
`%zu`	size_t	Equivalente a `printf("%zu")`.[1]
`%i`	int	Equivalente a `printf("%i")`.[1]
`%x`	int	Equivalente a `printf("%x")`.[1]
`%s`	const char*	Un arreglo de caracteres C terminados en nulo.
`%p`	const void*	La representación hexadecimal de un puntero en C. Principalmente equivalente a `printf("%p")` excepto que se garantiza que comience con el literal `0x`, independientemente de lo que produzca el `printf` 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. Part of the Stable ABI. Idéntica a *PyBytes_FromFormat()* excepto que toma exactamente dos argumentos.

PyObject ***PyBytes_FromObject**(*PyObject* *o)

Return value: New reference. Part of the Stable ABI. Retorna la representación en bytes del objeto *o* que implementa el protocolo de búfer.

Py_ssize_t **PyBytes_Size**(*PyObject* *o)

Part of the Stable ABI. 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)

Part of the Stable ABI. Retorna un puntero al contenido de *o*. El puntero se refiere al búfer interno de *o*, que consiste en `len(o) + 1` bytes. 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`.

[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.

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)

Part of the Stable ABI. Return the null-terminated contents of the object *obj* through the output variables *buffer* and *length*. Returns `0` on success.

Si *length* es `NULL`, el objeto bytes no puede contener bytes nulos incrustados; en caso contrario, la función retorna `-1` y se lanza un `ValueError`.

El búfer se refiere a un búfer interno de *obj*, que incluye un byte nulo adicional al final (no está considerado 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.

void **PyBytes_Concat**(*PyObject* **bytes, *PyObject* *newpart)

Part of the Stable ABI. 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)

Part of the Stable ABI. Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*. This version releases the *strong reference* to *newpart* (i.e. decrements its reference count).

PyObject ***PyBytes_Join**(*PyObject* *sep, *PyObject* *iterable)

Similar to `sep.join(iterable)` in Python.

sep must be Python `bytes` object. (Note that `PyUnicode_Join()` accepts `NULL` separator and treats it as a space, whereas `PyBytes_Join()` doesn't accept `NULL` separator.)

iterable must be an iterable object yielding objects that implement the *buffer protocol*.

On success, return a new `bytes` object. On error, set an exception and return `NULL`.

Added in version 3.14.

int **_PyBytes_Resize**(*PyObject* **bytes, *Py_ssize_t* newsize)

Resize a bytes object. *newsize* will be the new length of the bytes object. You can think of it as creating a new bytes object and destroying the old one, only more efficiently. Pass the address of an existing bytes object as an lvalue (it may be written into), and the new size desired. On success, **bytes* holds the resized bytes object and `0` is returned; the address in **bytes* may differ from its input value. If the reallocation fails, the original bytes object at **bytes* is deallocated, **bytes* is set to `NULL`, `MemoryError` is set, and `-1` is returned.

9.3.2 Objetos de arreglos de bytes (*bytearrays*)

type **PyByteArrayObject**

Este subtipo de `PyObject` representa un objeto arreglo de bytes de Python.

PyTypeObject **PyByteArray_Type**

Part of the Stable ABI. 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. Part of the Stable ABI. Retorna un nuevo objeto de arreglo de bytes de cualquier objeto, *o*, que implementa el *buffer protocol*.

Al fallar, retorna `NULL` con una excepción asignada.

PyObject ***PyByteArray_FromStringAndSize**(const char *string, *Py_ssize_t* len)

Return value: New reference. Part of the Stable ABI. Crea un nuevo objeto de arreglo de bytes a partir de *string* y su longitud, *len*.

Al fallar, retorna `NULL` con una excepción asignada.

PyObject ***PyByteArray_Concat**(*PyObject* *a, *PyObject* *b)

Return value: New reference. Part of the Stable ABI. Une los arreglos de bytes (*bytearrays*) *a* y *b* y retorna un nuevo arreglo de bytes (*bytearray*) con el resultado.

Al fallar, retorna `NULL` con una excepción asignada.

Py_ssize_t **PyByteArray_Size**(*PyObject* *bytearray)

Part of the Stable ABI. Retorna el tamaño de *bytearray* después de buscar un puntero `NULL`.

char ***PyByteArray_AsString**(*PyObject* *bytearray)

Part of the Stable ABI. Retorna el contenido de *bytearray* como un arreglo de caracteres después de verificar un puntero `NULL`. El arreglo retornado siempre tiene un byte nulo adicional agregado.

int **PyByteArray_Resize**(*PyObject* *bytearray, *Py_ssize_t* len)

Part of the Stable ABI. Resize the internal buffer of *bytearray* to *len*. Failure is a `-1` return with an exception set.

Distinto en la versión 3.14: A negative *len* will now result in an exception being set and -1 returned.

Macros

Estos macros intercambian seguridad por velocidad y no comprueban punteros.

char ***PyByteArray_AS_STRING**(*PyObject* *bytearray)

Similar a `PyByteArray_AsString()`, pero sin comprobación de errores.

Py_ssize_t **PyByteArray_GET_SIZE**(*PyObject* *bytearray)

Similar a `PyByteArray_Size()`, pero sin comprobación de errores.

9.3.3 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).

UTF-8 representation is created on demand and cached in the Unicode object.

Nota

The *Py_UNICODE* representation has been removed since Python 3.12 with deprecated APIs. See **PEP 623** for more information.

Tipo unicode

Estos son los tipos básicos de objetos Unicode utilizados para la implementación de Unicode en Python:

PyTypeObject **PyUnicode_Type**

Part of the Stable ABI. This instance of *PyTypeObject* represents the Python Unicode type. It is exposed to Python code as `str`.

PyTypeObject **PyUnicodeIter_Type**

Part of the Stable ABI. This instance of *PyTypeObject* represents the Python Unicode iterator type. It is used to iterate over Unicode string objects.

type **Py_UCS4**

type **Py_UCS2**

type **Py_UCS1**

Part of the Stable ABI. 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*.

Added in version 3.3.

type **PyASCIIObject**

type **PyCompactUnicodeObject**

type **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*.

Added in version 3.3.

The following APIs are C macros and static inlined functions for fast checks and access to internal read-only data of Unicode objects:

int **PyUnicode_Check**(*PyObject* *obj)

Return true if the object *obj* is a Unicode object or an instance of a Unicode subtype. This function always succeeds.

int **PyUnicode_CheckExact**(*PyObject* *obj)

Return true if the object *obj* is a Unicode object, but not an instance of a subtype. This function always succeeds.

Py_ssize_t **PyUnicode_GET_LENGTH**(*PyObject* *unicode)

Return the length of the Unicode string, in code points. *unicode* has to be a Unicode object in the «canonical» representation (not checked).

Added in version 3.3.

Py_UCS1 ***PyUnicode_1BYTE_DATA**(*PyObject* *unicode)

Py_UCS2 ***PyUnicode_2BYTE_DATA**(*PyObject* *unicode)

Py_UCS4 ***PyUnicode_4BYTE_DATA**(*PyObject* *unicode)

Return a pointer to the canonical representation cast to UCS1, UCS2 or UCS4 integer types for direct character access. No checks are performed if the canonical representation has the correct character size; use *PyUnicode_KIND()* to select the right function.

Added in version 3.3.

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND

PyUnicode_4BYTE_KIND

Retorna los valores de la macro *PyUnicode_KIND()*.

Added in version 3.3.

Distinto en la versión 3.12: `PyUnicode_WCHAR_KIND` has been removed.

int **PyUnicode_KIND**(*PyObject* *unicode)

Return one of the PyUnicode kind constants (see above) that indicate how many bytes per character this Unicode object uses to store its data. *unicode* has to be a Unicode object in the «canonical» representation (not checked).

Added in version 3.3.

void ***PyUnicode_DATA**(*PyObject* *unicode)

Return a void pointer to the raw Unicode buffer. *unicode* has to be a Unicode object in the «canonical» representation (not checked).

Added in version 3.3.

void **PyUnicode_WRITE**(int kind, void *data, *Py_ssize_t* index, *Py_UCS4* value)

Write the code point *value* to the given zero-based *index* in a string.

The *kind* value and *data* pointer must have been obtained from a string using *PyUnicode_KIND()* and *PyUnicode_DATA()* respectively. You must hold a reference to that string while calling `PyUnicode_WRITE()`. All requirements of *PyUnicode_WriteChar()* also apply.

The function performs no checks for any of its requirements, and is intended for usage in loops.

Added in version 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.

Added in version 3.3.

Py_UCS4 **PyUnicode_READ_CHAR**(*PyObject* *unicode, *Py_ssize_t* index)

Read a character from a Unicode object *unicode*, which must be in the «canonical» representation. This is less efficient than *PyUnicode_READ()* if you do multiple consecutive reads.

Added in version 3.3.

Py_UCS4 **PyUnicode_MAX_CHAR_VALUE**(*PyObject* *unicode)

Return the maximum code point that is suitable for creating another string based on *unicode*, which must be in the «canonical» representation. This is always an approximation but more efficient than iterating over the string.

Added in version 3.3.

int **PyUnicode_IsIdentifier**(*PyObject* *unicode)

Part of the Stable ABI. 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.

unsigned int **PyUnicode_IS_ASCII**(*PyObject* *unicode)

Return true if the string only contains ASCII characters. Equivalent to `str.isascii()`.

Added in version 3.2.

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)

Return `1` or `0` depending on whether *ch* is a printable character, in the sense of `str.isprintable()`.

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.

Py_UCS4 **Py_UNICODE_TOUPPER**(*Py_UCS4* ch)

Retorna el carácter *ch* convertido a mayúsculas.

Py_UCS4 **Py_UNICODE_TOTITLE**(*Py_UCS4* ch)

Retorna el carácter *ch* convertido a formato de título (*titlecase*).

int **Py_UNICODE_TODECIMAL**(*Py_UCS4* ch)

Return the character *ch* converted to a decimal positive integer. Return `-1` if this is not possible. This function does not raise exceptions.

int **Py_UNICODE_TODIGIT**(*Py_UCS4* ch)

Return the character *ch* converted to a single digit integer. Return `-1` if this is not possible. This function does not raise exceptions.

double **Py_UNICODE_TONUMERIC**(*Py_UCS4* ch)

Return the character *ch* converted to a double. Return `-1.0` if this is not possible. This function does not raise exceptions.

Estas API se pueden usar para trabajar con sustitutos:

int **Py_UNICODE_IS_SURROGATE**(*Py_UCS4* ch)

Comprueba si *ch* es un sustituto (`0xD800 <= ch <= 0xDFFF`).

int **Py_UNICODE_IS_HIGH_SURROGATE**(*Py_UCS4* ch)

Comprueba si *ch* es un sustituto alto (`0xD800 <= ch <= 0xDFFF`).

int **Py_UNICODE_IS_LOW_SURROGATE**(*Py_UCS4* ch)

Comprueba si *ch* es un sustituto bajo (`0xD800 <= ch <= 0xDFFF`).

Py_UCS4 **Py_UNICODE_JOIN_SURROGATES**(*Py_UCS4* high, *Py_UCS4* low)

Join two surrogate code points and return a single *Py_UCS4* value. *high* and *low* are respectively the leading and trailing surrogates in a surrogate pair. *high* must be in the range [0xD800; 0xDBFF] and *low* must be in the range [0xDC00; 0xDFFF].

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.

On error, set an exception and return `NULL`.

After creation, the string can be filled by *PyUnicode_WriteChar()*, *PyUnicode_CopyCharacters()*, *PyUnicode_Fill()*, *PyUnicode_WRITE()* or similar. Since strings are supposed to be immutable, take care to not "use" the result while it is being modified. In particular, before it's filled with its final contents, a string:

- must not be hashed,
- must not be *converted to UTF-8*, or another non-«canonical» representation,
- must not have its reference count changed,
- must not be shared with code that might do one of the above.

This list is not exhaustive. Avoiding these uses is your responsibility; Python does not always check these requirements.

To avoid accidentally exposing a partially-written string object, prefer using the *PyUnicodeWriter* API, or one of the `PyUnicode_From*` functions below.

Added in version 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.

Si es necesario, la entrada *buffer* se copia y se transforma en la representación canónica. Por ejemplo, si el *buffer* es una cadena de caracteres UCS4 (*PyUnicode_4BYTE_KIND*) y consta solo de puntos de código en el rango UCS1, se transformará en UCS1 (*PyUnicode_1BYTE_KIND*).

Added in version 3.3.

PyObject ***PyUnicode_FromStringAndSize**(const char *str, *Py_ssize_t* size)

Return value: New reference. Part of the Stable ABI. Create a Unicode object from the char buffer *str*. The bytes will be interpreted as being UTF-8 encoded. The buffer is copied into the new object. The return value might be a shared object, i.e. modification of the data is not allowed.

This function raises `SystemError` when:

- *size* < 0,

- *str* is `NULL` and *size* > 0

Distinto en la versión 3.12: *str* == `NULL` with *size* > 0 is not allowed anymore.

PyObject ***PyUnicode_FromString**(const char *str)

Return value: New reference. Part of the Stable ABI. Create a Unicode object from a UTF-8 encoded null-terminated char buffer *str*.

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

Return value: New reference. Part of the Stable ABI. Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python Unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* ASCII-encoded string.

A conversion specifier contains two or more characters and has the following components, which must occur in this order:

1. The `'%'` character, which marks the start of the specifier.
2. Conversion flags (optional), which affect the result of some conversion types.
3. Minimum field width (optional). If specified as an `'*'` (asterisk), the actual width is given in the next argument, which must be of type `int`, and the object to convert comes after the minimum field width and optional precision.
4. Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'` (an asterisk), the actual precision is given in the next argument, which must be of type `int`, and the value to convert comes after the precision.
5. Length modifier (optional).
6. Conversion type.

The conversion flag characters are:

Flag	Meaning
`0`	The conversion will be zero padded for numeric values.
`-`	The converted value is left adjusted (overrides the `0` flag if both are given).

The length modifiers for following integer conversions (`d`, `i`, `o`, `u`, `x`, or `X`) specify the type of the argument (`int` by default):

Modifier	Types
`l`	`long` or `unsigned long`
`ll`	`long long` or `unsigned long long`
`j`	`intmax_t` or `uintmax_t`
`z`	`size_t` or `ssize_t`
`t`	`ptrdiff_t`

The length modifier `l` for following conversions `s` or `V` specify that the type of the argument is `const wchar_t*`.

The conversion specifiers are:

Conversion Specifier	Tipo	Comentario
`%`	*n/a*	The literal `%` character.
`d`, `i`	Specified by the length modifier	The decimal representation of a signed C integer.
`u`	Specified by the length modifier	The decimal representation of an unsigned C integer.
`o`	Specified by the length modifier	The octal representation of an unsigned C integer.
`x`	Specified by the length modifier	The hexadecimal representation of an unsigned C integer (lowercase).
`X`	Specified by the length modifier	The hexadecimal representation of an unsigned C integer (uppercase).
`c`	`int`	A single character.
`s`	`const char*` or `const wchar_t*`	Un arreglo de caracteres de C terminada en nulo.
`p`	`const void*`	The hex representation of a C pointer. Mostly equivalent to `printf("%p")` except that it is guaranteed to start with the literal `0x` regardless of what the platform's `printf` yields.
`A`	*PyObject*`*`	El resultado de llamar `ascii()`.
`U`	*PyObject*`*`	Un objeto unicode.
`V`	*PyObject*`*`, `const char*` or `const wchar_t*`	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 *PyObject_Str()*.
`R`	*PyObject*`*`	El resultado de llamar *PyObject_Repr()*.
`T`	*PyObject*`*`	Get the fully qualified name of an object type; call *PyType_GetFullyQualifiedName()*.
`#T`	*PyObject*`*`	Similar to `T` format, but use a colon (`:`) as separator between the module name and the qualified name.
`N`	*PyTypeObject*`*`	Get the fully qualified name of a type; call *PyType_GetFullyQualifiedName()*.
`#N`	*PyTypeObject*`*`	Similar to `N` format, but use a colon (`:`) as separator between the module name and the qualified name.

Nota

The width formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes or `wchar_t` items (if the length modifier `l` is used) for `"%s"` and `"%V"` (if the `PyObject*` argument is `NULL`), and a number of characters for `"%A"`, `"%U"`, `"%S"`, `"%R"` and `"%V"` (if the `PyObject*` argument is not `NULL`).

Nota

Unlike to C `printf()` the `0` flag has effect even when a precision is given for integer conversions (`d`, `i`, `u`, `o`, `x`, or `X`).

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"`.

Distinto en la versión 3.12: Support for conversion specifiers `o` and `X`. Support for length modifiers `j` and `t`. Length modifiers are now applied to all integer conversions. Length modifier `l` is now applied to conversion specifiers `s` and `V`. Support for variable width and precision `*`. Support for flag `-`.

An unrecognized format character now sets a `SystemError`. In previous versions it caused all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

Distinto en la versión 3.13: Support for `%T`, `%#T`, `%N` and `%#N` formats added.

PyObject ***PyUnicode_FromFormatV**(const char *format, va_list vargs)

Return value: New reference. Part of the Stable ABI. Idéntico a *`PyUnicode_FromFormat()`* excepto que toma exactamente dos argumentos.

PyObject ***PyUnicode_FromObject**(*PyObject* *obj)

Return value: New reference. Part of the Stable ABI. Copy an instance of a Unicode subtype to a new true Unicode object if necessary. If *obj* is already a true Unicode object (not a subtype), return a new *strong reference* to the object.

Los objetos que no sean Unicode o sus subtipos causarán un `TypeError`.

PyObject ***PyUnicode_FromOrdinal**(int ordinal)

Return value: New reference. Part of the Stable ABI. Create a Unicode Object from the given Unicode code point *ordinal*.

The ordinal must be in `range(0x110000)`. A `ValueError` is raised in the case it is not.

PyObject ***PyUnicode_FromEncodedObject**(*PyObject* *obj, const char *encoding, const char *errors)

Return value: New reference. Part of the Stable ABI. 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.

void **PyUnicode_Append**(*PyObject* **p_left, *PyObject* *right)

Part of the Stable ABI. Append the string *right* to the end of *p_left*. *p_left* must point to a *strong reference* to a Unicode object; `PyUnicode_Append()` releases («steals») this reference.

On error, set **p_left* to `NULL` and set an exception.

On success, set **p_left* to a new strong reference to the result.

void **PyUnicode_AppendAndDel**(*PyObject* **p_left, *PyObject* *right)

Part of the Stable ABI. The function is similar to *`PyUnicode_Append()`*, with the only difference being that it decrements the reference count of *right* by one.

PyObject ***PyUnicode_BuildEncodingMap**(*PyObject* *string)

Return value: New reference. Part of the Stable ABI. Return a mapping suitable for decoding a custom single-byte encoding. Given a Unicode string *string* of up to 256 characters representing an encoding table, returns either a compact internal mapping object or a dictionary mapping character ordinals to byte values. Raises a `TypeError` and return `NULL` on invalid input.

Added in version 3.2.

const char ***PyUnicode_GetDefaultEncoding**(void)

Part of the Stable ABI. Return the name of the default string encoding, `"utf-8"`. See `sys.getdefaultencoding()`.

The returned string does not need to be freed, and is valid until interpreter shutdown.

Py_ssize_t **PyUnicode_GetLength**(*PyObject* *unicode)

Part of the Stable ABI *since version 3.7.* Retorna la longitud del objeto Unicode, en puntos de código.

On error, set an exception and return `-1`.

Added in version 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)

Copy characters from one Unicode object into another. This function performs character conversion when necessary and falls back to `memcpy()` if possible. Returns `-1` and sets an exception on error, otherwise returns the number of copied characters.

The string must not have been "used" yet. See *PyUnicode_New()* for details.

Added in version 3.3.

int **PyUnicode_Resize**(*PyObject* **unicode, *Py_ssize_t* length);

Part of the Stable ABI. Resize a Unicode object **unicode* to the new *length* in code points.

Try to resize the string in place (which is usually faster than allocating a new string and copying characters), or create a new string.

**unicode* is modified to point to the new (resized) object and `0` is returned on success. Otherwise, `-1` is returned and an exception is set, and **unicode* is left untouched.

The function doesn't check string content, the result may not be a string in canonical representation.

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.

The string must not have been "used" yet. See *PyUnicode_New()* for details.

Retorna el número de caracteres escritos o retorna `-1` y lanza una excepción en caso de error.

Added in version 3.3.

int **PyUnicode_WriteChar**(*PyObject* *unicode, *Py_ssize_t* index, *Py_UCS4* character)

Part of the Stable ABI *since version 3.7.* Write a *character* to the string *unicode* at the zero-based *index*. Return `0` on success, `-1` on error with an exception set.

This function checks that *unicode* is a Unicode object, that the index is not out of bounds, and that the object's reference count is one). See *PyUnicode_WRITE()* for a version that skips these checks, making them your responsibility.

The string must not have been "used" yet. See *PyUnicode_New()* for details.

Added in version 3.3.

Py_UCS4 **PyUnicode_ReadChar**(*PyObject* *unicode, *Py_ssize_t* index)

Part of the Stable ABI *since version 3.7.* Read a character from a string. This function checks that *unicode* is a Unicode object and the index is not out of bounds, in contrast to *PyUnicode_READ_CHAR()*, which performs no error checking.

Return character on success, `-1` on error with an exception set.

Added in version 3.3.

PyObject ***PyUnicode_Substring**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* end)

Return value: New reference. Part of the Stable ABI *since version 3.7.* Return a substring of *unicode*, from character index *start* (included) to character index *end* (excluded). Negative indices are not supported. On error, set an exception and return `NULL`.

Added in version 3.3.

Py_UCS4 ***PyUnicode_AsUCS4**(*PyObject* *unicode, *Py_UCS4* *buffer, *Py_ssize_t* buflen, int copy_null)

Part of the Stable ABI *since version 3.7.* Copy the string *unicode* into a UCS4 buffer, including a null character, if *copy_null* is set. Returns `NULL` and sets an exception on error (in particular, a `SystemError` if *buflen* is smaller than the length of *unicode*). *buffer* is returned on success.

Added in version 3.3.

Py_UCS4 ***PyUnicode_AsUCS4Copy**(*PyObject* *unicode)

Part of the Stable ABI *since version 3.7.* Copy the string *unicode* into a new UCS4 buffer that is allocated using `PyMem_Malloc()`. If this fails, `NULL` is returned with a `MemoryError` set. The returned buffer always has an extra null code point appended.

Added in version 3.3.

Codificación regional

La codificación local actual se puede utilizar para decodificar texto del sistema operativo.

PyObject ***PyUnicode_DecodeLocaleAndSize**(const char *str, *Py_ssize_t* length, const char *errors)

Return value: New reference. Part of the Stable ABI *since version 3.7.* 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.

Use `PyUnicode_DecodeFSDefaultAndSize()` to decode a string from the *filesystem encoding and error handler*.

Esta función ignora el modo Python UTF-8.

Ver también

La función `Py_DecodeLocale()`.

Added in version 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. Part of the Stable ABI *since version 3.7.* Similar to `PyUnicode_DecodeLocaleAndSize()`, but compute the string length using `strlen()`.

Added in version 3.3.

PyObject ***PyUnicode_EncodeLocale**(*PyObject* *unicode, const char *errors)

Return value: New reference. Part of the Stable ABI *since version 3.7.* 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 `"subrogateescape"` (**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.

Use `PyUnicode_EncodeFSDefault()` to encode a string to the *filesystem encoding and error handler*.

Esta función ignora el modo Python UTF-8.

Ver también

La función `Py_EncodeLocale()`.

Added in version 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_EncodeLocale()* se usaba para el `subrogateescape`, y la codificación local actual se usaba para `estricto`.

Codificación del sistema de archivos

Functions encoding to and decoding from the *filesystem encoding and error handler* (**PEP 383** and **PEP 529**).

To encode file names to `bytes` during argument parsing, the `"O&"` converter should be used, passing `PyUnicode_FSConverter()` as the conversion function:

int **PyUnicode_FSConverter**(*PyObject* *obj, void *result)

Part of the Stable ABI. *PyArg_Parse* converter*: encode `str` objects – obtained directly or through the `os.PathLike` interface – to `bytes` using *PyUnicode_EncodeFSDefault()*; `bytes` objects are output as-is. *result* must be an address of a C variable of type *PyObject** (or *PyBytesObject**). On success, set the variable to a new *strong reference* to a *bytes object* which must be released when it is no longer used and return a non-zero value (*Py_CLEANUP_SUPPORTED*). Embedded null bytes are not allowed in the result. On failure, return `0` with an exception set.

If *obj* is `NULL`, the function releases a strong reference stored in the variable referred by *result* and returns `1`.

Added in version 3.1.

Distinto en la versión 3.6: Acepta un objeto similar a una ruta (*path-like object*).

To decode file names to `str` during argument parsing, the `"O&"` converter should be used, passing `PyUnicode_FSDecoder()` as the conversion function:

int **PyUnicode_FSDecoder**(*PyObject* *obj, void *result)

Part of the Stable ABI. *PyArg_Parse* converter*: decode `bytes` objects – obtained either directly or indirectly through the `os.PathLike` interface – to `str` using *PyUnicode_DecodeFSDefaultAndSize()*; `str` objects are output as-is. *result* must be an address of a C variable of type *PyObject** (or *PyUnicodeObject**). On success, set the variable to a new *strong reference* to a *Unicode object* which must be released when it is no longer used and return a non-zero value (*Py_CLEANUP_SUPPORTED*). Embedded null characters are not allowed in the result. On failure, return `0` with an exception set.

If *obj* is `NULL`, release the strong reference to the object referred to by *result* and return `1`.

Added in version 3.2.

Distinto en la versión 3.6: Acepta un objeto similar a una ruta (*path-like object*).

PyObject ***PyUnicode_DecodeFSDefaultAndSize**(const char *str, *Py_ssize_t* size)

Return value: New reference. Part of the Stable ABI. Decodifica una cadena desde el *codificador de sistema de archivos y gestor de errores*.

If you need to decode a string from the current locale encoding, use *PyUnicode_DecodeLocaleAndSize()*.

Ver también

La función *Py_DecodeLocale()*.

Distinto en la versión 3.6: The *filesystem error handler* is now used.

PyObject ***PyUnicode_DecodeFSDefault**(const char *str)

Return value: New reference. Part of the Stable ABI. Decodifica una cadena terminada en nulo desde el *codificador de sistema de archivos y gestor de errores*.

If the string length is known, use *PyUnicode_DecodeFSDefaultAndSize()*.

Distinto en la versión 3.6: The *filesystem error handler* is now used.

PyObject ***PyUnicode_EncodeFSDefault**(*PyObject* *unicode)

Return value: New reference. Part of the Stable ABI. Encode a Unicode object to the *filesystem encoding and error handler*, and return `bytes`. Note that the resulting `bytes` object can contain null bytes.

If you need to encode a string to the current locale encoding, use *`PyUnicode_EncodeLocale()`*.

Ver también

La función *`Py_EncodeLocale()`*.

Added in version 3.2.

Distinto en la versión 3.6: The *filesystem error handler* is now used.

soporte wchar_t

`wchar_t` support for platforms which support it:

PyObject ***PyUnicode_FromWideChar**(const wchar_t *wstr, *Py_ssize_t* size)

Return value: New reference. Part of the Stable ABI. Create a Unicode object from the `wchar_t` buffer *wstr* of the given *size*. Passing `-1` as the *size* indicates that the function must itself compute the length, using `wcslen()`. Return `NULL` on failure.

Py_ssize_t ***PyUnicode_AsWideChar**(*PyObject* *unicode, wchar_t *wstr, *Py_ssize_t* size)

Part of the Stable ABI. Copy the Unicode object contents into the `wchar_t` buffer *wstr*. At most *size* `wchar_t` characters are copied (excluding a possibly trailing null termination character). Return the number of `wchar_t` characters copied or `-1` in case of an error.

When *wstr* is `NULL`, instead return the *size* that would be required to store all of *unicode* including a terminating null.

Note that the resulting `wchar_t*` string may or may not be null-terminated. It is the responsibility of the caller to make sure that the `wchar_t*` string is null-terminated in case this is required by the application. Also, note that the `wchar_t*` string might contain null characters, which would cause the string to be truncated when used with most C functions.

wchar_t ***PyUnicode_AsWideCharString**(*PyObject* *unicode, *Py_ssize_t* *size)

Part of the Stable ABI *since version 3.7*. Convert the Unicode object to a wide character string. The output string always ends with a null character. If *size* is not `NULL`, write the number of wide characters (excluding the trailing null termination character) into **size*. Note that the resulting `wchar_t` string might contain null characters, which would cause the string to be truncated when used with most C functions. If *size* is `NULL` and the `wchar_t*` string contains null characters a `ValueError` is raised.

Returns a buffer allocated by *`PyMem_New`* (use *`PyMem_Free()`* to free it) on success. On error, returns `NULL` and **size* is undefined. Raises a `MemoryError` if memory allocation is failed.

Added in version 3.2.

Distinto en la versión 3.7: Raises a `ValueError` if *size* is `NULL` and the `wchar_t*` string contains null characters.

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()`.

Setting encoding to `NULL` causes the default encoding to be used which is UTF-8. The file system calls should use *`PyUnicode_FSConverter()`* for encoding file names. This uses the *filesystem encoding and error handler* internally.

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

The following macro is provided:

`Py_UNICODE_REPLACEMENT_CHARACTER`

The Unicode code point `U+FFFD` (replacement character).

This Unicode character is used as the replacement character during decoding if the *errors* argument is set to «replace».

Estas son las APIs de códecs genéricos:

PyObject ***PyUnicode_Decode**(const char *str, *Py_ssize_t* size, const char *encoding, const char *errors)

Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the encoded string *str*. *encoding* and *errors* have the same meaning as the parameters of the same name in the `str()` built-in function. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsEncodedString**(*PyObject* *unicode, const char *encoding, const char *errors)

Return value: New reference. Part of the Stable ABI. 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.

Códecs UTF-8

Estas son las APIs del códec UTF-8:

PyObject ***PyUnicode_DecodeUTF8**(const char *str, *Py_ssize_t* size, const char *errors)

Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the UTF-8 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeUTF8Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. 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.

The function fails if the string contains surrogate code points (`U+D800` - `U+DFFF`).

const char ***PyUnicode_AsUTF8AndSize**(*PyObject* *unicode, *Py_ssize_t* *size)

Part of the Stable ABI *since version 3.10.* 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.

On error, set an exception, set *size* to `-1` (if it's not NULL) and return `NULL`.

The function fails if the string contains surrogate code points (`U+D800` - `U+DFFF`).

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.

Added in version 3.3.

Distinto en la versión 3.7: El tipo de retorno ahora es `const char *` en lugar de `char *`.

Distinto en la versión 3.10: This function is a part of the *limited API*.

const char ***PyUnicode_AsUTF8**(*PyObject* *unicode)

Como *PyUnicode_AsUTF8AndSize()*, pero no almacena el tamaño.

Advertencia

This function does not have any special behavior for null characters embedded within *unicode*. As a result, strings containing null characters will remain in the returned string, which some C functions might interpret as the end of the string, leading to truncation. If truncation is an issue, it is recommended to use *PyUnicode_AsUTF8AndSize()* instead.

Added in version 3.3.

Distinto en la versión 3.7: El tipo de retorno ahora es `const char *` en lugar de `char *`.

Códecs UTF-32

Estas son las APIs de códecs para UTF-32:

PyObject ***PyUnicode_DecodeUTF32**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

Return value: New reference. Part of the Stable ABI. 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 *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. 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.

Códecs UTF-16

Estas son las APIs de códecs para UTF-16:

PyObject ***PyUnicode_DecodeUTF16**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

Return value: New reference. Part of the Stable ABI. 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 `\ufeff` 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 *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. 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.

Códecs UTF-7

Estas son las APIs del códec UTF-7:

PyObject ***PyUnicode_DecodeUTF7**(const char *str, *Py_ssize_t* size, const char *errors)

Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the UTF-7 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeUTF7Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Return value: New reference. Part of the Stable ABI. 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*.

Códecs Unicode escapado

Estas son las APIs de códecs para Unicode escapado:

PyObject ***PyUnicode_DecodeUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the Unicode-Escape encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsUnicodeEscapeString**(*PyObject* *unicode)

Return value: New reference. Part of the Stable ABI. 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.

Códecs para Unicode escapado en bruto

Estas son las API del códec Unicode escapado en bruto (*Raw Unicode Escape*):

PyObject ***PyUnicode_DecodeRawUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the Raw-Unicode-Escape encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsRawUnicodeEscapeString**(*PyObject* *unicode)

Return value: New reference. Part of the Stable ABI. 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.

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 *str, *Py_ssize_t* size, const char *errors)

Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the Latin-1 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsLatin1String**(*PyObject* *unicode)

Return value: New reference. Part of the Stable ABI. 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.

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 *str, *Py_ssize_t* size, const char *errors)

Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the ASCII encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsASCIIString**(*PyObject* *unicode)

Return value: New reference. Part of the Stable ABI. 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.

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 *str, *Py_ssize_t* length, *PyObject* *mapping, const char *errors)

Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the encoded string *str* using the given *mapping* object. Return `NULL` if an exception was raised by the codec.

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`, `0xFFFE` o `'\ ufffe'`, se tratan como asignaciones indefinidas y causan un error.

PyObject ***PyUnicode_AsCharmapString**(*PyObject* *unicode, *PyObject* *mapping)

Return value: New reference. Part of the Stable ABI. 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.

La siguiente API de códec es especial en que asigna Unicode a Unicode.

PyObject ***PyUnicode_Translate**(*PyObject* *unicode, *PyObject* *table, const char *errors)

Return value: New reference. Part of the Stable ABI. 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).

Mapping tables need only provide the `__getitem__()` interface; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a `LookupError`) are left untouched and are copied as-is.

errors tiene el significado habitual para los códecs. Puede ser `NULL`, lo que indica que debe usar el manejo de errores predeterminado.

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 *str, *Py_ssize_t* size, const char *errors)

Return value: New reference. Part of the Stable ABI *on Windows since version 3.7.* Create a Unicode object by decoding *size* bytes of the MBCS encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeMBCSStateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Return value: New reference. Part of the Stable ABI *on Windows since version 3.7.* 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_DecodeCodePageStateful**(int code_page, const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Return value: New reference. Part of the Stable ABI *on Windows since version 3.7.* Similar to *`PyUnicode_DecodeMBCSStateful()`*, except uses the code page specified by *code_page*.

PyObject ***PyUnicode_AsMBCSString**(*PyObject* *unicode)

Return value: New reference. Part of the Stable ABI *on Windows since version 3.7.* 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. Part of the Stable ABI *on Windows since version 3.7.* Encode the Unicode object using the specified code page and return a Python bytes object. Return `NULL` if an exception was raised by the codec. Use `CP_ACP` code page to get the MBCS encoder.

Added in version 3.3.

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. Part of the Stable ABI. Une dos cadenas de caracteres que dan una nueva cadena de caracteres Unicode.

PyObject ***PyUnicode_Split**(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

Return value: New reference. Part of the Stable ABI. 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.

On error, return `NULL` with an exception set.

Equivalent to `str.split()`.

PyObject ***PyUnicode_RSplit**(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

Return value: New reference. Part of the Stable ABI. Similar to *PyUnicode_Split()*, but splitting will be done beginning at the end of the string.

On error, return `NULL` with an exception set.

Equivalent to `str.rsplit()`.

PyObject ***PyUnicode_Splitlines**(*PyObject* *unicode, int keepends)

Return value: New reference. Part of the Stable ABI. Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If *keepends* is `0`, the Line break characters are not included in the resulting strings.

PyObject ***PyUnicode_Partition**(*PyObject* *unicode, *PyObject* *sep)

Return value: New reference. Part of the Stable ABI. Split a Unicode string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

sep must not be empty.

On error, return `NULL` with an exception set.

Equivalent to `str.partition()`.

PyObject ***PyUnicode_RPartition**(*PyObject* *unicode, *PyObject* *sep)

Return value: New reference. Part of the Stable ABI. Similar to *PyUnicode_Partition()*, but split a Unicode string at the last occurrence of *sep*. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

sep must not be empty.

On error, return `NULL` with an exception set.

Equivalent to `str.rpartition()`.

PyObject ***PyUnicode_Join**(*PyObject* *separator, *PyObject* *seq)

Return value: New reference. Part of the Stable ABI. 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* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Part of the Stable ABI. Return `1` if *substr* matches `unicode[start:end]` at the given tail end (*direction* == `-1` means to do a prefix match, *direction* == `1` a suffix match), `0` otherwise. Return `-1` if an error occurred.

Py_ssize_t **PyUnicode_Find**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Part of the Stable ABI. Return the first position of *substr* in `unicode[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Py_ssize_t **PyUnicode_FindChar**(*PyObject* *unicode, *Py_UCS4* ch, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Part of the Stable ABI *since version 3.7.* Return the first position of the character *ch* in `unicode[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Added in version 3.3.

Distinto en la versión 3.7: *start* and *end* are now adjusted to behave like `unicode[start:end]`.

Py_ssize_t **PyUnicode_Count**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end)

Part of the Stable ABI. Return the number of non-overlapping occurrences of *substr* in `unicode[start:end]`. Return `-1` if an error occurred.

PyObject ***PyUnicode_Replace**(*PyObject* *unicode, *PyObject* *substr, *PyObject* *replstr, *Py_ssize_t* maxcount)

Return value: New reference. Part of the Stable ABI. Replace at most *maxcount* occurrences of *substr* in *unicode* with *replstr* and return the resulting Unicode object. *maxcount* == `-1` means replace all occurrences.

int **PyUnicode_Compare**(*PyObject* *left, *PyObject* *right)

Part of the Stable ABI. 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.

Ver también

The *PyUnicode_Equal()* function.

int **PyUnicode_Equal**(*PyObject* *a, *PyObject* *b)

Part of the Stable ABI *since version 3.14.* Test if two strings are equal:

- Return `1` if *a* is equal to *b*.
- Return `0` if *a* is not equal to *b*.
- Set a `TypeError` exception and return `-1` if *a* or *b* is not a `str` object.

The function always succeeds if *a* and *b* are `str` objects.

The function works for `str` subclasses, but does not honor custom `__eq__()` method.

Ver también

The *PyUnicode_Compare()* function.

Added in version 3.14.

int **PyUnicode_EqualToUTF8AndSize**(*PyObject* *unicode, const char *string, *Py_ssize_t* size)

Part of the Stable ABI *since version 3.13.* Compare a Unicode object with a char buffer which is interpreted as being UTF-8 or ASCII encoded and return true (`1`) if they are equal, or false (`0`) otherwise. If the Unicode object contains surrogate code points (`U+D800` - `U+DFFF`) or the C string is not valid UTF-8, false (`0`) is returned.

Esta función no lanza excepciones.

Added in version 3.13.

int **PyUnicode_EqualToUTF8**(*PyObject* *unicode, const char *string)

Part of the Stable ABI *since version 3.13.* Similar to `PyUnicode_EqualToUTF8AndSize()`, but compute *string* length using `strlen()`. If the Unicode object contains null characters, false (`0`) is returned.

Added in version 3.13.

int **PyUnicode_CompareWithASCIIString**(*PyObject* *unicode, const char *string)

Part of the Stable ABI. Compare a Unicode object, *unicode*, with *string* and return `-1`, `0`, `1` for less than, equal, and greater than, respectively. It is best to pass only ASCII-encoded strings, but the function interprets the input string as ISO-8859-1 if it contains non-ASCII characters.

Esta función no lanza excepciones.

PyObject ***PyUnicode_RichCompare**(*PyObject* *left, *PyObject* *right, int op)

Return value: New reference. Part of the Stable ABI. 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` or `Py_False` for successful comparisons
- `Py_NotImplemented` in case the type combination is unknown

Possible values for *op* are `Py_GT`, `Py_GE`, `Py_EQ`, `Py_NE`, `Py_LT`, and `Py_LE`.

PyObject ***PyUnicode_Format**(*PyObject* *format, *PyObject* *args)

Return value: New reference. Part of the Stable ABI. Retorna un nuevo objeto de cadena de caracteres desde *format* y *args*; esto es análogo al `format % args`.

int **PyUnicode_Contains**(*PyObject* *unicode, *PyObject* *substr)

Part of the Stable ABI. Check whether *substr* is contained in *unicode* and return true or false accordingly.

substr has to coerce to a one element Unicode string. `-1` is returned if there was an error.

void **PyUnicode_InternInPlace**(*PyObject* **p_unicode)

Part of the Stable ABI. Intern the argument `*p_unicode` in place. The argument must be the address of a pointer variable pointing to a Python Unicode string object. If there is an existing interned string that is the same as `*p_unicode`, it sets `*p_unicode` to it (releasing the reference to the old string object and creating a new *strong reference* to the interned string object), otherwise it leaves `*p_unicode` alone and interns it.

(Clarification: even though there is a lot of talk about references, think of this function as reference-neutral. You must own the object you pass in; after the call you no longer own the passed-in reference, but you newly own the result.)

This function never raises an exception. On error, it leaves its argument unchanged without interning it.

Instances of subclasses of `str` may not be interned, that is, `PyUnicode_CheckExact(*p_unicode)` must be true. If it is not, then – as with any other error – the argument is left unchanged.

Note that interned strings are not "immortal". You must keep a reference to the result to benefit from interning.

PyObject ***PyUnicode_InternFromString**(const char *str)

Return value: New reference. Part of the Stable ABI. A combination of `PyUnicode_FromString()` and `PyUnicode_InternInPlace()`, meant for statically allocated strings.

Return a new («owned») reference to either a new Unicode string object that has been interned, or an earlier interned string object with the same value.

Python may keep a reference to the result, or make it *immortal*, preventing it from being garbage-collected promptly. For interning an unbounded number of different strings, such as ones coming from user input, prefer calling `PyUnicode_FromString()` and `PyUnicode_InternInPlace()` directly.

unsigned int **PyUnicode_CHECK_INTERNED**(*PyObject* *str)

Return a non-zero value if *str* is interned, zero if not. The *str* argument must be a string; this is not checked. This function always succeeds.

Detalles de implementación de CPython: A non-zero return value may carry additional information about *how* the string is interned. The meaning of such non-zero values, as well as each specific string's intern-related details, may change between CPython versions.

PyUnicodeWriter

The *PyUnicodeWriter* API can be used to create a Python `str` object.

Added in version 3.14.

type **PyUnicodeWriter**

A Unicode writer instance.

The instance must be destroyed by *PyUnicodeWriter_Finish()* on success, or *PyUnicodeWriter_Discard()* on error.

PyUnicodeWriter ***PyUnicodeWriter_Create**(*Py_ssize_t* length)

Create a Unicode writer instance.

length must be greater than or equal to `0`.

If *length* is greater than `0`, preallocate an internal buffer of *length* characters.

Set an exception and return `NULL` on error.

PyObject ***PyUnicodeWriter_Finish**(*PyUnicodeWriter* *writer)

Return the final Python `str` object and destroy the writer instance.

Set an exception and return `NULL` on error.

The writer instance is invalid after this call.

void **PyUnicodeWriter_Discard**(*PyUnicodeWriter* *writer)

Discard the internal Unicode buffer and destroy the writer instance.

If *writer* is `NULL`, no operation is performed.

The writer instance is invalid after this call.

int **PyUnicodeWriter_WriteChar**(*PyUnicodeWriter* *writer, *Py_UCS4* ch)

Write the single Unicode character *ch* into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteUTF8**(*PyUnicodeWriter* *writer, const char *str, *Py_ssize_t* size)

Decode the string *str* from UTF-8 in strict mode and write the output into *writer*.

size is the string length in bytes. If *size* is equal to `-1`, call `strlen(str)` to get the string length.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

See also *PyUnicodeWriter_DecodeUTF8Stateful()*.

int **PyUnicodeWriter_WriteASCII**(*PyUnicodeWriter* *writer, const char *str, *Py_ssize_t* size)

Write the ASCII string *str* into *writer*.

size is the string length in bytes. If *size* is equal to `-1`, call `strlen(str)` to get the string length.

str must only contain ASCII characters. The behavior is undefined if *str* contains non-ASCII characters.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

Added in version 3.14.

int **PyUnicodeWriter_WriteWideChar**(*PyUnicodeWriter* *writer, const wchar_t *str, *Py_ssize_t* size)

Write the wide string *str* into *writer*.

size is a number of wide characters. If *size* is equal to `-1`, call `wcslen(str)` to get the string length.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteUCS4**(*PyUnicodeWriter* *writer, *Py_UCS4* *str, *Py_ssize_t* size)

Writer the UCS4 string *str* into *writer*.

size is a number of UCS4 characters.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteStr**(*PyUnicodeWriter* *writer, *PyObject* *obj)

Call *PyObject_Str()* on *obj* and write the output into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteRepr**(*PyUnicodeWriter* *writer, *PyObject* *obj)

Call *PyObject_Repr()* on *obj* and write the output into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteSubstring**(*PyUnicodeWriter* *writer, *PyObject* *str, *Py_ssize_t* start, *Py_ssize_t* end)

Write the substring `str[start:end]` into *writer*.

str must be Python `str` object. *start* must be greater than or equal to 0, and less than or equal to *end*. *end* must be less than or equal to *str* length.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_Format**(*PyUnicodeWriter* *writer, const char *format, ...)

Similar to *PyUnicode_FromFormat()*, but write the output directly into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_DecodeUTF8Stateful**(*PyUnicodeWriter* *writer, const char *string, *Py_ssize_t* length, const char *errors, *Py_ssize_t* *consumed)

Decode the string *str* from UTF-8 with *errors* error handler and write the output into *writer*.

size is the string length in bytes. If *size* is equal to `-1`, call `strlen(str)` to get the string length.

errors is an error handler name, such as `"replace"`. If *errors* is `NULL`, use the strict error handler.

If *consumed* is not `NULL`, set **consumed* to the number of decoded bytes on success. If *consumed* is `NULL`, treat trailing incomplete UTF-8 byte sequences as an error.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

See also *PyUnicodeWriter_WriteUTF8()*.

Deprecated API

The following API is deprecated.

type **Py_UNICODE**

This is a typedef of `wchar_t`, which is a 16-bit type or 32-bit type depending on the platform. Please use `wchar_t` directly instead.

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.

Deprecated since version 3.13, will be removed in version 3.15.

int **PyUnicode_READY**(*PyObject* *unicode)

Do nothing and return `0`. This API is kept only for backward compatibility, but there are no plans to remove it.

Added in version 3.3.

Obsoleto desde la versión 3.10: This API does nothing since Python 3.12. Previously, this needed to be called for each string created using the old API (`PyUnicode_FromUnicode()` or similar).

unsigned int **PyUnicode_IS_READY**(*PyObject* *unicode)

Do nothing and return `1`. This API is kept only for backward compatibility, but there are no plans to remove it.

Added in version 3.3.

Obsoleto desde la versión 3.14: This API does nothing since Python 3.12. Previously, this could be called to check if *PyUnicode_READY()* is necessary.

9.3.4 Objetos tupla

type **PyTupleObject**

Este subtipo de *PyObject* representa un objeto tupla de Python.

PyTypeObject **PyTuple_Type**

Part of the Stable ABI. 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. Part of the Stable ABI. Return a new tuple object of size *len*, or `NULL` with an exception set on failure.

PyObject ***PyTuple_Pack**(*Py_ssize_t* n, ...)

Return value: New reference. Part of the Stable ABI. Return a new tuple object of size *n*, or `NULL` with an exception set on failure. The tuple values are initialized to the subsequent *n* C arguments pointing to Python objects. `PyTuple_Pack(2, a, b)` is equivalent to `Py_BuildValue("(OO)", a, b)`.

Py_ssize_t **PyTuple_Size**(*PyObject* *p)

Part of the Stable ABI. Take a pointer to a tuple object, and return the size of that tuple. On error, return `-1` and with an exception set.

Py_ssize_t **PyTuple_GET_SIZE**(*PyObject* *p)

Like *PyTuple_Size()*, but without error checking.

PyObject ***PyTuple_GetItem**(*PyObject* *p, *Py_ssize_t* pos)

Return value: Borrowed reference. Part of the Stable ABI. Retorna el objeto en la posición *pos* en la tupla señalada por *p*. Si *pos* es negativo o está fuera de los límites, retorna `NULL` y establece una excepción `IndexError`.

The returned reference is borrowed from the tuple *p* (that is: it is only valid as long as you hold a reference to *p*). To get a *strong reference*, use *Py_NewRef(PyTuple_GetItem(...))* or *PySequence_GetItem()*.

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. Part of the Stable ABI. Return the slice of the tuple pointed to by *p* between *low* and *high*, or `NULL` with an exception set on failure.

This is the equivalent of the Python expression `p[low:high]`. Indexing from the end of the tuple is not supported.

int **PyTuple_SetItem**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Part of the Stable ABI. 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.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

> **Nota**
>
> Esta función «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.

> **Advertencia**
>
> This macro should *only* be used on tuples that are newly created. Using this macro on a tuple that is already in use (or in other words, has a refcount > 1) could lead to undefined behavior.

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`.

9.3.5 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.

PyTypeObject ***PyStructSequence_NewType**(*PyStructSequence_Desc* *desc)

Return value: New reference. Part of the Stable ABI. 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()*.

Return `NULL` with an exception set on failure.

void **PyStructSequence_InitType**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

Inicializa una secuencia de estructura tipo *type* desde *desc* en su lugar.

int **PyStructSequence_InitType2**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

Like `PyStructSequence_InitType()`, but returns `0` on success and `-1` with an exception set on failure.

Added in version 3.4.

type **PyStructSequence_Desc**

Part of the Stable ABI *(including all members).* Contiene la meta información de un tipo de secuencia de estructura para crear.

const char ***name**

Fully qualified name of the type; null-terminated UTF-8 encoded. The name must contain the module name.

const char ***doc**

Puntero al *docstring* para el tipo o `NULL` para omitir

PyStructSequence_Field ***fields**

Puntero al arreglo terminado en `NULL` con nombres de campo del nuevo tipo

int **n_in_sequence**

Cantidad de campos visibles para el lado de Python (si se usa como tupla)

type **PyStructSequence_Field**

Part of the Stable ABI *(including all members).* 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.

const char ***name**

Nombre para el campo o `NULL` para finalizar la lista de campos con nombre, establece en `PyStructSequence_UnnamedField` para dejar sin nombre

const char ***doc**

Campo *docstring* o `NULL` para omitir

const char *const **PyStructSequence_UnnamedField**

Part of the Stable ABI *since version 3.11.* 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**(*PyTypeObject* *type)

Return value: New reference. Part of the Stable ABI. Crea una instancia de *type*, que debe haberse creado con `PyStructSequence_NewType()`.

Return `NULL` with an exception set on failure.

PyObject ***PyStructSequence_GetItem**(*PyObject* *p, *Py_ssize_t* pos)

Return value: Borrowed reference. Part of the Stable ABI. Return the object at position *pos* in the struct sequence pointed to by *p*.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

PyObject ***PyStructSequence_GET_ITEM**(*PyObject* *p, *Py_ssize_t* pos)

Return value: Borrowed reference. Alias to `PyStructSequence_GetItem()`.

Distinto en la versión 3.13: Now implemented as an alias to `PyStructSequence_GetItem()`.

void **PyStructSequence_SetItem**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Part of the Stable ABI. 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.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

> **Nota**
>
> Esta función «roba» una referencia a *o*.

void **PyStructSequence_SET_ITEM**(*PyObject* *p, *Py_ssize_t* *pos, *PyObject* *o)

Alias to `PyStructSequence_SetItem()`.

Distinto en la versión 3.13: Now implemented as an alias to `PyStructSequence_SetItem()`.

9.3.6 Objetos lista

type **PyListObject**

Este subtipo de `PyObject` representa un objeto lista de Python.

PyTypeObject **PyList_Type**

Part of the Stable ABI. Esta instancia de `PyTypeObject` representa el tipo 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 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. Part of the Stable ABI. 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()` o `PyList_SET_ITEM()`. Las siguientes APIs son APIs seguras antes de que la lista esté completamente inicializada: `PyList_SetItem()` y `PyList_SET_ITEM()`.

Py_ssize_t **PyList_Size**(*PyObject* *list)

Part of the Stable ABI. 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)

Similar a `PyList_Size()`, pero sin comprobación de errores.

PyObject ***PyList_GetItemRef**(*PyObject* *list, *Py_ssize_t* index)

Return value: New reference. Part of the Stable ABI *since version 3.13.* 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 >= len(list)*), retorna `NULL` y establece una excepción `IndexError`.

Added in version 3.13.

PyObject ***PyList_GetItem**(*PyObject* *list, *Py_ssize_t* index)

Return value: Borrowed reference. Part of the Stable ABI. Como `PyList_GetItemRef()`, pero retorna una *referencia prestada* en lugar de una *referencia fuerte*.

PyObject ***PyList_GET_ITEM**(*PyObject* *list, *Py_ssize_t* i)

Return value: Borrowed reference. Similar a `PyList_GetItem()`, pero sin comprobación de errores.

int **PyList_SetItem**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

Part of the Stable ABI. 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.

La verificación de límites se realiza como una aserción si Python se compila en modo de depuración o `con aserciones`.

> **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)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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. Part of the Stable ABI. 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)

Part of the Stable ABI. 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_Extend**(*PyObject* *list, *PyObject* *iterable)

Extiende *list* con los contenidos de *iterable*. Esto es lo mismo que `PyList_SetSlice(list, PY_SSIZE_T_MAX, PY_SSIZE_T_MAX, iterable)` y análogo a `list.extend(iterable)` o `list += iterable`.

Lanza una excepción y retorna `-1` si *list* no es un objeto `list`. Retorna 0 en caso de éxito.

Added in version 3.13.

int **PyList_Clear**(*PyObject* *list)

Elimina todos los elementos de *list*. Esto es lo mismo que `PyList_SetSlice(list, 0, PY_SSIZE_T_MAX, NULL)` y análogo a `list.clear()` o `del list[:]`.

Lanza una excepción y retorna `-1` si *list* no es un objeto `list`. Retorna 0 en caso de éxito.

Added in version 3.13.

int **PyList_Sort**(*PyObject* *list)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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. Part of the Stable ABI. Retorna un nuevo objeto tupla que contiene el contenido de *list*; equivalente a `tuple(list)`.

9.4 Objetos contenedor

9.4.1 Objetos diccionario

type **PyDictObject**

Este subtipo de *PyObject* representa un objeto diccionario de Python.

PyTypeObject **PyDict_Type**

Part of the Stable ABI. 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. Part of the Stable ABI. Retorna un nuevo diccionario vacío, o `NULL` en caso de falla.

PyObject ***PyDictProxy_New**(*PyObject* *mapping)

Return value: New reference. Part of the Stable ABI. 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)

Part of the Stable ABI. Vacía un diccionario existente de todos los pares clave-valor (*key*-*value*).

int **PyDict_Contains**(*PyObject* *p, *PyObject* *key)

Part of the Stable ABI. 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`.

int **PyDict_ContainsString**(*PyObject* *p, const char *key)

This is the same as *PyDict_Contains()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

Added in version 3.13.

PyObject ***PyDict_Copy**(*PyObject* *p)

Return value: New reference. Part of the Stable ABI. Retorna un nuevo diccionario que contiene los mismos pares clave-valor (*key*-*value*) que *p*.

int **PyDict_SetItem**(*PyObject* *p, *PyObject* *key, *PyObject* *val)

Part of the Stable ABI. 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)

Part of the Stable ABI. This is the same as *PyDict_SetItem()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

int **PyDict_DelItem**(*PyObject* *p, *PyObject* *key)

Part of the Stable ABI. Remove the entry in dictionary *p* with key *key*. *key* must be *hashable*; if it isn't, `TypeError` is raised. If *key* is not in the dictionary, `KeyError` is raised. Return `0` on success or `-1` on failure.

int **PyDict_DelItemString**(*PyObject* *p, const char *key)

Part of the Stable ABI. This is the same as *PyDict_DelItem()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

int **PyDict_GetItemRef**(*PyObject* *p, *PyObject* *key, *PyObject* **result)

Part of the Stable ABI *since version 3.13*. Return a new *strong reference* to the object from dictionary *p* which has a key *key*:

- If the key is present, set **result* to a new *strong reference* to the value and return `1`.
- If the key is missing, set **result* to `NULL` and return `0`.
- On error, raise an exception and return `-1`.

Added in version 3.13.

See also the *PyObject_GetItem()* function.

PyObject ***PyDict_GetItem**(*PyObject* *p, *PyObject* *key)

Return value: Borrowed reference. Part of the Stable ABI. Return a *borrowed reference* to the object from dictionary *p* which has a key *key*. Return `NULL` if the key *key* is missing *without* setting an exception.

> **Nota**
>
> Exceptions that occur while this calls `__hash__()` and `__eq__()` methods are silently ignored. Prefer the *PyDict_GetItemWithError()* function instead.

Distinto en la versión 3.10: Calling this API without an *attached thread state* had been allowed for historical reason. It is no longer allowed.

PyObject ***PyDict_GetItemWithError**(*PyObject* *p, *PyObject* *key)

Return value: Borrowed reference. Part of the Stable ABI. 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. Part of the Stable ABI. This is the same as *PyDict_GetItem()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

> **Nota**
>
> Exceptions that occur while this calls `__hash__()` and `__eq__()` methods or while creating the temporary `str` object are silently ignored. Prefer using the *PyDict_GetItemWithError()* function with your own *PyUnicode_FromString()* *key* instead.

int **PyDict_GetItemStringRef**(*PyObject* *p, const char *key, *PyObject* **result)

Part of the Stable ABI *since version 3.13.* Similar to `PyDict_GetItemRef()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Added in version 3.13.

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.

Added in version 3.4.

int **PyDict_SetDefaultRef**(*PyObject* *p, *PyObject* *key, *PyObject* *default_value, *PyObject* **result)

Inserts *default_value* into the dictionary *p* with a key of *key* if the key is not already present in the dictionary. If *result* is not `NULL`, then **result* is set to a *strong reference* to either *default_value*, if the key was not present, or the existing value, if *key* was already present in the dictionary. Returns `1` if the key was present and *default_value* was not inserted, or `0` if the key was not present and *default_value* was inserted. On failure, returns `-1`, sets an exception, and sets `*result` to `NULL`.

For clarity: if you have a strong reference to *default_value* before calling this function, then after it returns, you hold a strong reference to both *default_value* and **result* (if it's not `NULL`). These may refer to the same object: in that case you hold two separate references to it.

Added in version 3.13.

int **PyDict_Pop**(*PyObject* *p, *PyObject* *key, *PyObject* **result)

Remove *key* from dictionary *p* and optionally return the removed value. Do not raise `KeyError` if the key missing.

- If the key is present, set **result* to a new reference to the removed value if *result* is not `NULL`, and return `1`.
- If the key is missing, set **result* to `NULL` if *result* is not `NULL`, and return `0`.
- On error, raise an exception and return `-1`.

Similar to `dict.pop()`, but without the default value and not raising `KeyError` if the key missing.

Added in version 3.13.

int **PyDict_PopString**(*PyObject* *p, const char *key, *PyObject* **result)

Similar to `PyDict_Pop()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Added in version 3.13.

PyObject ***PyDict_Items**(*PyObject* *p)

Return value: New reference. Part of the Stable ABI. Retorna un `PyListObject` que contiene todos los elementos del diccionario.

PyObject ***PyDict_Keys**(*PyObject* *p)

Return value: New reference. Part of the Stable ABI. Retorna un `PyListObject` que contiene todas las claves del diccionario.

PyObject ***PyDict_Values**(*PyObject* *p)

Return value: New reference. Part of the Stable ABI. Retorna un `PyListObject` que contiene todos los valores del diccionario *p*.

Py_ssize_t **PyDict_Size**(*PyObject* *p)

Part of the Stable ABI. 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)

Part of the Stable ABI. Iterate over all key-value pairs in the dictionary *p*. The `Py_ssize_t` referred to by *ppos* must be initialized to `0` prior to the first call to this function to start the iteration; the function returns true for each pair in the dictionary, and false once all pairs have been reported. The parameters *pkey* and *pvalue* should either point to `PyObject*` variables that will be filled in with each key and value, respectively, or may be `NULL`. Any references returned through them are borrowed. *ppos* should not be altered during iteration. Its value represents offsets within the internal dictionary structure, and since the structure is sparse, the offsets are not consecutive.

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;

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);
}
```

The function is not thread-safe in the *free-threaded* build without external synchronization. You can use `Py_BEGIN_CRITICAL_SECTION` to lock the dictionary while iterating over it:

```
Py_BEGIN_CRITICAL_SECTION(self->dict);
while (PyDict_Next(self->dict, &pos, &key, &value)) {
    ...
}
Py_END_CRITICAL_SECTION();
```

Nota

On the free-threaded build, this function can be used safely inside a critical section. However, the references returned for *pkey* and *pvalue* are *borrowed* and are only valid while the critical section is held. If you need to use these objects outside the critical section or when the critical section can be suspended, create a *strong reference* (for example, using `Py_NewRef()`).

int **PyDict_Merge**(*PyObject* *a, *PyObject* *b, int override)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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
```

int **PyDict_AddWatcher**(*PyDict_WatchCallback* callback)

Register *callback* as a dictionary watcher. Return a non-negative integer id which must be passed to future calls to *PyDict_Watch()*. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.12.

int **PyDict_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from *PyDict_AddWatcher()*. Return `0` on success, `-1` on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.12.

int **PyDict_Watch**(int watcher_id, *PyObject* *dict)

Mark dictionary *dict* as watched. The callback granted *watcher_id* by *PyDict_AddWatcher()* will be called when *dict* is modified or deallocated. Return `0` on success or `-1` on error.

Added in version 3.12.

int **PyDict_Unwatch**(int watcher_id, *PyObject* *dict)

Mark dictionary *dict* as no longer watched. The callback granted *watcher_id* by *PyDict_AddWatcher()* will no longer be called when *dict* is modified or deallocated. The dict must previously have been watched by this watcher. Return `0` on success or `-1` on error.

Added in version 3.12.

type **PyDict_WatchEvent**

Enumeration of possible dictionary watcher events: `PyDict_EVENT_ADDED`, `PyDict_EVENT_MODIFIED`, `PyDict_EVENT_DELETED`, `PyDict_EVENT_CLONED`, `PyDict_EVENT_CLEARED`, or `PyDict_EVENT_DEALLOCATED`.

Added in version 3.12.

typedef int (***PyDict_WatchCallback**)(*PyDict_WatchEvent* event, *PyObject* *dict, *PyObject* *key, *PyObject* *new_value)

Type of a dict watcher callback function.

If *event* is `PyDict_EVENT_CLEARED` or `PyDict_EVENT_DEALLOCATED`, both *key* and *new_value* will be `NULL`. If *event* is `PyDict_EVENT_ADDED` or `PyDict_EVENT_MODIFIED`, *new_value* will be the new value for *key*. If *event* is `PyDict_EVENT_DELETED`, *key* is being deleted from the dictionary and *new_value* will be `NULL`.

`PyDict_EVENT_CLONED` occurs when *dict* was previously empty and another dict is merged into it. To maintain efficiency of this operation, per-key `PyDict_EVENT_ADDED` events are not issued in this case; instead a single `PyDict_EVENT_CLONED` is issued, and *key* will be the source dictionary.

The callback may inspect but must not modify *dict*; doing so could have unpredictable effects, including infinite recursion. Do not trigger Python code execution in the callback, as it could modify the dict as a side effect.

If *event* is `PyDict_EVENT_DEALLOCATED`, taking a new reference in the callback to the about-to-be-destroyed dictionary will resurrect it and prevent it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

Callbacks occur before the notified modification to *dict* takes place, so the prior state of *dict* can be inspected.

If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using *PyErr_WriteUnraisable()*. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.12.

9.4.2 Objetos conjunto

Esta sección detalla la API pública de los objetos `set` y `frozenset`. Cualquier funcionalidad que no esté listada a continuación se accede mejor utilizando el protocolo abstracto de objetos (incluyendo *PyObject_CallMethod()*, *PyObject_RichCompareBool()*, *PyObject_Hash()*, *PyObject_Repr()*, *PyObject_IsTrue()*, *PyObject_Print()*, y *PyObject_GetIter()*) o el protocolo numérico abstracto (incluyendo *PyNumber_And()*, *PyNumber_Subtract()*, *PyNumber_Or()*, *PyNumber_Xor()*, *PyNumber_InPlaceAnd()*, *PyNumber_InPlaceSubtract()*, *PyNumber_InPlaceOr()*, y *PyNumber_InPlaceXor()*).

type **PySetObject**

Este subtipo de *PyObject* se utiliza para mantener los datos internos de los objetos `set` y `frozenset`. Es como un *PyDictObject* en el sentido de que tiene un tamaño fijo para los conjuntos pequeños (muy parecido al almacenamiento de tuplas) y apuntará a un bloque de memoria separado y de tamaño variable para los conjuntos de tamaño medio y grande (muy parecido al almacenamiento de listas). Ninguno de los campos de esta estructura debe considerarse público y todos están sujetos a cambios. Todo el acceso debe hacerse a través de la API documentada en lugar de manipular los valores de la estructura.

PyTypeObject **PySet_Type**

Part of the Stable ABI. Esta es una instancia de *PyTypeObject* que representa el tipo Python `set`.

PyTypeObject **PyFrozenSet_Type**

Part of the Stable ABI. Esta es una instancia de *PyTypeObject* 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 **PySet_CheckExact**(*PyObject* *p)

Retorna verdadero si *p* es un objeto `set` pero no una instancia de un subtipo. Esta función siempre finaliza con éxito.

Added in version 3.10.

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. Part of the Stable ABI. 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. Part of the Stable ABI. 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)

Part of the Stable ABI. Return the length of a `set` or `frozenset` object. Equivalent to `len(anyset)`. Raises a `SystemError` if *anyset* is not a `set`, `frozenset`, or an instance of a subtype.

Py_ssize_t **PySet_GET_SIZE**(*PyObject* *anyset)

Forma macro de *PySet_Size()* sin comprobación de errores.

int **PySet_Contains**(*PyObject* *anyset, *PyObject* *key)

Part of the Stable ABI. Return `1` if found, `0` if not found, and `-1` if an error is encountered. Unlike the Python `__contains__()` method, this function does not automatically convert unhashable sets into temporary frozensets. Raise a `TypeError` if the *key* is unhashable. Raise `SystemError` if *anyset* is not a `set`, `frozenset`, or an instance of a subtype.

int **PySet_Add**(*PyObject* *set, *PyObject* *key)

Part of the Stable ABI. Añade *key* a una instancia `set`. También funciona con instancias de `frozenset` (al igual que *PyTuple_SetItem()* puede usarse para rellenar los valores de nuevos frozensets antes de que sean expuestos a otro código). Retorna `0` en caso de éxito o `-1` en caso de fallo. Lanza un error `TypeError` si la *key* no se puede intercambiar. Lanza un `MemoryError` si no hay espacio para crecer. Lanza un `SystemError` si *set* no es una instancia de `set` o su subtipo.

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)

Part of the Stable ABI. Return `1` if found and removed, `0` if not found (no action taken), and `-1` if an error is encountered. Does not raise `KeyError` for missing keys. Raise a `TypeError` if the *key* is unhashable. Unlike the Python `discard()` method, this function does not automatically convert unhashable sets into temporary frozensets. Raise `SystemError` if *set* is not an instance of `set` or its subtype.

PyObject ***PySet_Pop**(*PyObject* *set)

Return value: New reference. Part of the Stable ABI. 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)

Part of the Stable ABI. Empty an existing set of all elements. Return `0` on success. Return `-1` and raise `SystemError` if *set* is not an instance of `set` or its subtype.

9.5 Objetos de función

9.5.1 Objetos función

Hay algunas funciones específicas para las funciones de Python.

type **PyFunctionObject**

La estructura C utilizada para las funciones.

PyTypeObject **PyFunction_Type**

Esta es una instancia de *PyTypeObject* 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.

The function's docstring and name are retrieved from the code object. `__module__` is retrieved from *globals*. The argument defaults, annotations and closure are set to `NULL`. `__qualname__` is set to the same value as the code object's `co_qualname` field.

PyObject ***PyFunction_NewWithQualName**(*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)

Return value: New reference. As *PyFunction_New()*, but also allows setting the function object's `__qualname__` attribute. *qualname* should be a unicode object or `NULL`; if `NULL`, the `__qualname__` attribute is set to the same value as the code object's `co_qualname` field.

Added in version 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. Return a *borrowed reference* to the `__module__` attribute of the function object *op*. It can be *NULL*.

This is normally a `string` containing the module name, but can be set to any other object by Python code.

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.

void **PyFunction_SetVectorcall**(*PyFunctionObject* *func, *vectorcallfunc* vectorcall)

Set the vectorcall field of a given function object *func*.

Warning: extensions using this API must preserve the behavior of the unaltered (default) vectorcall function!

Added in version 3.12.

PyObject ***PyFunction_GetKwDefaults**(*PyObject* *op)

Return value: Borrowed reference. Return the keyword-only argument default values of the function object *op*. This can be a dictionary of arguments or `NULL`.

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.

PyObject ***PyFunction_GET_CODE**(*PyObject* *op)

PyObject ***PyFunction_GET_GLOBALS**(*PyObject* *op)

PyObject ***PyFunction_GET_MODULE**(*PyObject* *op)

PyObject ***PyFunction_GET_DEFAULTS**(*PyObject* *op)

PyObject ***PyFunction_GET_KW_DEFAULTS**(*PyObject* *op)

PyObject ***PyFunction_GET_CLOSURE**(*PyObject* *op)

PyObject ***PyFunction_GET_ANNOTATIONS**(*PyObject* *op)

Return value: Borrowed reference. These functions are similar to their `PyFunction_Get*` counterparts, but do not do type checking. Passing anything other than an instance of *`PyFunction_Type`* is undefined behavior.

int **PyFunction_AddWatcher**(*PyFunction_WatchCallback* callback)

Register *callback* as a function watcher for the current interpreter. Return an ID which may be passed to *`PyFunction_ClearWatcher()`*. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.12.

int **PyFunction_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from *`PyFunction_AddWatcher()`* for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.12.

type **PyFunction_WatchEvent**

Enumeration of possible function watcher events:

- `PyFunction_EVENT_CREATE`
- `PyFunction_EVENT_DESTROY`
- `PyFunction_EVENT_MODIFY_CODE`
- `PyFunction_EVENT_MODIFY_DEFAULTS`
- `PyFunction_EVENT_MODIFY_KWDEFAULTS`

Added in version 3.12.

typedef int (***PyFunction_WatchCallback**)(*PyFunction_WatchEvent* event, *PyFunctionObject* *func, *PyObject* *new_value)

Type of a function watcher callback function.

If *event* is `PyFunction_EVENT_CREATE` or `PyFunction_EVENT_DESTROY` then *new_value* will be `NULL`. Otherwise, *new_value* will hold a *borrowed reference* to the new value that is about to be stored in *func* for the attribute that is being modified.

The callback may inspect but must not modify *func*; doing so could have unpredictable effects, including infinite recursion.

If *event* is `PyFunction_EVENT_CREATE`, then the callback is invoked after *func* has been fully initialized. Otherwise, the callback is invoked before the modification to *func* takes place, so the prior state of *func* can be inspected. The runtime is permitted to optimize away the creation of function objects when possible. In such cases no event will be emitted. Although this creates the possibility of an observable difference of runtime behavior depending on optimization decisions, it does not change the semantics of the Python code being executed.

If *event* is `PyFunction_EVENT_DESTROY`, Taking a reference in the callback to the about-to-be-destroyed function will resurrect it, preventing it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using *PyErr_WriteUnraisable()*. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.12.

9.5.2 Objetos de método de instancia

An instance method is a wrapper for a *PyCFunction* and the new way to bind a *PyCFunction* to a class object. It replaces the former call `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.

9.5.3 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.

PyTypeObject **PyMethod_Type**

Esta instancia de *PyTypeObject* 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.

9.5.4 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.

type **PyCellObject**

La estructura C utilizada para objetos celda.

PyTypeObject **PyCell_Type**

El objeto tipo correspondiente a los objetos celda.

int **PyCell_Check**(*PyObject* *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. Return the contents of the cell *cell*, which can be `NULL`. If *cell* is not a cell object, returns `NULL` with an exception set.

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)

Set the contents of the cell object *cell* to *value*. This releases the reference to any current content of the cell. *value* may be `NULL`. *cell* must be non-`NULL`.

On success, return `0`. If *cell* is not a cell object, set an exception and return `-1`.

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.

9.5.5 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.

type **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**

This is an instance of `PyTypeObject` representing the Python code object.

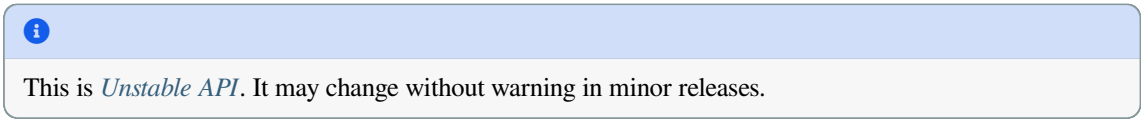
int **PyCode_Check**(*PyObject* *co)

Return true if *co* is a code object. This function always succeeds.

Py_ssize_t **PyCode_GetNumFree**(*PyCodeObject* *co)

Return the number of *free (closure) variables* in a code object.

int **PyUnstable_Code_GetFirstFree**(*PyCodeObject* *co)

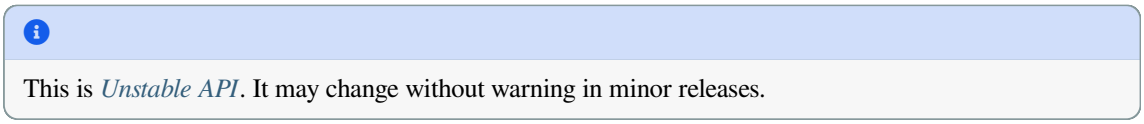


This is *Unstable API*. It may change without warning in minor releases.

Return the position of the first *free (closure) variable* in a code object.

Distinto en la versión 3.13: Renamed from `PyCode_GetFirstFree` as part of *Unstable C API*. The old name is deprecated, but will remain available until the signature changes again.

PyCodeObject ***PyUnstable_Code_New**(int argcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)



This is *Unstable API*. It may change without warning in minor releases.

Return a new code object. If you need a dummy code object to create a frame, use `PyCode_NewEmpty()` instead.

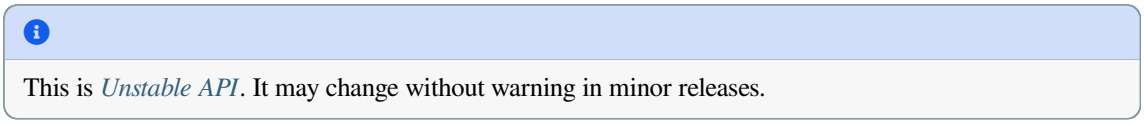
Since the definition of the bytecode changes often, calling `PyUnstable_Code_New()` directly can bind you to a precise Python version.

The many arguments of this function are inter-dependent in complex ways, meaning that subtle changes to values are likely to result in incorrect execution or VM crashes. Use this function only with extreme care.

Distinto en la versión 3.11: Added `qualname` and `exceptiontable` parameters.

Distinto en la versión 3.12: Renamed from `PyCode_New` as part of *Unstable C API*. The old name is deprecated, but will remain available until the signature changes again.

PyCodeObject ***PyUnstable_Code_NewWithPosOnlyArgs**(int argcount, int posonlyargcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)



This is *Unstable API*. It may change without warning in minor releases.

Similar to *PyUnstable_Code_New()*, but with an extra «posonlyargcount» for positional-only arguments. The same caveats that apply to PyUnstable_Code_New also apply to this function.

Added in version 3.8: as PyCode_NewWithPosOnlyArgs

Distinto en la versión 3.11: Added qualname and exceptiontable parameters.

Distinto en la versión 3.12: Renamed to PyUnstable_Code_NewWithPosOnlyArgs. The old name is deprecated, but will remain available until the signature changes again.

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. Si el objeto código resultante es ejecutado, lanzará una Exception.

int **PyCode_Addr2Line**(*PyCodeObject* *co, int byte_offset)

Retorna el número de línea de la instrucción que se produce en o antes de byte_offset y finaliza después. Si solo necesita el número de línea de un marco, use *PyFrame_GetLineNumber()* en su lugar.

For efficiently iterating over the line numbers in a code object, use **the API described in PEP 626**.

int **PyCode_Addr2Location**(*PyObject* *co, int byte_offset, int *start_line, int *start_column, int *end_line, int *end_column)

Establece los punteros int pasados en los números de línea y columna del código fuente para las instrucciones en byte_offset. Establece el valor en 0 cuando la información no está disponible para algún elemento en particular.

Retorna 1 si la función fue exitosa y 0 de lo contrario.

Added in version 3.11.

PyObject ***PyCode_GetCode**(*PyCodeObject* *co)

Equivalente al código Python getattr(co, 'co_code'). Retorna una referencia fuerte a un *PyBytesObject* representando el bytecode en un objecto código. En caso de error se retorna NULL y se lanza una excepción.

Este PyBytesObject puede ser creado a pedido del intérprete y no necesariamente representa el bytecode que es realmente ejecutado por CPython. Los casos de uso principales para esta función son depuradores y perfiladores.

Added in version 3.11.

PyObject ***PyCode_GetVarnames**(*PyCodeObject* *co)

Equivalente al código Python getattr(co, 'co_varnames'). Retorna una nueva referencia a un *PyTupleObject* que contiene los nombres de las variables locales. En caso de error, retorna NULL y lanza una excepción.

Added in version 3.11.

PyObject ***PyCode_GetCellvars**(*PyCodeObject* *co)

Equivalente al código Python getattr(co, 'co_cellvars'). Retorna una nueva referencia a un *PyTupleObject* que contiene los nombres de las variables locales referenciadas por funciones anidadas. En caso de error, retorna NULL y lanza una excepción.

Added in version 3.11.

PyObject ***PyCode_GetFreevars**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_freevars')`. Returns a new reference to a `PyTupleObject` containing the names of the *free (closure) variables*. On error, `NULL` is returned and an exception is raised.

Added in version 3.11.

int **PyCode_AddWatcher**(*PyCode_WatchCallback* callback)

Register *callback* as a code object watcher for the current interpreter. Return an ID which may be passed to `PyCode_ClearWatcher()`. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.12.

int **PyCode_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from `PyCode_AddWatcher()` for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.12.

type **PyCodeEvent**

Enumeration of possible code object watcher events: - `PY_CODE_EVENT_CREATE` - `PY_CODE_EVENT_DESTROY`

Added in version 3.12.

typedef int (***PyCode_WatchCallback**)(*PyCodeEvent* event, *PyCodeObject* *co)

Type of a code object watcher callback function.

If *event* is `PY_CODE_EVENT_CREATE`, then the callback is invoked after *co* has been fully initialized. Otherwise, the callback is invoked before the destruction of *co* takes place, so the prior state of *co* can be inspected.

If *event* is `PY_CODE_EVENT_DESTROY`, taking a reference in the callback to the about-to-be-destroyed code object will resurrect it and prevent it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

Users of this API should not rely on internal runtime implementation details. Such details may include, but are not limited to, the exact order and timing of creation and destruction of code objects. While changes in these details may result in differences observable by watchers (including whether a callback is invoked or not), it does not change the semantics of the Python code being executed.

If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using `PyErr_WriteUnraisable()`. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.12.

9.5.6 Code Object Flags

Code objects contain a bit-field of flags, which can be retrieved as the `co_flags` Python attribute (for example using `PyObject_GetAttrString()`), and set using a *flags* argument to `PyUnstable_Code_New()` and similar functions.

Flags whose names start with `CO_FUTURE_` correspond to features normally selectable by future statements. These flags can be used in `PyCompilerFlags.cf_flags`. Note that many `CO_FUTURE_` flags are mandatory in current versions of Python, and setting them has no effect.

The following flags are available. For their meaning, see the linked documentation of their Python equivalents.

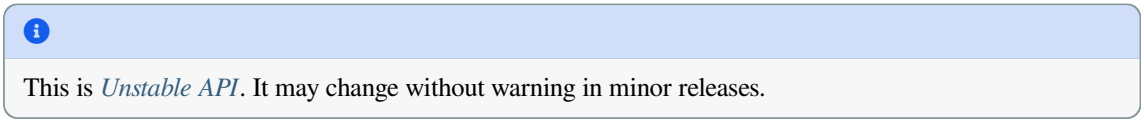
Flag	Meaning
CO_OPTIMIZED	inspect.CO_OPTIMIZED
CO_NEWLOCALS	inspect.CO_NEWLOCALS
CO_VARARGS	inspect.CO_VARARGS
CO_VARKEYWORDS	inspect.CO_VARKEYWORDS
CO_NESTED	inspect.CO_NESTED
CO_GENERATOR	inspect.CO_GENERATOR
CO_COROUTINE	inspect.CO_COROUTINE
CO_ITERABLE_COROUTINE	inspect.CO_ITERABLE_COROUTINE
CO_ASYNC_GENERATOR	inspect.CO_ASYNC_GENERATOR
CO_HAS_DOCSTRING	inspect.CO_HAS_DOCSTRING
CO_METHOD	inspect.CO_METHOD
CO_FUTURE_DIVISION	no effect (__future__.division)
CO_FUTURE_ABSOLUTE_IMPORT	no effect (__future__.absolute_import)
CO_FUTURE_WITH_STATEMENT	no effect (__future__.with_statement)
CO_FUTURE_PRINT_FUNCTION	no effect (__future__.print_function)
CO_FUTURE_UNICODE_LITERALS	no effect (__future__.unicode_literals)
CO_FUTURE_GENERATOR_STOP	no effect (__future__.generator_stop)
CO_FUTURE_ANNOTATIONS	__future__.annotations

9.5.7 Extra information

To support low-level extensions to frame evaluation, such as external just-in-time compilers, it is possible to attach arbitrary extra data to code objects.

These functions are part of the unstable C API tier: this functionality is a CPython implementation detail, and the API may change without deprecation warnings.

Py_ssize_t **PyUnstable_Eval_RequestCodeExtraIndex**(*freefunc* free)



This is *Unstable API*. It may change without warning in minor releases.

Return a new an opaque index value used to adding data to code objects.

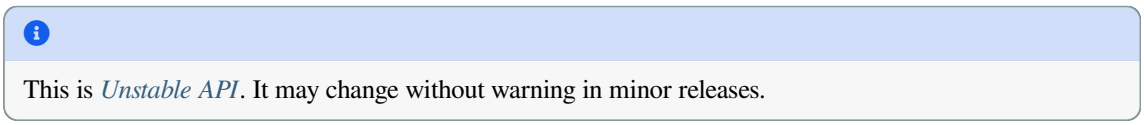
You generally call this function once (per interpreter) and use the result with `PyCode_GetExtra` and `PyCode_SetExtra` to manipulate data on individual code objects.

If *free* is not `NULL`: when a code object is deallocated, *free* will be called on non-`NULL` data stored under the new index. Use `Py_DecRef()` when storing `PyObject`.

Added in version 3.6: as `_PyEval_RequestCodeExtraIndex`

Distinto en la versión 3.12: Renamed to `PyUnstable_Eval_RequestCodeExtraIndex`. The old private name is deprecated, but will be available until the API changes.

int **PyUnstable_Code_GetExtra**(*PyObject* *code, *Py_ssize_t* index, void **extra)



This is *Unstable API*. It may change without warning in minor releases.

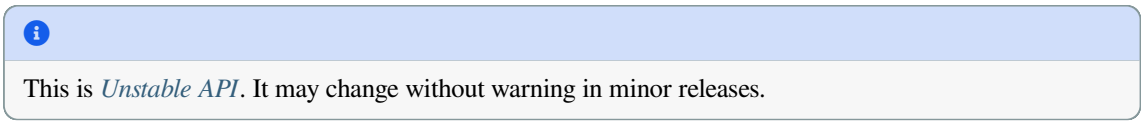
Set *extra* to the extra data stored under the given index. Return 0 on success. Set an exception and return -1 on failure.

If no data was set under the index, set *extra* to `NULL` and return 0 without setting an exception.

Added in version 3.6: as `_PyCode_GetExtra`

Distinto en la versión 3.12: Renamed to `PyUnstable_Code_GetExtra`. The old private name is deprecated, but will be available until the API changes.

int **PyUnstable_Code_SetExtra**(*PyObject* *code, *Py_ssize_t* index, void *extra)



This is *Unstable API*. It may change without warning in minor releases.

Set the extra data stored under the given index to *extra*. Return 0 on success. Set an exception and return -1 on failure.

Added in version 3.6: as `_PyCode_SetExtra`

Distinto en la versión 3.12: Renamed to `PyUnstable_Code_SetExtra`. The old private name is deprecated, but will be available until the API changes.

9.6 Otros objetos

9.6.1 Objetos archivo

These APIs are a minimal emulation of the Python 2 C API for built-in file objects, which used to rely on the buffered I/O (`FILE*`) support from the C standard library. In Python 3, files and streams use the new `io` module, which defines several layers over the low-level unbuffered I/O of the operating system. The functions described below are convenience C wrappers over these new APIs, and meant mostly for internal error reporting in the interpreter; third-party code is advised to access the `io` APIs instead.

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. Part of the Stable ABI. 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)

Part of the Stable ABI. Return the file descriptor associated with *p* as an `int`. If the object is an integer, its value is returned. If not, the object's `fileno()` method is called if it exists; the method must return an integer, which is returned as the file descriptor value. Sets an exception and returns `-1` on failure.

PyObject ***PyFile_GetLine**(*PyObject* *p, int n)

Return value: New reference. Part of the Stable ABI. 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.

The *handler* is a function of type:

typedef *PyObject* *(***Py_OpenCodeHookFunction**)(*PyObject**, void*)

Equivalent of `PyObject *(*)(PyObject *path, void *userData)`, where *path* is guaranteed to be `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.

Added in version 3.8.

int **PyFile_WriteObject**(*PyObject* *obj, *PyObject* *p, int flags)

Part of the Stable ABI. Write object *obj* to file object *p*. The only supported flag for *flags* is `Py_PRINT_RAW`; if given, the `str()` of the object is written instead of the `repr()`. Return `0` on success or `-1` on failure; the appropriate exception will be set.

int **PyFile_WriteString**(const char *s, *PyObject* *p)

Part of the Stable ABI. 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.

9.6.2 Objetos módulo

PyTypeObject **PyModule_Type**

Part of the Stable ABI. Esta instancia de `PyTypeObject` 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. Part of the Stable ABI *since version 3.7.* Return a new module object with `module.__name__` set to *name*. The module's `__name__`, `__doc__`, `__package__` and `__loader__` attributes are filled in (all but `__name__` are set to `None`). The caller is responsible for setting a `__file__` attribute.

Return `NULL` with an exception set on error.

Added in version 3.3.

Distinto en la versión 3.4: `__package__` and `__loader__` are now set to `None`.

PyObject ***PyModule_New**(const char *name)

Return value: New reference. Part of the Stable ABI. 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. Part of the Stable ABI. 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), se lanza `SystemError` y se retorna `NULL`.

It is recommended extensions use other `PyModule_*` and `PyObject_*` functions rather than directly manipulate a module's `__dict__`.

PyObject ***PyModule_GetNameObject**(*PyObject* *module)

Return value: New reference. Part of the Stable ABI *since version 3.7.* Return *module*'s `__name__` value. If the module does not provide one, or if it is not a string, `SystemError` is raised and `NULL` is returned.

Added in version 3.3.

const char ***PyModule_GetName**(*PyObject* *module)

Part of the Stable ABI. Similar a `PyModule_GetNameObject()` pero retorna el nombre codificado a `'utf-8'`.

void *__PyModule_GetState__(*PyObject* *module)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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. Part of the Stable ABI. Return the name of the file from which *module* was loaded using *module*'s `__file__` attribute. If this is not defined, or if it is not a string, raise `SystemError` and return `NULL`; otherwise return a reference to a Unicode object.

Added in version 3.2.

const char *__PyModule_GetFilename__(*PyObject* *module)

Part of the Stable ABI. Similar a *PyModule_GetFilenameObject()* pero retorna el nombre de archivo codificado a "utf-8".

Obsoleto desde la versión 3.2: *PyModule_GetFilename()* raises `UnicodeEncodeError` on unencodable filenames, use *PyModule_GetFilenameObject()* instead.

9.6.3 Module definitions

The functions in the previous section work on any module object, including modules imported from Python code.

Modules defined using the C API typically use a *module definition*, *PyModuleDef* – a statically allocated, constant "description» of how a module should be created.

The definition is usually used to define an extension's "main" module object (see *Defining extension modules* for details). It is also used to *create extension modules dynamically*.

Unlike *PyModule_New()*, the definition allows management of *module state* – a piece of memory that is allocated and cleared together with the module object. Unlike the module's Python attributes, Python code cannot replace or delete data stored in module state.

type __PyModuleDef__

Part of the Stable ABI *(including all members).* The module definition struct, which holds all information needed to create a module object. This structure must be statically allocated (or be otherwise guaranteed to be valid while any modules created from it exist). Usually, there is only one variable of this type for each extension module.

PyModuleDef_Base __m_base__

Always initialize this member to `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.

This memory area is allocated based on *m_size* on module creation, and freed when the module object is deallocated, after the *m_free* function has been called, if present.

Setting it to a non-negative value means that the module can be re-initialized and specifies the additional amount of memory it requires for its state.

Setting `m_size` to `-1` means that the module does not support sub-interpreters, because it has global state. Negative `m_size` is only allowed when using *legacy single-phase initialization* or when *creating modules dynamically*.

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**

An array of slot definitions for multi-phase initialization, terminated by a `{0, NULL}` entry. When using legacy single-phase initialization, *m_slots* must be `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.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (*Py_mod_exec* function). More precisely, this function is not called if *m_size* is greater than 0 and the module state (as returned by *PyModule_GetState()*) is `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.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (*Py_mod_exec* function). More precisely, this function is not called if *m_size* is greater than 0 and the module state (as returned by *PyModule_GetState()*) is `NULL`.

Tal como *PyTypeObject.tp_clear*, esta función no *siempre* es llamada antes de la designación de un módulo. Por ejemplo, cuando el recuento de referencias está listo para determinar que un objeto no se usa más, la recolección de basura cíclica no se involucra y se llama a *m_free* directamente.

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.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (*Py_mod_exec* function). More precisely, this function is not called if *m_size* is greater than 0 and the module state (as returned by *PyModule_GetState()*) is `NULL`.

Distinto en la versión 3.9: Ya no se llama antes de que se asigne el estado del módulo.

Module slots

type **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.

Added in version 3.5.

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*.

Py_mod_multiple_interpreters

Specifies one of the following values:

Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED

The module does not support being imported in subinterpreters.

Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED

The module supports being imported in subinterpreters, but only when they share the main interpreter's GIL. (See isolating-extensions-howto.)

Py_MOD_PER_INTERPRETER_GIL_SUPPORTED

The module supports being imported in subinterpreters, even when they have their own GIL. (See isolating-extensions-howto.)

This slot determines whether or not importing this module in a subinterpreter will fail.

Multiple `Py_mod_multiple_interpreters` slots may not be specified in one module definition.

If `Py_mod_multiple_interpreters` is not specified, the import machinery defaults to `Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED`.

Added in version 3.12.

Py_mod_gil

Specifies one of the following values:

Py_MOD_GIL_USED

The module depends on the presence of the global interpreter lock (GIL), and may access global state without synchronization.

Py_MOD_GIL_NOT_USED

The module is safe to run without an active GIL.

This slot is ignored by Python builds not configured with `--disable-gil`. Otherwise, it determines whether or not importing this module will cause the GIL to be automatically enabled. See whatsnew313-free-threaded-cpython for more detail.

Multiple `Py_mod_gil` slots may not be specified in one module definition.

If `Py_mod_gil` is not specified, the import machinery defaults to `Py_MOD_GIL_USED`.

Added in version 3.13.

9.6.4 Creating extension modules dynamically

The following functions may be used to create a module outside of an extension's *initialization function*. They are also used in *single-phase initialization*.

PyObject ***PyModule_Create**(*PyModuleDef* *def)

Return value: New reference. Create a new module object, given the definition in *def*. This is a macro that calls `PyModule_Create2()` with *module_api_version* set to `PYTHON_API_VERSION`, or to `PYTHON_ABI_VERSION` if using the *limited API*.

PyObject ***PyModule_Create2**(*PyModuleDef* *def, int module_api_version)

Return value: New reference. Part of the Stable ABI. 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`.

Return `NULL` with an exception set on error.

This function does not support slots. The `m_slots` member of *def* must be `NULL`.

> **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.

PyObject ***PyModule_FromDefAndSpec**(*PyModuleDef* *def, *PyObject* *spec)

Return value: New reference. This macro calls `PyModule_FromDefAndSpec2()` with *module_api_version* set to `PYTHON_API_VERSION`, or to `PYTHON_ABI_VERSION` if using the *limited API*.

Added in version 3.5.

PyObject ***PyModule_FromDefAndSpec2**(*PyModuleDef* *def, *PyObject* *spec, int module_api_version)

Return value: New reference. Part of the Stable ABI *since version 3.7.* Create a new module object, given the definition in *def* and the ModuleSpec *spec*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

Return `NULL` with an exception set on error.

Note that this does not process execution slots (`Py_mod_exec`). Both `PyModule_FromDefAndSpec` and `PyModule_ExecDef` must be called to fully initialize a module.

> **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.

Added in version 3.5.

int **PyModule_ExecDef**(*PyObject* *module, *PyModuleDef* *def)

Part of the Stable ABI *since version 3.7.* Procesa cualquier ranura de ejecución (`Py_mod_exec`) dado en *def*.

Added in version 3.5.

PYTHON_API_VERSION

The C API version. Defined for backwards compatibility.

Currently, this constant is not updated in new Python versions, and is not useful for versioning. This may change in the future.

PYTHON_ABI_VERSION

Defined as `3` for backwards compatibility.

Currently, this constant is not updated in new Python versions, and is not useful for versioning. This may change in the future.

9.6.5 Funciones de soporte

The following functions are provided to help initialize a module state. They are intended for a module's execution slots (`Py_mod_exec`), the initialization function for legacy *single-phase initialization*, or code that creates modules dynamically.

int **PyModule_AddObjectRef**(*PyObject* *module, const char *name, *PyObject* *value)

Part of the Stable ABI *since version 3.10.* Agrega un objeto a *module* como *name*. Esta es una función de conveniencia que se puede usar desde la función de inicialización del módulo.

En caso de éxito, retorna `0`. En caso de error, lanza una excepción y retorna `-1`.

Ejemplo de uso

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    if (obj == NULL) {
        return -1;
    }
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_DECREF(obj);
    return res;
}
```

To be convenient, the function accepts `NULL` *value* with an exception set. In this case, return `-1` and just leave the raised exception unchanged.

El ejemplo puede también ser escrito sin verificar explicitamente si *obj* es `NULL`:

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_XDECREF(obj);
    return res;
}
```

Note que `Py_XDECREF()` debería ser usado en vez de `Py_DECREF()` en este caso, ya que *obj* puede ser `NULL`.

The number of different *name* strings passed to this function should be kept small, usually by only using statically allocated strings as *name*. For names that aren't known at compile time, prefer calling `PyUnicode_FromString()` and `PyObject_SetAttr()` directly. For more details, see `PyUnicode_InternFromString()`, which may be used internally to create a key object.

Added in version 3.10.

int **PyModule_Add**(*PyObject* *module, const char *name, *PyObject* *value)

Part of the Stable ABI *since version 3.13*. Similar to `PyModule_AddObjectRef()`, but «steals» a reference to *value*. It can be called with a result of function that returns a new reference without bothering to check its result or even saving it to a variable.

Ejemplo de uso

```
if (PyModule_Add(module, "spam", PyBytes_FromString(value)) < 0) {
    goto error;
}
```

Added in version 3.13.

int **PyModule_AddObject**(*PyObject* *module, const char *name, *PyObject* *value)

Part of the Stable ABI. Similar a `PyModule_AddObjectRef()`, pero roba una referencia a *value* en caso de éxito (en este caso retorna `0`).

The new `PyModule_Add()` or `PyModule_AddObjectRef()` functions are recommended, since it is easy to introduce reference leaks by misusing the `PyModule_AddObject()` function.

Nota

Unlike other functions that steal references, `PyModule_AddObject()` only releases the reference to *value* **on success**.

This means that its return value must be checked, and calling code must `Py_XDECREF()` *value* manually on error.

Ejemplo de uso

```
PyObject *obj = PyBytes_FromString(value);
if (PyModule_AddObject(module, "spam", obj) < 0) {
    // If 'obj' is not NULL and PyModule_AddObject() failed,
    // 'obj' strong reference must be deleted with Py_XDECREF().
    // If 'obj' is NULL, Py_XDECREF() does nothing.
    Py_XDECREF(obj);
    goto error;
}
// PyModule_AddObject() stole a reference to obj:
// Py_XDECREF(obj) is not needed here.
```

Obsoleto desde la versión 3.13: `PyModule_AddObject()` is *soft deprecated*.

int **PyModule_AddIntConstant**(*PyObject* *module, const char *name, long value)

Part of the Stable ABI. Add an integer constant to *module* as *name*. This convenience function can be used from the module's initialization function. Return `-1` with an exception set on error, `0` on success.

This is a convenience function that calls `PyLong_FromLong()` and `PyModule_AddObjectRef()`; see their documentation for details.

int **PyModule_AddStringConstant**(*PyObject* *module, const char *name, const char *value)

Part of the Stable ABI. Add a string constant to *module* as *name*. This convenience function can be used from the module's initialization function. The string *value* must be `NULL`-terminated. Return `-1` with an exception set on error, `0` on success.

This is a convenience function that calls `PyUnicode_InternFromString()` and `PyModule_AddObjectRef()`; see their documentation for details.

PyModule_AddIntMacro(module, macro)

Add an int constant to *module*. The name and the value are taken from *macro*. For example `PyModule_AddIntMacro(module, AF_INET)` adds the int constant *AF_INET* with the value of *AF_INET* to *module*. Return `-1` with an exception set on error, `0` on success.

PyModule_AddStringMacro(module, macro)

Agrega una constante de cadena de caracteres a *module*.

int **PyModule_AddType**(*PyObject* *module, *PyTypeObject* *type)

Part of the Stable ABI *since version 3.10.* Add a type object to *module*. The type object is finalized by calling internally `PyType_Ready()`. The name of the type object is taken from the last component of `tp_name` after dot. Return `-1` with an exception set on error, `0` on success.

Added in version 3.9.

int **PyModule_AddFunctions**(*PyObject* *module, *PyMethodDef* *functions)

Part of the Stable ABI *since version 3.7.* Add the functions from the `NULL` terminated *functions* array to *module*. Refer to the `PyMethodDef` documentation for details on individual entries (due to the lack of a shared module namespace, module level «functions» implemented in C typically receive the module as their first parameter, making them similar to instance methods on Python classes).

This function is called automatically when creating a module from `PyModuleDef` (such as when using *Multi-phase initialization*, `PyModule_Create`, or `PyModule_FromDefAndSpec`). Some module authors may prefer defining functions in multiple `PyMethodDef` arrays; in that case they should call this function directly.

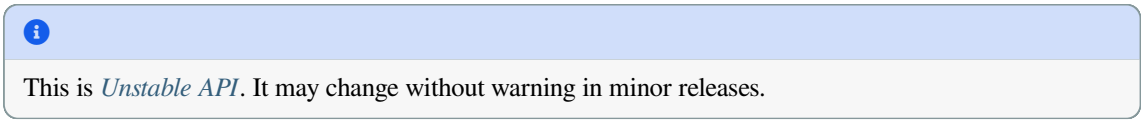
Added in version 3.5.

int **PyModule_SetDocString**(*PyObject* *module, const char *docstring)

Part of the Stable ABI *since version 3.7.* Set the docstring for *module* to *docstring*. This function is called automatically when creating a module from `PyModuleDef` (such as when using *Multi-phase initialization*, `PyModule_Create`, or `PyModule_FromDefAndSpec`).

Added in version 3.5.

int **PyUnstable_Module_SetGIL**(*PyObject* *module, void *gil)



This is *Unstable API*. It may change without warning in minor releases.

Indicate that *module* does or does not support running without the global interpreter lock (GIL), using one of the values from `Py_mod_gil`. It must be called during *module*'s initialization function when using *Legacy single-phase initialization*. If this function is not called during module initialization, the import machinery assumes the module does not support running without the GIL. This function is only available in Python builds configured with `--disable-gil`. Return `-1` with an exception set on error, `0` on success.

Added in version 3.13.

Module lookup (single-phase initialization)

The legacy *single-phase initialization* initialization scheme creates singleton modules that can be looked up in the context of the current interpreter. This allows the module object to be retrieved later with only a reference to the module definition.

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. Part of the Stable ABI. 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)

Part of the Stable ABI *since version 3.3.* 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 calls `PyState_AddModule` automatically after importing a module that uses *single-phase initialization*, so it is unnecessary (but harmless) to call it from module initialization code. An explicit call is needed only if the module's own init code subsequently calls `PyState_FindModule`. The function is mainly intended for implementing alternative import mechanisms (either by calling it directly, or by referring to its implementation for details of the required state updates).

If a module was attached previously using the same *def*, it is replaced by the new *module*.

The caller must have an *attached thread state*.

Return `-1` with an exception set on error, `0` on success.

Added in version 3.3.

int **PyState_RemoveModule**(*PyModuleDef* *def)

Part of the Stable ABI *since version 3.3.* Removes the module object created from *def* from the interpreter state. Return `-1` with an exception set on error, `0` on success.

The caller must have an *attached thread state*.

Added in version 3.3.

9.6.6 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 en la secuencia y finalizando la iteración cuando se retorna el valor centinela.

PyTypeObject **PySeqIter_Type**

Part of the Stable ABI. 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**(*PyObject* *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. Part of the Stable ABI. 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.

PyTypeObject **PyCallIter_Type**

Part of the Stable ABI. Objeto tipo para los objetos iteradores retornados por `PyCallIter_New()` y la forma de dos argumentos de la función incorporada `iter()`.

int **PyCallIter_Check**(*PyObject* *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. Part of the Stable ABI. 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.

9.6.7 Objetos descriptores

Los «descriptores» son objetos que describen algún atributo de un objeto. Se encuentran en el diccionario de objetos tipo.

PyTypeObject **PyProperty_Type**

Part of the Stable ABI. El objeto de tipo para los tipos de descriptor incorporado.

PyObject ***PyDescr_NewGetSet**(*PyTypeObject* *type, struct *PyGetSetDef* *getset)

Return value: New reference. Part of the Stable ABI.

PyObject ***PyDescr_NewMember**(*PyTypeObject* *type, struct *PyMemberDef* *meth)

Return value: New reference. Part of the Stable ABI.

PyObject ***PyDescr_NewMethod**(*PyTypeObject* *type, struct *PyMethodDef* *meth)

Return value: New reference. Part of the Stable ABI.

PyObject ***PyDescr_NewWrapper**(*PyTypeObject* *type, struct wrapperbase *wrapper, void *wrapped)

Return value: New reference.

PyObject ***PyDescr_NewClassMethod**(*PyTypeObject* *type, *PyMethodDef* *method)

Return value: New reference. Part of the Stable ABI.

int **PyDescr_IsData**(*PyObject* *descr)

Retorna distinto de cero si el descriptor objetos *descr* describe un atributo de datos, o `0` si describe un método. *descr* debe ser un objeto descriptor; no hay comprobación de errores.

PyObject ***PyWrapper_New**(*PyObject**, *PyObject**)

Return value: New reference. Part of the Stable ABI.

9.6.8 Objetos rebanada (*slice*)

PyTypeObject **PySlice_Type**

Part of the Stable ABI. 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 funciona correctamente.

PyObject ***PySlice_New**(*PyObject* *start, *PyObject* *stop, *PyObject* *step)

Return value: New reference. Part of the Stable ABI. 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` con una excepción establecida si no se ha podido 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)

Part of the Stable ABI. 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 una 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)

Part of the Stable ABI. 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 una 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) {
    // retorna un error
}
```

es reemplazado por:

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {
    // retorna un 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 se 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)

Part of the Stable ABI *since version 3.7.* 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` con una excepción establecida en caso de error, `0` en caso de éxito.

Added in version 3.6.1.

Py_ssize_t **PySlice_AdjustIndices**(*Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* step)

Part of the Stable ABI *since version 3.7.* 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.

Added in version 3.6.1.

Objeto elipsis

PyTypeObject **PyEllipsis_Type**

Part of the Stable ABI. The type of Python `Ellipsis` object. Same as `types.EllipsisType` in the Python layer.

PyObject ***Py_Ellipsis**

El objeto `Ellipsis` de Python. Este objeto no tiene métodos. Al igual que *Py_None*, es un objeto singleton *immortal*.

Distinto en la versión 3.12: *Py_Ellipsis* es inmortal.

9.6.9 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. Part of the Stable ABI. 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.

PyBUF_READ

Indicador para solicitar un búfer de solo lectura.

PyBUF_WRITE

Indicador para solicitar un búfer escribible.

PyObject ***PyMemoryView_FromMemory**(char *mem, *Py_ssize_t* size, int flags)

Return value: New reference. Part of the Stable ABI *since version 3.7.* Crea un objeto de vista de memoria usando *mem* como el búfer subyacente. *flags* pueden ser uno de *PyBUF_READ* o *PyBUF_WRITE*.

Added in version 3.3.

PyObject ***PyMemoryView_FromBuffer**(const *Py_buffer* *view)

Return value: New reference. Part of the Stable ABI *since version 3.11.* 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. Part of the Stable ABI. 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.

buffertype puede ser uno de *PyBUF_READ* o *PyBUF_WRITE*.

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.

PyObject ***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*.

9.6.10 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**(*PyObject* *ob)

Retorna un valor distinto de cero si *ob* es una referencia o un objeto proxy. Esta función siempre finaliza con éxito.

int **PyWeakref_CheckRef**(*PyObject* *ob)

Retorna un valor distinto de cero si *ob* es un objeto de referencia. Esta función siempre finaliza con éxito.

int **PyWeakref_CheckProxy**(*PyObject* *ob)

Retorna un valor distinto de cero si *ob* es un objeto proxy. Esta función siempre finaliza con éxito.

PyObject ***PyWeakref_NewRef**(*PyObject* *ob, *PyObject* *callback)

Return value: New reference. Part of the Stable ABI. 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 lanzará un `TypeError`.

PyObject ***PyWeakref_NewProxy**(*PyObject* *ob, *PyObject* *callback)

Return value: New reference. Part of the Stable ABI. 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 lanzará un `TypeError`.

int **PyWeakref_GetRef**(*PyObject* *ref, *PyObject* **pobj)

Part of the Stable ABI *since version 3.13*. Obtiene un *strong reference* al objeto referenciado desde una referencia débil, *ref*, en **pobj*.

- En caso de éxito, asigna **pobj* en un nuevo *strong reference* al objeto referenciado y retorna 1.
- Si la referencia está inactiva, asigna **pobj* a `NULL` y retorna 0.
- En caso de error, lanza una excepción y devuelve -1.

Added in version 3.13.

PyObject ***PyWeakref_GetObject**(*PyObject* *ref)

Return value: Borrowed reference. Part of the Stable ABI. Retorna un *borrowed reference* del objeto referenciado desde una referencia débil, *ref*. Si el referente ya no está activo, retorna `Py_None`.

> **Nota**
>
> Esta función retorna una referencia *borrowed reference* al objeto referenciado. Esto significa que siempre debe llamar a *Py_INCREF()* sobre el objeto excepto cuando no pueda ser destruido antes del último uso de la referencia prestada.

Deprecated since version 3.13, will be removed in version 3.15: Utiliza *PyWeakref_GetRef()* en su lugar.

PyObject ***PyWeakref_GET_OBJECT**(*PyObject* *ref)

Return value: Borrowed reference. Similar a *PyWeakref_GetObject()*, pero no realiza ninguna comprobación de errores.

Deprecated since version 3.13, will be removed in version 3.15: Utiliza *PyWeakref_GetRef()* en su lugar.

int **PyWeakref_IsDead**(*PyObject* *ref)

Test if the weak reference *ref* is dead. Returns 1 if the reference is dead, 0 if it is alive, and -1 with an error set if *ref* is not a weak reference object.

Added in version 3.14.

void **PyObject_ClearWeakRefs**(*PyObject* *object)

Part of the Stable ABI. Esta función es invocada por el gestor *tp_dealloc* para limpiar referencias débiles.

Esto recorre las referencias débiles de *object* e invoca retrollamadas para aquellas referencias que tengan una. Retorna cuando se han intentado todas las retrollamadas.

void **PyUnstable_Object_ClearWeakRefsNoCallbacks**(*PyObject* *object)

> This is *Unstable API*. It may change without warning in minor releases.

Borra las referencias débiles para *object* sin llamar a las retrollamadas.

Esta función es invocada por el gestor `tp_dealloc` para tipos con finalizadores (i.e., `__del__()`). El gestor de esos objetos primero llama a `PyObject_ClearWeakRefs()` para borrar las referencias débiles y llamar a sus callbacks, luego al finalizador, y finalmente a esta función para borrar cualquier referencia débil que pueda haber sido creada por el finalizador.

En la mayoría de los casos, es más apropiado utilizar `PyObject_ClearWeakRefs()` para borrar las referencias débiles en lugar de esta función.

Added in version 3.13.

9.6.11 Cápsulas

Consulta using-capsules para obtener más información sobre el uso de estos objetos.

Added in version 3.1.

type **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.

type **PyCapsule_Destructor**

Part of the Stable ABI. 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. Part of the Stable ABI. 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)

Part of the Stable ABI. Recupera el *pointer* almacenado en la cápsula. En caso de falla, establece una excepción y retorna `NULL`.

El parámetro *name* debe coincidir exactamente con el nombre almacenado en la cápsula. Si el nombre almacenado en la cápsula es `NULL`, el *name* también debe ser `NULL`. Python utiliza la función C `strcmp()` para comparar los nombres de las cápsulas.

PyCapsule_Destructor **PyCapsule_GetDestructor**(*PyObject* *capsule)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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.

This function splits *name* on the `.` character, and imports the first element. It then processes further elements using attribute lookups.

Retorna el puntero *pointer* interno de la cápsula en caso de éxito. En caso de falla, establece una excepción y retorna `NULL`.

> **Nota**
>
> If *name* points to an attribute of some submodule or subpackage, this submodule or subpackage must be previously imported using other means (for example, by using *PyImport_ImportModule()*) for the attribute lookups to succeed.

Distinto en la versión 3.3: *no_block* ya no tiene efecto.

int **PyCapsule_IsValid**(*PyObject* *capsule, const char *name)

Part of the Stable ABI. 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, se garantiza que las llamadas a cualquiera de las funciones de acceso (cualquier función que comience con `PyCapsule_Get`) tendrán 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)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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.

9.6.12 Objetos frame

type **PyFrameObject**

Part of the Limited API *(as an opaque struct).* La estructura C de los objetos utilizados para describir los objetos del frame.

No hay miembros públicos en esta estructura.

Distinto en la versión 3.11: Los miembros de esta estructura se han eliminado de la API pública de C. Consulte la entrada Novedades para más detalles.

Las funciones *PyEval_GetFrame()* y *PyThreadState_GetFrame()* pueden utilizarse para obtener un objeto frame.

Véase también *Reflexión*.

PyTypeObject **PyFrame_Type**

The type of frame objects. It is the same object as `types.FrameType` in the Python layer.

Distinto en la versión 3.11: Previously, this type was only available after including `<frameobject.h>`.

int **PyFrame_Check**(*PyObject* *obj)

Return non-zero if *obj* is a frame object.

Distinto en la versión 3.11: Previously, this function was only available after including `<frameobject.h>`.

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

Return value: New reference. Obtiene el *frame* exterior siguiente.

Retorna una *strong reference*, o `NULL` si *frame* no tiene frame exterior.

Added in version 3.9.

PyObject ***PyFrame_GetBuiltins**(*PyFrameObject* *frame)

Return value: New reference. Get the *frame*'s `f_builtins` attribute.

Retorna una *strong reference*, o `NULL` si *frame* no tiene frame exterior.

Added in version 3.11.

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

Return value: New reference. Part of the Stable ABI *since version 3.10.* Obtenga el código *frame*.

Retorna un *strong reference*.

El resultado (frame code) no puede ser `NULL`.

Added in version 3.9.

PyObject ***PyFrame_GetGenerator**(*PyFrameObject* *frame)

Return value: New reference. Obtiene el generador, rutina o generador asíncrono al que pertenece este frame, o `NULL` si este frame no es propiedad de un generador. No lanza una excepción, incluso si el valor de retorno es `NULL`.

Retorna un *strong reference*, o `NULL`.

Added in version 3.11.

PyObject ***PyFrame_GetGlobals**(*PyFrameObject* *frame)

Return value: New reference. Get the *frame*'s `f_globals` attribute.

Retorna una *strong reference*, o `NULL` si *frame* no tiene frame exterior.

Added in version 3.11.

int **PyFrame_GetLasti**(*PyFrameObject* *frame)

Get the *frame*'s `f_lasti` attribute.

Retorna -1 si `frame.f_lasti` es `None`.

Added in version 3.11.

PyObject ***PyFrame_GetVar**(*PyFrameObject* *frame, *PyObject* *name)

Return value: New reference. Get the variable *name* of *frame*.

- Return a *strong reference* to the variable value on success.
- Raise `NameError` and return `NULL` if the variable does not exist.
- Raise an exception and return `NULL` on error.

name type must be a `str`.

Added in version 3.12.

PyObject ***PyFrame_GetVarString**(*PyFrameObject* *frame, const char *name)

Return value: New reference. Similar to *`PyFrame_GetVar()`*, but the variable name is a C string encoded in UTF-8.

Added in version 3.12.

PyObject ***PyFrame_GetLocals**(*PyFrameObject* *frame)

Return value: New reference. Get the *frame*'s `f_locals` attribute. If the frame refers to an *optimized scope*, this returns a write-through proxy object that allows modifying the locals. In all other cases (classes, modules, `exec()`, `eval()`) it returns the mapping representing the frame locals directly (as described for `locals()`).

Retorna un *strong reference*.

Added in version 3.11.

Distinto en la versión 3.13: As part of **PEP 667**, return an instance of *`PyFrameLocalsProxy_Type`*.

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

Part of the Stable ABI *since version 3.10.* Retorna el número de línea en la que se está ejecutando el *frame*.

Frame Locals Proxies

Added in version 3.13.

The `f_locals` attribute on a frame object is an instance of a «frame-locals proxy». The proxy object exposes a write-through view of the underlying locals dictionary for the frame. This ensures that the variables exposed by `f_locals` are always up to date with the live local variables in the frame itself.

See **PEP 667** for more information.

PyTypeObject **PyFrameLocalsProxy_Type**

The type of frame `locals()` proxy objects.

int **PyFrameLocalsProxy_Check**(*PyObject* *obj)

Return non-zero if *obj* is a frame `locals()` proxy.

Internal Frames

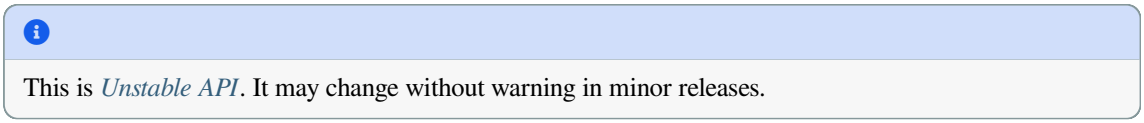
Unless using **PEP 523**, you will not need this.

struct **_PyInterpreterFrame**

The interpreter's internal frame representation.

Added in version 3.11.

PyObject ***PyUnstable_InterpreterFrame_GetCode**(struct *_PyInterpreterFrame* *frame);

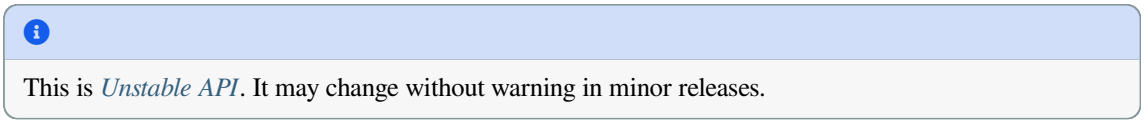


This is *Unstable API*. It may change without warning in minor releases.

Return a *strong reference* to the code object for the frame.

Added in version 3.12.

int **PyUnstable_InterpreterFrame_GetLasti**(struct *_PyInterpreterFrame* *frame);

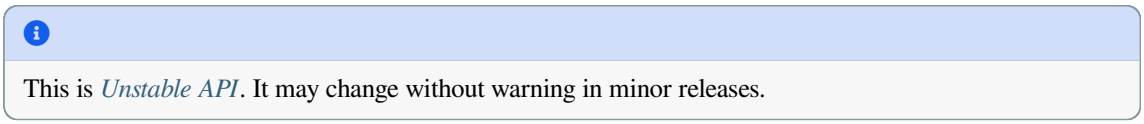


This is *Unstable API*. It may change without warning in minor releases.

Return the byte offset into the last executed instruction.

Added in version 3.12.

int **PyUnstable_InterpreterFrame_GetLine**(struct *_PyInterpreterFrame* *frame);



This is *Unstable API*. It may change without warning in minor releases.

Return the currently executing line number, or -1 if there is no line number.

Added in version 3.12.

9.6.13 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()*.

type **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`.

9.6.14 Objetos corrutina

Added in version 3.5.

Los objetos de corrutina son las funciones declaradas con un retorno de palabra clave `async`.

type **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`.

9.6.15 Objetos de variables de contexto

Added in version 3.7.

Distinto en la versión 3.7.1:

Nota

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:

```
// en 3.7.0:
PyContext *PyContext_New(void);

// en 3.7.1+:
PyObject *PyContext_New(void);
```

Ver bpo-34762 para más detalles.

Esta sección detalla la API pública de C para el módulo `contextvars`.

type **PyContext**

La estructura C utilizada para representar un objeto `contextvars.Context`.

type **PyContextVar**

La estructura C utilizada para representar un objeto `contextvars.ContextVar`.

type **PyContextToken**

La estructura C solía representar un objeto `contextvars.Token`.

PyTypeObject **PyContext_Type**

El objeto de tipo que representa el tipo *context*.

PyTypeObject **PyContextVar_Type**

El objeto tipo que representa el tipo *variable de contexto*.

PyTypeObject **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.

int **PyContext_AddWatcher**(*PyContext_WatchCallback* callback)

Register *callback* as a context object watcher for the current interpreter. Return an ID which may be passed to *PyContext_ClearWatcher()*. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Added in version 3.14.

int **PyContext_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from *PyContext_AddWatcher()* for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

Added in version 3.14.

type **PyContextEvent**

Enumeration of possible context object watcher events:

- `Py_CONTEXT_SWITCHED`: The *current context* has switched to a different context. The object passed to the watch callback is the now-current `contextvars.Context` object, or None if no context is current.

Added in version 3.14.

typedef int (***PyContext_WatchCallback**)(*PyContextEvent* event, *PyObject* *obj)

Context object watcher callback function. The object passed to the callback is event-specific; see *PyContextEvent* for details.

If the callback returns with an exception set, it must return `-1`; this exception will be printed as an unraisable exception using *PyErr_FormatUnraisable()*. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Added in version 3.14.

Funciones variables de contexto:

PyObject ***PyContextVar_New**(const char *name, *PyObject* *def)

Return value: New reference. Crea un nuevo objeto `ContextVar`. El parámetro *name* se usa para propósitos de introspección y depuración. El parámetro *def* especifica el valor predeterminado para la variable de contexto, o `NULL` para no especificar un valor predeterminado. Si se ha producido un error, esta función retorna `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`

A excepción de `NULL`, la función retorna una nueva referencia.

PyObject ***PyContextVar_Set**(*PyObject* *var, *PyObject* *value)

Return value: New reference. Establece el valor de *var* en *value* en el contexto actual. Retorna un nuevo objeto token para este cambio, o `NULL` si se ha producido un error.

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.

9.6.16 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.

type **PyDateTime_Date**

Este subtipo de *PyObject* representa un *date object* de Python.

type **PyDateTime_DateTime**

Este subtipo de *PyObject* representa un *datetime object* de Python.

type **PyDateTime_Time**

Este subtipo de *PyObject* representa un *time object* de Python.

type **PyDateTime_Delta**

Este subtipo de *PyObject* representa la diferencia entre dos valores *datetime*.

PyTypeObject **PyDateTime_DateType**

Esta instancia de *PyTypeObject* representa el *date type* de Python; es el mismo objeto que `datetime.date` en la capa de Python.

PyTypeObject **PyDateTime_DateTimeType**

Esta instancia de *PyTypeObject* representa el *datetime type* de Python; es el mismo objeto que `datetime.datetime` en la capa de Python.

PyTypeObject **PyDateTime_TimeType**

Esta instancia de *PyObject* representa el *time type* de Python; es el mismo objeto que `datetime.time` en la capa de Python.

PyTypeObject **PyDateTime_DeltaType**

Esta instancia de *PyTypeObject* representa un tipo en *Python* para la diferencia entre dos valore *datetime*; es el mismo objeto que `datetime.timedelta` en la capa de Python.

PyTypeObject **PyDateTime_TZInfoType**

Esta instancia de *PyTypeObject* representa el tipo de Python de *time zone info*; es el mismo objeto que `datetime.tzinfo` de la capa de Python.

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`.

Added in version 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.

Added in version 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.

Added in version 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**(*PyObject* *offset)

Return value: New reference. Retorna un objeto datetime.timezone con un desplazamiento fijo sin nombre representado por el argumento *offset*.

Added in version 3.7.

PyObject ***PyTimeZone_FromOffsetAndName**(*PyObject* *offset, *PyObject* *name)

Return value: New reference. Retorna un objeto datetime.timezone con un desplazamiento fijo representado por el argumento *offset* y con tzname *name*.

Added in version 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 se comprueba.

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)

Retorna el *fold*, como int de 0 a 1.

Added in version 3.6.

PyObject ***PyDateTime_DATE_GET_TZINFO**(*PyDateTime_DateTime* *o)

Retorna el tzinfo (que puede ser `None`).

Added in version 3.10.

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 es comprobado:

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)

Retorna el *fold*, como int de 0 a 1.

Added in version 3.6.

PyObject ***PyDateTime_TIME_GET_TZINFO**(*PyDateTime_Time* *o)

Retorna el tzinfo (que puede ser `None`).

Added in version 3.10.

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 es comprobado:

int **PyDateTime_DELTA_GET_DAYS**(*PyDateTime_Delta* *o)

Retorna el número de días, como un int desde -999999999 a 999999999.

Added in version 3.3.

int **PyDateTime_DELTA_GET_SECONDS**(*PyDateTime_Delta* *o)

Retorna el número de segundos, como un int de 0 a 86399.

Added in version 3.3.

int **PyDateTime_DELTA_GET_MICROSECONDS**(*PyDateTime_Delta* *o)

Retorna el número de micro segundos, como un int de 0 a 999999.

Added in version 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()`.

9.6.17 Objetos para indicaciones de tipado

Se proporcionan varios tipos incorporados para indicaciones de tipado. Actualmente existen dos tipos – GenericAlias y Union. Solo `GenericAlias` es expuesto a C.

PyObject ***Py_GenericAlias**(*PyObject* *origin, *PyObject* *args)

Part of the Stable ABI *since version 3.9.* Crea un objeto GenericAlias. Equivalente a llamar la clase de Python `types.GenericAlias`. Los argumentos *origin* y *args* establecen los atributos `__origin__` y `__args__` de `GenericAlias` respectivamente. *origin* debe ser un *PyTypeObject**, y *args* puede ser un *PyTupleObject** o cualquier `PyObject*`. Si *args* no es una tupla, se construye automáticamente una tupla de un elemento y `__args__` se establece como `(args,)`. Se realiza una verificación mínima para los argumentos, por lo que la función tendrá éxito incluso si *origin* no es un tipo. El atributo `__parameters__` de `GenericAlias` se construye de manera perezosa a partir de `__args__`. En caso de falla, se levantará una excepción y retorna `NULL`.

Aquí hay un ejemplo sobre cómo hacer un tipo de extensión genérica:

```
static PyMethodDef my_obj_methods[] = {
    // Otros Métodos.
    ...
    {"__class_getitem__", Py_GenericAlias, METH_O|METH_CLASS, "Ver PEP 585"}
    ...

}
```

Ver también

El método del modelo de datos `__class_getitem__()`.

Added in version 3.9.

PyTypeObject **Py_GenericAliasType**

Part of the Stable ABI *since version 3.9.* El tipo en C del objeto retornado por *Py_GenericAlias()*. Equivalente a `types.GenericAlias` en Python.

Added in version 3.9.

CAPÍTULO 10

Inicialización, finalización e hilos

See *Python Initialization Configuration* for details on how to configure the interpreter prior to initialization.

10.1 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:

- Functions that initialize the interpreter:
  - `Py_Initialize()`
  - `Py_InitializeEx()`
  - `Py_InitializeFromConfig()`
  - `Py_BytesMain()`
  - `Py_Main()`
  - the runtime pre-initialization functions covered in *Configuración de inicialización de Python*
- Funciones de configuración:
  - `PyImport_AppendInittab()`
  - `PyImport_ExtendInittab()`
  - `PyInitFrozenExtensions()`
  - `PyMem_SetAllocator()`
  - `PyMem_SetupDebugHooks()`
  - `PyObject_SetArenaAllocator()`
  - `Py_SetProgramName()`
  - `Py_SetPythonHome()`
  - `PySys_ResetWarnOptions()`
  - the configuration functions covered in *Configuración de inicialización de Python*
- Funciones informativas:

  - *Py_IsInitialized()*
  - *PyMem_GetAllocator()*
  - *PyObject_GetArenaAllocator()*
  - *Py_GetBuildInfo()*
  - *Py_GetCompiler()*
  - *Py_GetCopyright()*
  - *Py_GetPlatform()*
  - *Py_GetVersion()*
  - *Py_IsInitialized()*
- Utilidades:
  - *Py_DecodeLocale()*
  - the status reporting and utility functions covered in *Configuración de inicialización de Python*
- Asignadores de memoria:
  - *PyMem_RawMalloc()*
  - *PyMem_RawRealloc()*
  - *PyMem_RawCalloc()*
  - *PyMem_RawFree()*
- Synchronization:
  - *PyMutex_Lock()*
  - *PyMutex_Unlock()*

Nota

Despite their apparent similarity to some of the functions listed above, the following functions **should not be called** before the interpreter has been initialized: *Py_EncodeLocale()*, *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()*, *Py_GetProgramFullPath()*, *Py_GetPythonHome()*, *Py_GetProgramName()*, *PyEval_InitThreads()*, and *Py_RunMain()*.

10.2 Variables de configuración global

Python tiene variables para la configuración global para controlar diferentes características y opciones. De forma predeterminada, 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**

This API is kept for backward compatibility: setting *PyConfig.bytes_warning* should be used instead, see *Python Initialization Configuration*.

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`.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_DebugFlag**

This API is kept for backward compatibility: setting *PyConfig.parser_debug* should be used instead, see *Python Initialization Configuration*.

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`.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_DontWriteBytecodeFlag**

This API is kept for backward compatibility: setting *PyConfig.write_bytecode* should be used instead, see *Python Initialization Configuration*.

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`.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_FrozenFlag**

This API is kept for backward compatibility: setting *PyConfig.pathconfig_warnings* should be used instead, see *Python Initialization Configuration*.

Suprime los mensajes de error al calcular la ruta de búsqueda del módulo en *Py_GetPath()*.

Indicador privado utilizado por los programas `_freeze_module` y `frozenmain`.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_HashRandomizationFlag**

This API is kept for backward compatibility: setting *PyConfig.hash_seed* and *PyConfig.use_hash_seed* should be used instead, see *Python Initialization Configuration*.

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.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_IgnoreEnvironmentFlag**

This API is kept for backward compatibility: setting *PyConfig.use_environment* should be used instead, see *Python Initialization Configuration*.

Ignore all `PYTHON*` environment variables, e.g. `PYTHONPATH` and `PYTHONHOME`, that might be set.

Establecido por las opciones `-E` y `-I`.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_InspectFlag**

This API is kept for backward compatibility: setting *PyConfig.inspect* should be used instead, see *Python Initialization Configuration*.

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`.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_InteractiveFlag**

This API is kept for backward compatibility: setting *PyConfig.interactive* should be used instead, see *Python Initialization Configuration*.

Establecido por la opción `-i`.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_IsolatedFlag**

This API is kept for backward compatibility: setting *PyConfig.isolated* should be used instead, see *Python Initialization Configuration*.

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-pacages*).

Establecido por la opción `-I`.

Added in version 3.4.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_LegacyWindowsFSEncodingFlag**

This API is kept for backward compatibility: setting *PyPreConfig.legacy_windows_fs_encoding* should be used instead, see *Python Initialization Configuration*.

Si la bandera no es cero, utilice la codificación `mbcs` con el gestor de errores `replace` en lugar de la codificación UTF-8 con el gestor de error `surrogatepass`, para la *filesystem encoding and error handler* (codificación del sistema de archivos y gestor de errores).

Establece en `1` si la variable de entorno `PYTHONLEGACYWINDOWSFSENCODING` está configurada en una cadena de caracteres no vacía.

Ver **PEP 529** para más detalles.

Availability: Windows.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_LegacyWindowsStdioFlag**

This API is kept for backward compatibility: setting *PyConfig.legacy_windows_stdio* should be used instead, see *Python Initialization Configuration*.

If the flag is non-zero, use `io.FileIO` instead of `io._WindowsConsoleIO` for `sys` standard streams.

Establece en `1` si la variable de entorno `PYTHONLEGACYWINDOWSSTDIO` está configurada en una cadena de caracteres no vacía.

Ver **PEP 528** para más detalles.

Availability: Windows.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_NoSiteFlag**

This API is kept for backward compatibility: setting *PyConfig.site_import* should be used instead, see *Python Initialization Configuration*.

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`.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_NoUserSiteDirectory**

This API is kept for backward compatibility: setting *PyConfig.user_site_directory* should be used instead, see *Python Initialization Configuration*.

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`.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_OptimizeFlag**

This API is kept for backward compatibility: setting *PyConfig.optimization_level* should be used instead, see *Python Initialization Configuration*.

Establecido por la opción `-O` y la variable de entorno `PYTHONOPTIMIZE`.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_QuietFlag**

This API is kept for backward compatibility: setting *PyConfig.quiet* should be used instead, see *Python Initialization Configuration*.

No muestre los mensajes de *copyright* y de versión incluso en modo interactivo.

Establecido por la opción `-q`.

Added in version 3.2.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_UnbufferedStdioFlag**

This API is kept for backward compatibility: setting *PyConfig.buffered_stdio* should be used instead, see *Python Initialization Configuration*.

Obliga a las secuencias *stdout* y *stderr* a que no tengan búfer.

Establecido por la opción `-u` y la variable de entorno `PYTHONUNBUFFERED`.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_VerboseFlag**

This API is kept for backward compatibility: setting *PyConfig.verbose* should be used instead, see *Python Initialization Configuration*.

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`.

Deprecated since version 3.12, will be removed in version 3.15.

10.3 Inicializando y finalizando el intérprete

void **Py_Initialize**()

Part of the Stable ABI. 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.

This initializes the table of loaded modules (`sys.modules`), and creates the fundamental modules `builtins`, `__main__` and `sys`. It also initializes the module search path (`sys.path`). It does not set `sys.argv`; use the *Python Initialization Configuration* API for that. This is a no-op when called for a second time (without calling *Py_FinalizeEx()* first). There is no return value; it is a fatal error if the initialization fails.

Use *Py_InitializeFromConfig()* to customize the *Python Initialization Configuration*.

> **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)

Part of the Stable ABI. This function works like `Py_Initialize()` if *initsigs* is `1`. If *initsigs* is `0`, it skips initialization registration of signal handlers, which may be useful when CPython is embedded as part of a larger application.

Use `Py_InitializeFromConfig()` to customize the *Python Initialization Configuration*.

PyStatus **Py_InitializeFromConfig**(const *PyConfig* *config)

Initialize Python from *config* configuration, as described in *Inicialización con PyConfig*.

See the *Configuración de inicialización de Python* section for details on pre-initializing the interpreter, populating the runtime configuration structure, and querying the returned status structure.

int **Py_IsInitialized**()

Part of the Stable ABI. 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_IsFinalizing**()

Part of the Stable ABI *since version 3.13.* Return true (non-zero) if the main Python interpreter is *shutting down*. Return false (zero) otherwise.

Added in version 3.13.

int **Py_FinalizeEx**()

Part of the Stable ABI *since version 3.6.* Undo all initializations made by `Py_Initialize()` and subsequent use of Python/C API functions, and destroy all sub-interpreters (see `Py_NewInterpreter()` below) that were created and not yet destroyed since the last call to `Py_Initialize()`. This is a no-op when called for a second time (without calling `Py_Initialize()` again first).

Since this is the reverse of `Py_Initialize()`, it should be called in the same thread with the same interpreter active. That means the main thread and the main interpreter. This should never be called while `Py_RunMain()` is running.

Normally the return value is `0`. If there were errors during finalization (flushing buffered data), `-1` is returned.

Note that Python will do a best effort at freeing all memory allocated by the Python interpreter. Therefore, any C-Extension should make sure to correctly clean up all of the preveiously allocated PyObjects before using them in subsequent calls to `Py_Initialize()`. Otherwise it could introduce vulnerabilities and incorrect behavior.

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.

Bugs and caveats: The destruction of modules and objects in modules is done in random order; this may cause destructors (`__del__()` methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Interned strings will all be deallocated regardless of their reference count. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once; this can happen if an application calls `Py_Initialize()` and `Py_FinalizeEx()` more than once. `Py_FinalizeEx()` must not be called recursively from within itself. Therefore, it must not be called by any code that may be run as part of the interpreter

shutdown process, such as `atexit` handlers, object finalizers, or any code that may be run while flushing the stdout and stderr files.

Genera un evento de auditoría `cpython._PySys_ClearAuditHooks` sin argumentos.

Added in version 3.6.

void **Py_Finalize**()

Part of the Stable ABI. Esta es una versión compatible con versiones anteriores de *Py_FinalizeEx()* que ignora el valor de retorno.

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

Part of the Stable ABI *since version 3.8.* Similar to *Py_Main()* but *argv* is an array of bytes strings, allowing the calling application to delegate the text decoding step to the CPython runtime.

Added in version 3.8.

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

Part of the Stable ABI. The main program for the standard interpreter, encapsulating a full initialization/finalization cycle, as well as additional behaviour to implement reading configurations settings from the environment and command line, and then executing `__main__` in accordance with using-on-cmdline.

This is made available for programs which wish to support the full CPython command line interface, rather than just embedding a Python runtime in a larger application.

The *argc* and *argv* parameters are similar to those which are passed to a C program's `main()` function, except that the *argv* entries are first converted to `wchar_t` using *Py_DecodeLocale()*. It is also important to note that the argument list entries may be modified to point to strings other than those passed in (however, the contents of the strings pointed to by the argument list are not modified).

The return value is `2` if the argument list does not represent a valid Python command line, and otherwise the same as *Py_RunMain()*.

In terms of the CPython runtime configuration APIs documented in the *runtime configuration* section (and without accounting for error handling), `Py_Main` is approximately equivalent to:

```
PyConfig config;
PyConfig_InitPythonConfig(&config);
PyConfig_SetArgv(&config, argc, argv);
Py_InitializeFromConfig(&config);
PyConfig_Clear(&config);

Py_RunMain();
```

In normal usage, an embedding application will call this function *instead* of calling *Py_Initialize()*, *Py_InitializeEx()* or *Py_InitializeFromConfig()* directly, and all settings will be applied as described elsewhere in this documentation. If this function is instead called *after* a preceding runtime initialization API call, then exactly which environmental and command line configuration settings will be updated is version dependent (as it depends on which settings correctly support being modified after they have already been set once when the runtime was first initialized).

int **Py_RunMain**(void)

Executes the main module in a fully configured CPython runtime.

Executes the command (*PyConfig.run_command*), the script (*PyConfig.run_filename*) or the module (*PyConfig.run_module*) specified on the command line or in the configuration. If none of these values are set, runs the interactive Python prompt (REPL) using the `__main__` module's global namespace.

If *PyConfig.inspect* is not set (the default), the return value will be `0` if the interpreter exits normally (that is, without raising an exception), the exit status of an unhandled `SystemExit`, or `1` for any other unhandled exception.

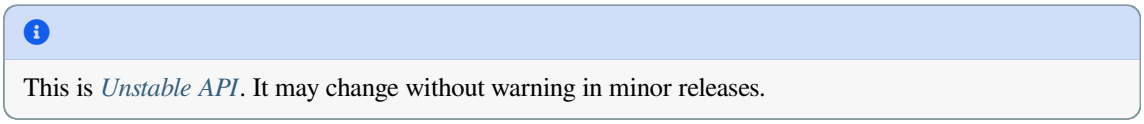
If *PyConfig.inspect* is set (such as when the `-i` option is used), rather than returning when the interpreter exits, execution will instead resume in an interactive Python prompt (REPL) using the `__main__` module's

global namespace. If the interpreter exited with an exception, it is immediately raised in the REPL session. The function return value is then determined by the way the *REPL session* terminates: `0`, `1`, or the status of a `SystemExit`, as specified above.

This function always finalizes the Python interpreter before it returns.

See *Python Configuration* for an example of a customized Python that always runs in isolated mode using *Py_RunMain()*.

int **PyUnstable_AtExit**(*PyInterpreterState* *interp, void (*func)(void*), void *data)



This is *Unstable API*. It may change without warning in minor releases.

Register an `atexit` callback for the target interpreter *interp*. This is similar to *Py_AtExit()*, but takes an explicit interpreter and data pointer for the callback.

There must be an *attached thread state* for *interp*.

Added in version 3.13.

10.4 Parámetros de todo el proceso

void **Py_SetProgramName**(const wchar_t *name)

Part of the Stable ABI. Esta API se mantiene para la compatibilidad con versiones anteriores: en su lugar, se debe usar la configuración de *PyConfig.program_name*, consulta *Configuración de inicialización de Python*.

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()* to decode a bytes string to get a `wchar_t*` string.

Deprecated since version 3.11, will be removed in version 3.15.

wchar_t ***Py_GetProgramName**()

Part of the Stable ABI. Return the program name set with *PyConfig.program_name*, or the default. The returned string points into static storage; the caller should not modify its value.

Esta función ya no se puede llamar antes de *Py_Initialize()*, de otra forma retornará `NULL`.

Distinto en la versión 3.10: Todas las siguientes funciones deben llamarse después de *Py_Initialize()*, de lo contrario retornará `NULL`.

Deprecated since version 3.13, will be removed in version 3.15: Use *PyConfig_Get("executable")* (`sys.executable`) instead.

wchar_t ***Py_GetPrefix**()

Part of the Stable ABI. Return the *prefix* for installed platform-independent files. This is derived through a number of complicated rules from the program name set with *PyConfig.program_name* and some environment variables; for example, if the program name is `'/usr/local/bin/python'`, the prefix is `'/usr/local'`. The returned string points into static storage; the caller should not modify its value. This corresponds to the **prefix** variable in the top-level `Makefile` and the `--prefix` argument to the **configure** script at build time. The value is available to Python code as `sys.base_prefix`. It is only useful on Unix. See also the next function.

Esta función ya no se puede llamar antes de *Py_Initialize()*, de otra forma retornará `NULL`.

Distinto en la versión 3.10: Todas las siguientes funciones deben llamarse después de *Py_Initialize()*, de lo contrario retornará NULL.

Deprecated since version 3.13, will be removed in version 3.15: Use *PyConfig_Get("base_prefix")* (sys.base_prefix) instead. Use *PyConfig_Get("prefix")* (sys.prefix) if virtual environments need to be handled.

wchar_t ***Py_GetExecPrefix**()

Part of the Stable ABI. Return the *exec-prefix* for installed platform-*dependent* files. This is derived through a number of complicated rules from the program name set with *PyConfig.program_name* and some environment variables; for example, if the program name is '/usr/local/bin/python', the exec-prefix is '/usr/local'. The returned string points into static storage; the caller should not modify its value. This corresponds to the **exec_prefix** variable in the top-level Makefile and the --exec-prefix argument to the **configure** script at build time. The value is available to Python code as sys.base_exec_prefix. It is only useful on 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.

Esta función ya no se puede llamar antes de *Py_Initialize()*, de otra forma retornará NULL.

Distinto en la versión 3.10: Todas las siguientes funciones deben llamarse después de *Py_Initialize()*, de lo contrario retornará NULL.

Deprecated since version 3.13, will be removed in version 3.15: Use *PyConfig_Get("base_exec_prefix")* (sys.base_exec_prefix) instead. Use *PyConfig_Get("exec_prefix")* (sys.exec_prefix) if virtual environments need to be handled.

wchar_t ***Py_GetProgramFullPath**()

Part of the Stable ABI. Return the full program name of the Python executable; this is computed as a side-effect of deriving the default module search path from the program name (set by *PyConfig.program_name*). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as sys.executable.

Esta función ya no se puede llamar antes de *Py_Initialize()*, de otra forma retornará NULL.

Distinto en la versión 3.10: Todas las siguientes funciones deben llamarse después de *Py_Initialize()*, de lo contrario retornará NULL.

Deprecated since version 3.13, will be removed in version 3.15: Use *PyConfig_Get("executable")* (sys.executable) instead.

wchar_t ***Py_GetPath**()

Part of the Stable ABI. Return the default module search path; this is computed from the program name (set by *PyConfig.program_name*) 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.

Esta función ya no se puede llamar antes de *Py_Initialize()*, de otra forma retornará NULL.

Distinto en la versión 3.10: Todas las siguientes funciones deben llamarse después de *Py_Initialize()*, de lo contrario retornará NULL.

Deprecated since version 3.13, will be removed in version 3.15: Use *PyConfig_Get("module_search_paths")* (sys.path) instead.

const char ***Py_GetVersion**()

Part of the Stable ABI. 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]"
```

La primera palabra (hasta el primer carácter de espacio) es la versión actual de Python; los primeros tres caracteres son la versión mayor y menor separados por un punto. 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 sys.version.

Consulta también la constante *Py_Version*.

const char ***Py_GetPlatform**()

Part of the Stable ABI. Retorna el identificador de plataforma para la plataforma actual. En Unix, esto se forma a partir del nombre «oficial» del sistema operativo, convertido a minúsculas, seguido del número de revisión principal; por ejemplo, para Solaris 2.x, que también se conoce como SunOS 5.x, el valor es 'sunos5'. En macOS, es 'darwin'. En Windows, es 'win'. 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.platform.

const char ***Py_GetCopyright**()

Part of the Stable ABI. 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**()

Part of the Stable ABI. 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**()

Part of the Stable ABI. 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)

Part of the Stable ABI. Esta API se mantiene para la compatibilidad con versiones anteriores: en su lugar, se debe usar la configuración de *PyConfig.argv*, *PyConfig.parse_argv* y *PyConfig.safe_path*, consulta *Configuración de inicialización de Python*.

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()* to decode a bytes string to get a `wchar_t*` string.

Consulta también los miembros de *PyConfig.orig_argv* y *PyConfig.argv* de *Python Initialization Configuration*.

Nota

It is recommended that applications embedding the Python interpreter for purposes other than executing a single script pass `0` as *updatepath*, and update `sys.path` themselves if desired. See **CVE 2008-5983**.

En las versiones anteriores a 3.1.3, puede lograr el mismo efecto quitando manualmente el primer elemento (*popping*) `sys.path` después de haber llamado *PySys_SetArgv()*, por ejemplo usando

```
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```

Added in version 3.1.3.

Deprecated since version 3.11, will be removed in version 3.15.

void **PySys_SetArgv**(int argc, wchar_t **argv)

Part of the Stable ABI. Esta API se mantiene para la compatibilidad con versiones anteriores: en su lugar, se debe usar la configuración de *PyConfig.argv* y *PyConfig.parse_argv*, consulta *Configuración de inicialización de Python*.

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()* to decode a bytes string to get a `wchar_t*` string.

Consulta también los miembros de *PyConfig.orig_argv* y *PyConfig.argv* de *Python Initialization Configuration*.

Distinto en la versión 3.4: El valor *updatepath* depende de la opción `-I`.

Deprecated since version 3.11, will be removed in version 3.15.

void **Py_SetPythonHome**(const wchar_t *home)

Part of the Stable ABI. Esta API se mantiene para la compatibilidad con versiones anteriores: en su lugar, se debe usar la configuración de *PyConfig.home*, consulta *Configuración de inicialización de Python*.

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()* to decode a bytes string to get a `wchar_t*` string.

Deprecated since version 3.11, will be removed in version 3.15.

wchar_t *__Py_GetPythonHome__()

Part of the Stable ABI. Return the default «home», that is, the value set by *PyConfig.home*, or the value of the PYTHONHOME environment variable if it is set.

Esta función ya no se puede llamar antes de *Py_Initialize()*, de otra forma retornará NULL.

Distinto en la versión 3.10: Todas las siguientes funciones deben llamarse después de *Py_Initialize()*, de lo contrario retornará NULL.

Deprecated since version 3.13, will be removed in version 3.15: Use *PyConfig_Get("home")* or the PYTHONHOME environment variable instead.

10.5 Estado del hilo y el bloqueo global del intérprete

Unless on a *free-threaded* build of *CPython*, the Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the *global interpreter lock* or *GIL*, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

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.

The Python interpreter keeps some thread-specific bookkeeping information inside a data structure called *PyThreadState*, known as a *thread state*. Each OS thread has a thread-local pointer to a *PyThreadState*; a thread state referenced by this pointer is considered to be *attached*.

A thread can only have one *attached thread state* at a time. An attached thread state is typically analogous with holding the *GIL*, except on *free-threaded* builds. On builds with the *GIL* enabled, *attaching* a thread state will block until the *GIL* can be acquired. However, even on builds with the *GIL* disabled, it is still required to have an attached thread state to call most of the C API.

In general, there will always be an *attached thread state* when using Python's C API. Only in some specific cases (such as in a *Py_BEGIN_ALLOW_THREADS* block) will the thread not have an attached thread state. If uncertain, check if *PyThreadState_GetUnchecked()* returns NULL.

10.5.1 Detaching the thread state from extension code

Most extension code manipulating the *thread state* has the following simple structure:

```
Save the thread state in a local variable.
... Do some blocking I/O operation ...
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);
```

Here is how these functions work:

The *attached thread state* holds the *GIL* for the entire interpreter. When detaching the *attached thread state*, the *GIL* is released, allowing other threads to attach a thread state to their own thread, thus getting the *GIL* and can start executing. The pointer to the prior *attached thread state* is stored as a local variable. Upon reaching `Py_END_ALLOW_THREADS`, the thread state that was previously *attached* is passed to `PyEval_RestoreThread()`. This function will block until another releases its *thread state*, thus allowing the old *thread state* to get re-attached and the C API can be called again.

For *free-threaded* builds, the *GIL* is normally out of the question, but detaching the *thread state* is still required for blocking I/O and long operations. The difference is that threads don't have to wait for the *GIL* to be released to attach their thread state, allowing true multi-core parallelism.

Nota

Calling system I/O functions is the most common use case for detaching the *thread state*, but it can also be useful before calling long-running computations which don't need access to Python objects, such as compression or cryptographic functions operating over memory buffers. For example, the standard `zlib` and `hashlib` modules detach the *thread state* when compressing or hashing data.

10.5.2 Hilos creados sin Python

When threads are created using the dedicated Python APIs (such as the `threading` module), a thread state is automatically associated to them and the code showed above is therefore correct. However, when threads are created from C (for example by a third-party library with its own thread management), they don't hold the *GIL*, because they don't have an *attached thread state*.

If you need to call Python code from these threads (often this will be part of a callback API provided by the aforementioned third-party library), you must first register these threads with the interpreter by creating an *attached thread state* before you can start using the Python/C API. When you are done, you should detach the *thread state*, and finally free it.

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);
```

Note that the `PyGILState_*` functions assume there is only one global interpreter (created automatically by `Py_Initialize()`). Python supports the creation of additional interpreters (using `Py_NewInterpreter()`), but mixing multiple interpreters and the `PyGILState_*` API is unsupported. This is because `PyGILState_Ensure()` and similar functions default to *attaching* a *thread state* for the main interpreter, meaning that the thread can't safely interact with the calling subinterpreter.

10.5.3 Supporting subinterpreters in non-Python threads

If you would like to support subinterpreters with non-Python created threads, you must use the `PyThreadState_*` API instead of the traditional `PyGILState_*` API.

In particular, you must store the interpreter state from the calling function and pass it to `PyThreadState_New()`, which will ensure that the *thread state* is targeting the correct interpreter:

```
/* The return value of PyInterpreterState_Get() from the
   function that created this thread. */
PyInterpreterState *interp = ThreadData->interp;
PyThreadState *tstate = PyThreadState_New(interp);
PyThreadState_Swap(tstate);

/* GIL of the subinterpreter is now held.
   Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Destroy the thread state. No Python API allowed beyond this point. */
PyThreadState_Clear(tstate);
PyThreadState_DeleteCurrent();
```

10.5.4 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.

The fact that only the «current» thread remains means any locks held by other threads will never be released. Python solves this for `os.fork()` by acquiring the locks it uses internally before the fork, and releasing them afterwards. In addition, it resets any lock-objects in the child. When extending or embedding Python, there is no way to inform Python of additional (non-Python) locks that need to be acquired before or reset after a fork. OS facilities such as `pthread_atfork()` would need to be used to accomplish the same thing. Additionally, when extending or embedding Python, calling `fork()` directly rather than through `os.fork()` (and returning to or calling into Python) may result in a deadlock by one of Python's internal locks being held by a thread that is defunct after the fork. *PyOS_AfterFork_Child()* tries to reset the necessary locks, but is not always able to.

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.

10.5.5 Cautions regarding runtime finalization

In the late stage of *interpreter shutdown*, after attempting to wait for non-daemon threads to exit (though this can be interrupted by `KeyboardInterrupt`) and running the `atexit` functions, the runtime is marked as *finalizing*: *Py_IsFinalizing()* and `sys.is_finalizing()` return true. At this point, only the *finalization thread* that initiated finalization (typically the main thread) is allowed to acquire the *GIL*.

If any thread, other than the finalization thread, attempts to attach a *thread state* during finalization, either explicitly or implicitly, the thread enters **a permanently blocked state** where it remains until the program exits. In most cases this is harmless, but this can result in deadlock if a later stage of finalization attempts to acquire a lock owned by the blocked thread, or otherwise waits on the blocked thread.

Gross? Yes. This prevents random crashes and/or unexpectedly skipped C++ finalizations further up the call stack when such threads were forcibly exited here in CPython 3.13 and earlier. The CPython runtime *thread state* C APIs have never had any error reporting or handling expectations at *thread state* attachment time that would've allowed for graceful exit from this situation. Changing that would require new stable C APIs and rewriting the majority of C code in the CPython ecosystem to use those with error handling.

10.5.6 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:

type **PyInterpreterState**

Part of the Limited API *(as an opaque struct).* 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.

Distinto en la versión 3.12: **PEP 684** introduced the possibility of a *per-interpreter GIL*. See `Py_NewInterpreterFromConfig()`.

type **PyThreadState**

Part of the Limited API *(as an opaque struct).* This data structure represents the state of a single thread. The only public data member is:

PyInterpreterState ***interp**

This thread's interpreter state.

void **PyEval_InitThreads**()

Part of the Stable ABI. 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()`.

Obsoleto desde la versión 3.9.

PyThreadState ***PyEval_SaveThread**()

Part of the Stable ABI. Detach the *attached thread state* and return it. The thread will have no *thread state* upon returning.

void **PyEval_RestoreThread**(*PyThreadState* *tstate)

Part of the Stable ABI. Set the *attached thread state* to *tstate*. The passed *thread state* **should not** be *attached*, otherwise deadlock ensues. *tstate* will be attached upon returning.

> **Nota**
>
> Calling this function from a thread when the runtime is finalizing will hang the thread until the program exits, even if the thread was not created by Python. Refer to *Cautions regarding runtime finalization* for more details.

Distinto en la versión 3.14: Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

PyThreadState ***PyThreadState_Get**()

Part of the Stable ABI. Return the *attached thread state*. If the thread has no attached thread state, (such as when inside of `Py_BEGIN_ALLOW_THREADS` block), then this issues a fatal error (so that the caller needn't check for `NULL`).

See also `PyThreadState_GetUnchecked()`.

PyThreadState ***PyThreadState_GetUnchecked**()

Similar to *PyThreadState_Get()*, but don't kill the process with a fatal error if it is NULL. The caller is responsible to check if the result is NULL.

Added in version 3.13: In Python 3.5 to 3.12, the function was private and known as `_PyThreadState_UncheckedGet()`.

PyThreadState ***PyThreadState_Swap**(*PyThreadState* *tstate)

Part of the Stable ABI. Set the *attached thread state* to *tstate*, and return the *thread state* that was attached prior to calling.

This function is safe to call without an *attached thread state*; it will simply return `NULL` indicating that there was no prior thread state.

Nota

Similar to *PyGILState_Ensure()*, this function will hang the thread if the runtime is finalizing.

Las siguientes funciones utilizan almacenamiento local de hilos y no son compatibles con subinterpretes:

PyGILState_STATE **PyGILState_Ensure**()

Part of the Stable ABI. Ensure that the current thread is ready to call the Python C API regardless of the current state of Python, or of the *attached thread state*. This may be called as many times as desired by a thread as long as each call is matched with a call to *PyGILState_Release()*. In general, other thread-related APIs may be used between *PyGILState_Ensure()* and *PyGILState_Release()* calls as long as the thread state is restored to its previous state before the Release(). For example, normal usage of the *Py_BEGIN_ALLOW_THREADS* and *Py_END_ALLOW_THREADS* macros is acceptable.

The return value is an opaque «handle» to the *attached thread state* when *PyGILState_Ensure()* was called, and must be passed to *PyGILState_Release()* to ensure Python is left in the same state. Even though recursive calls are allowed, these handles *cannot* be shared - each unique call to *PyGILState_Ensure()* must save the handle for its call to *PyGILState_Release()*.

When the function returns, there will be an *attached thread state* and the thread will be able to call arbitrary Python code. Failure is a fatal error.

Advertencia

Calling this function when the runtime is finalizing is unsafe. Doing so will either hang the thread until the program ends, or fully crash the interpreter in rare cases. Refer to *Cautions regarding runtime finalization* for more details.

Distinto en la versión 3.14: Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

void **PyGILState_Release**(PyGILState_STATE)

Part of the Stable ABI. 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**()

Part of the Stable ABI. Get the *attached thread state* for this thread. May return `NULL` if no GILState API has been used on the current thread. Note that the main thread always has such a thread-state, even if no auto-thread-state call has been made on the main thread. This is mainly a helper/diagnostic function.

> **Nota**
>
> This function does not account for *thread states* created by something other than `PyGILState_Ensure()` (such as `PyThreadState_New()`). Prefer `PyThreadState_Get()` or `PyThreadState_GetUnchecked()` for most cases.

int **PyGILState_Check**()

Return `1` if the current thread is holding the *GIL* and `0` otherwise. This function can be called from any thread at any time. Only if it has had its *thread state* initialized via `PyGILState_Ensure()` will it return `1`. This is mainly a helper/diagnostic function. It can be useful for example in callback contexts or memory allocation functions when knowing that the *GIL* is locked can allow the caller to perform sensitive actions or otherwise behave differently.

> **Nota**
>
> If the current Python process has ever created a subinterpreter, this function will *always* return `1`. Prefer `PyThreadState_GetUnchecked()` for most cases.

Added in version 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

Part of the Stable ABI. 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

Part of the Stable ABI. 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

Part of the Stable ABI. Esta macro se expande a `PyEval_RestoreThread(_save);`: es equivalente a `Py_END_ALLOW_THREADS` sin la llave de cierre.

Py_UNBLOCK_THREADS

Part of the Stable ABI. 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.

10.5.7 API de bajo nivel

Todas las siguientes funciones deben llamarse después de `Py_Initialize()`.

Distinto en la versión 3.7: `Py_Initialize()` now initializes the *GIL* and sets an *attached thread state*.

PyInterpreterState ***PyInterpreterState_New**()

Part of the Stable ABI. Create a new interpreter state object. An *attached thread state* is not needed, but may optionally exist if it is necessary to serialize calls to this function.

Genera un evento de auditoría `python.PyInterpreterState_New` sin argumentos.

void **PyInterpreterState_Clear**(*PyInterpreterState* *interp)

Part of the Stable ABI. Reset all information in an interpreter state object. There must be an *attached thread state* for the interpreter.

Lanza una eventos de auditoría `python.PyInterpreterState_Clear` sin argumentos.

void **PyInterpreterState_Delete**(*PyInterpreterState* *interp)

Part of the Stable ABI. Destroy an interpreter state object. There **should not** be an *attached thread state* for the target interpreter. The interpreter state must have been reset with a previous call to `PyInterpreterState_Clear()`.

PyThreadState ***PyThreadState_New**(*PyInterpreterState* *interp)

Part of the Stable ABI. Create a new thread state object belonging to the given interpreter object. An *attached thread state* is not needed.

void **PyThreadState_Clear**(*PyThreadState* *tstate)

Part of the Stable ABI. Reset all information in a *thread state* object. *tstate* must be *attached*

Distinto en la versión 3.9: Esta función ahora llama a la retrollamada `PyThreadState.on_delete`. Anteriormente, eso sucedía en `PyThreadState_Delete()`.

Distinto en la versión 3.13: The `PyThreadState.on_delete` callback was removed.

void **PyThreadState_Delete**(*PyThreadState* *tstate)

Part of the Stable ABI. Destroy a *thread state* object. *tstate* should not be *attached* to any thread. *tstate* must have been reset with a previous call to `PyThreadState_Clear()`.

void **PyThreadState_DeleteCurrent**(void)

Detach the *attached thread state* (which must have been reset with a previous call to `PyThreadState_Clear()`) and then destroy it.

No *thread state* will be *attached* upon returning.

PyFrameObject ***PyThreadState_GetFrame**(*PyThreadState* *tstate)

Part of the Stable ABI *since version 3.10.* Obtiene el marco actual del estado del hilo de Python *tstate*.

Retorna una *strong reference* (referencia sólida). Retorna `NULL` si no se está ejecutando ningún cuadro.

Vea también `PyEval_GetFrame()`.

tstate must not be `NULL`, and must be *attached*.

Added in version 3.9.

uint64_t **PyThreadState_GetID**(*PyThreadState* *tstate)

Part of the Stable ABI *since version 3.10.* Get the unique *thread state* identifier of the Python thread state *tstate*.

tstate must not be `NULL`, and must be *attached*.

Added in version 3.9.

PyInterpreterState ***PyThreadState_GetInterpreter**(*PyThreadState* *tstate)

Part of the Stable ABI *since version 3.10.* Obtiene el intérprete del estado del hilo de Python *tstate*.

tstate must not be `NULL`, and must be *attached*.

Added in version 3.9.

void **PyThreadState_EnterTracing**(*PyThreadState* *tstate)

Suspender el seguimiento y el perfilado en el estado del hilo de Python *tstate*.

Reanudelos usando la función `PyThreadState_LeaveTracing()`.

Added in version 3.11.

void **PyThreadState_LeaveTracing**(*PyThreadState* *tstate)

Reanudar el seguimiento y el perfilado en el estado del hilo de Python *tstate* suspendido por la función `PyThreadState_EnterTracing()`.

Consulte también las funciones `PyEval_SetTrace()` y `PyEval_SetProfile()`.

Added in version 3.11.

PyInterpreterState ***PyInterpreterState_Get**(void)

Part of the Stable ABI *since version 3.9.* Obtiene el intérprete actual.

Issue a fatal error if there no *attached thread state*. It cannot return NULL.

Added in version 3.9.

int64_t **PyInterpreterState_GetID**(*PyInterpreterState* *interp)

Part of the Stable ABI *since version 3.7.* Retorna la identificación única del intérprete. Si hubo algún error al hacerlo, entonces se retorna `-1` y se establece un error.

The caller must have an *attached thread state*.

Added in version 3.7.

PyObject ***PyInterpreterState_GetDict**(*PyInterpreterState* *interp)

Return value: Borrowed reference. Part of the Stable ABI *since version 3.8.* 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.

The returned dictionary is borrowed from the interpreter and is valid until interpreter shutdown.

Added in version 3.8.

typedef *PyObject* *(***_PyFrameEvalFunction**)(*PyThreadState* *tstate, *_PyInterpreterFrame* *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*.

Distinto en la versión 3.11: El parámetro *frame* cambió de `PyFrameObject*` a `_PyInterpreterFrame*`.

_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».

Added in version 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».

Added in version 3.9.

PyObject ***PyThreadState_GetDict**()

Return value: Borrowed reference. Part of the Stable ABI. Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no *thread state* is *attached*. If this function returns `NULL`, no exception has been raised and the caller should assume no thread state is attached.

int **PyThreadState_SetAsyncExc**(unsigned long id, *PyObject* *exc)

Part of the Stable ABI. Asynchronously raise an exception in a thread. The *id* argument is the thread id of the target thread; *exc* is the exception object to be raised. This function does not steal any references to *exc*. To prevent naive misuse, you must write your own C extension to call this. Must be called with an *attached thread state*. Returns the number of thread states modified; this is normally one, but will be zero if the thread id isn't found. If *exc* is `NULL`, the pending exception (if any) for the thread is cleared. This raises no exceptions.

Distinto en la versión 3.7: El tipo del parámetro *id* cambia de `long` a `unsigned long`.

void **PyEval_AcquireThread**(*PyThreadState* *tstate)

Part of the Stable ABI. *Attach tstate* to the current thread, which must not be `NULL` or already *attached*.

The calling thread must not already have an *attached thread state*.

> **Nota**
>
> Calling this function from a thread when the runtime is finalizing will hang the thread until the program exits, even if the thread was not created by Python. Refer to *Cautions regarding runtime finalization* for more details.

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.

Distinto en la versión 3.14: Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

`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)

Part of the Stable ABI. Detach the *attached thread state*. The *tstate* argument, which must not be `NULL`, is only used to check that it represents the *attached thread state* — if it isn't, a fatal error is reported.

`PyEval_SaveThread()` es una función de nivel superior que siempre está disponible (incluso cuando los hilos no se han inicializado).

10.6 Soporte de subinterprete

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 subinterpretes 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 subinterpretes, 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:

type **PyInterpreterConfig**

Structure containing most parameters to configure a sub-interpreter. Its values are used only in `Py_NewInterpreterFromConfig()` and never modified by the runtime.

Added in version 3.12.

Structure fields:

int **use_main_obmalloc**

If this is `0` then the sub-interpreter will use its own «object» allocator state. Otherwise it will use (share) the main interpreter's.

If this is `0` then `check_multi_interp_extensions` must be `1` (non-zero). If this is `1` then `gil` must not be `PyInterpreterConfig_OWN_GIL`.

int **allow_fork**

If this is `0` then the runtime will not support forking the process in any thread where the sub-interpreter is currently active. Otherwise fork is unrestricted.

Note that the `subprocess` module still works when fork is disallowed.

int **allow_exec**

If this is `0` then the runtime will not support replacing the current process via exec (e.g. `os.execv()`) in any thread where the sub-interpreter is currently active. Otherwise exec is unrestricted.

Note that the `subprocess` module still works when exec is disallowed.

int **allow_threads**

If this is `0` then the sub-interpreter's `threading` module won't create threads. Otherwise threads are allowed.

int **allow_daemon_threads**

If this is `0` then the sub-interpreter's `threading` module won't create daemon threads. Otherwise daemon threads are allowed (as long as *`allow_threads`* is non-zero).

int **check_multi_interp_extensions**

If this is `0` then all extension modules may be imported, including legacy (single-phase init) modules, in any thread where the sub-interpreter is currently active. Otherwise only multi-phase init extension modules (see **PEP 489**) may be imported. (Also see *`Py_mod_multiple_interpreters`*.)

This must be `1` (non-zero) if *`use_main_obmalloc`* is `0`.

int **gil**

This determines the operation of the GIL for the sub-interpreter. It may be one of the following:

PyInterpreterConfig_DEFAULT_GIL

Use the default selection (*`PyInterpreterConfig_SHARED_GIL`*).

PyInterpreterConfig_SHARED_GIL

Use (share) the main interpreter's GIL.

PyInterpreterConfig_OWN_GIL

Use the sub-interpreter's own GIL.

If this is *`PyInterpreterConfig_OWN_GIL`* then *`PyInterpreterConfig.use_main_obmalloc`* must be `0`.

PyStatus **Py_NewInterpreterFromConfig**(*PyThreadState* **tstate_p, const *PyInterpreterConfig* *config)

Crea un nuevo subinterprete. 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).

The given *config* controls the options with which the interpreter is initialized.

Upon success, *tstate_p* will be set to the first *thread state* created in the new sub-interpreter. This thread state is *attached*. Note that no actual thread is created; see the discussion of thread states below. If creation of the new interpreter is unsuccessful, *tstate_p* is set to `NULL`; no exception is set since the exception state is stored in the *attached thread state*, which might not exist.

Like all other Python/C API functions, an *attached thread state* must be present before calling this function, but it might be detached upon returning. On success, the returned thread state will be *attached*. If the sub-interpreter is created with its own *GIL* then the *attached thread state* of the calling interpreter will be detached. When the function returns, the new interpreter's *thread state* will be *attached* to the current thread and the previous interpreter's *attached thread state* will remain detached.

Added in version 3.12.

Sub-interpreters are most effective when isolated from each other, with certain functionality restricted:

```
PyInterpreterConfig config = {
    .use_main_obmalloc = 0,
```

(continúe en la próxima página)

(proviene de la página anterior)

```
    .allow_fork = 0,
    .allow_exec = 0,
    .allow_threads = 1,
    .allow_daemon_threads = 0,
    .check_multi_interp_extensions = 1,
    .gil = PyInterpreterConfig_OWN_GIL,
};
PyThreadState *tstate = NULL;
PyStatus status = Py_NewInterpreterFromConfig(&tstate, &config);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}
```

Note that the config is used only briefly and does not get modified. During initialization the config's values are converted into various *PyInterpreterState* values. A read-only copy of the config may be stored internally on the *PyInterpreterState*.

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.

PyThreadState ***Py_NewInterpreter**(void)

Part of the Stable ABI. Create a new sub-interpreter. This is essentially just a wrapper around *Py_NewInterpreterFromConfig()* with a config that preserves the existing behavior. The result is an unisolated sub-interpreter that shares the main interpreter's GIL, allows fork/exec, allows daemon threads, and allows single-phase init modules.

void **Py_EndInterpreter**(*PyThreadState* *tstate)

Part of the Stable ABI. Destroy the (sub-)interpreter represented by the given *thread state*. The given thread state must be *attached*. When the call returns, there will be no *attached thread state*. All thread states associated with this interpreter are destroyed.

Py_FinalizeEx() will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

10.6.1 A Per-Interpreter GIL

Using *Py_NewInterpreterFromConfig()* you can create a sub-interpreter that is completely isolated from other interpreters, including having its own GIL. The most important benefit of this isolation is that such an interpreter can execute Python code without being blocked by other interpreters or blocking any others. Thus a single Python process can truly take advantage of multiple CPU cores when running Python code. The isolation also encourages a different approach to concurrency than that of just using threads. (See **PEP 554** and **PEP 684**.)

Using an isolated interpreter requires vigilance in preserving that isolation. That especially means not sharing any objects or mutable state without guarantees about thread-safety. Even objects that are otherwise immutable (e.g. `None`, `(1, 5)`) can't normally be shared because of the refcount. One simple but less-efficient approach around this

is to use a global lock around all use of some state (or object). Alternately, effectively immutable objects (like integers or strings) can be made safe in spite of their refcounts by making them *immortal*. In fact, this has been done for the builtin singletons, small integers, and a number of other builtin objects.

If you preserve isolation then you will have access to proper multi-core computing without the complications that come with free-threading. Failure to preserve isolation will expose you to the full consequences of free-threading, including races and hard-to-debug crashes.

Aside from that, one of the main challenges of using multiple isolated interpreters is how to communicate between them safely (not break isolation) and efficiently. The runtime and stdlib do not provide any standard approach to this yet. A future stdlib module would help mitigate the effort of preserving isolation and expose effective tools for communicating (and sharing) data between interpreters.

Added in version 3.12.

10.6.2 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 subinterprete 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_*` APIs es delicada, porque estas APIs 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 APIs para permitir la llamada de código Python desde hilos no creados por Python probablemente se rompan cuando se usan subinterpretes.

10.7 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)

Part of the Stable ABI. 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*;
- with the main thread holding an *attached thread state* (*func* can therefore use the full C API).

func must return `0` on success, or `-1` on failure with an exception set. *func* won't be interrupted to perform another asynchronous notification recursively, but it can still be interrupted to switch threads if the *thread state* is detached.

This function doesn't need an *attached thread state*. However, to call this function in a subinterpreter, the caller must have an *attached thread state*. Otherwise, the function *func* can be scheduled to be called from the wrong interpreter.

> **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*.

Added in version 3.1.

Distinto en la versión 3.9: Si esta función se llama en un subinterprete, la función *func* ahora está programada para ser llamada desde el subinterprete, en lugar de ser llamada desde el intérprete principal. Cada subinterprete ahora tiene su propia lista de llamadas programadas.

10.8 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.

typedef int (***Py_tracefunc**)(*PyObject* *obj, *PyFrameObject* *frame, int what, *PyObject* *arg)

The type of the trace function registered using `PyEval_SetProfile()` and `PyEval_SetTrace()`. The first parameter is the object passed to the registration function as *obj*, *frame* is the frame object to which the event pertains, *what* is one of the constants `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION`, `PyTrace_C_RETURN`, or `PyTrace_OPCODE`, and *arg* depends on the value of *what*:

Valor de *what*	Significado de *arg*
`PyTrace_CALL`	Siempre `Py_None`.
`PyTrace_EXCEPTION`	Información de excepción retornada por `sys.exc_info()`.
`PyTrace_LINE`	Siempre `Py_None`.
`PyTrace_RETURN`	Valor retornado al que llama, o `NULL` si es causado por una excepción.
`PyTrace_C_CALL`	Objeto función que se llaman.
`PyTrace_C_EXCEPTION`	Objeto función que se llaman.
`PyTrace_C_RETURN`	Objeto función que se llaman.
`PyTrace_OPCODE`	Siempre `Py_None`.

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**

The value passed as the *what* parameter to a `Py_tracefunc` function (but not a profiling function) when a

line-number event is being reported. It may be disabled for a frame by setting `f_trace_lines` to *0* on that frame.

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**

The value for the *what* parameter to *Py_tracefunc* functions (but not profiling functions) when a new opcode is about to be executed. This event is not emitted by default: it must be explicitly requested by setting `f_trace_opcodes` to *1* on the frame.

void **PyEval_SetProfile**(*Py_tracefunc* func, *PyObject* *obj)

Set the profiler function to *func*. The *obj* parameter is passed to the function as its first parameter, and may be any Python object, or `NULL`. If the profile function needs to maintain state, using a different value for *obj* for each thread provides a convenient and thread-safe place to store it. The profile function is called for all monitored events except *PyTrace_LINE* *PyTrace_OPCODE* and *PyTrace_EXCEPTION*.

Consulte también la función `sys.setprofile()`.

The caller must have an *attached thread state*.

void **PyEval_SetProfileAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

Like *PyEval_SetProfile()* but sets the profile function in all running threads belonging to the current interpreter instead of the setting it only on the current thread.

The caller must have an *attached thread state*.

As *PyEval_SetProfile()*, this function ignores any exceptions raised while setting the profile functions in all threads.

Added in version 3.12.

void **PyEval_SetTrace**(*Py_tracefunc* func, *PyObject* *obj)

Set the tracing function to *func*. This is similar to *PyEval_SetProfile()*, except the tracing function does receive line-number events and per-opcode events, but does not receive any event related to C function objects being called. Any trace function registered using *PyEval_SetTrace()* will not receive *PyTrace_C_CALL*, *PyTrace_C_EXCEPTION* or *PyTrace_C_RETURN* as a value for the *what* parameter.

Consulte también la función `sys.settrace()`.

The caller must have an *attached thread state*.

void **PyEval_SetTraceAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

Like *PyEval_SetTrace()* but sets the tracing function in all running threads belonging to the current interpreter instead of the setting it only on the current thread.

The caller must have an *attached thread state*.

As *PyEval_SetTrace()*, this function ignores any exceptions raised while setting the trace functions in all threads.

Added in version 3.12.

10.9 Reference tracing

Added in version 3.13.

typedef int (*__PyRefTracer__)(*PyObject**, int event, void *data)

The type of the trace function registered using `PyRefTracer_SetTracer()`. The first parameter is a Python object that has been just created (when **event** is set to `PyRefTracer_CREATE`) or about to be destroyed (when **event** is set to `PyRefTracer_DESTROY`). The **data** argument is the opaque pointer that was provided when `PyRefTracer_SetTracer()` was called.

Added in version 3.13.

int **PyRefTracer_CREATE**

The value for the *event* parameter to `PyRefTracer` functions when a Python object has been created.

int **PyRefTracer_DESTROY**

The value for the *event* parameter to `PyRefTracer` functions when a Python object has been destroyed.

int **PyRefTracer_SetTracer**(*PyRefTracer* tracer, void *data)

Register a reference tracer function. The function will be called when a new Python has been created or when an object is going to be destroyed. If **data** is provided it must be an opaque pointer that will be provided when the tracer function is called. Return `0` on success. Set an exception and return `-1` on error.

Not that tracer functions **must not** create Python objects inside or otherwise the call will be re-entrant. The tracer also **must not** clear any existing exception or set an exception. A *thread state* will be active every time the tracer function is called.

There must be an *attached thread state* when calling this function.

Added in version 3.13.

PyRefTracer **PyRefTracer_GetTracer**(void **data)

Get the registered reference tracer function and the value of the opaque data pointer that was registered when `PyRefTracer_SetTracer()` was called. If no tracer was registered this function will return NULL and will set the **data** pointer to NULL.

There must be an *attached thread state* when calling this function.

Added in version 3.13.

10.10 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`.

10.11 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 APIs 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.

A *thread state* does *not* need to be *attached* when calling these functions; they suppl their own locking.

Tenga en cuenta que `Python.h` no incluye la declaración de las API de TLS, debe incluir `pythread.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.

10.11.1 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.

Added in version 3.7.

Ver también

«Una nueva C-API para *Thread-Local Storage* en CPython» (**PEP 539**)

type **`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`**`()`

Part of the Stable ABI *since version 3.7.* 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`)`

Part of the Stable ABI *since version 3.7.* Libera la *clave* 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 *clave* es `NULL`.

> **Nota**
>
> Una clave liberada se convierte en un puntero colgante. Debería restablecer la clave a `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)

Part of the Stable ABI *since version 3.7.* 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)

Part of the Stable ABI *since version 3.7.* 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)

Part of the Stable ABI *since version 3.7.* 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)

Part of the Stable ABI *since version 3.7.* 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)

Part of the Stable ABI *since version 3.7.* 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.

10.11.2 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**()

Part of the Stable ABI.

void **PyThread_delete_key**(int key)

Part of the Stable ABI.

int **PyThread_set_key_value**(int key, void *value)

Part of the Stable ABI.

void *PyThread_get_key_value(int key)

Part of the Stable ABI.

void PyThread_delete_key_value(int key)

Part of the Stable ABI.

void PyThread_ReInitTLS()

Part of the Stable ABI.

10.12 Synchronization Primitives

The C-API provides a basic mutual exclusion lock.

type PyMutex

A mutual exclusion lock. The `PyMutex` should be initialized to zero to represent the unlocked state. For example:

```
PyMutex mutex = {0};
```

Instances of `PyMutex` should not be copied or moved. Both the contents and address of a `PyMutex` are meaningful, and it must remain at a fixed, writable location in memory.

> **Nota**
>
> A `PyMutex` currently occupies one byte, but the size should be considered unstable. The size may change in future Python releases without a deprecation period.

Added in version 3.13.

void PyMutex_Lock(*PyMutex* *m)

Lock mutex *m*. If another thread has already locked it, the calling thread will block until the mutex is unlocked. While blocked, the thread will temporarily detach the *thread state* if one exists.

Added in version 3.13.

void PyMutex_Unlock(*PyMutex* *m)

Unlock mutex *m*. The mutex must be locked — otherwise, the function will issue a fatal error.

Added in version 3.13.

int PyMutex_IsLocked(*PyMutex* *m)

Returns non-zero if the mutex *m* is currently locked, zero otherwise.

> **Nota**
>
> This function is intended for use in assertions and debugging only and should not be used to make concurrency control decisions, as the lock state may change immediately after the check.

Added in version 3.14.

10.12.1 Python Critical Section API

The critical section API provides a deadlock avoidance layer on top of per-object locks for *free-threaded* CPython. They are intended to replace reliance on the *global interpreter lock*, and are no-ops in versions of Python with the global interpreter lock.

Critical sections are intended to be used for custom types implemented in C-API extensions. They should generally not be used with built-in types like `list` and `dict` because their public C-APIs already use critical sections internally, with the notable exception of `PyDict_Next()`, which requires critical section to be acquired externally.

Critical sections avoid deadlocks by implicitly suspending active critical sections, hence, they do not provide exclusive access such as provided by traditional locks like *PyMutex*. When a critical section is started, the per-object lock for the object is acquired. If the code executed inside the critical section calls C-API functions then it can suspend the critical section thereby releasing the per-object lock, so other threads can acquire the per-object lock for the same object.

Variants that accept *PyMutex* pointers rather than Python objects are also available. Use these variants to start a critical section in a situation where there is no *PyObject* – for example, when working with a C type that does not extend or wrap *PyObject* but still needs to call into the C API in a manner that might lead to deadlocks.

The functions and structs used by the macros are exposed for cases where C macros are not available. They should only be used as in the given macro expansions. Note that the sizes and contents of the structures may change in future Python versions.

Nota

Operations that need to lock two objects at once must use *Py_BEGIN_CRITICAL_SECTION2*. You *cannot* use nested critical sections to lock more than one object at once, because the inner critical section may suspend the outer critical sections. This API does not provide a way to lock more than two objects at once.

Example usage:

```
static PyObject *
set_field(MyObject *self, PyObject *value)
{
   Py_BEGIN_CRITICAL_SECTION(self);
   Py_SETREF(self->field, Py_XNewRef(value));
   Py_END_CRITICAL_SECTION();
   Py_RETURN_NONE;
}
```

In the above example, *Py_SETREF* calls *Py_DECREF*, which can call arbitrary code through an object's deallocation function. The critical section API avoids potential deadlocks due to reentrancy and lock ordering by allowing the runtime to temporarily suspend the critical section if the code triggered by the finalizer blocks and calls *PyEval_SaveThread()*.

Py_BEGIN_CRITICAL_SECTION(op)

Acquires the per-object lock for the object *op* and begins a critical section.

In the free-threaded build, this macro expands to:

```
{
    PyCriticalSection _py_cs;
    PyCriticalSection_Begin(&_py_cs, (PyObject*)(op))
```

In the default build, this macro expands to `{`.

Added in version 3.13.

Py_BEGIN_CRITICAL_SECTION_MUTEX(m)

Locks the mutex *m* and begins a critical section.

In the free-threaded build, this macro expands to:

```
{
     PyCriticalSection _py_cs;
     PyCriticalSection_BeginMutex(&_py_cs, m)
```

Note that unlike *Py_BEGIN_CRITICAL_SECTION*, there is no cast for the argument of the macro - it must be a *PyMutex* pointer.

On the default build, this macro expands to `{`.

Added in version 3.14.

Py_END_CRITICAL_SECTION()

Ends the critical section and releases the per-object lock.

In the free-threaded build, this macro expands to:

```
    PyCriticalSection_End(&_py_cs);
}
```

In the default build, this macro expands to `}`.

Added in version 3.13.

Py_BEGIN_CRITICAL_SECTION2(a, b)

Acquires the per-objects locks for the objects *a* and *b* and begins a critical section. The locks are acquired in a consistent order (lowest address first) to avoid lock ordering deadlocks.

In the free-threaded build, this macro expands to:

```
{
    PyCriticalSection2 _py_cs2;
    PyCriticalSection2_Begin(&_py_cs2, (PyObject*)(a), (PyObject*)(b))
```

In the default build, this macro expands to `{`.

Added in version 3.13.

Py_BEGIN_CRITICAL_SECTION2_MUTEX(m1, m2)

Locks the mutexes *m1* and *m2* and begins a critical section.

In the free-threaded build, this macro expands to:

```
{
     PyCriticalSection2 _py_cs2;
     PyCriticalSection2_BeginMutex(&_py_cs2, m1, m2)
```

Note that unlike *Py_BEGIN_CRITICAL_SECTION2*, there is no cast for the arguments of the macro - they must be *PyMutex* pointers.

On the default build, this macro expands to `{`.

Added in version 3.14.

Py_END_CRITICAL_SECTION2()

Ends the critical section and releases the per-object locks.

In the free-threaded build, this macro expands to:

```
    PyCriticalSection2_End(&_py_cs2);
}
```

In the default build, this macro expands to `}`.

Added in version 3.13.

CAPÍTULO 11

Configuración de inicialización de Python

11.1 PyInitConfig C API

Added in version 3.14.

Python can be initialized with *Py_InitializeFromInitConfig()*.

La función *Py_RunMain()* se puede utilizar para escribir un programa Python personalizado.

Consulte también *Inicialización, finalización y subprocesos*.

Ver también

PEP 741 «Python Configuration C API».

11.1.1 Ejemplo

Example of customized Python always running with the Python Development Mode enabled; return `-1` on error:

```
int init_python(void)
{
    PyInitConfig *config = PyInitConfig_Create();
    if (config == NULL) {
        printf("PYTHON INIT ERROR: memory allocation failed\n");
        return -1;
    }

    // Enable the Python Development Mode
    if (PyInitConfig_SetInt(config, "dev_mode", 1) < 0) {
        goto error;
    }

    // Initialize Python with the configuration
    if (Py_InitializeFromInitConfig(config) < 0) {
        goto error;
    }
    PyInitConfig_Free(config);
```

(continúe en la próxima página)

(proviene de la página anterior)

```
    return 0;

error:
    {
        // Display the error message.
        //
        // This uncommon braces style is used, because you cannot make
        // goto targets point to variable declarations.
        const char *err_msg;
        (void)PyInitConfig_GetError(config, &err_msg);
        printf("PYTHON INIT ERROR: %s\n", err_msg);
        PyInitConfig_Free(config);
        return -1;
    }
}
```

11.1.2 Create Config

struct **PyInitConfig**

Opaque structure to configure the Python initialization.

PyInitConfig ***PyInitConfig_Create**(void)

Create a new initialization configuration using *Isolated Configuration* default values.

It must be freed by *PyInitConfig_Free()*.

Return `NULL` on memory allocation failure.

void **PyInitConfig_Free**(*PyInitConfig* *config)

Free memory of the initialization configuration *config*.

If *config* is `NULL`, no operation is performed.

11.1.3 Error Handling

int **PyInitConfig_GetError**(*PyInitConfig* *config, const char **err_msg)

Get the *config* error message.

- Set **err_msg* and return `1` if an error is set.
- Set **err_msg* to `NULL` and return `0` otherwise.

An error message is an UTF-8 encoded string.

If *config* has an exit code, format the exit code as an error message.

The error message remains valid until another `PyInitConfig` function is called with *config*. The caller doesn't have to free the error message.

int **PyInitConfig_GetExitCode**(*PyInitConfig* *config, int *exitcode)

Get the *config* exit code.

- Set **exitcode* and return `1` if *config* has an exit code set.
- Return `0` if *config* has no exit code set.

Only the `Py_InitializeFromInitConfig()` function can set an exit code if the `parse_argv` option is non-zero.

An exit code can be set when parsing the command line failed (exit code `2`) or when a command line option asks to display the command line help (exit code `0`).

11.1.4 Get Options

The configuration option *name* parameter must be a non-NULL null-terminated UTF-8 encoded string. See *Configuration Options*.

int **PyInitConfig_HasOption**(*PyInitConfig* *config, const char *name)

Test if the configuration has an option called *name*.

Return `1` if the option exists, or return `0` otherwise.

int **PyInitConfig_GetInt**(*PyInitConfig* *config, const char *name, int64_t *value)

Get an integer configuration option.

- Set **value*, and return `0` on success.
- Set an error in *config* and return `-1` on error.

int **PyInitConfig_GetStr**(*PyInitConfig* *config, const char *name, char **value)

Get a string configuration option as a null-terminated UTF-8 encoded string.

- Set **value*, and return `0` on success.
- Set an error in *config* and return `-1` on error.

**value* can be set to `NULL` if the option is an optional string and the option is unset.

On success, the string must be released with `free(value)` if it's not `NULL`.

int **PyInitConfig_GetStrList**(*PyInitConfig* *config, const char *name, size_t *length, char ***items)

Get a string list configuration option as an array of null-terminated UTF-8 encoded strings.

- Set **length* and **value*, and return `0` on success.
- Set an error in *config* and return `-1` on error.

On success, the string list must be released with `PyInitConfig_FreeStrList(length, items)`.

void **PyInitConfig_FreeStrList**(size_t length, char **items)

Free memory of a string list created by `PyInitConfig_GetStrList()`.

11.1.5 Set Options

The configuration option *name* parameter must be a non-NULL null-terminated UTF-8 encoded string. See *Configuration Options*.

Some configuration options have side effects on other options. This logic is only implemented when `Py_InitializeFromInitConfig()` is called, not by the «Set» functions below. For example, setting `dev_mode` to `1` does not set `faulthandler` to `1`.

int **PyInitConfig_SetInt**(*PyInitConfig* *config, const char *name, int64_t value)

Set an integer configuration option.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.

int **PyInitConfig_SetStr**(*PyInitConfig* *config, const char *name, const char *value)

Set a string configuration option from a null-terminated UTF-8 encoded string. The string is copied.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.

int **PyInitConfig_SetStrList**(*PyInitConfig* *config, const char *name, size_t length, char *const *items)

Set a string list configuration option from an array of null-terminated UTF-8 encoded strings. The string list is copied.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.

11.1.6 Module

int **PyInitConfig_AddModule**(*PyInitConfig* *config, const char *name, *PyObject* *(*initfunc)(void))

Add a built-in extension module to the table of built-in modules.

The new module can be imported by the name *name*, and uses the function *initfunc* as the initialization function called on the first attempted import.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.

If Python is initialized multiple times, `PyInitConfig_AddModule()` must be called at each Python initialization.

Similar to the *PyImport_AppendInittab()* function.

11.1.7 Initialize Python

int **Py_InitializeFromInitConfig**(*PyInitConfig* *config)

Initialize Python from the initialization configuration.

- Return `0` on success.
- Set an error in *config* and return `-1` on error.
- Set an exit code in *config* and return `-1` if Python wants to exit.

See `PyInitConfig_GetExitcode()` for the exit code case.

11.2 Configuration Options

Option	PyConfig/PyPreConfig member	Type	Visibility
"allocator"	*allocator*	int	Read-only
"argv"	*argv*	list[str]	Public
"base_exec_prefix"	*base_exec_prefix*	str	Public
"base_executable"	*base_executable*	str	Public
"base_prefix"	*base_prefix*	str	Public
"buffered_stdio"	*buffered_stdio*	bool	Read-only
"bytes_warning"	*bytes_warning*	int	Public
"check_hash_pycs_mode"	*check_hash_pycs_mode*	str	Read-only
"code_debug_ranges"	*code_debug_ranges*	bool	Read-only
"coerce_c_locale"	*coerce_c_locale*	bool	Read-only
"coerce_c_locale_warn"	*coerce_c_locale_warn*	bool	Read-only
"configure_c_stdio"	*configure_c_stdio*	bool	Read-only
"configure_locale"	*configure_locale*	bool	Read-only
"cpu_count"	*cpu_count*	int	Public
"dev_mode"	*dev_mode*	bool	Read-only
"dump_refs"	*dump_refs*	bool	Read-only
"dump_refs_file"	*dump_refs_file*	str	Read-only
"exec_prefix"	*exec_prefix*	str	Public
"executable"	*executable*	str	Public
"faulthandler"	*faulthandler*	bool	Read-only
"filesystem_encoding"	*filesystem_encoding*	str	Read-only
"filesystem_errors"	*filesystem_errors*	str	Read-only
"hash_seed"	*hash_seed*	int	Read-only
"home"	*home*	str	Read-only
"import_time"	*import_time*	int	Read-only
"inspect"	*inspect*	bool	Public

continúe en la próxima página

Tabla 1 – proviene de la página anterior

Option	PyConfig/PyPreConfig member	Type	Visibility
"install_signal_handlers"	*install_signal_handlers*	bool	Read-only
"int_max_str_digits"	*int_max_str_digits*	int	Public
"interactive"	*interactive*	bool	Public
"isolated"	*isolated*	bool	Read-only
"legacy_windows_fs_encoding"	*legacy_windows_fs_encoding*	bool	Read-only
"legacy_windows_stdio"	*legacy_windows_stdio*	bool	Read-only
"malloc_stats"	*malloc_stats*	bool	Read-only
"module_search_paths"	*module_search_paths*	list[str]	Public
"optimization_level"	*optimization_level*	int	Public
"orig_argv"	*orig_argv*	list[str]	Read-only
"parse_argv"	*parse_argv*	bool	Read-only
"parser_debug"	*parser_debug*	bool	Public
"pathconfig_warnings"	*pathconfig_warnings*	bool	Read-only
"perf_profiling"	*perf_profiling*	bool	Read-only
"platlibdir"	*platlibdir*	str	Public
"prefix"	*prefix*	str	Public
"program_name"	*program_name*	str	Read-only
"pycache_prefix"	*pycache_prefix*	str	Public
"quiet"	*quiet*	bool	Public
"run_command"	*run_command*	str	Read-only
"run_filename"	*run_filename*	str	Read-only
"run_module"	*run_module*	str	Read-only
"run_presite"	*run_presite*	str	Read-only
"safe_path"	*safe_path*	bool	Read-only
"show_ref_count"	*show_ref_count*	bool	Read-only
"site_import"	*site_import*	bool	Read-only
"skip_source_first_line"	*skip_source_first_line*	bool	Read-only
"stdio_encoding"	*stdio_encoding*	str	Read-only
"stdio_errors"	*stdio_errors*	str	Read-only
"stdlib_dir"	*stdlib_dir*	str	Public
"tracemalloc"	*tracemalloc*	int	Read-only
"use_environment"	*use_environment*	bool	Public
"use_frozen_modules"	*use_frozen_modules*	bool	Read-only
"use_hash_seed"	*use_hash_seed*	bool	Read-only
"use_system_logger"	*use_system_logger*	bool	Read-only
"user_site_directory"	*user_site_directory*	bool	Read-only
"utf8_mode"	*utf8_mode*	bool	Read-only
"verbose"	*verbose*	int	Public
"warn_default_encoding"	*warn_default_encoding*	bool	Read-only
"warnoptions"	*warnoptions*	list[str]	Public
"write_bytecode"	*write_bytecode*	bool	Public
"xoptions"	*xoptions*	dict[str, str]	Public
"_pystats"	*_pystats*	bool	Read-only

Visibility:

- Public: Can by get by *PyConfig_Get()* and set by *PyConfig_Set()*.
- Read-only: Can by get by *PyConfig_Get()*, but cannot be set by *PyConfig_Set()*.

11.3 Runtime Python configuration API

At runtime, it's possible to get and set configuration options using *PyConfig_Get()* and *PyConfig_Set()* functions.

The configuration option *name* parameter must be a non-NULL null-terminated UTF-8 encoded string. See *Configuration Options*.

Some options are read from the `sys` attributes. For example, the option `"argv"` is read from `sys.argv`.

PyObject ***PyConfig_Get**(const char *name)

Get the current runtime value of a configuration option as a Python object.

- Return a new reference on success.
- Set an exception and return `NULL` on error.

The object type depends on the configuration option. It can be:

- `bool`
- `int`
- `str`
- `list[str]`
- `dict[str, str]`

The caller must have an *attached thread state*. The function cannot be called before Python initialization nor after Python finalization.

Added in version 3.14.

int **PyConfig_GetInt**(const char *name, int *value)

Similar to *PyConfig_Get()*, but get the value as a C int.

- Return `0` on success.
- Set an exception and return `-1` on error.

Added in version 3.14.

PyObject ***PyConfig_Names**(void)

Get all configuration option names as a `frozenset`.

- Return a new reference on success.
- Set an exception and return `NULL` on error.

The caller must have an *attached thread state*. The function cannot be called before Python initialization nor after Python finalization.

Added in version 3.14.

int **PyConfig_Set**(const char *name, *PyObject* *value)

Set the current runtime value of a configuration option.

- Raise a `ValueError` if there is no option *name*.
- Raise a `ValueError` if *value* is an invalid value.
- Raise a `ValueError` if the option is read-only (cannot be set).
- Raise a `TypeError` if *value* has not the proper type.

The caller must have an *attached thread state*. The function cannot be called before Python initialization nor after Python finalization.

Raises an auditing event `cpython.PyConfig_Set` with arguments `name`, `value`.

Added in version 3.14.

11.4 PyConfig C API

Added in version 3.8.

Python se puede inicializar con `Py_InitializeFromConfig()` y la estructura `PyConfig`. Se puede preinicializar con `Py_PreInitialize()` y la estructura `PyPreConfig`.

Hay dos tipos de configuración:

- The *Python Configuration* can be used to build a customized Python which behaves as the regular Python. For example, environment variables and command line arguments are used to configure Python.
- The *Isolated Configuration* can be used to embed Python into an application. It isolates Python from the system. For example, environment variables are ignored, the LC_CTYPE locale is left unchanged and no signal handler is registered.

La función `Py_RunMain()` se puede utilizar para escribir un programa Python personalizado.

Consulte también *Inicialización, finalización y subprocesos*.

Ver también

PEP 587 «Configuración de inicialización de Python».

11.4.1 Ejemplo

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 exception;
    }

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

    return Py_RunMain();

exception:
    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);
}
```

11.4.2 PyWideStringList

type **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 must be greater than or equal to `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.

11.4.3 PyStatus

type **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.

err_msg no debe ser `NULL`.

PyStatus **PyStatus_NoMemory**(void)

Error de asignación de memoria (sin memoria).

PyStatus **PyStatus_Exit**(int exitcode)

Sale 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();
    }
    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;
}
```

11.4.4 PyPreConfig

type **PyPreConfig**

Estructura utilizada para preinicializar Python.

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 de los asignadores de memoria de Python:

- `PYMEM_ALLOCATOR_NOT_SET` (`0`): no cambia asignadores de memoria (usa 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`): usa `malloc()` de la biblioteca C.
- `PYMEM_ALLOCATOR_MALLOC_DEBUG` (`4`): fuerza el uso de `malloc()` con *ganchos de depuración*.
- `PYMEM_ALLOCATOR_PYMALLOC` (`5`): *asignador de memoria pymalloc de Python*
- `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` (`6`): *asignador de memoria pymalloc de Python* con *ganchos de depuración*.
- `PYMEM_ALLOCATOR_MIMALLOC` (`6`): use `mimalloc`, a fast malloc replacement.
- `PYMEM_ALLOCATOR_MIMALLOC_DEBUG` (`7`): use `mimalloc`, a fast malloc replacement with *debug hooks*.

`PYMEM_ALLOCATOR_PYMALLOC` y `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` no son compatibles si Python es `configurado usando --without-pymalloc`.

`PYMEM_ALLOCATOR_MIMALLOC` and `PYMEM_ALLOCATOR_MIMALLOC_DEBUG` are not supported if Python is `configured using --without-mimalloc` or if the underlying atomic support isn't available.

Ver *Administración de memorias*.

Predeterminado: `PYMEM_ALLOCATOR_NOT_SET`.

int **configure_locale**

Set the LC_CTYPE locale to the user preferred locale.

If equals to `0`, set *coerce_c_locale* and *coerce_c_locale_warn* members to `0`.

Vea el *locale encoding*.

Predeterminado: `1` en la configuración de Python, `0` en la configuración aislada.

int **coerce_c_locale**

If equals to `2`, coerce the C locale.

If equals to `1`, read the LC_CTYPE locale to decide if it should be coerced.

Vea el *locale encoding*.

Predeterminado: `-1` en la configuración de Python, `0` en la configuración aislada.

int **coerce_c_locale_warn**

Si no es cero, emita una advertencia si la configuración regional C está coaccionada.

Predeterminado: `-1` en la configuración de Python, `0` en la configuración aislada.

int **dev_mode**

Python Development Mode: see *PyConfig.dev_mode*.

Por defecto: `-1` en modo Python, `0` en modo aislado.

int **isolated**

Modo aislado: consulte *PyConfig.isolated*.

Por defecto: `0` en modo Python, `1` en modo aislado.

int **legacy_windows_fs_encoding**

Si no es cero:

- Establezca *PyPreConfig.utf8_mode* en `0`,

- Establezca *PyConfig.filesystem_encoding* en "mbcs",
- Establezca *PyConfig.filesystem_errors* en "replace".

Initialized from the PYTHONLEGACYWINDOWSFSENCODING environment variable value.

Solo disponible en Windows. La macro #ifdef MS_WINDOWS se puede usar para el código específico de Windows.

Predeterminado: 0.

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.

Predeterminado: 1 en la configuración de Python, 0 en la configuración aislada.

int **use_environment**

¿Utiliza variables de entorno? Consulte *PyConfig.use_environment*.

Predeterminado: 1 en la configuración de Python y 0 en la configuración aislada.

int **utf8_mode**

Si es distinto de cero, habilite Python UTF-8 Mode.

Set to 0 or 1 by the -X utf8 command line option and the PYTHONUTF8 environment variable.

También se pone a 1 si la configuración regional LC_CTYPE es C o POSIX.

Predeterminado: -1 en la configuración de Python y 0 en la configuración aislada.

11.4.5 Preinicialización de Python con PyPreConfig

La preinicialización de Python:

- Establecer los asignadores de memoria de Python (*PyPreConfig.allocator*)
- Configurar la configuración regional LC_CTYPE (*locale encoding*)
- Establecer el Python UTF-8 Mode (*PyPreConfig.utf8_mode*)

La preconfiguración actual (tipo PyPreConfig) se almacena en _PyRuntime.preconfig.

Funciones para preinicializar Python:

PyStatus **Py_PreInitialize**(const *PyPreConfig* *preconfig)

Preinicializa Python desde la preconfiguración *preconfig*.

preconfig no debe ser NULL.

PyStatus **Py_PreInitializeFromBytesArgs**(const *PyPreConfig* *preconfig, int argc, char *const *argv)

Preinicializa Python desde la preconfiguración *preconfig*.

Analice los argumentos de la línea de comando *argv* (cadenas de bytes) si *parse_argv* de *preconfig* no es cero.

preconfig no debe ser NULL.

PyStatus **Py_PreInitializeFromArgs**(const *PyPreConfig* *preconfig, int argc, wchar_t *const *argv)

Preinicializa Python desde la preconfiguración *preconfig*.

Analice los argumentos de la línea de comando *argv* (cadenas anchas) si *parse_argv* de *preconfig* no es cero.

preconfig no debe ser NULL.

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 la 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 Python UTF-8 Mode.

`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. No se debe llamar a *Py_DecodeLocale()* antes de la preinicialización de Python.

Ejemplo usando la preinicialización para habilitar el Python UTF-8 Mode

```
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 speaks UTF-8 */

Py_Initialize();
/* ... use Python API here ... */
Py_Finalize();
```

11.4.6 PyConfig

type **PyConfig**

Estructura que contiene la mayoría de los parámetros para configurar Python.

Cuando termine, se debe utilizar la función *PyConfig_Clear()* para liberar la memoria de configuración.

Métodos de estructura:

void **PyConfig_InitPythonConfig**(*PyConfig* *config)

Inicialice la configuración con la *Configuración de Python*.

void **PyConfig_InitIsolatedConfig**(*PyConfig* *config)

Inicialice la configuración con la *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)

Decodifique *str* usando *Py_DecodeLocale()* y establezca el resultado en `*config_str`.

Preinicializa Python si es necesario.

PyStatus **PyConfig_SetArgv**(*PyConfig* *config, int argc, wchar_t *const *argv)

Configure los argumentos de la línea de comando (miembro *argv* de *config*) de la lista *argv* 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 (miembro `argv` de *config*) de la lista *argv* de cadenas de bytes. Decodifica 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.

Los campos para la *configuración de ruta* ya no se calculan ni modifican al llamar a esta función, a partir de Python 3.11.

The `PyConfig_Read()` function only parses `PyConfig.argv` arguments once: `PyConfig.parse_argv` is set to `2` after arguments are parsed. Since Python arguments are stripped from `PyConfig.argv`, parsing arguments twice would parse the application options as Python options.

Preinicializa Python si es necesario.

Distinto en la versión 3.10: Los argumentos `PyConfig.argv` ahora solo se analizan una vez, `PyConfig.parse_argv` se establece en `2` después de analizar los argumentos y los argumentos solo se analizan si `PyConfig.parse_argv` es igual a `1`.

Distinto en la versión 3.11: `PyConfig_Read()` ya no calcula todas las rutas, por lo que los campos listados en *Python Path Configuration* ya no pueden ser actualizados hasta que se llame a `Py_InitializeFromConfig()`.

void **PyConfig_Clear**(*PyConfig* *config)

Libera memoria de configuración.

La mayoría de los método `PyConfig` *preinitializan Python* si es necesario. En ese caso, la configuración de preinicialización de Python (`PyPreConfig`) se basa en `PyConfig`. Si se ajustan los campos de configuración que son comunes con `PyPreConfig`, deben establecerse antes de llamar a un método `PyConfig`:

- `PyConfig.dev_mode`
- `PyConfig.isolated`
- `PyConfig.parse_argv`
- `PyConfig.use_environment`

Moreover, if `PyConfig_SetArgv()` or `PyConfig_SetBytesArgv()` is used, this method must be called before other methods, since the preinitialization configuration depends on command line arguments (if `parse_argv` is non-zero).

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**

Set `sys.argv` command line arguments based on `argv`. These parameters are similar to those passed to the program's `main()` function with the difference that the first entry should refer to the script file to be executed rather than the executable hosting the Python interpreter. If there isn't a script that will be run, the first entry in `argv` can be an empty string.

Configure `parse_argv` en `1` para analizar `argv` de la misma manera que Python normal analiza los argumentos de la línea de comandos de Python y luego quita los argumentos de Python de `argv`.

Si *argv* está vacío, se agrega una cadena vacía para garantizar que `sys.argv` siempre exista y nunca esté vacío.

Valor predeterminado: `NULL`.

Consulte también el miembro *orig_argv*.

int **safe_path**

Si es igual a cero, `Py_RunMain()` agrega una ruta potencialmente insegura a `sys.path` al inicio:

- Si *argv[0]* es igual a `L"-m"` (`python -m module`), añade el directorio de trabajo actual.
- If running a script (`python script.py`), prepend the script's directory. If it's a symbolic link, resolve symbolic links.
- En caso contrario (`python -c code` and `python`), añade una cadena vacía, que significa el directorio de trabajo actual.

Set to `1` by the `-P` command line option and the `PYTHONSAFEPATH` environment variable.

Default: `0` in Python config, `1` in isolated config.

Added in version 3.11.

wchar_t ***base_exec_prefix**

`sys.base_exec_prefix`.

Valor predeterminado: `NULL`.

Parte de la salida *Python Path Configuration*.

See also *PyConfig.exec_prefix*.

wchar_t ***base_executable**

Ejecutable base de Python: `sys._base_executable`.

Establecido por la variable de entorno `__PYVENV_LAUNCHER__`.

Establecido desde *PyConfig.executable* si `NULL`.

Valor predeterminado: `NULL`.

Parte de la salida *Python Path Configuration*.

See also *PyConfig.executable*.

wchar_t ***base_prefix**

`sys.base_prefix`.

Valor predeterminado: `NULL`.

Parte de la salida *Python Path Configuration*.

See also *PyConfig.prefix*.

int **buffered_stdio**

If equals to `0` and *configure_c_stdio* is non-zero, disable buffering on the C streams stdout and stderr.

Set to `0` by the `-u` command line option and the `PYTHONUNBUFFERED` environment variable.

stdin siempre se abre en modo de búfer.

Predeterminado: `1`.

int **bytes_warning**

If equals to `1`, issue a warning when comparing `bytes` or `bytearray` with `str`, or comparing `bytes` with `int`.

If equal or greater to `2`, raise a `BytesWarning` exception in these cases.

Incrementado por la opción de línea de comando `-b`.

Predeterminado: `0`.

int **warn_default_encoding**

Si no es cero, emite una advertencia `EncodingWarning` cuando `io.TextIOWrapper` usa su codificación predeterminada. Consulte io-encoding-warning para obtener más detalles.

Predeterminado: `0`.

Added in version 3.10.

int **code_debug_ranges**

If equals to `0`, disables the inclusion of the end line and column mappings in code objects. Also disables traceback printing carets to specific error locations.

Set to `0` by the `PYTHONNODEBUGRANGES` environment variable and by the `-X no_debug_ranges` command line option.

Predeterminado: `1`.

Added in version 3.11.

wchar_t ***check_hash_pycs_mode**

Controla el comportamiento de validación de archivos `.pyc` basados en hash: valor de la opción de línea de comando `--check-hash-based-pycs`.

Valores válidos:

- `L"always"`: Calcula el hash el archivo fuente para invalidación independientemente del valor de la marca "check_source".
- `L"never"`: suponga que los pycs basados en hash siempre son válidos.
- `L"default"`: El indicador "check_source" en pycs basados en hash determina la invalidación.

Predeterminado: `L"default"`.

Consulte también **PEP 552** «Pycs deterministas».

int **configure_c_stdio**

Si es distinto de cero, configure los flujos estándar de C:

- En Windows, configure el modo binario (`O_BINARY`) en stdin, stdout y stderr.
- Si *buffered_stdio* es igual a cero, deshabilite el almacenamiento en búfer de los flujos stdin, stdout y stderr.
- Si *interactive* no es cero, habilite el almacenamiento en búfer de flujo en stdin y stdout (solo stdout en Windows).

Predeterminado: `1` en la configuración de Python, `0` en la configuración aislada.

int **dev_mode**

Si es distinto de cero, habilita Modo de desarrollo de Python.

Set to `1` by the `-X dev` option and the `PYTHONDEVMODE` environment variable.

Por defecto: `-1` en modo Python, `0` en modo aislado.

int **dump_refs**

Dump Python references?

Si no es cero, volcar todos los objetos que aún están vivos en la salida.

Establecido en `1` por la variable de entorno `PYTHONDUMPREFS`.

Needs a special build of Python with the `Py_TRACE_REFS` macro defined: see the `configure --with-trace-refs option`.

Predeterminado: `0`.

wchar_t ***dump_refs_file**

Filename where to dump Python references.

Set by the `PYTHONDUMPREFSFILE` environment variable.

Valor predeterminado: `NULL`.

Added in version 3.11.

wchar_t ***exec_prefix**

El prefijo de directorio específico del sitio donde se instalan los archivos Python dependientes de la plataforma: `sys.exec_prefix`.

Valor predeterminado: `NULL`.

Parte de la salida *Python Path Configuration*.

See also *`PyConfig.base_exec_prefix`*.

wchar_t ***executable**

La ruta absoluta del binario ejecutable para el intérprete de Python: `sys.executable`.

Valor predeterminado: `NULL`.

Parte de la salida *Python Path Configuration*.

See also *`PyConfig.base_executable`*.

int **faulthandler**

¿Habilitar administrador de fallas?

Si no es cero, llama a `faulthandler.enable()` al inicio.

Establecido en `1` por `-X faulthandler` y la variable de entorno `PYTHONFAULTHANDLER`.

Por defecto: `-1` en modo Python, `0` en modo aislado.

wchar_t ***filesystem_encoding**

Codificación del sistema de archvios: `sys.getfilesystemencoding()`.

En macOS, Android y VxWorks: use `"utf-8"` de forma predeterminada.

En Windows: utilice `"utf-8"` de forma predeterminada o `"mbcs"` si *`legacy_windows_fs_encoding`* de *`PyPreConfig`* no es cero.

Codificación predeterminada en otras plataformas:

- `"utf-8"` si *`PyPreConfig.utf8_mode`* es distinto de cero.
- `"ascii"` if Python detects that `nl_langinfo(CODESET)` announces the ASCII encoding, whereas the `mbstowcs()` function decodes from a different encoding (usually Latin1).
- `"utf-8"` si `nl_langinfo(CODESET)` retorna una cadena vacía.
- De lo contrario, utilice el resultado *locale encoding*: `nl_langinfo(CODESET)`.

Al inicio de Python, el nombre de codificación se normaliza al nombre del códec de Python. Por ejemplo, `"ANSI_X3.4-1968"` se reemplaza por `"ascii"`.

Consulte también el miembro *`filesystem_errors`*.

wchar_t ***filesystem_errors**

Manejador de errores del sistema de archivos: `sys.getfilesystemencodeerrors()`.

En Windows: utilice `"surrogatepass"` de forma predeterminada o `"replace"` si *`legacy_windows_fs_encoding`* de *`PyPreConfig`* no es cero.

En otras plataformas: utilice `"surrogateescape"` de forma predeterminada.

Controladores de errores admitidos:

- `"strict"`

- `"surrogateescape"`
- `"surrogatepass"` (solo compatible con la codificación UTF-8)

Consulte también el miembro *filesystem_encoding*.

int **use_frozen_modules**

If non-zero, use frozen modules.

Set by the `PYTHON_FROZEN_MODULES` environment variable.

Default: `1` in a release build, or `0` in a debug build.

unsigned long **hash_seed**

int **use_hash_seed**

Funciones de semillas aleatorias hash.

Si *use_hash_seed* es cero, se elige una semilla al azar en el inicio de Python y se ignora *hash_seed*.

Establecido por la variable de entorno `PYTHONHASHSEED`.

Valor predeterminado de *use_hash_seed*: `-1` en modo Python, `0` en modo aislado.

wchar_t ***home**

Set the default Python «home» directory, that is, the location of the standard Python libraries (see `PYTHONHOME`).

Establecido por la variable de entorno `PYTHONHOME`.

Valor predeterminado: `NULL`.

Parte de la entrada *Python Path Configuration*.

int **import_time**

If `1`, profile import time. If `2`, include additional output that indicates when an imported module has already been loaded.

Set by the `-X importtime` option and the `PYTHONPROFILEIMPORTTIME` environment variable.

Predeterminado: `0`.

Distinto en la versión 3.14: Added support for `import_time = 2`

int **inspect**

Ingresa al modo interactivo después de ejecutar un script o un comando.

If greater than `0`, enable inspect: when a script is passed as first argument or the -c option is used, enter interactive mode after executing the script or the command, even when `sys.stdin` does not appear to be a terminal.

Incrementado por la opción de línea de comando `-i`. Establecido en `1` si la variable de entorno `PYTHONINSPECT` no está vacía.

Predeterminado: `0`.

int **install_signal_handlers**

¿Instalar controladores de señales de Python?

Por defecto: `1` en modo Python, `0` en modo aislado.

int **interactive**

If greater than `0`, enable the interactive mode (REPL).

Incrementado por la opción de línea de comando `-i`.

Predeterminado: `0`.

int **int_max_str_digits**

Configures the integer string conversion length limitation. An initial value of `-1` means the value will be taken from the command line or environment or otherwise default to 4300 (`sys.int_info.default_max_str_digits`). A value of `0` disables the limitation. Values greater than zero but less than 640 (`sys.int_info.str_digits_check_threshold`) are unsupported and will produce an error.

Configured by the `-X int_max_str_digits` command line flag or the `PYTHONINTMAXSTRDIGITS` environment variable.

Default: `-1` in Python mode. 4300 (`sys.int_info.default_max_str_digits`) in isolated mode.

Added in version 3.12.

int **cpu_count**

If the value of *cpu_count* is not `-1` then it will override the return values of `os.cpu_count()`, `os.process_cpu_count()`, and `multiprocessing.cpu_count()`.

Configured by the `-X cpu_count=`*n*`|default` command line flag or the `PYTHON_CPU_COUNT` environment variable.

Default: `-1`.

Added in version 3.13.

int **isolated**

If greater than `0`, enable isolated mode:

- Set *safe_path* to `1`: don't prepend a potentially unsafe path to `sys.path` at Python startup, such as the current directory, the script's directory or an empty string.
- Set *use_environment* to `0`: ignore `PYTHON` environment variables.
- Set *user_site_directory* to `0`: don't add the user site directory to `sys.path`.
- Python REPL no importa `readline` ni habilita la configuración predeterminada de readline en mensajes interactivos.

Set to `1` by the `-I` command line option.

Por defecto: `0` en modo Python, `1` en modo aislado.

See also the *Isolated Configuration* and *PyPreConfig.isolated*.

int **legacy_windows_stdio**

If non-zero, use `io.FileIO` instead of `io._WindowsConsoleIO` for `sys.stdin`, `sys.stdout` and `sys.stderr`.

Establecido en `1` si la variable de entorno `PYTHONLEGACYWINDOWSSTDIO` está establecida en una cadena no vacía.

Solo disponible en Windows. La macro `#ifdef MS_WINDOWS` se puede usar para el código específico de Windows.

Predeterminado: `0`.

Consulte también **PEP 528** (Cambiar la codificación de la consola de Windows a UTF-8).

int **malloc_stats**

Si no es cero, volcar las estadísticas en *Asignador de memoria Python pymalloc* en la salida.

Establecido en `1` por la variable de entorno `PYTHONMALLOCSTATS`.

La opción se ignora si Python es `configurado usando la opción --without-pymalloc`.

Predeterminado: `0`.

wchar_t ***platlibdir**

Nombre del directorio de la biblioteca de la plataforma: `sys.platlibdir`.

Establecido por la variable de entorno `PYTHONPLATLIBDIR`.

Default: value of the `PLATLIBDIR` macro which is set by the `configure --with-platlibdir option` (default: `"lib"`, or `"DLLs"` on Windows).

Parte de la entrada *Python Path Configuration*.

Added in version 3.9.

Distinto en la versión 3.11: Esta macro se utiliza ahora en Windows para localizar los módulos de extensión de la biblioteca estándar, normalmente en `DLLs`. Sin embargo, por compatibilidad, tenga en cuenta que este valor se ignora para cualquier disposición no estándar, incluyendo las construcciones dentro del árbol y los entornos virtuales.

wchar_t ***pythonpath_env**

Module search paths (`sys.path`) as a string separated by `DELIM` (`os.pathsep`).

Establecido por la variable de entorno `PYTHONPATH`.

Valor predeterminado: `NULL`.

Parte de la entrada *Python Path Configuration*.

PyWideStringList **module_search_paths**

int **module_search_paths_set**

Rutas de búsqueda del módulo: `sys.path`.

If *module_search_paths_set* is equal to `0`, *Py_InitializeFromConfig()* will replace *module_search_paths* and sets *module_search_paths_set* to `1`.

Por defecto: lista vacía (`module_search_paths`) y `0` (`module_search_paths_set`).

Parte de la salida *Python Path Configuration*.

int **optimization_level**

Nivel de optimización de compilación:

- `0`: Optimizador de mirilla, configure `__debug__` en `True`.
- `1`: Nivel 0, elimina las aserciones, establece `__debug__` en `False`.
- `2`: Nivel 1, elimina docstrings.

Incrementado por la opción de línea de comando `-O`. Establecido en el valor de la variable de entorno `PYTHONOPTIMIZE`.

Predeterminado: `0`.

PyWideStringList **orig_argv**

La lista de los argumentos originales de la línea de comandos pasados al ejecutable de Python: `sys.orig_argv`.

Si la lista *orig_argv* está vacía y *argv* no es una lista que solo contiene una cadena vacía, *PyConfig_Read()* copia *argv* en *orig_argv* antes de modificar *argv* (si *parse_argv* no es cero).

Consulte también el miembro *argv* y la función *Py_GetArgcArgv()*.

Predeterminado: lista vacía.

Added in version 3.10.

int **parse_argv**

¿Analizar los argumentos de la línea de comando?

Si es igual a `1`, analiza *argv* de la misma forma que Python normal analiza argumentos de línea de comando y elimina los argumentos de Python de *argv*.

The *PyConfig_Read()* function only parses *PyConfig.argv* arguments once: *PyConfig.parse_argv* is set to 2 after arguments are parsed. Since Python arguments are stripped from *PyConfig.argv*, parsing arguments twice would parse the application options as Python options.

Por defecto: 1 en modo Python, 0 en modo aislado.

Distinto en la versión 3.10: Los argumentos *PyConfig.argv* ahora solo se analizan si *PyConfig.parse_argv* es igual a 1.

int **parser_debug**

Parser debug mode. If greater than 0, turn on parser debugging output (for expert only, depending on compilation options).

Incrementado por la opción de línea de comando -d. Establecido en el valor de la variable de entorno PYTHONDEBUG.

Needs a debug build of Python (the Py_DEBUG macro must be defined).

Predeterminado: 0.

int **pathconfig_warnings**

If non-zero, calculation of path configuration is allowed to log warnings into stderr. If equals to 0, suppress these warnings.

Por defecto: 1 en modo Python, 0 en modo aislado.

Parte de la entrada *Python Path Configuration*.

Distinto en la versión 3.11: Ahora también se aplica en Windows.

wchar_t ***prefix**

El prefijo de directorio específico del sitio donde se instalan los archivos Python independientes de la plataforma: sys.prefix.

Valor predeterminado: NULL.

Parte de la salida *Python Path Configuration*.

See also *PyConfig.base_prefix*.

wchar_t ***program_name**

Nombre del programa utilizado para inicializar *executable* y en los primeros mensajes de error durante la inicialización de Python.

- En macOS, usa la variable de entorno PYTHONEXECUTABLE si está configurada.
- Si se define la macro WITH_NEXT_FRAMEWORK, utiliza la variable de entorno __PYVENV_LAUNCHER__ si está configurada.
- Utiliza argv[0] de *argv* si está disponible y no está vacío.
- De lo contrario, utiliza L"python" en Windows o L"python3" en otras plataformas.

Valor predeterminado: NULL.

Parte de la entrada *Python Path Configuration*.

wchar_t ***pycache_prefix**

Directorio donde se escriben los archivos .pyc almacenados en caché: sys.pycache_prefix.

Set by the -X pycache_prefix=PATH command line option and the PYTHONPYCACHEPREFIX environment variable. The command-line option takes precedence.

Si NULL, sys.pycache_prefix es establecido a None.

Valor predeterminado: NULL.

int **quiet**

Quiet mode. If greater than `0`, don't display the copyright and version at Python startup in interactive mode.

Incrementado por la opción de línea de comando `-q`.

Predeterminado: `0`.

wchar_t ***run_command**

Valor de la opción de línea de comando `-c`.

Usado por *Py_RunMain()*.

Valor predeterminado: `NULL`.

wchar_t ***run_filename**

Filename passed on the command line: trailing command line argument without `-c` or `-m`. It is used by the *Py_RunMain()* function.

For example, it is set to `script.py` by the `python3 script.py arg` command line.

See also the *PyConfig.skip_source_first_line* option.

Valor predeterminado: `NULL`.

wchar_t ***run_module**

Valor de la opción de línea de comando `-m`.

Usado por *Py_RunMain()*.

Valor predeterminado: `NULL`.

wchar_t ***run_presite**

`package.module` path to module that should be imported before `site.py` is run.

Set by the `-X presite=package.module` command-line option and the `PYTHON_PRESITE` environment variable. The command-line option takes precedence.

Needs a debug build of Python (the `Py_DEBUG` macro must be defined).

Valor predeterminado: `NULL`.

int **show_ref_count**

Show total reference count at exit (excluding *immortal* objects)?

Set to `1` by `-X showrefcount` command line option.

Needs a debug build of Python (the `Py_REF_DEBUG` macro must be defined).

Predeterminado: `0`.

int **site_import**

¿Importar el módulo `site` al inicio?

Si es igual a cero, desactive la importación del sitio del módulo y las manipulaciones dependientes del sitio de `sys.path` que conlleva.

También deshabilite estas manipulaciones si el módulo `site` se importa explícitamente más tarde (llame a `site.main()` si desea que se activen).

Establecido en `0` mediante la opción de línea de comando `-S`.

`sys.flags.no_site` is set to the inverted value of *site_import*.

Predeterminado: `1`.

int **skip_source_first_line**

Si no es cero, omita la primera línea de la fuente *PyConfig.run_filename*.

Permite el uso de formas de `#!cmd` que no son Unix. Esto está destinado únicamente a un truco específico de DOS.

Establecido en `1` mediante la opción de línea de comando `-x`.

Predeterminado: `0`.

wchar_t ***stdio_encoding**

wchar_t ***stdio_errors**

Codificación y errores de codificación de `sys.stdin`, `sys.stdout` y `sys.stderr` (pero `sys.stderr` siempre usa el controlador de errores `"backslashreplace"`).

Utilice la variable de entorno `PYTHONIOENCODING` si no está vacía.

Codificación predeterminada:

- `"UTF-8"` si *PyPreConfig.utf8_mode* es distinto de cero.
- De lo contrario, usa el *locale encoding*.

Manejador de errores predeterminado:

- En Windows: use `"surrogateescape"`.
- `"surrogateescape"` si *PyPreConfig.utf8_mode* no es cero o si la configuración regional LC_CTYPE es «C» o «POSIX».
- `"strict"` de lo contrario.

See also *PyConfig.legacy_windows_stdio*.

int **tracemalloc**

¿Habilitar tracemalloc?

Si no es cero, llama a `tracemalloc.start()` al inicio.

Establecido por la opción de línea de comando `-X tracemalloc=N` y por la variable de entorno `PYTHONTRACEMALLOC`.

Por defecto: `-1` en modo Python, `0` en modo aislado.

int **perf_profiling**

Enable the Linux `perf` profiler support?

If equals to `1`, enable support for the Linux `perf` profiler.

If equals to `2`, enable support for the Linux `perf` profiler with DWARF JIT support.

Set to `1` by `-X perf` command-line option and the `PYTHONPERFSUPPORT` environment variable.

Set to `2` by the `-X perf_jit` command-line option and the `PYTHON_PERF_JIT_SUPPORT` environment variable.

Default: `-1`.

Ver también

See perf_profiling for more information.

Added in version 3.12.

wchar_t ***stdlib_dir**

Directory of the Python standard library.

Valor predeterminado: `NULL`.

Added in version 3.11.

int **use_environment**

¿Utiliza variables de entorno?

Si es igual a cero, ignora las variables de entorno.

Set to `0` by the `-E` environment variable.

Predeterminado: `1` en la configuración de Python y `0` en la configuración aislada.

int **use_system_logger**

If non-zero, `stdout` and `stderr` will be redirected to the system log.

Only available on macOS 10.12 and later, and on iOS.

Default: `0` (don't use the system log) on macOS; `1` on iOS (use the system log).

Added in version 3.14.

int **user_site_directory**

Si es distinto de cero, agregue el directorio del sitio del usuario a `sys.path`.

Establecido en `0` por las opciones de línea de comando `-s` y `-I`.

Establecido en `0` por la variable de entorno `PYTHONNOUSERSITE`.

Por defecto: `1` en modo Python, `0` en modo aislado.

int **verbose**

Verbose mode. If greater than `0`, print a message each time a module is imported, showing the place (filename or built-in module) from which it is loaded.

If greater than or equal to `2`, print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Incrementado por la opción de línea de comando `-v`.

Set by the `PYTHONVERBOSE` environment variable value.

Predeterminado: `0`.

PyWideStringList **warnoptions**

Opciones del módulo `warnings` para crear filtros de advertencias, de menor a mayor prioridad: `sys.warnoptions`.

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).

Las opciones de la línea de comando `-W` agregan su valor a *warnoptions*, se puede usar varias veces.

La variable de entorno `PYTHONWARNINGS` también se puede utilizar para agregar opciones de advertencia. Se pueden especificar varias opciones, separadas por comas (`,`).

Predeterminado: lista vacía.

int **write_bytecode**

If equal to `0`, Python won't try to write `.pyc` files on the import of source modules.

Establecido en `0` por la opción de línea de comando `-B` y la variable de entorno `PYTHONDONTWRITEBYTECODE`.

`sys.dont_write_bytecode` se inicializa al valor invertido de *write_bytecode*.

Predeterminado: `1`.

PyWideStringList **xoptions**

Valores de las opciones de la línea de comando `-X`: `sys._xoptions`.

Predeterminado: lista vacía.

int **_pystats**

If non-zero, write performance statistics at Python exit.

Need a special build with the `Py_STATS` macro: see `--enable-pystats`.

Predeterminado: `0`.

Si *parse_argv* no es cero, los argumentos de *argv* se analizan de la misma forma que Python normal analiza argumentos de línea de comando, y los argumentos de Python se eliminan de *argv*.

Las opciones de *xoptions* se analizan para establecer otras opciones: consulte la opción de línea de comando `-X`.

Distinto en la versión 3.9: El campo `show_alloc_count` fue removido.

11.4.7 Inicialización con PyConfig

Initializing the interpreter from a populated configuration struct is handled by calling *Py_InitializeFromConfig()*.

La persona que llama es responsable de manejar las excepciones (error o salida) usando *PyStatus_Exception()* y *Py_ExitStatusException()*.

Si se utilizan *PyImport_FrozenModules()*, *PyImport_AppendInittab()* o *PyImport_ExtendInittab()*, deben establecerse o llamarse después de la preinicialización de Python y antes de la inicialización de Python. Si Python se inicializa varias veces, se debe llamar a *PyImport_AppendInittab()* o *PyImport_ExtendInittab()* antes de cada inicialización de Python.

La configuración actual (tipo `PyConfig`) se almacena en `PyInterpreterState.config`.

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 exception;
    }

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

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

More complete example modifying the default configuration, read the configuration, and then override some parameters. Note that since 3.11, many parameters are not calculated until initialization, and so values cannot be read from the configuration structure. Any values set before initialize is called will be left unchanged by initialization:

```
PyStatus init_python(const char *program_name)
{
    PyStatus status;

    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;
    }

    /* Specify sys.path explicitly */
    /* If you want to modify the default set of paths, finish
       initialization first and then use PySys_GetObject("path") */
    config.module_search_paths_set = 1;
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/stdlib");
    if (PyStatus_Exception(status)) {
        goto done;
    }
    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;
}
```

11.4.8 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 se siguen utilizando con esta configuración para determinar las rutas que no se especifican. Asegúrese de que se especifica *PyConfig.home* para evitar que se calcule la configuración de la ruta por defecto.

11.4.9 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 habilita la coerción de configuración regional C (**PEP 538**) y Python UTF-8 Mode (**PEP 540**) según la configuración regional LC_CTYPE, las variables de entorno `PYTHONUTF8` y `PYTHONCOERCECLOCALE`.

11.4.10 Configuración de la ruta de Python

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 `__PYVENV_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 al menos un «campo de salida» no está establecido, Python calcula la configuración de la ruta para rellenar los campos no establecidos. Si *module_search_paths_set* es igual a `0`, *module_search_paths* se anula y *module_search_paths_set* se establece en `1`.

It is possible to completely ignore the function calculating the default path configuration by setting explicitly all path configuration output fields listed above. A string is considered as set even if it is non-empty. `module_search_paths`

is considered as set if `module_search_paths_set` is set to `1`. In this case, `module_search_paths` will be used without modification.

Set *pathconfig_warnings* to `0` to suppress warnings when calculating the path configuration (Unix only, Windows does not log any warning).

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`
- archivo `._pth` (ej: `python._pth`)
- `pybuilddir.txt` (sólo Unix)

Si un archivo `._pth` está presente:

- Set *isolated* to `1`.
- Set *use_environment* to `0`.
- Set *site_import* to `0`.
- Set *safe_path* to `1`.

If *home* is not set and a `pyvenv.cfg` file is present in the same directory as *executable*, or its parent, *prefix* and *exec_prefix* are set that location. When this happens, *base_prefix* and *base_exec_prefix* still keep their value, pointing to the base installation. See sys-path-init-virtual-environments for more information.

The `__PYVENV_LAUNCHER__` environment variable is used to set *PyConfig.base_executable*.

Distinto en la versión 3.14: *prefix*, and *exec_prefix*, are now set to the `pyvenv.cfg` directory. This was previously done by `site`, therefore affected by `-S`.

11.5 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.

Ver también el miembro *PyConfig.orig_argv*.

11.6 Delaying main module execution

In some embedding use cases, it may be desirable to separate interpreter initialization from the execution of the main module.

This separation can be achieved by setting `PyConfig.run_command` to the empty string during initialization (to prevent the interpreter from dropping into the interactive prompt), and then subsequently executing the desired main module code using `__main__.__dict__` as the global namespace.

CAPÍTULO 12

Gestión de la memoria

12.1 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)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyBytes_FromString(buf);
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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.

12.2 Dominios del asignador

All allocating functions belong to one of three different «domains» (see also `PyMemAllocatorDomain`). These domains represent different allocation strategies and are optimized for different purposes. The specific details on how every domain allocates memory or what internal functions each domain calls is considered an implementation detail, but for debugging purposes a simplified table can be found at *here*. The APIs used to allocate and free a block of memory must be from the same domain. For example, `PyMem_Free()` must be used to free memory allocated using `PyMem_Malloc()`.

Los tres dominios de asignación son:

- Raw domain: intended for allocating memory for general-purpose memory buffers where the allocation *must* go to the system allocator or where the allocator can operate without an *attached thread state*. The memory is requested directly from the system. See *Raw Memory Interface*.
- «Mem» domain: intended for allocating memory for Python buffers and general-purpose memory buffers where the allocation must be performed with an *attached thread state*. The memory is taken from the Python private heap. See *Memory Interface*.
- Object domain: intended for allocating memory for Python objects. The memory is taken from the Python private heap. See *Object allocators*.

Nota

The *free-threaded* build requires that only Python objects are allocated using the «object» domain and that all Python objects are allocated using that domain. This differs from the prior Python versions, where this was only a best practice and not a hard requirement.

For example, buffers (non-Python objects) should be allocated using `PyMem_Malloc()`, `PyMem_RawMalloc()`, or `malloc()`, but not `PyObject_Malloc()`.

See Memory Allocation APIs.

12.3 Interfaz de memoria sin procesar

The following function sets are wrappers to the system allocator. These functions are thread-safe, so a *thread state* does not need to be *attached*.

The *default raw memory allocator* uses the following functions: `malloc()`, `calloc()`, `realloc()` and `free()`; call `malloc(1)` (or `calloc(1, 1)`) when requesting zero bytes.

Added in version 3.4.

void ***PyMem_RawMalloc**(size_t n)

Part of the Stable ABI *since version 3.13.* 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)

Part of the Stable ABI *since version 3.13.* 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)`.

Added in version 3.5.

void ***PyMem_RawRealloc**(void *p, size_t n)

Part of the Stable ABI *since version 3.13.* 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)

Part of the Stable ABI *since version 3.13.* 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.

12.4 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 memorya pymalloc*.

Advertencia

There must be an *attached thread state* when using these functions.

Distinto en la versión 3.6: El asignador predeterminado ahora es pymalloc en lugar del `malloc()` del sistema.

void *__PyMem_Malloc__(size_t n)

Part of the Stable ABI. 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)

Part of the Stable ABI *since version 3.7.* 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)`.

Added in version 3.5.

void *__PyMem_Realloc__(void *p, size_t n)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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.

__PyMem_New__(TYPE, n)

Same as *PyMem_Malloc()*, but allocates `(n * sizeof(TYPE))` bytes of memory. Returns a pointer cast to `TYPE*`. The memory will not have been initialized in any way.

__PyMem_Resize__(p, TYPE, n)

Same as *PyMem_Realloc()*, but the memory block is resized to `(n * sizeof(TYPE))` bytes. Returns a pointer cast to `TYPE*`. On return, *p* will be a pointer to the new memory area, or `NULL` in the event of failure.

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)`

12.5 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.

Nota

No hay garantía de que la memoria retornada por estos asignadores se pueda convertir con éxito en un objeto Python al interceptar las funciones de asignación en este dominio mediante los métodos descritos en la sección *Personalizar Asignadores de Memoria*.

El *asignador predeterminado de objetos* usa el *asignador de memoria pymalloc*.

Advertencia

There must be an *attached thread state* when using these functions.

void ***PyObject_Malloc**(size_t n)

Part of the Stable ABI. 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)

Part of the Stable ABI *since version 3.7.* 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)`.

Added in version 3.5.

void ***PyObject_Realloc**(void *p, size_t n)

Part of the Stable ABI. 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)

Part of the Stable ABI. 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.

Do not call this directly to free an object's memory; call the type's *`tp_free`* slot instead.

Do not use this for memory allocated by *PyObject_GC_New* or *PyObject_GC_NewVar*; use *PyObject_GC_Del()* instead.

Ver también

- *PyObject_GC_Del()* is the equivalent of this function for memory allocated by types that support garbage collection.
- *PyObject_Malloc()*
- *PyObject_Realloc()*
- *PyObject_Calloc()*
- *PyObject_New*
- *PyObject_NewVar*
- *PyType_GenericAlloc()*
- *tp_free*

12.6 Asignadores de memoria predeterminados

Asignadores de memoria predeterminados:

Configuración	Nombre	PyMem_RawMalloc	PyMem_Malloc	PyObject_Malloc
Lanzamiento de compilación	`"pymalloc"`	`malloc`	`malloc` + debug	`malloc` + debug
Compilación de depuración	`"pymalloc_debu`	`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*.
- `mimalloc`: *mimalloc memory allocator*. The pymalloc allocator will be used if mimalloc support isn't available.
- «+ debug»: con *enlaces de depuración en los asignadores de memoria de Python*.
- «Debug build»: Compilación de Python en modo de depuración.

12.7 Personalizar asignadores de memoria

Added in version 3.4.

type **PyMemAllocatorEx**

Estructura utilizada para describir un asignador de bloque de memoria. La estructura tiene cuatro campos:

Campo	Significado
`void *ctx`	contexto de usuario pasado como primer argumento
`void* malloc(void *ctx, size_t size)`	asignar un bloque de memoria
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	asignar un bloque de memoria inicializado con ceros
`void* realloc(void *ctx, void *ptr, size_t new_size)`	asignar o cambiar el tamaño de un bloque de memoria
`void free(void *ctx, void *ptr)`	liberar un bloque de memoria

Distinto en la versión 3.5: The `PyMemAllocator` structure was renamed to *PyMemAllocatorEx* and a new `calloc` field was added.

type **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.

For the *PYMEM_DOMAIN_RAW* domain, the allocator must be thread-safe: a *thread state* is not *attached* when the allocator is called.

For the remaining domains, the allocator must also be thread-safe: the allocator may be called in different interpreters that do not share a *GIL*.

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.

Vea también `PyPreConfig.allocator` y *Preinicialización de Python con PyPreConfig*.

> **Advertencia**
>
> `PyMem_SetAllocator()` does have the following contract:
>
> - It can be called after `Py_PreInitialize()` and before `Py_InitializeFromConfig()` to install a custom memory allocator. There are no restrictions over the installed allocator other than the ones imposed by the domain (for instance, the Raw Domain allows the allocator to be called without an *attached thread state*). See *the section on allocator domains* for more information.
> - If called after Python has finish initializing (after `Py_InitializeFromConfig()` has been called) the allocator **must** wrap the existing allocator. Substituting the current allocator for some other arbitrary one is **not supported**.

Distinto en la versión 3.12: All allocators must be thread-safe.

void **PyMem_SetupDebugHooks**(void)

Configurar *enlaces de depuración en los asignadores de memoria de Python* para detectar errores de memoria.

12.8 Configurar enlaces para detectar errores en las funciones del asignador de memoria de Python

Cuando Python está construido en modo de depuración, la función `PyMem_SetupDebugHooks()` se llama en *Preinicialización de Python* para configurar los enlaces de depuración en Python asignadores de memoria para detectar errores de memoria.

La variable de entorno `PYTHONMALLOC` se puede utilizar para instalar enlaces de depuración en un Python compilado en modo de lanzamiento (por ejemplo: `PYTHONMALLOC=debug`).

La función `PyMem_SetupDebugHooks()` se puede utilizar para establecer enlaces de depuración después de llamar a `PyMem_SetAllocator()`.

Estos enlaces de depuración llenan bloques de memoria asignados dinámicamente con patrones de bits especiales y reconocibles. La memoria recién asignada se llena con el byte `0xCD` (`PYMEM_CLEANBYTE`), la memoria liberada se llena con el byte `0xDD` (`PYMEM_DEADBYTE`). Los bloques de memoria están rodeados por «bytes prohibidos» rellenos con el byte `0xFD` (`PYMEM_FORBIDDENBYTE`). Es poco probable que las cadenas de estos bytes sean direcciones válidas, flotantes o cadenas ASCII.

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)
- Check that there is an *attached thread state* when allocator functions of `PYMEM_DOMAIN_OBJ` (ex: `PyObject_Malloc()`) and `PYMEM_DOMAIN_MEM` (ex: `PyMem_Malloc()`) domains are called.

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.

Sea *S* = `sizeof(size_t)`. Se agregan `2*S` bytes en cada extremo de cada bloque de *N* bytes solicitados. El diseño de la memoria es así, donde p representa la dirección retornada por una función similar a malloc o realloc (`p[i:j]` significa el segmento de bytes de `*(p+i)` inclusive hasta `*(p+j)` exclusivo; tenga en cuenta que el tratamiento de los índices negativos difiere de un segmento de Python):

p[-2*S:-S]

Número de bytes solicitados originalmente. Este es un size_t, big-endian (más fácil de leer en un volcado de memoria).

p[-S]
Identificador de API (carácter ASCII):

- `'r'` for *PYMEM_DOMAIN_RAW*.
- `'m'` for *PYMEM_DOMAIN_MEM*.
- `'o'` for *PYMEM_DOMAIN_OBJ*.

p[-S+1:0]
Copias de PYMEM_FORBIDDENBYTE. Se utiliza para detectar suscripciones y lecturas.

p[0:N]
La memoria solicitada, llena de copias de PYMEM_CLEANBYTE, utilizada para capturar la referencia a la memoria no inicializada. Cuando se llama a una función similar a realloc solicitando un bloque de memoria más grande, los nuevos bytes en exceso también se llenan con PYMEM_CLEANBYTE. Cuando se llama a una función de tipo free, se sobrescriben con PYMEM_DEADBYTE, para captar la referencia a la memoria liberada. Cuando se llama a una función similar a la realloc solicitando un bloque de memoria más pequeño, los bytes antiguos sobrantes también se llenan con PYMEM_DEADBYTE.

p[N:N+S]
Copias de PYMEM_FORBIDDENBYTE. Se utiliza para detectar sobrescrituras y lecturas.

p[N+S:N+2*S]
Solo se utiliza si la macro `PYMEM_DEBUG_SERIALNO` está definida (no definida por defecto).

A serial number, incremented by 1 on each call to a malloc-like or realloc-like function. Big-endian `size_t`. If «bad memory» is detected later, the serial number gives an excellent way to set a breakpoint on the next run, to capture the instant at which this block was passed out. The static function bumpserialno() in obmalloc.c is the only place the serial number is incremented, and exists so you can set such a breakpoint easily.

Una función de tipo realloc o de tipo free primero verifica que los bytes PYMEM_FORBIDDENBYTE en cada extremo estén intactos. Si se han modificado, la salida de diagnóstico se escribe en stderr y el programa se aborta mediante Py_FatalError(). El otro modo de falla principal es provocar un error de memoria cuando un programa lee uno de los patrones de bits especiales e intenta usarlo como una dirección. Si ingresa a un depurador y observa el objeto, es probable que vea que está completamente lleno de PYMEM_DEADBYTE (lo que significa que se está usando la memoria liberada) o PYMEM_CLEANBYTE (que significa que se está usando la memoria no inicializada).

Distinto en la versión 3.6: The *PyMem_SetupDebugHooks()* function now also works on Python compiled in release mode. On error, the debug hooks now use `tracemalloc` to get the traceback where a memory block was allocated. The debug hooks now also check if there is an *attached thread state* when functions of *PYMEM_DOMAIN_OBJ* and *PYMEM_DOMAIN_MEM* domains are called.

Distinto en la versión 3.8: Los patrones de bytes `0xCB` (`PYMEM_CLEANBYTE`), `0xDB` (`PYMEM_DEADBYTE`) y `0xFB` (`PYMEM_FORBIDDENBYTE`) se han reemplazado por `0xCD`, `0xDD` y `0xFD` para usar los mismos valores que la depuración de Windows CRT `malloc()` y `free()`.

12.9 El asignador pymalloc

Python has a *pymalloc* allocator optimized for small objects (smaller or equal to 512 bytes) with a short lifetime. It uses memory mappings called «arenas» with a fixed size of either 256 KiB on 32-bit platforms or 1 MiB on 64-bit platforms. It falls back to *PyMem_RawMalloc()* and *PyMem_RawRealloc()* for allocations larger than 512 bytes.

pymalloc is the *default allocator* of the *PYMEM_DOMAIN_MEM* (ex: *PyMem_Malloc()*) and *PYMEM_DOMAIN_OBJ* (ex: *PyObject_Malloc()*) domains.

El asignador de arena utiliza las siguientes funciones:

- `VirtualAlloc()` and `VirtualFree()` on Windows,
- `mmap()` and `munmap()` if available,
- `malloc()` y `free()` en caso contrario.

Este asignador está deshabilitado si Python está configurado con la opción `--without-pymalloc`. También se puede deshabilitar en tiempo de ejecución usando la variable de entorno `PYTHONMALLOC` (por ejemplo: `PYTHONMALLOC=malloc`).

Typically, it makes sense to disable the pymalloc allocator when building Python with AddressSanitizer (`--with-address-sanitizer`) which helps uncover low level bugs within the C code.

12.9.1 Personalizar asignador de arena de pymalloc

Added in version 3.4.

type **PyObjectArenaAllocator**

Estructura utilizada para describir un asignador de arena. La estructura tiene tres campos:

Campo	Significado
`void *ctx`	contexto de usuario pasado como primer argumento
`void* alloc(void *ctx, size_t size)`	asignar una arena de bytes de tamaño
`void free(void *ctx, void *ptr, size_t size)`	liberar la arena

void **PyObject_GetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

Consigue el asignador de arena.

void **PyObject_SetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

Establecer el asignador de arena.

12.10 The mimalloc allocator

Added in version 3.13.

Python supports the mimalloc allocator when the underlying platform support is available. mimalloc «is a general purpose allocator with excellent performance characteristics. Initially developed by Daan Leijen for the runtime systems of the Koka and Lean languages.»

12.11 tracemalloc C API

Added in version 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`.

12.12 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_Free(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_Free()  */
```

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with *PyObject_New*, *PyObject_NewVar* and *PyObject_Free()*.

Esto se explicará en el próximo capítulo sobre cómo definir e implementar nuevos tipos de objetos en C.

CAPÍTULO 13

Soporte de implementación de objetos

Este capítulo describe las funciones, los tipos y las macros utilizados al definir nuevos tipos de objetos.

13.1 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. Part of the Stable ABI. Initialize a newly allocated object *op* with its type and initial reference. Returns the initialized object. Other fields of the object are not initialized. Despite its name, this function is unrelated to the object's `__init__()` method (*tp_init* slot). Specifically, this function does **not** call the object's `__init__()` method.

In general, consider this function to be a low-level routine. Use *tp_alloc* where possible. For implementing `tp_alloc` for your type, prefer *PyType_GenericAlloc()* or *PyObject_New()*.

> **Nota**
>
> This function only initializes the object's memory corresponding to the initial *PyObject* structure. It does not zero the rest.

PyVarObject ***PyObject_InitVar**(*PyVarObject* *op, *PyTypeObject* *type, *Py_ssize_t* size)

Return value: Borrowed reference. Part of the Stable ABI. Esto hace todo lo que *PyObject_Init()* hace, y también inicializa la información de longitud para un objeto de tamaño variable.

> **Nota**
>
> This function only initializes some of the object's memory. It does not zero the rest.

PyObject_New(TYPE, typeobj)

Allocates a new Python object using the C structure type *TYPE* and the Python type object *typeobj* (`PyTypeObject*`) by calling *PyObject_Malloc()* to allocate memory and initializing it like *PyObject_Init()*. The caller will own the only reference to the object (i.e. its reference count will be one).

Avoid calling this directly to allocate memory for an object; call the type's *tp_alloc* slot instead.

When populating a type's *tp_alloc* slot, *PyType_GenericAlloc()* is preferred over a custom function that simply calls this macro.

This macro does not call *tp_alloc*, *tp_new* (`__new__()`), or *tp_init* (`__init__()`).

This cannot be used for objects with *Py_TPFLAGS_HAVE_GC* set in *tp_flags*; use *PyObject_GC_New* instead.

Memory allocated by this macro must be freed with *PyObject_Free()* (usually called via the object's *tp_free* slot).

Nota

The returned memory is not guaranteed to have been completely zeroed before it was initialized.

Nota

This macro does not construct a fully initialized object of the given type; it merely allocates memory and prepares it for further initialization by *tp_init*. To construct a fully initialized object, call *typeobj* instead. For example:

```
PyObject *foo = PyObject_CallNoArgs((PyObject *)&PyFoo_Type);
```

Ver también

- *PyObject_Free()*
- *PyObject_GC_New*
- *PyType_GenericAlloc()*
- *tp_alloc*

PyObject_NewVar(TYPE, typeobj, size)

Like *PyObject_New* except:

- It allocates enough memory for the *TYPE* structure plus *size* (`Py_ssize_t`) fields of the size given by the *tp_itemsize* field of *typeobj*.
- The memory is initialized like *PyObject_InitVar()*.

This is useful for implementing objects like tuples, which are able to determine their size at construction time. Embedding the array of fields into the same allocation decreases the number of allocations, improving the memory management efficiency.

Avoid calling this directly to allocate memory for an object; call the type's *tp_alloc* slot instead.

When populating a type's *tp_alloc* slot, *PyType_GenericAlloc()* is preferred over a custom function that simply calls this macro.

This cannot be used for objects with *Py_TPFLAGS_HAVE_GC* set in *tp_flags*; use *PyObject_GC_NewVar* instead.

Memory allocated by this function must be freed with *PyObject_Free()* (usually called via the object's *tp_free* slot).

Nota

The returned memory is not guaranteed to have been completely zeroed before it was initialized.

Nota

This macro does not construct a fully initialized object of the given type; it merely allocates memory and prepares it for further initialization by *tp_init*. To construct a fully initialized object, call *typeobj* instead. For example:

```
PyObject *list_instance = PyObject_CallNoArgs((PyObject *)&PyList_Type);
```

Ver también

- *PyObject_Free()*
- *PyObject_GC_NewVar*
- *PyType_GenericAlloc()*
- *tp_alloc*

void **PyObject_Del**(void *op)

Same as *PyObject_Free()*.

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

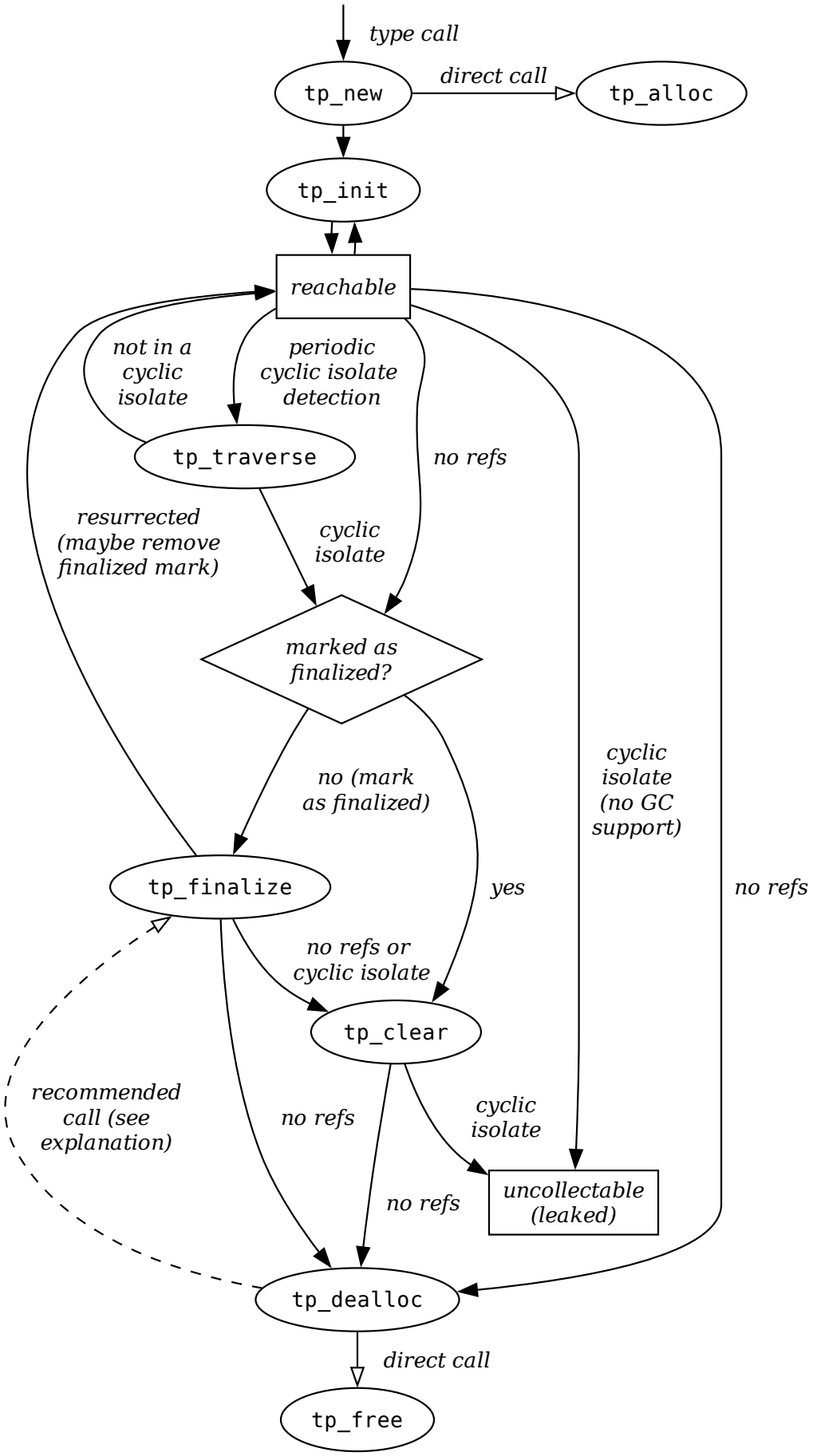
Objetos módulo
Para asignar y crear módulos de extensión.

13.2 Object Life Cycle

This section explains how a type's slots relate to each other throughout the life of an object. It is not intended to be a complete canonical reference for the slots; instead, refer to the slot-specific documentation in *Type Object Structures* for details about a particular slot.

13.2.1 Life Events

The figure below illustrates the order of events that can occur throughout an object's life. An arrow from *A* to *B* indicates that event *B* can occur after event *A* has occurred, with the arrow's label indicating the condition that must be true for *B* to occur after *A*.


type call
tp_new
direct call
tp_alloc
tp_init
reachable
not in a cyclic isolate
periodic cyclic isolate detection
tp_traverse
no refs
resurrected (maybe remove finalized mark)
cyclic isolate
marked as finalized?
no (mark as finalized)
cyclic isolate (no GC support)
tp_finalize
yes
no refs
no refs or cyclic isolate
tp_clear
recommended call (see explanation)
no refs
cyclic isolate
uncollectable (leaked)
no refs
tp_dealloc
direct call
tp_free

Explanation:

- When a new object is constructed by calling its type:
  1. *tp_new* is called to create a new object.
  2. *tp_alloc* is directly called by *tp_new* to allocate the memory for the new object.
  3. *tp_init* initializes the newly created object. `tp_init` can be called again to re-initialize an object, if desired. The `tp_init` call can also be skipped entirely, for example by Python code calling `__new__()`.
- After `tp_init` completes, the object is ready to use.
- Some time after the last reference to an object is removed:
  1. If an object is not marked as *finalized*, it might be finalized by marking it as *finalized* and calling its *tp_finalize* function. Python does *not* finalize an object when the last reference to it is deleted; use *PyObject_CallFinalizerFromDealloc()* to ensure that *tp_finalize* is always called.
  2. If the object is marked as finalized, *tp_clear* might be called by the garbage collector to clear references held by the object. It is *not* called when the object's reference count reaches zero.
  3. *tp_dealloc* is called to destroy the object. To avoid code duplication, *tp_dealloc* typically calls into *tp_clear* to free up the object's references.
  4. When *tp_dealloc* finishes object destruction, it directly calls *tp_free* (usually set to *PyObject_Free()* or *PyObject_GC_Del()* automatically as appropriate for the type) to deallocate the memory.
- The *tp_finalize* function is permitted to add a reference to the object if desired. If it does, the object is *resurrected*, preventing its pending destruction. (Only `tp_finalize` is allowed to resurrect an object; *tp_clear* and *tp_dealloc* cannot without calling into `tp_finalize`.) Resurrecting an object may or may not cause the object's *finalized* mark to be removed. Currently, Python does not remove the *finalized* mark from a resurrected object if it supports garbage collection (i.e., the *Py_TPFLAGS_HAVE_GC* flag is set) but does remove the mark if the object does not support garbage collection; either or both of these behaviors may change in the future.
- *tp_dealloc* can optionally call *tp_finalize* via *PyObject_CallFinalizerFromDealloc()* if it wishes to reuse that code to help with object destruction. This is recommended because it guarantees that `tp_finalize` is always called before destruction. See the *tp_dealloc* documentation for example code.
- If the object is a member of a *cyclic isolate* and either *tp_clear* fails to break the reference cycle or the cyclic isolate is not detected (perhaps `gc.disable()` was called, or the *Py_TPFLAGS_HAVE_GC* flag was erroneously omitted in one of the involved types), the objects remain indefinitely uncollectable (they «leak»). See `gc.garbage`.

If the object is marked as supporting garbage collection (the *Py_TPFLAGS_HAVE_GC* flag is set in *tp_flags*), the following events are also possible:

- The garbage collector occasionally calls *tp_traverse* to identify *cyclic isolates*.
- When the garbage collector discovers a *cyclic isolate*, it finalizes one of the objects in the group by marking it as *finalized* and calling its *tp_finalize* function, if it has one. This repeats until the cyclic isolate doesn't exist or all of the objects have been finalized.
- *tp_finalize* is permitted to resurrect the object by adding a reference from outside the *cyclic isolate*. The new reference causes the group of objects to no longer form a cyclic isolate (the reference cycle may still exist, but if it does the objects are no longer isolated).
- When the garbage collector discovers a *cyclic isolate* and all of the objects in the group have already been marked as *finalized*, the garbage collector clears one or more of the uncleared objects in the group (possibly concurrently) by calling each's *tp_clear* function. This repeats as long as the cyclic isolate still exists and not all of the objects have been cleared.

13.2.2 Cyclic Isolate Destruction

Listed below are the stages of life of a hypothetical *cyclic isolate* that continues to exist after each member object is finalized or cleared. It is a memory leak if a cyclic isolate progresses through all of these stages; it should vanish once all objects are cleared, if not sooner. A cyclic isolate can vanish either because the reference cycle is broken or because the objects are no longer isolated due to finalizer resurrection (see `tp_finalize`).

0. **Reachable** (not yet a cyclic isolate): All objects are in their normal, reachable state. A reference cycle could exist, but an external reference means the objects are not yet isolated.
1. **Unreachable but consistent:** The final reference from outside the cyclic group of objects has been removed, causing the objects to become isolated (thus a cyclic isolate is born). None of the group's objects have been finalized or cleared yet. The cyclic isolate remains at this stage until some future run of the garbage collector (not necessarily the next run because the next run might not scan every object).
2. **Mix of finalized and not finalized:** Objects in a cyclic isolate are finalized one at a time, which means that there is a period of time when the cyclic isolate is composed of a mix of finalized and non-finalized objects. Finalization order is unspecified, so it can appear random. A finalized object must behave in a sane manner when non-finalized objects interact with it, and a non-finalized object must be able to tolerate the finalization of an arbitrary subset of its referents.
3. **All finalized:** All objects in a cyclic isolate are finalized before any of them are cleared.
4. **Mix of finalized and cleared:** The objects can be cleared serially or concurrently (but with the *GIL* held); either way, some will finish before others. A finalized object must be able to tolerate the clearing of a subset of its referents. **PEP 442** calls this stage «cyclic trash».
5. **Leaked:** If a cyclic isolate still exists after all objects in the group have been finalized and cleared, then the objects remain indefinitely uncollectable (see `gc.garbage`). It is a bug if a cyclic isolate reaches this stage—it means the `tp_clear` methods of the participating objects have failed to break the reference cycle as required.

If `tp_clear` did not exist, then Python would have no way to safely break a reference cycle. Simply destroying an object in a cyclic isolate would result in a dangling pointer, triggering undefined behavior when an object referencing the destroyed object is itself destroyed. The clearing step makes object destruction a two-phase process: first `tp_clear` is called to partially destroy the objects enough to detangle them from each other, then `tp_dealloc` is called to complete the destruction.

Unlike clearing, finalization is not a phase of destruction. A finalized object must still behave properly by continuing to fulfill its design contracts. An object's finalizer is allowed to execute arbitrary Python code, and is even allowed to prevent the impending destruction by adding a reference. The finalizer is only related to destruction by call order—if it runs, it runs before destruction, which starts with `tp_clear` (if called) and concludes with `tp_dealloc`.

The finalization step is not necessary to safely reclaim the objects in a cyclic isolate, but its existence makes it easier to design types that behave in a sane manner when objects are cleared. Clearing an object might necessarily leave it in a broken, partially destroyed state—it might be unsafe to call any of the cleared object's methods or access any of its attributes. With finalization, only finalized objects can possibly interact with cleared objects; non-finalized objects are guaranteed to interact with only non-cleared (but potentially finalized) objects.

To summarize the possible interactions:

- A non-finalized object might have references to or from non-finalized and finalized objects, but not to or from cleared objects.
- A finalized object might have references to or from non-finalized, finalized, and cleared objects.
- A cleared object might have references to or from finalized and cleared objects, but not to or from non-finalized objects.

Without any reference cycles, an object can be simply destroyed once its last reference is deleted; the finalization and clearing steps are not necessary to safely reclaim unused objects. However, it can be useful to automatically call `tp_finalize` and `tp_clear` before destruction anyway because type design is simplified when all objects always experience the same series of events regardless of whether they participated in a cyclic isolate. Python currently only calls `tp_finalize` and `tp_clear` as needed to destroy a cyclic isolate; this may change in a future version.

13.2.3 Functions

To allocate and free memory, see *Asignación de objetos en el montículo*.

void **PyObject_CallFinalizer**(*PyObject* *op)

Finalizes the object as described in `tp_finalize`. Call this function (or `PyObject_CallFinalizerFromDealloc()`) instead of calling `tp_finalize` directly because this function may deduplicate multiple calls to `tp_finalize`. Currently, calls are only deduplicated if the type supports garbage collection (i.e., the `Py_TPFLAGS_HAVE_GC` flag is set); this may change in the future.

int **PyObject_CallFinalizerFromDealloc**(*PyObject* *op)

Same as `PyObject_CallFinalizer()` but meant to be called at the beginning of the object's destructor (`tp_dealloc`). There must not be any references to the object. If the object's finalizer resurrects the object, this function returns -1; no further destruction should happen. Otherwise, this function returns 0 and destruction can continue normally.

Ver también

`tp_dealloc` for example code.

13.3 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.

13.3.1 Tipos objeto base y macros

All Python objects ultimately share a small number of fields at the beginning of the object's representation in memory. These are represented by the `PyObject` and `PyVarObject` types, which are defined, in turn, by the expansions of some macros also used, whether directly or indirectly, in the definition of all other Python objects. Additional macros can be found under *reference counting*.

type **PyObject**

Part of the Limited API. *(Only some members are part of the stable ABI.)* All object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In a normal «release» build, it contains only the object's reference count and a pointer to the corresponding type object. Nothing is actually declared to be a `PyObject`, but every pointer to a Python object can be cast to a `PyObject*`.

The members must not be accessed directly; instead use macros such as `Py_REFCNT` and `Py_TYPE`.

Py_ssize_t **ob_refcnt**

Part of the Stable ABI. The object's reference count, as returned by `Py_REFCNT`. Do not use this field directly; instead use functions and macros such as `Py_REFCNT`, `Py_INCREF()` and `Py_DecRef()`.

The field type may be different from `Py_ssize_t`, depending on build configuration and platform.

PyTypeObject ***ob_type**

Part of the Stable ABI. The object's type. Do not use this field directly; use `Py_TYPE` and `Py_SET_TYPE()` instead.

type **PyVarObject**

Part of the Limited API. *(Only some members are part of the stable ABI.)* An extension of `PyObject` that adds the `ob_size` field. This is intended for objects that have some notion of *length*.

As with `PyObject`, the members must not be accessed directly; instead use macros such as `Py_SIZE`, `Py_REFCNT` and `Py_TYPE`.

Py_ssize_t **ob_size**

Part of the Stable ABI. A size field, whose contents should be considered an object's internal implementation detail.

Do not use this field directly; use *Py_SIZE* instead.

Object creation functions such as *PyObject_NewVar()* will generally set this field to the requested size (number of items). After creation, arbitrary values can be stored in `ob_size` using *Py_SET_SIZE*.

To get an object's publicly exposed length, as returned by the Python function `len()`, use *PyObject_Length()* instead.

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.

PyTypeObject **PyBaseObject_Type**

Part of the Stable ABI. The base class of all other objects, the same as `object` in Python.

int **Py_Is**(*PyObject* *x, *PyObject* *y)

Part of the Stable ABI *since version 3.10.* Prueba si el objeto *x* es el objeto *y*, lo mismo que `x is y` en Python.

Added in version 3.10.

int **Py_IsNone**(*PyObject* *x)

Part of the Stable ABI *since version 3.10.* Prueba si un objeto es la instancia única `None`, lo mismo que `x is None` en Python.

Added in version 3.10.

int **Py_IsTrue**(*PyObject* *x)

Part of the Stable ABI *since version 3.10.* Prueba si un objeto es la instancia única `True`, lo mismo que `x is True` en Python.

Added in version 3.10.

int **Py_IsFalse**(*PyObject* *x)

Part of the Stable ABI *since version 3.10.* Prueba si un objeto es la instancia única `False`, lo mismo que `x is False` en Python.

Added in version 3.10.

PyTypeObject ***Py_TYPE**(*PyObject* *o)

Return value: Borrowed reference. Part of the Stable ABI *since version 3.14.* Obtiene el tipo de objeto Python *o*.

The returned reference is *borrowed* from *o*. Do not release it with *Py_DECREF()* or similar.

Distinto en la versión 3.11: *Py_TYPE()* se cambia a una función estática inline. El tipo de parámetro ya no es `const` *PyObject**.

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`.

Added in version 3.9.

void **Py_SET_TYPE**(*PyObject* *o, *PyTypeObject* *type)

Set the type of object *o* to *type*, without any checking or reference counting.

This is a very low-level operation. Consider instead setting the Python attribute `__class__` using *PyObject_SetAttrString()* or similar.

Note that assigning an incompatible type can lead to undefined behavior.

If *type* is a *heap type*, the caller must create a new reference to it. Similarly, if the old type of *o* is a heap type, the caller must release a reference to that type.

Added in version 3.9.

Py_ssize_t **Py_SIZE**(*PyVarObject* *o)

Get the *ob_size* field of *o*.

Distinto en la versión 3.11: *Py_SIZE()* se cambia a una función estática inline. El tipo de parámetro ya no es `const PyVarObject*`.

void **Py_SET_SIZE**(*PyVarObject* *o, *Py_ssize_t* size)

Set the *ob_size* field of *o* to *size*.

Added in version 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)

This is a macro which expands to initialization values for a new *PyVarObject* type, including the *ob_size* field. This macro expands to:

```
_PyObject_EXTRA_INIT
1, type, size,
```

13.3.2 Implementando funciones y métodos

type **PyCFunction**

Part of the Stable ABI. Type of the functions used to implement most Python callables in C. Functions of this type take two *PyObject** parameters and return one such value. If the return value is `NULL`, an exception shall have been set. If not `NULL`, the return value is interpreted as the return value of the function as exposed in Python. The function must return a new reference.

La firma de la función es:

```
PyObject *PyCFunction(PyObject *self,
                      PyObject *args);
```

type **PyCFunctionWithKeywords**

Part of the Stable ABI. Type of the functions used to implement Python callables in C with signature *METH_VARARGS | METH_KEYWORDS*. The function signature is:

```
PyObject *PyCFunctionWithKeywords(PyObject *self,
                                  PyObject *args,
                                  PyObject *kwargs);
```

type **PyCFunctionFast**

Part of the Stable ABI *since version 3.13.* Type of the functions used to implement Python callables in C with signature *METH_FASTCALL*. The function signature is:

```
PyObject *PyCFunctionFast(PyObject *self,
                          PyObject *const *args,
                          Py_ssize_t nargs);
```

type **PyCFunctionFastWithKeywords**

Part of the Stable ABI *since version 3.13.* Type of the functions used to implement Python callables in C with signature *METH_FASTCALL | METH_KEYWORDS*. The function signature is:

```
PyObject *PyCFunctionFastWithKeywords(PyObject *self,
                                      PyObject *const *args,
                                      Py_ssize_t nargs,
                                      PyObject *kwnames);
```

type **PyCMethod**

Type of the functions used to implement Python callables in C with signature *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*. The function signature is:

```
PyObject *PyCMethod(PyObject *self,
                    PyTypeObject *defining_class,
                    PyObject *const *args,
                    Py_ssize_t nargs,
                    PyObject *kwnames)
```

Added in version 3.9.

type **PyMethodDef**

Part of the Stable ABI *(including all members).* Estructura utiliza para describir un método de un tipo de extensión. Esta estructura tiene cuatro campos:

const char ***ml_name**

Name of the method.

PyCFunction **ml_meth**

Pointer to the C implementation.

int **ml_flags**

Flags bits indicating how the call should be constructed.

const char ***ml_doc**

Points to the contents of the docstring.

The *ml_meth* is a C function pointer. The functions may be of different types, but they always return *PyObject**. If the function is not of the *PyCFunction*, the compiler will require a cast in the method table. Even though *PyCFunction* defines the first parameter as *PyObject**, it is common that the method implementation uses the specific C type of the *self* object.

The *ml_flags* field is a bitfield which can include the following flags. The individual flags indicate either a calling convention or a binding convention.

Existen estas convenciones de llamada:

METH_VARARGS

This is the typical calling convention, where the methods have the type *PyCFunction*. The function expects two *PyObject** values. The first one is the *self* object for methods; for module functions, it is the module object. The second parameter (often called *args*) is a tuple object representing all arguments. This parameter is typically processed using *PyArg_ParseTuple()* or *PyArg_UnpackTuple()*.

METH_KEYWORDS

Can only be used in certain combinations with other flags: *METH_VARARGS | METH_KEYWORDS*, *METH_FASTCALL | METH_KEYWORDS* and *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*.

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

Fast calling convention supporting only positional arguments. The methods have the type *PyCFunctionFast*. The first parameter is *self*, the second parameter is a C array of *PyObject** values indicating the arguments and the third parameter is the number of arguments (the length of the array).

Added in version 3.7.

Distinto en la versión 3.10: `METH_FASTCALL` is now part of the *stable ABI*.

METH_FASTCALL* | *METH_KEYWORDS

Extension of *METH_FASTCALL* supporting also keyword arguments, with methods of type *PyCFunctionFastWithKeywords*. Keyword arguments are passed the same way as in the *vectorcall protocol*: there is an additional fourth *PyObject** parameter which is a tuple representing the names of the keyword arguments (which are guaranteed to be strings) or possibly `NULL` if there are no keywords. The values of the keyword arguments are stored in the *args* array, after the positional arguments.

Added in version 3.7.

METH_METHOD

Can only be used in the combination with other flags: *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*.

METH_METHOD* | *METH_FASTCALL* | *METH_KEYWORDS

Extension of *METH_FASTCALL | METH_KEYWORDS* supporting the *defining class*, that is, the class that contains the method in question. The defining class might be a superclass of `Py_TYPE(self)`.

El método debe ser de tipo *PyCMethod*, lo mismo que para `METH_FASTCALL | METH_KEYWORDS` con el argumento `defining_clase` añadido después de `self`.

Added in version 3.9.

METH_NOARGS

Methods without parameters don't need to check whether arguments are given if they are listed with the *METH_NOARGS* flag. They need to be of type *PyCFunction*. The first parameter is typically named *self* and will hold a reference to the module or object instance. In all cases the second parameter will be `NULL`.

La función debe tener 2 parámetros. Dado que el segundo parámetro no se usa, *Py_UNUSED* se puede usar para evitar una advertencia del compilador.

METH_O

Methods with a single object argument can be listed with the *METH_O* flag, instead of invoking *PyArg_ParseTuple()* with a `"O"` argument. They have the type *PyCFunction*, with the *self* parameter, and a *PyObject** parameter representing the single argument.

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

The method will be loaded in place of existing definitions. Without *METH_COEXIST*, the default is to skip repeated definitions. Since slot wrappers are loaded before the method table, the existence of a *sq_contains* slot, for example, would generate a wrapped method named `__contains__()` and preclude the loading of a corresponding PyCFunction with the same name. With the flag defined, the PyCFunction will be loaded in place of the wrapper object and will co-exist with the slot. This is helpful because calls to PyCFunctions are optimized more than wrapper object calls.

PyObject ***PyCMethod_New**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module, *PyTypeObject* *cls)

Return value: New reference. Part of the Stable ABI *since version 3.9.* Turn *ml* into a Python *callable* object. The caller must ensure that *ml* outlives the *callable*. Typically, *ml* is defined as a static variable.

The *self* parameter will be passed as the *self* argument to the C function in `ml->ml_meth` when invoked. *self* can be `NULL`.

The *callable* object's `__module__` attribute can be set from the given *module* argument. *module* should be a Python string, which will be used as name of the module the function is defined in. If unavailable, it can be set to `None` or `NULL`.

Ver también

`function.__module__`

The *cls* parameter will be passed as the *defining_class* argument to the C function. Must be set if *METH_METHOD* is set on `ml->ml_flags`.

Added in version 3.9.

PyObject ***PyCFunction_NewEx**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module)

Return value: New reference. Part of the Stable ABI. Equivalent to `PyCMethod_New(ml, self, module, NULL)`.

PyObject ***PyCFunction_New**(*PyMethodDef* *ml, *PyObject* *self)

Return value: New reference. Part of the Stable ABI *since version 3.4.* Equivalent to `PyCMethod_New(ml, self, NULL, NULL)`.

13.3.3 Acceder a atributos de tipos de extensión

type **PyMemberDef**

Part of the Stable ABI *(including all members).* Structure which describes an attribute of a type which corresponds to a C struct member. When defining a class, put a NULL-terminated array of these structures in the *tp_members* slot.

Its fields are, in order:

const char ***name**

Name of the member. A NULL value marks the end of a `PyMemberDef[]` array.

The string should be static, no copy is made of it.

int **type**

The type of the member in the C struct. See *Member types* for the possible values.

Py_ssize_t **offset**

The offset in bytes that the member is located on the type's object struct.

int **flags**

Zero or more of the *Member flags*, combined using bitwise OR.

const char ***doc**

The docstring, or NULL. The string should be static, no copy is made of it. Typically, it is defined using *PyDoc_STR*.

By default (when *flags* is `0`), members allow both read and write access. Use the *Py_READONLY* flag for read-only access. Certain types, like *Py_T_STRING*, imply *Py_READONLY*. Only *Py_T_OBJECT_EX* (and legacy *T_OBJECT*) members can be deleted.

For heap-allocated types (created using *PyType_FromSpec()* or similar), `PyMemberDef` may contain a definition for the special member `"__vectorcalloffset__"`, corresponding to *tp_vectorcall_offset* in type objects. This member must be defined with `Py_T_PYSSIZET`, and either `Py_READONLY` or `Py_READONLY | Py_RELATIVE_OFFSET`. For example:

```
static PyMemberDef spam_type_members[] = {
    {"__vectorcalloffset__", Py_T_PYSSIZET,
     offsetof(Spam_object, vectorcall), Py_READONLY},
    {NULL}  /* Sentinel */
};
```

(You may need to `#include <stddef.h>` for `offsetof()`.)

The legacy offsets *tp_dictoffset* and *tp_weaklistoffset* can be defined similarly using `"__dictoffset__"` and `"__weaklistoffset__"` members, but extensions are strongly encouraged to use *Py_TPFLAGS_MANAGED_DICT* and *Py_TPFLAGS_MANAGED_WEAKREF* instead.

Distinto en la versión 3.12: `PyMemberDef` is always available. Previously, it required including `"structmember.h"`.

Distinto en la versión 3.14: *Py_RELATIVE_OFFSET* is now allowed for `"__vectorcalloffset__"`, `"__dictoffset__"` and `"__weaklistoffset__"`.

PyObject ***PyMember_GetOne**(const char *obj_addr, struct *PyMemberDef* *m)

Part of the Stable ABI. Obtiene un atributo que pertenece al objeto en la dirección *obj_addr*. El atributo se describe por `PyMemberDef` *m*. Retorna `NULL` en caso de error.

Distinto en la versión 3.12: `PyMember_GetOne` is always available. Previously, it required including `"structmember.h"`.

int **PyMember_SetOne**(char *obj_addr, struct *PyMemberDef* *m, *PyObject* *o)

Part of the Stable ABI. Establece un atributo que pertenece al objeto en la dirección *obj_addr* al objeto *o*. El atributo a establecer se describe por `PyMemberDef` *m*. Retorna `0` si tiene éxito y un valor negativo si falla.

Distinto en la versión 3.12: `PyMember_SetOne` is always available. Previously, it required including `"structmember.h"`.

Member flags

The following flags can be used with *PyMemberDef.flags*:

Py_READONLY

Not writable.

Py_AUDIT_READ

Emit an `object.__getattr__` audit event before reading.

Py_RELATIVE_OFFSET

Indicates that the *offset* of this PyMemberDef entry indicates an offset from the subclass-specific data, rather than from PyObject.

Can only be used as part of *Py_tp_members slot* when creating a class using negative *basicsize*. It is mandatory in that case.

This flag is only used in *PyType_Slot*. When setting *tp_members* during class creation, Python clears it and sets *PyMemberDef.offset* to the offset from the PyObject struct.

Distinto en la versión 3.10: The RESTRICTED, READ_RESTRICTED and WRITE_RESTRICTED macros available with #include "structmember.h" are deprecated. READ_RESTRICTED and RESTRICTED are equivalent to *Py_AUDIT_READ*; WRITE_RESTRICTED does nothing.

Distinto en la versión 3.12: The READONLY macro was renamed to *Py_READONLY*. The PY_AUDIT_READ macro was renamed with the Py_ prefix. The new names are now always available. Previously, these required #include "structmember.h". The header is still available and it provides the old names.

Member types

PyMemberDef.type can be one of the following macros corresponding to various C types. When the member is accessed in Python, it will be converted to the equivalent Python type. When it is set from Python, it will be converted back to the C type. If that is not possible, an exception such as TypeError or ValueError is raised.

Unless marked (D), attributes defined this way cannot be deleted using e.g. del or delattr().

Nombre de la macro	Tipo C	Python type
Py_T_BYTE	`char`	`int`
Py_T_SHORT	`short`	`int`
Py_T_INT	`int`	`int`
Py_T_LONG	`long`	`int`
Py_T_LONGLONG	`long long`	`int`
Py_T_UBYTE	`unsigned char`	`int`
Py_T_UINT	`unsigned int`	`int`
Py_T_USHORT	`unsigned short`	`int`
Py_T_ULONG	`unsigned long`	`int`
Py_T_ULONGLONG	`unsigned long long`	`int`
Py_T_PYSSIZET	*Py_ssize_t*	`int`
Py_T_FLOAT	`float`	`float`
Py_T_DOUBLE	`double`	`float`
Py_T_BOOL	`char` (written as 0 or 1)	`bool`
Py_T_STRING	`const char*` (*)	`str` (RO)
Py_T_STRING_INPLACE	`const char[]` (*)	`str` (RO)
Py_T_CHAR	`char` (0-127)	`str` (**)
Py_T_OBJECT_EX	*PyObject**	`object` (D)

(*): Zero-terminated, UTF8-encoded C string. With `Py_T_STRING` the C representation is a pointer; with `Py_T_STRING_INPLACE` the string is stored directly in the structure.

(**): String of length 1. Only ASCII is accepted.

(RO): Implies *Py_READONLY*.

(D): Can be deleted, in which case the pointer is set to NULL. Reading a NULL pointer raises AttributeError.

Added in version 3.12: In previous versions, the macros were only available with #include "structmember.h" and were named without the Py_ prefix (e.g. as T_INT). The header is still available and contains the old names, along with the following deprecated types:

T_OBJECT

Like Py_T_OBJECT_EX, but NULL is converted to None. This results in surprising behavior in Python: deleting the attribute effectively sets it to None.

T_NONE

Always None. Must be used with *Py_READONLY*.

Defining Getters and Setters

type **PyGetSetDef**

Part of the Stable ABI *(including all members).* Estructura para definir el acceso para un tipo como el de una propiedad. Véase también la descripción de la ranura *PyTypeObject.tp_getset*.

const char ***name**

nombre del atributo

getter **get**

C function to get the attribute.

setter **set**

Optional C function to set or delete the attribute. If NULL, the attribute is read-only.

const char ***doc**

docstring opcional

void ***closure**

Optional user data pointer, providing additional data for getter and setter.

typedef *PyObject* *(***getter**)(*PyObject**, void*)

Part of the Stable ABI. The get function takes one *PyObject** parameter (the instance) and a user data pointer (the associated closure):

Debe retornar una nueva referencia en caso de éxito o NULL con una excepción establecida en caso de error.

typedef int (***setter**)(*PyObject**, *PyObject**, void*)

Part of the Stable ABI. set functions take two *PyObject** parameters (the instance and the value to be set) and a user data pointer (the associated closure):

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.

13.4 Type Object Structures

Perhaps one of the most important structures of the Python object system is the structure that defines a new type: the *PyTypeObject* structure. Type objects can be handled using any of the PyObject_* or PyType_* functions, but do not offer much that's interesting to most Python applications. These objects are fundamental to how objects behave, so they are very important to the interpreter itself and to any extension module that implements new types.

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`.

13.4.1 Referencia rápida

«ranuras *tp*» (*tp slots*)

Ranura `PyTypeObject`[Página 320, 1]	*Type*	métodos/atributos especiales	Información[Página 320, 2]			
			C	T	D	I
<R> `tp_name`	const char *	__name__	X	X		
`tp_basicsize`	`Py_ssize_t`		X	X		X
`tp_itemsize`	`Py_ssize_t`			X		X
`tp_dealloc`	`destructor`		X	X		X
`tp_vectorcall_offset`	`Py_ssize_t`			X		X
(`tp_getattr`)	`getattrfunc`	__getattribute__, __getattr__				G
(`tp_setattr`)	`setattrfunc`	__setattr__, __delattr__				G
`tp_as_async`	`PyAsyncMethods` *	*sub-ranuras (sub-slots)*				%
`tp_repr`	`reprfunc`	__repr__	X	X		X
`tp_as_number`	`PyNumberMethods` *	*sub-ranuras (sub-slots)*				%
`tp_as_sequence`	`PySequenceMethods` *	*sub-ranuras (sub-slots)*				%
`tp_as_mapping`	`PyMappingMethods` *	*sub-ranuras (sub-slots)*				%
`tp_hash`	`hashfunc`	__hash__	X			G
`tp_call`	`ternaryfunc`	__call__		X		X
`tp_str`	`reprfunc`	__str__	X			X
`tp_getattro`	`getattrofunc`	__getattribute__, __getattr__	X	X		G
`tp_setattro`	`setattrofunc`	__setattr__, __delattr__	X	X		G
`tp_as_buffer`	`PyBufferProcs` *	*sub-ranuras (sub-slots)*				%
`tp_flags`	unsigned long		X	X		?
`tp_doc`	const char *	__doc__	X	X		
`tp_traverse`	`traverseproc`			X		G
`tp_clear`	`inquiry`			X		G
`tp_richcompare`	`richcmpfunc`	__lt__, __le__, __eq__, __ne__, __gt__, __ge__	X			G
(`tp_weaklistoffset`)	`Py_ssize_t`			X		?
`tp_iter`	`getiterfunc`	__iter__				X
`tp_iternext`	`iternextfunc`	__next__				X
`tp_methods`	`PyMethodDef` []		X	X		
`tp_members`	`PyMemberDef` []			X		
`tp_getset`	`PyGetSetDef` []		X	X		
`tp_base`	`PyTypeObject` *	__base__			X	
`tp_dict`	`PyObject` *	__dict__			?	
`tp_descr_get`	`descrgetfunc`	__get__				X
`tp_descr_set`	`descrsetfunc`	__set__, __delete__				X
(`tp_dictoffset`)	`Py_ssize_t`			X		?
`tp_init`	`initproc`	__init__	X	X		X
`tp_alloc`	`allocfunc`		X		?	?
`tp_new`	`newfunc`	__new__	X	X	?	?
`tp_free`	`freefunc`		X	X	?	?
`tp_is_gc`	`inquiry`			X		X
<`tp_bases`>	`PyObject` *	__bases__			~	
<`tp_mro`>	`PyObject` *	__mro__			~	
[`tp_cache`]	`PyObject` *					
[`tp_subclasses`]	void *	__subclasses__				

continúe en la próxima página

Tabla 1 – proviene de la página anterior

Ranura `PyTypeObject`[1]	*Type*	métodos/atributos especiales	Información[2] C T D I
[*tp_weaklist*]	*PyObject* *		
(*tp_del*)	*destructor*		
[*tp_version_tag*]	unsigned int		
tp_finalize	*destructor*	__del__	X
tp_vectorcall	*vectorcallfunc*		
[*tp_watched*]	unsigned char		

sub-ranuras (*sub-slots*)

Ranuras (*Slot*)	*Type*	métodos especiales
am_await	*unaryfunc*	__await__
am_aiter	*unaryfunc*	__aiter__
am_anext	*unaryfunc*	__anext__
am_send	*sendfunc*	
nb_add	*binaryfunc*	__add__ __radd__
nb_inplace_add	*binaryfunc*	__iadd__
nb_subtract	*binaryfunc*	__sub__ __rsub__
nb_inplace_subtract	*binaryfunc*	__isub__
nb_multiply	*binaryfunc*	__mul__ __rmul__
nb_inplace_multiply	*binaryfunc*	__imul__
nb_remainder	*binaryfunc*	__mod__ __rmod__
nb_inplace_remainder	*binaryfunc*	__imod__
nb_divmod	*binaryfunc*	__divmod__ __rdivmod__
nb_power	*ternaryfunc*	__pow__ __rpow__
nb_inplace_power	*ternaryfunc*	__ipow__
nb_negative	*unaryfunc*	__neg__
nb_positive	*unaryfunc*	__pos__
nb_absolute	*unaryfunc*	__abs__
nb_bool	*inquiry*	__bool__

continúe en la próxima página

[1] (): A slot name in parentheses indicates it is (effectively) deprecated.
<>: Names in angle brackets should be initially set to `NULL` and treated as read-only.
[]: Names in square brackets are for internal use only.
<R> (as a prefix) means the field is required (must be non-`NULL`).

[2] Columnas:
«O»: set on *PyBaseObject_Type*
«T»: set on *PyType_Type*
«D»: por defecto (si la ranura está establecida como `NULL`)

```
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

```
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 (*Slot*)	*Type*	métodos especiales
nb_invert	*unaryfunc*	__invert__
nb_lshift	*binaryfunc*	__lshift__ __rlshift__
nb_inplace_lshift	*binaryfunc*	__ilshift__
nb_rshift	*binaryfunc*	__rshift__ __rrshift__
nb_inplace_rshift	*binaryfunc*	__irshift__
nb_and	*binaryfunc*	__and__ __rand__
nb_inplace_and	*binaryfunc*	__iand__
nb_xor	*binaryfunc*	__xor__ __rxor__
nb_inplace_xor	*binaryfunc*	__ixor__
nb_or	*binaryfunc*	__or__ __ror__
nb_inplace_or	*binaryfunc*	__ior__
nb_int	*unaryfunc*	__int__
nb_reserved	void *	
nb_float	*unaryfunc*	__float__
nb_floor_divide	*binaryfunc*	__floordiv__
nb_inplace_floor_divide	*binaryfunc*	__ifloordiv__
nb_true_divide	*binaryfunc*	__truediv__
nb_inplace_true_divide	*binaryfunc*	__itruediv__
nb_index	*unaryfunc*	__index__
nb_matrix_multiply	*binaryfunc*	__matmul__ __rmatmul__
nb_inplace_matrix_multiply	*binaryfunc*	__imatmul__
mp_length	*lenfunc*	__len__
mp_subscript	*binaryfunc*	__getitem__
mp_ass_subscript	*objobjargproc*	__setitem__, __delitem__
sq_length	*lenfunc*	__len__
sq_concat	*binaryfunc*	__add__
sq_repeat	*ssizeargfunc*	__mul__
sq_item	*ssizeargfunc*	__getitem__
sq_ass_item	*ssizeobjargproc*	__setitem__ __delitem__
sq_contains	*objobjproc*	__contains__
sq_inplace_concat	*binaryfunc*	__iadd__
sq_inplace_repeat	*ssizeargfunc*	__imul__
bf_getbuffer	*getbufferproc()*	__buffer__
bf_releasebuffer	*releasebufferproc()*	__release_buffer__

ranura de *typedefs*

typedef	Tipos parámetros	Tipo de retorno
allocfunc	*PyTypeObject* * *Py_ssize_t*	*PyObject* *
destructor	*PyObject* *	void
freefunc	void *	void
traverseproc	*PyObject* * *visitproc* void *	int
newfunc	*PyTypeObject* * *PyObject* * *PyObject* *	*PyObject* *
initproc	*PyObject* * *PyObject* * *PyObject* *	int
reprfunc	*PyObject* *	*PyObject* *
getattrfunc	*PyObject* * const char *	*PyObject* *
setattrfunc	*PyObject* * const char * *PyObject* *	int
getattrofunc	*PyObject* * *PyObject* *	*PyObject* *
setattrofunc	*PyObject* * *PyObject* * *PyObject* *	int
descrgetfunc	*PyObject* * *PyObject* * *PyObject* *	*PyObject* *
descrsetfunc	*PyObject* * *PyObject* * *PyObject* *	int
hashfunc	*PyObject* *	Py_hash_t
richcmpfunc		*PyObject* *

Vea *Tipo Ranura typedefs* abajo para más detalles.

13.4.2 Definición de `PyTypeObject`

The structure definition for *PyTypeObject* can be found in `Include/cpython/object.h`. For convenience of reference, this repeats the definition found there:

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

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

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

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

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

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

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

    const char *tp_doc; /* Documentation string */

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

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

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

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;
```

(continúe en la próxima página)

(proviene de la página anterior)

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

    /* Attribute descriptor and subclassing stuff */
    PyMethodDef *tp_methods;
    PyMemberDef *tp_members;
    PyGetSetDef *tp_getset;
    // Strong reference on a heap type, borrowed reference on a static type
    PyTypeObject *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; /* no longer used */
    void *tp_subclasses;  /* for static builtin types this is an index */
    PyObject *tp_weaklist; /* not used for static builtin types */
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6.
     * If zero, the cache is invalid and must be initialized.
     */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;

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

    /* Number of tp_version_tag values used.
     * Set to _Py_ATTR_CACHE_UNUSED if the attribute cache is
     * disabled for this type (e.g. due to custom MRO entries).
     * Otherwise, limited to MAX_VERSIONS_PER_CLASS (defined elsewhere).
     */
    uint16_t tp_versions_used;
} PyTypeObject;
```

13.4.3 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.ob_refcnt

The type object's reference count is initialized to `1` by the `PyObject_HEAD_INIT` macro. Note that for *statically allocated type objects*, the type's instances (objects whose *ob_type* points back to the type) do *not* count as references. But for *dynamically allocated type objects*, the instances *do* count as references.

Herencia:

Este campo no es heredado por los subtipos.

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;
```

This should be done before any instances of the type are created. *PyType_Ready()* checks if *ob_type* is `NULL`, and if so, initializes it to the *ob_type* field of the base class. *PyType_Ready()* will not change this field if it is non-zero.

Herencia:

Este campo es heredado por subtipos.

13.4.4 Ranuras `PyVarObject`

PyVarObject.ob_size

Para *objetos de tipo estáticamente asignados*, debe inicializarse a cero. Para *objetos de tipo dinámicamente asignados*, este campo tiene un significado interno especial.

This field should be accessed using the *Py_SIZE()* macro.

Herencia:

Este campo no es heredado por los subtipos.

13.4.5 Ranuras `PyTypeObject`

Each slot has a section describing inheritance. If *PyType_Ready()* may set a value when the field is set to `NULL` then there will also be a «Default» section. (Note that many fields set on *PyBaseObject_Type* and *PyType_Type* effectively act as defaults.)

const char **PyTypeObject*.**tp_name**

Pointer to a NUL-terminated string containing the name of the type. For types that are accessible as module globals, the string should be the full module name, followed by a dot, followed by the type name; for built-in types, it should be just the type name. If the module is a submodule of a package, the full package name is part of the full module name. For example, a type named `T` defined in module `M` in subpackage `Q` in package `P` should have the *tp_name* initializer `"P.Q.M.T"`.

Para *objetos de tipo dinámicamente asignados*, éste debe ser sólo el nombre del tipo, y el nombre del módulo almacenado explícitamente en el dict tipo que el valor de `'__module__'` clave.

For *statically allocated type objects*, the *tp_name* field should contain a dot. Everything before the last dot is made accessible as the `__module__` attribute, and everything after the last dot is made accessible as the `__name__` attribute.

If no dot is present, the entire *tp_name* field is made accessible as the `__name__` attribute, and the `__module__` attribute is undefined (unless explicitly set in the dictionary, as explained above). This means your type will be impossible to pickle. Additionally, it will not be listed in module documentations created with pydoc.

Este campo no debe ser `NULL`. Es el único campo obligatorio en *PyTypeObject()* (que no sea potencialmente *tp_itemsize*).

Herencia:

Este campo no es heredado por los subtipos.

Py_ssize_t *PyTypeObject*.**tp_basicsize**

Py_ssize_t *PyTypeObject*.**tp_itemsize**

Estos campos permiten calcular el tamaño en bytes de instancias del tipo.

There are two kinds of types: types with fixed-length instances have a zero `tp_itemsize` field, types with variable-length instances have a non-zero `tp_itemsize` field. For a type with fixed-length instances, all instances have the same size, given in `tp_basicsize`. (Exceptions to this rule can be made using *PyUnstable_Object_GC_NewWithExtraData()*.)

For a type with variable-length instances, the instances must have an *ob_size* field, and the instance size is `tp_basicsize` plus N times `tp_itemsize`, where N is the «length» of the object.

Functions like *PyObject_NewVar()* will take the value of N as an argument, and store in the instance's *ob_size* field. Note that the *ob_size* field may later be used for other purposes. For example, `int` instances use the bits of *ob_size* in an implementation-defined way; the underlying storage and its size should be accessed using *PyLong_Export()*.

Nota

The *ob_size* field should be accessed using the *Py_SIZE()* and *Py_SET_SIZE()* macros.

Also, the presence of an *ob_size* field in the instance layout doesn't mean that the instance structure is variable-length. For example, the `list` type has fixed-length instances, yet those instances have a *ob_size* field. (As with `int`, avoid reading lists" `ob_size` directly. Call *PyList_Size()* instead.)

The `tp_basicsize` includes size needed for data of the type's *tp_base*, plus any extra data needed by each instance.

The correct way to set `tp_basicsize` is to use the `sizeof` operator on the struct used to declare the instance layout. This struct must include the struct used to declare the base type. In other words, `tp_basicsize` must be greater than or equal to the base's `tp_basicsize`.

Since every type is a subtype of `object`, this struct must include *PyObject* or *PyVarObject* (depending on whether *ob_size* should be included). These are usually defined by the macro *PyObject_HEAD* or *PyObject_VAR_HEAD*, respectively.

The basic size does not include the GC header size, as that header is not part of *PyObject_HEAD*.

For cases where struct used to declare the base type is unknown, see *PyType_Spec.basicsize* and *PyType_FromMetaclass()*.

Notes about alignment:

- `tp_basicsize` must be a multiple of `_Alignof(PyObject)`. When using `sizeof` on a `struct` that includes *PyObject_HEAD*, as recommended, the compiler ensures this. When not using a C `struct`, or when using compiler extensions like `__attribute__((packed))`, it is up to you.
- If the variable items require a particular alignment, `tp_basicsize` and `tp_itemsize` must each be a multiple of that alignment. For example, if a type's variable part stores a `double`, it is your responsibility that both fields are a multiple of `_Alignof(double)`.

Herencia:

These fields are inherited separately by subtypes. (That is, if the field is set to zero, *PyType_Ready()* will copy the value from the base type, indicating that the instances do not need additional storage.)

If the base type has a non-zero *tp_itemsize*, it is generally not safe to set *tp_itemsize* to a different non-zero value in a subtype (though this depends on the implementation of the base type).

destructor *PyTypeObject*.**tp_dealloc**

A pointer to the instance destructor function. The function signature is:

```
void tp_dealloc(PyObject *self);
```

The destructor function should remove all references which the instance owns (e.g., call *Py_CLEAR()*), free all memory buffers owned by the instance, and call the type's *tp_free* function to free the object itself.

If you may call functions that may set the error indicator, you must use *PyErr_GetRaisedException()* and *PyErr_SetRaisedException()* to ensure you don't clobber a preexisting error indicator (the deallocation could have occurred while processing a different error):

```
static void
foo_dealloc(foo_object *self)
{
    PyObject *et, *ev, *etb;
    PyObject *exc = PyErr_GetRaisedException();
    ...
    PyErr_SetRaisedException(exc);
}
```

The dealloc handler itself must not raise an exception; if it hits an error case it should call *PyErr_FormatUnraisable()* to log (and clear) an unraisable exception.

No guarantees are made about when an object is destroyed, except:

- Python will destroy an object immediately or some time after the final reference to the object is deleted, unless its finalizer (*tp_finalize*) subsequently resurrects the object.
- An object will not be destroyed while it is being automatically finalized (*tp_finalize*) or automatically cleared (*tp_clear*).

CPython currently destroys an object immediately from *Py_DECREF()* when the new reference count is zero, but this may change in a future version.

It is recommended to call *PyObject_CallFinalizerFromDealloc()* at the beginning of `tp_dealloc` to guarantee that the object is always finalized before destruction.

If the type supports garbage collection (the *Py_TPFLAGS_HAVE_GC* flag is set), the destructor should call *PyObject_GC_UnTrack()* before clearing any member fields.

It is permissible to call *tp_clear* from `tp_dealloc` to reduce code duplication and to guarantee that the object is always cleared before destruction. Beware that `tp_clear` might have already been called.

If the type is heap allocated (*Py_TPFLAGS_HEAPTYPE*), the deallocator should release the owned reference to its type object (via *Py_DECREF()*) after calling the type deallocator. See the example code below.:

```
static void
foo_dealloc(PyObject *op)
{
   foo_object *self = (foo_object *) op;
   PyObject_GC_UnTrack(self);
   Py_CLEAR(self->ref);
   Py_TYPE(self)->tp_free(self);
}
```

`tp_dealloc` must leave the exception status unchanged. If it needs to call something that might raise an exception, the exception state must be backed up first and restored later (after logging any exceptions with *PyErr_WriteUnraisable()*).

Example:

```
static void
foo_dealloc(PyObject *self)
{
    PyObject *exc = PyErr_GetRaisedException();

    if (PyObject_CallFinalizerFromDealloc(self) < 0) {
```

(continúe en la próxima página)

(proviene de la página anterior)

```
        // self was resurrected.
        goto done;
    }

    PyTypeObject *tp = Py_TYPE(self);

    if (tp->tp_flags & Py_TPFLAGS_HAVE_GC) {
        PyObject_GC_UnTrack(self);
    }

    // Optional, but convenient to avoid code duplication.
    if (tp->tp_clear && tp->tp_clear(self) < 0) {
        PyErr_WriteUnraisable(self);
    }

    // Any additional destruction goes here.

    tp->tp_free(self);
    self = NULL;  // In case PyErr_WriteUnraisable() is called below.

    if (tp->tp_flags & Py_TPFLAGS_HEAPTYPE) {
        Py_CLEAR(tp);
    }

done:
    // Optional, if something was called that might have raised an
    // exception.
    if (PyErr_Occurred()) {
        PyErr_WriteUnraisable(self);
    }
    PyErr_SetRaisedException(exc);
}
```

`tp_dealloc` may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a garbage collection on any thread). This is not a problem for Python API calls, since the thread on which `tp_dealloc` is called with an *attached thread state*. However, if the object being destroyed in turn destroys objects from some other C library, care should be taken to ensure that destroying those objects on the thread which called `tp_dealloc` will not violate any assumptions of the library.

Herencia:

Este campo es heredado por subtipos.

Ver también

Object Life Cycle for details about how this slot relates to other slots.

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`.

This field is only used if the flag `Py_TPFLAGS_HAVE_VECTORCALL` is set. If so, this must be a positive integer containing the offset in the instance of a `vectorcallfunc` pointer.

The *vectorcallfunc* pointer may be `NULL`, in which case the instance behaves as if `Py_TPFLAGS_HAVE_VECTORCALL` was not set: calling the instance falls back to `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()*.

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.

Distinto en la versión 3.12: Before version 3.12, it was not recommended for *mutable heap types* to implement the vectorcall protocol. When a user sets `__call__` in Python code, only *tp_call* is updated, likely making it inconsistent with the vectorcall function. Since 3.12, setting `__call__` will disable vectorcall optimization by clearing the *Py_TPFLAGS_HAVE_VECTORCALL* flag.

Herencia:

This field is always inherited. However, the *Py_TPFLAGS_HAVE_VECTORCALL* flag is not always inherited. If it's not set, then the subclass won't use *vectorcall*, except when *PyVectorcall_Call()* is explicitly called.

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:

Group: *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 *PyTypeObject*.**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:

Group: *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 **PyTypeObject*.**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.

Added in version 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 *PyTypeObject*.**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 **PyTypeObject.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 **PyTypeObject.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 secuencia*.

Herencia:

El campo *tp_as_sequence* no se hereda, pero los campos contenidos se heredan individualmente.

PyMappingMethods **PyTypeObject.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 *PyTypeObject*.**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`.

When this field is not set (*and* *tp_richcompare* is not set), an attempt to take the hash of the object raises `TypeError`. This is the same as setting it to *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:

Group: *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`.

Por defecto:

PyBaseObject_Type uses *PyObject_GenericHash()*.

ternaryfunc *PyTypeObject*.**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 *PyTypeObject*.**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:

Group: *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 uses *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:

Group: *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 uses *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 *PyTypeObject*.**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:

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type's value of the flag bit is copied into the subtype together with a pointer to the extension structure. The *Py_TPFLAGS_HAVE_GC* flag bit is inherited together with the *tp_traverse* and *tp_clear* fields, i.e. if the *Py_TPFLAGS_HAVE_GC* flag bit is clear in the subtype and the *tp_traverse* and *tp_clear* fields in the subtype exist and have `NULL` values.

Por defecto:

PyBaseObject_Type uses `Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE`.

Máscaras de bits:

Las siguientes máscaras de bits están definidas actualmente; estos se pueden unidos 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

This bit is set when the type object itself is allocated on the heap, for example, types created dynamically using *PyType_FromSpec()*. In this case, the *ob_type* field of its instances is considered a reference to the type, and the type object is INCREF'ed when a new instance is created, and DECREF'ed when an instance is destroyed (this does not apply to instances of subtypes; only the type referenced by the instance's ob_type gets INCREF'ed or DECREF'ed). Heap types should also *support garbage collection* as they can form a reference cycle with their own module object.

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

This bit is set when the object supports garbage collection. If this bit is set, memory for new instances (see *tp_alloc*) must be allocated using *PyObject_GC_New* or *PyType_GenericAlloc()* and deallocated (see *tp_free*) using *PyObject_GC_Del()*. More information in section *Apoyo a la recolección de basura cíclica*.

Herencia:

Group: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

The *Py_TPFLAGS_HAVE_GC* flag bit is inherited together with the *tp_traverse* and *tp_clear* fields, i.e. if the *Py_TPFLAGS_HAVE_GC* flag bit is clear in the subtype and the *tp_traverse* and *tp_clear* fields in the subtype exist and have `NULL` values.

Py_TPFLAGS_DEFAULT

This is a bitmask of all the bits that pertain to the existence of certain fields in the type object and its extension structures. Currently, it includes the following bits: `Py_TPFLAGS_HAVE_STACKLESS_EXTENSION`.

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`.

Added in version 3.8.

Herencia:

This flag is never inherited by types without the *Py_TPFLAGS_IMMUTABLETYPE* flag set. For extension types, it is inherited whenever *tp_descr_get* is inherited.

Py_TPFLAGS_MANAGED_DICT

This bit indicates that instances of the class have a `__dict__` attribute, and that the space for the dictionary is managed by the VM.

If this flag is set, *Py_TPFLAGS_HAVE_GC* should also be set.

The type traverse function must call *PyObject_VisitManagedDict()* and its clear function must call *PyObject_ClearManagedDict()*.

Added in version 3.12.

Herencia:

This flag is inherited unless the *tp_dictoffset* field is set in a superclass.

Py_TPFLAGS_MANAGED_WEAKREF

This bit indicates that instances of the class should be weakly referenceable.

Added in version 3.12.

Herencia:

This flag is inherited unless the *tp_weaklistoffset* field is set in a superclass.

Py_TPFLAGS_ITEMS_AT_END

Only usable with variable-size types, i.e. ones with non-zero *tp_itemsize*.

Indicates that the variable-sized portion of an instance of this type is at the end of the instance's memory area, at an offset of `Py_TYPE(obj)->tp_basicsize` (which may be different in each subclass).

When setting this flag, be sure that all superclasses either use this memory layout, or are not variable-sized. Python does not check this.

Added in version 3.12.

Herencia:

This flag is inherited.

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.

Added in version 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:

This bit is inherited if *tp_call* is also inherited.

Added in version 3.9.

Distinto en la versión 3.12: This flag is now removed from a class when the class's `__call__()` method is reassigned.

This flag can now be inherited by mutable classes.

Py_TPFLAGS_IMMUTABLETYPE

Este bit se establece para objetos de tipo que son inmutables: los atributos de tipo no se pueden establecer ni eliminar.

PyType_Ready() aplica automáticamente este indicador a *static types*.

Herencia:

Este flag no se hereda.

Added in version 3.10.

Py_TPFLAGS_DISALLOW_INSTANTIATION

No permita la creación de instancias del tipo: establezca *tp_new* en NULL y no cree la clave `__new__` en el diccionario de tipos.

La bandera debe establecerse antes de crear el tipo, no después. Por ejemplo, debe establecerse antes de que se llame a *PyType_Ready()* en el tipo.

La bandera se establece automáticamente en *static types* si *tp_base* es NULL o `&PyBaseObject_Type` y *tp_new* es NULL.

Herencia:

This flag is not inherited. However, subclasses will not be instantiable unless they provide a non-NULL *tp_new* (which is only possible via the C API).

Nota

To disallow instantiating a class directly but allow instantiating its subclasses (e.g. for an *abstract base class*), do not use this flag. Instead, make *tp_new* only succeed for subclasses.

Added in version 3.10.

Py_TPFLAGS_MAPPING

Este bit indica que las instancias de la clase pueden coincidir con los patrones de correspondencia cuando se utilizan como sujeto de un bloque `match`. Se configura automáticamente al registrar o subclasificar `collections.abc.Mapping`, y se desarma al registrar `collections.abc.Sequence`.

Nota

Py_TPFLAGS_MAPPING and *Py_TPFLAGS_SEQUENCE* are mutually exclusive; it is an error to enable both flags simultaneously.

Herencia:

This flag is inherited by types that do not already set *Py_TPFLAGS_SEQUENCE*.

Ver también

PEP 634 - Coincidencia de patrones estructurales: especificación

Added in version 3.10.

Py_TPFLAGS_SEQUENCE

Este bit indica que las instancias de la clase pueden coincidir con los patrones de secuencia cuando se utilizan como sujeto de un bloque `match`. Se configura automáticamente al registrar o subclasificar `collections.abc.Sequence`, y se desarma al registrar `collections.abc.Mapping`.

> **Nota**
>
> *Py_TPFLAGS_MAPPING* and *Py_TPFLAGS_SEQUENCE* are mutually exclusive; it is an error to enable both flags simultaneously.

Herencia:

This flag is inherited by types that do not already set *Py_TPFLAGS_MAPPING*.

> **Ver también**
>
> **PEP 634** - Coincidencia de patrones estructurales: especificación

Added in version 3.10.

Py_TPFLAGS_VALID_VERSION_TAG

Internal. Do not set or unset this flag. To indicate that a class has changed call *PyType_Modified()*

> **Advertencia**
>
> This flag is present in header files, but is not be used. It will be removed in a future version of CPython

const char ***PyTypeObject*.tp_doc**

An optional pointer to a NUL-terminated C string giving the docstring for this type object. This is exposed as the `__doc__` attribute on the type and instances of the type.

Herencia:

Este campo es *no* heredado por los subtipos.

traverseproc *PyTypeObject*.**tp_traverse**

An optional pointer to a traversal function for the garbage collector. This is only used if the *Py_TPFLAGS_HAVE_GC* flag bit is set. The signature is:

```
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*.

The *tp_traverse* pointer is used by the garbage collector to detect reference cycles. A typical implementation of a *tp_traverse* function simply calls *Py_VISIT()* on each of the instance's members that are Python objects that the instance owns. For example, this is function `local_traverse()` from the `_thread` extension module:

```
static int
local_traverse(PyObject *op, visitproc visit, void *arg)
{
    localobject *self = (localobject *) op;
    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 incluirá.

Heap types (*Py_TPFLAGS_HEAPTYPE*) must visit their type with:

```
Py_VISIT(Py_TYPE(self));
```

It is only needed since Python 3.9. To support Python 3.8 and older, this line must be conditional:

```
#if PY_VERSION_HEX >= 0x03090000
    Py_VISIT(Py_TYPE(self));
#endif
```

If the *Py_TPFLAGS_MANAGED_DICT* bit is set in the *tp_flags* field, the traverse function must call *PyObject_VisitManagedDict()* like this:

```
PyObject_VisitManagedDict((PyObject*)self, visit, arg);
```

Advertencia

Al implementar *tp_traverse*, solo se deben visitar los miembros que tienen la instancia *owns* (por tener *referencias fuertes*). Por ejemplo, si un objeto admite referencias débiles a través de la ranura *tp_weaklist*, el puntero que respalda la lista vinculada (a lo que apunta *tp_weaklist*) **no** debe visitarse ya que la instancia no posee directamente las referencias débiles a sí misma (la lista de referencias débiles está ahí para respaldar la maquinaria de referencia débil, pero la instancia no tiene una fuerte referencia a los elementos dentro de ella, ya que pueden eliminarse incluso si la instancia todavía está viva).

Note that *Py_VISIT()* requires the *visit* and *arg* parameters to `local_traverse()` to have these specific names; don't name them just anything.

Las instancias de *heap-allocated types* 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.

Nota

The *tp_traverse* function can be called from any thread.

Distinto en la versión 3.9: Se espera que los tipos asignados al heap 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:

Group: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

This field is inherited by subtypes together with *tp_clear* and the *Py_TPFLAGS_HAVE_GC* flag bit: the flag bit, *tp_traverse*, and *tp_clear* are all inherited from the base type if they are all zero in the subtype.

inquiry *PyTypeObject*.**tp_clear**

An optional pointer to a clear function. The signature is:

```
int tp_clear(PyObject *);
```

The purpose of this function is to break reference cycles that are causing a *cyclic isolate* so that the objects can be safely destroyed. A cleared object is a partially destroyed object; the object is not obligated to satisfy design invariants held during normal use.

`tp_clear` does not need to delete references to objects that can't participate in reference cycles, such as Python strings or Python integers. However, it may be convenient to clear all references, and write the type's

tp_dealloc function to invoke `tp_clear` to avoid code duplication. (Beware that `tp_clear` might have already been called. Prefer calling idempotent functions like *Py_CLEAR()*.)

Any non-trivial cleanup should be performed in *tp_finalize* instead of `tp_clear`.

Nota

If `tp_clear` fails to break a reference cycle then the objects in the *cyclic isolate* may remain indefinitely uncollectable («leak»). See `gc.garbage`.

Nota

Referents (direct and indirect) might have already been cleared; they are not guaranteed to be in a consistent state.

Nota

The *tp_clear* function can be called from any thread.

Nota

An object is not guaranteed to be automatically cleared before its destructor (*tp_dealloc*) is called.

This function differs from the destructor (*tp_dealloc*) in the following ways:

- The purpose of clearing an object is to remove references to other objects that might participate in a reference cycle. The purpose of the destructor, on the other hand, is a superset: it must release *all* resources it owns, including references to objects that cannot participate in a reference cycle (e.g., integers) as well as the object's own memory (by calling *tp_free*).
- When `tp_clear` is called, other objects might still hold references to the object being cleared. Because of this, `tp_clear` must not deallocate the object's own memory (*tp_free*). The destructor, on the other hand, is only called when no (strong) references exist, and as such, must safely destroy the object itself by deallocating it.
- `tp_clear` might never be automatically called. An object's destructor, on the other hand, will be automatically called some time after the object becomes unreachable (i.e., either there are no references to the object or the object is a member of a *cyclic isolate*).

No guarantees are made about when, if, or how often Python automatically clears an object, except:

- Python will not automatically clear an object if it is reachable, i.e., there is a reference to it and it is not a member of a *cyclic isolate*.
- Python will not automatically clear an object if it has not been automatically finalized (see *tp_finalize*). (If the finalizer resurrected the object, the object may or may not be automatically finalized again before it is cleared.)
- If an object is a member of a *cyclic isolate*, Python will not automatically clear it if any member of the cyclic isolate has not yet been automatically finalized (*tp_finalize*).
- Python will not destroy an object until after any automatic calls to its `tp_clear` function have returned. This ensures that the act of breaking a reference cycle does not invalidate the `self` pointer while `tp_clear` is still executing.
- Python will not automatically call `tp_clear` multiple times concurrently.

CPython currently only automatically clears objects as needed to break reference cycles in a *cyclic isolate*, but future versions might clear objects regularly before their destruction.

Taken together, all *tp_clear* functions in the system must combine to break all reference cycles. This is subtle, and if in any doubt supply a *tp_clear* function. For example, the tuple type does not implement a *tp_clear* function, because it's possible to prove that no reference cycle can be composed entirely of tuples. Therefore the *tp_clear* functions of other types are responsible for breaking any cycle containing a tuple. This isn't immediately obvious, and there's rarely a good reason to avoid implementing *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(PyObject *op)
{
    localobject *self = (localobject *) op;
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

The *Py_CLEAR()* macro should be used, because clearing references is delicate: the reference to the contained object must not be released (via *Py_DECREF()*) until after the pointer to the contained object is set to `NULL`. This is because releasing the reference may cause the contained object to become trash, triggering a chain of reclamation activity that may include invoking arbitrary Python code (due to finalizers, or weakref callbacks, associated with the contained object). If it's possible for such code to reference *self* again, it's important that the pointer to the contained object be `NULL` at that time, so that *self* knows the contained object can no longer be used. The *Py_CLEAR()* macro performs the operations in a safe order.

If the *Py_TPFLAGS_MANAGED_DICT* bit is set in the *tp_flags* field, the traverse function must call *PyObject_ClearManagedDict()* like this:

```
PyObject_ClearManagedDict((PyObject*)self);
```

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:

Group: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

This field is inherited by subtypes together with *tp_traverse* and the *Py_TPFLAGS_HAVE_GC* flag bit: the flag bit, *tp_traverse*, and *tp_clear* are all inherited from the base type if they are all zero in the subtype.

Ver también

Object Life Cycle for details about how this slot relates to other slots.

richcmpfunc *PyTypeObject*.**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 *PyTypeObject*.

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
Py_LT	<
Py_LE	<=
Py_EQ	==
Py_NE	!=
Py_GT	>
Py_GE	>=

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()*.

The returned value is a new *strong reference*.

En caso de error, establece una excepción y retorna `NULL` de la función.

Added in version 3.7.

Herencia:

Group: *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 provides a *tp_richcompare* implementation, which may be inherited. However, if only *tp_hash* is defined, not even the inherited function is used and instances of the type will not be able to participate in any comparisons.

Py_ssize_t *PyTypeObject*.**tp_weaklistoffset**

While this field is still supported, *Py_TPFLAGS_MANAGED_WEAKREF* should be used instead, if at all possible.

If the instances of this type are weakly referenceable, this field is greater than zero and contains the offset in the instance structure of the weak reference list head (ignoring the GC header, if present); this offset is used by *PyObject_ClearWeakRefs()* and the `PyWeakref_*` functions. The instance structure needs to include a field of type *PyObject*** which is initialized to `NULL`.

No confunda este campo con *tp_weaklist*; ese es el encabezado de la lista para referencias débiles al objeto de tipo en sí.

It is an error to set both the *Py_TPFLAGS_MANAGED_WEAKREF* bit and *tp_weaklistoffset*.

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.

Por defecto:

If the *Py_TPFLAGS_MANAGED_WEAKREF* bit is set in the *tp_flags* field, then *tp_weaklistoffset* will be set to a negative value, to indicate that it is unsafe to use this field.

getiterfunc *PyTypeObject*.**tp_iter**

An optional pointer to a function that returns an *iterator* for the object. Its presence normally signals that the instances of this type are *iterable* (although sequences may be iterable without this function).

Esta función tiene la misma firma que *PyObject_GetIter()*:

```
PyObject *tp_iter(PyObject *self);
```

Herencia:

Este campo es heredado por subtipos.

iternextfunc *PyTypeObject*.**tp_iternext**

An optional pointer to a function that returns the next item in an *iterator*. The signature is:

```
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* **PyTypeObject*.**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* **PyTypeObject*.**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* **PyTypeObject*.**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).

PyTypeObject ***PyTypeObject.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.
>
> However, the unary "&" operator applied to a non-static variable like *PyBaseObject_Type* is not required to produce an address constant. Compilers may support this (gcc does), MSVC does not. Both compilers are strictly standard conforming in this particular behavior.
>
> 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 ***PyTypeObject.tp_dict**

El diccionario del tipo se almacena aquí por *PyType_Ready()*.

This field should normally be initialized to `NULL` before PyType_Ready is called; it may also be initialized to a dictionary containing initial attributes for the type. Once *PyType_Ready()* has initialized the type, extra attributes for the type may be added to this dictionary only if they don't correspond to overloaded operations (like `__add__()`). Once initialization for the type has finished, this field should be treated as read-only.

Some types may not store their dictionary in this slot. Use *PyType_GetDict()* to retrieve the dictionary for an arbitrary type.

Distinto en la versión 3.12: Internals detail: For static builtin types, this is always `NULL`. Instead, the dict for such types is stored on `PyInterpreterState`. Use *PyType_GetDict()* to get the dict for an arbitrary type.

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**

While this field is still supported, *Py_TPFLAGS_MANAGED_DICT* should be used instead, if at all possible.

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.

The value specifies the offset of the dictionary from the start of the instance structure.

El *tp_dictoffset* debe considerarse como de solo escritura. Para obtener el puntero al diccionario, llame a *PyObject_GenericGetDict()*. Llamar a *PyObject_GenericGetDict()* puede necesitar asignar memoria para el diccionario, por lo que puede ser más eficiente llamar a *PyObject_GetAttr()* cuando se accede a un atributo en el objeto.

It is an error to set both the *Py_TPFLAGS_MANAGED_DICT* bit and *tp_dictoffset*.

Herencia:

This field is inherited by subtypes. A subtype should not override this offset; doing so could be unsafe, if C code tries to access the dictionary at the previous offset. To properly support inheritance, use *Py_TPFLAGS_MANAGED_DICT*.

Por defecto:

This slot has no default. For *static types*, if the field is `NULL` then no `__dict__` gets created for instances.

If the *Py_TPFLAGS_MANAGED_DICT* bit is set in the *tp_flags* field, then *tp_dictoffset* will be set to `-1`, to indicate that it is unsafe to use this field.

initproc *PyTypeObject*.**tp_init**

Un puntero opcional a una función de inicialización de instancia.

This function corresponds to the `__init__()` method of classes. Like `__init__()`, it is possible to create an instance without calling `__init__()`, and it is possible to reinitialize an instance by calling its `__init__()` method again.

La firma de la función es:

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwds);
```

The self argument is the instance to be initialized; the *args* and *kwds* arguments represent positional and keyword arguments of the call to `__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 *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:

Static subtypes inherit this slot, which will be *PyType_GenericAlloc()* if inherited from `object`.

Heap subtypes do not inherit this slot.

Por defecto:

For heap subtypes, this field is always set to *PyType_GenericAlloc()*.

For static subtypes, this slot is inherited (see above).

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 *kwds);
```

El argumento *subtype* es el tipo de objeto que se está creando; los argumentos *args* y *kwds* 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*.

Set the *Py_TPFLAGS_DISALLOW_INSTANTIATION* flag to disallow creating instances of the type in Python.

Herencia:

Este campo se hereda por subtipos, excepto que no lo heredan *tipos estáticos* cuyo *tp_base* es `NULL` o `&PyBaseObject_Type`.

Por defecto:

Para *tipos estáticos*, este campo no tiene ningún valor predeterminado. Esto significa que si la ranura se define como `NULL`, no se puede llamar al tipo para crear nuevas instancias; presumiblemente hay alguna 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);
```

This function must free the memory allocated by *tp_alloc*.

Herencia:

Static subtypes inherit this slot, which will be *PyObject_Free()* if inherited from `object`. Exception: If the type supports garbage collection (i.e., the *Py_TPFLAGS_HAVE_GC* flag is set in *tp_flags*) and it would inherit *PyObject_Free()*, then this slot is not inherited but instead defaults to *PyObject_GC_Del()*.

Heap subtypes do not inherit this slot.

Por defecto:

For *heap subtypes*, this slot defaults to a deallocator suitable to match *PyType_GenericAlloc()* and the value of the *Py_TPFLAGS_HAVE_GC* flag.

For static subtypes, this slot is inherited (see above).

inquiry *PyTypeObject*.**tp_is_gc**

Un puntero opcional a una función llamada por el recolector de basura.

The garbage collector needs to know whether a particular object is collectible or not. Normally, it is sufficient to look at the object's type's *tp_flags* field, and check the *Py_TPFLAGS_HAVE_GC* flag bit. But some types have a mixture of statically and dynamically allocated instances, and the statically allocated instances are not collectible. Such types should define this function; it should return `1` for a collectible instance, and `0` for a non-collectible instance. The signature is:

```
int tp_is_gc(PyObject *self);
```

(El único ejemplo de esto son los tipos en sí. El metatipo, *PyType_Type*, define esta función para distinguir entre tipos estática y *dinámicamente asignados*.)

Herencia:

Este campo es heredado por subtipos.

Por defecto:

This slot has no default. If this field is `NULL`, *Py_TPFLAGS_HAVE_GC* is used as the functional equivalent.

PyObject ***PyTypeObject**.**tp_bases**

Tupla de tipos base.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is *initialized*.

For dynamically created classes, the `Py_tp_bases` *slot* can be used instead of the *bases* argument of *PyType_FromSpecWithBases()*. The argument form is preferred.

Advertencia

Multiple inheritance does not work well for statically defined types. If you set `tp_bases` to a tuple, Python will not raise an error, but some slots will only be inherited from the first base.

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.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is *initialized*.

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.

void ***PyTypeObject.tp_subclasses**

A collection of subclasses. Internal use only. May be an invalid pointer.

To get a list of subclasses, call the Python method `__subclasses__()`.

Distinto en la versión 3.12: For some types, this field does not hold a valid *PyObject**. The type was changed to `void*` to indicate this.

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.

Distinto en la versión 3.12: Internals detail: For the static builtin types this is always `NULL`, even if weakrefs are added. Instead, the weakrefs for each are stored on `PyInterpreterState`. Use the public C-API or the internal `_PyObject_GET_WEAKREFS_LISTPTR()` macro to avoid the distinction.

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**

An optional pointer to an instance finalization function. This is the C implementation of the `__del__()` special method. Its signature is:

```
void tp_finalize(PyObject *self);
```

The primary purpose of finalization is to perform any non-trivial cleanup that must be performed before the object is destroyed, while the object and any other objects it directly or indirectly references are still in a consistent state. The finalizer is allowed to execute arbitrary Python code.

Before Python automatically finalizes an object, some of the object's direct or indirect referents might have themselves been automatically finalized. However, none of the referents will have been automatically cleared (*tp_clear*) yet.

Other non-finalized objects might still be using a finalized object, so the finalizer must leave the object in a sane state (e.g., invariants are still met).

Nota

After Python automatically finalizes an object, Python might start automatically clearing (*tp_clear*) the object and its referents (direct and indirect). Cleared objects are not guaranteed to be in a consistent state; a finalized object must be able to tolerate cleared referents.

Nota

An object is not guaranteed to be automatically finalized before its destructor (*tp_dealloc*) is called. It is recommended to call *PyObject_CallFinalizerFromDealloc()* at the beginning of `tp_dealloc` to guarantee that the object is always finalized before destruction.

Nota

The *tp_finalize* function can be called from any thread, although the *GIL* will be held.

Nota

The `tp_finalize` function can be called during shutdown, after some global variables have been deleted. See the documentation of the `__del__()` method for details.

When Python finalizes an object, it behaves like the following algorithm:

1. Python might mark the object as *finalized*. Currently, Python always marks objects whose type supports garbage collection (i.e., the *Py_TPFLAGS_HAVE_GC* flag is set in *tp_flags*) and never marks other types of objects; this might change in a future version.
2. If the object is not marked as *finalized* and its `tp_finalize` finalizer function is non-`NULL`, the finalizer function is called.
3. If the finalizer function was called and the finalizer made the object reachable (i.e., there is a reference to the object and it is not a member of a *cyclic isolate*), then the finalizer is said to have *resurrected* the object. It is unspecified whether the finalizer can also resurrect the object by adding a new reference to the object that does not make it reachable, i.e., the object is (still) a member of a cyclic isolate.
4. If the finalizer resurrected the object, the object's pending destruction is canceled and the object's *finalized* mark might be removed if present. Currently, Python never removes the *finalized* mark; this might change in a future version.

Automatic finalization refers to any finalization performed by Python except via calls to *PyObject_CallFinalizer()* or *PyObject_CallFinalizerFromDealloc()*. No guarantees are made about when, if, or how often an object is automatically finalized, except:

- Python will not automatically finalize an object if it is reachable, i.e., there is a reference to it and it is not a member of a *cyclic isolate*.
- Python will not automatically finalize an object if finalizing it would not mark the object as *finalized*. Currently, this applies to objects whose type does not support garbage collection, i.e., the *Py_TPFLAGS_HAVE_GC* flag is not set. Such objects can still be manually finalized by calling *PyObject_CallFinalizer()* or *PyObject_CallFinalizerFromDealloc()*.
- Python will not automatically finalize any two members of a *cyclic isolate* concurrently.
- Python will not automatically finalize an object after it has automatically cleared (*tp_clear*) the object.
- If an object is a member of a *cyclic isolate*, Python will not automatically finalize it after automatically clearing (see *tp_clear*) any other member.

- Python will automatically finalize every member of a *cyclic isolate* before it automatically clears (see `tp_clear`) any of them.
- If Python is going to automatically clear an object (`tp_clear`), it will automatically finalize the object first.

Python currently only automatically finalizes objects that are members of a *cyclic isolate*, but future versions might finalize objects regularly before their destruction.

To manually finalize an object, do not call this function directly; call `PyObject_CallFinalizer()` or `PyObject_CallFinalizerFromDealloc()` instead.

`tp_finalize` should leave the current exception status unchanged. The recommended way to write a non-trivial finalizer is to back up the exception at the beginning by calling `PyErr_GetRaisedException()` and restore the exception at the end by calling `PyErr_SetRaisedException()`. If an exception is encountered in the middle of the finalizer, log and clear it with `PyErr_WriteUnraisable()` or `PyErr_FormatUnraisable()`. For example:

```
static void
foo_finalize(PyObject *self)
{
    // Save the current exception, if any.
    PyObject *exc = PyErr_GetRaisedException();

    // ...

    if (do_something_that_might_raise() != success_indicator) {
        PyErr_WriteUnraisable(self);
        goto done;
    }

done:
    // Restore the saved exception.  This silently discards any exception
    // raised above, so be sure to call PyErr_WriteUnraisable first if
    // necessary.
    PyErr_SetRaisedException(exc);
}
```

Herencia:

Este campo es heredado por subtipos.

Added in version 3.4.

Distinto en la versión 3.8: Before version 3.8 it was necessary to set the `Py_TPFLAGS_HAVE_FINALIZE` flags bit in order for this field to be used. This is no longer required.

Ver también

- **PEP 442**: «Safe object finalization»
- *Object Life Cycle* for details about how this slot relates to other slots.
- `PyObject_CallFinalizer()`
- `PyObject_CallFinalizerFromDealloc()`

vectorcallfunc `PyTypeObject`.**tp_vectorcall**

A *vectorcall function* to use for calls of this type object (rather than instances). In other words, `tp_vectorcall` can be used to optimize `type.__call__`, which typically returns a new instance of *type*.

As with any vectorcall function, if `tp_vectorcall` is `NULL`, the *tp_call* protocol (`Py_TYPE(type)->tp_call`) is used instead.

> **Nota**
>
> The *vectorcall protocol* requires that the vectorcall function has the same behavior as the corresponding `tp_call`. This means that `type->tp_vectorcall` must match the behavior of `Py_TYPE(type)->tp_call`.
>
> Specifically, if *type* uses the default metaclass, `type->tp_vectorcall` must behave the same as *PyType_Type*`->tp_call`, which:
>
> - calls `type->tp_new`,
> - if the result is a subclass of *type*, calls `type->tp_init` on the result of `tp_new`, and
> - returns the result of `tp_new`.
>
> Typically, `tp_vectorcall` is overridden to optimize this process for specific *tp_new* and *tp_init*. When doing this for user-subclassable types, note that both can be overridden (using `__new__()` and `__init__()`, respectively).

Herencia:

Este campo nunca se hereda.

Added in version 3.9: (el campo existe desde 3.8 pero solo se usa desde 3.9)

unsigned char *PyTypeObject*`.`**`tp_watched`**

Internal. Do not use.

Added in version 3.12.

13.4.6 Tipos estáticos

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 interpretador.

Also, since *PyTypeObject* is only part of the *Limited API* as an opaque struct, any extension modules using static types must be compiled for a specific Python minor version.

13.4.7 Tipos Heap

An alternative to *static types* is *heap-allocated types*, or *heap types* for short, which correspond closely to classes created by Python's `class` statement. Heap types have the *Py_TPFLAGS_HEAPTYPE* flag set.

This is done by filling a *PyType_Spec* structure and calling *PyType_FromSpec()*, *PyType_FromSpecWithBases()*, *PyType_FromModuleAndSpec()*, or *PyType_FromMetaclass()*.

13.4.8 Estructuras de objetos de números

type **`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;
     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

The *`nb_reserved`* field should always be `NULL`. It was previously called `nb_long`, and was renamed in

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**

13.4.9 Estructuras de objetos mapeo

type **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()* retorna `1`, de lo contrario puede ser `NULL`.

objobjargproc *PyMappingMethods*.**mp_ass_subscript**

This function is used by *PyObject_SetItem()*, *PyObject_DelItem()*, *PySequence_SetSlice()* and *PySequence_DelSlice()*. It has the same signature as `PyObject_SetItem()`, but *v* can also be set to `NULL` to delete an item. If this slot is `NULL`, the object does not support item assignment and deletion.

13.4.10 Estructuras de objetos secuencia

type **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()* retorna `1`, de lo contrario puede ser `NULL`.

Negative indexes are handled as follows: if the *sq_length* slot is filled, it is called and the sequence length is used to compute a positive index which is passed to *sq_item*. If `sq_length` is `NULL`, the index is passed as is to the function.

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*.

13.4.11 Estructuras de objetos búfer

type **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) Check if the request can be met. If not, raise BufferError, set view->obj to NULL and return -1.

(2) Rellene los campos solicitados.

(3) Incrementa un contador interno para el número de exportaciones (*exports*).

(4) Set view->obj to *exporter* and increment 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-export: Each member of the tree acts as the exporting object and sets view->obj to a new reference to itself.
- Redirect: The buffer request is redirected to the root object of the tree. Here, view->obj will be a new reference to the root object.

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*.

This function MUST NOT decrement `view->obj`, since that is done automatically in *PyBuffer_Release()* (this scheme is useful for breaking reference cycles).

PyBuffer_Release() es la interfaz para el consumidor que envuelve esta función.

13.4.12 Estructuras de objetos asíncronos

Added in version 3.5.

type **PyAsyncMethods**

Esta estructura contiene punteros a las funciones requeridas para implementar objetos «esperable» (*awaitable*) y «iterador asincrónico» (*asynchronous iterator*).

Aquí está la definición de la estructura:

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
    sendfunc am_send;
} PyAsyncMethods;
```

unaryfunc *PyAsyncMethods*.**am_await**

La firma de esta función es:

```
PyObject *am_await(PyObject *self);
```

The returned object must be an *iterator*, i.e. *PyIter_Check()* must return `1` for it.

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 asincrónica.

unaryfunc *PyAsyncMethods*.**am_anext**

La firma de esta función es:

```
PyObject *am_anext(PyObject *self);
```

Must return an *awaitable* object. See `__anext__()` for details. This slot may be set to `NULL`.

sendfunc `PyAsyncMethods`.**am_send**

La firma de esta función es:

```
PySendResult am_send(PyObject *self, PyObject *arg, PyObject **result);
```

Consulte `PyIter_Send()` para obtener más detalles. Esta ranura se puede establecer en `NULL`.

Added in version 3.10.

13.4.13 Tipo Ranura *typedefs*

typedef *PyObject* *(***allocfunc**)(*PyTypeObject* *cls, *Py_ssize_t* nitems)

Part of the Stable ABI. The purpose of this function is to separate memory allocation from memory initialization. It should return a pointer to a block of memory of adequate length for the instance, suitably aligned, and initialized to zeros, but with `ob_refcnt` set to `1` and `ob_type` set to the type argument. If the type's `tp_itemsize` is non-zero, the object's `ob_size` field should be initialized to *nitems* and the length of the allocated memory block should be `tp_basicsize + nitems*tp_itemsize`, rounded up to a multiple of `sizeof(void*)`; otherwise, *nitems* is not used and the length of the block should be `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`.

typedef void (***destructor**)(*PyObject**)

Part of the Stable ABI.

typedef void (***freefunc**)(void*)

Consulte `tp_free`.

typedef *PyObject* *(***newfunc**)(*PyTypeObject**, *PyObject**, *PyObject**)

Part of the Stable ABI. Consulte `tp_new`.

typedef int (***initproc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the Stable ABI. Consulte `tp_init`.

typedef *PyObject* *(***reprfunc**)(*PyObject**)

Part of the Stable ABI. Consulte `tp_repr`.

typedef *PyObject* *(***getattrfunc**)(*PyObject* *self, char *attr)

Part of the Stable ABI. Retorna el valor del atributo nombrado para el objeto.

typedef int (***setattrfunc**)(*PyObject* *self, char *attr, *PyObject* *value)

Part of the Stable ABI. Establece el valor del atributo nombrado para el objeto. El argumento del valor se establece en `NULL` para eliminar el atributo.

typedef *PyObject* *(***getattrofunc**)(*PyObject* *self, *PyObject* *attr)

Part of the Stable ABI. Retorna el valor del atributo nombrado para el objeto.

Consulte `tp_getattro`.

typedef int (***setattrofunc**)(*PyObject* *self, *PyObject* *attr, *PyObject* *value)

Part of the Stable ABI. Establece el valor del atributo nombrado para el objeto. El argumento del valor se establece en `NULL` para eliminar el atributo.

Consulte `tp_setattro`.

typedef *PyObject* *(***descrgetfunc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the Stable ABI. See `tp_descr_get`.

typedef int (***descrsetfunc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the Stable ABI. See `tp_descr_set`.

typedef *Py_hash_t* (***hashfunc**)(*PyObject**)

Part of the Stable ABI. Consulte `tp_hash`.

typedef *PyObject* *(***richcmpfunc**)(*PyObject**, *PyObject**, int)

Part of the Stable ABI. Consulte `tp_richcompare`.

typedef *PyObject* *(***getiterfunc**)(*PyObject**)

Part of the Stable ABI. Consulte `tp_iter`.

typedef *PyObject* *(***iternextfunc**)(*PyObject**)

Part of the Stable ABI. Consulte `tp_iternext`.

typedef *Py_ssize_t* (***lenfunc**)(*PyObject**)

Part of the Stable ABI.

typedef int (***getbufferproc**)(*PyObject**, *Py_buffer**, int)

Part of the Stable ABI *since version 3.12.*

typedef void (***releasebufferproc**)(*PyObject**, *Py_buffer**)

Part of the Stable ABI *since version 3.12.*

typedef *PyObject* *(***unaryfunc**)(*PyObject**)

Part of the Stable ABI.

typedef *PyObject* *(***binaryfunc**)(*PyObject**, *PyObject**)

Part of the Stable ABI.

typedef *PySendResult* (***sendfunc**)(*PyObject**, *PyObject**, *PyObject***)

Consulte `am_send`.

typedef *PyObject* *(***ternaryfunc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the Stable ABI.

typedef *PyObject* *(***ssizeargfunc**)(*PyObject**, *Py_ssize_t*)

Part of the Stable ABI.

typedef int (***ssizeobjargproc**)(*PyObject**, *Py_ssize_t*, *PyObject**)

Part of the Stable ABI.

typedef int (***objobjproc**)(*PyObject**, *PyObject**)

Part of the Stable ABI.

typedef int (***objobjargproc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the Stable ABI.

13.4.14 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;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_new = myobj_new,
```

(continúe en la próxima página)

(proviene de la página anterior)

```
    .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 MyObject_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 */
    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;
} MyObject;

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

(continúe en la próxima página)

(proviene de la página anterior)

```
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
        Py_TPFLAGS_HAVE_GC | Py_TPFLAGS_MANAGED_DICT |
        Py_TPFLAGS_MANAGED_WEAKREF,
    .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,
};
```

A str subclass that cannot be subclassed and cannot be called to create instances (e.g. uses a separate factory func) using *Py_TPFLAGS_DISALLOW_INSTANTIATION* flag:

```
typedef struct {
    PyUnicodeObject raw;
    char *extra;
} MyStr;

static PyTypeObject 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 | Py_TPFLAGS_DISALLOW_INSTANTIATION,
    .tp_repr = (reprfunc)myobj_repr,
};
```

El *tipo estático* más simple con instancias de longitud fija:

```
typedef struct {
    PyObject_HEAD
} MyObject;

static PyTypeObject 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 PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject) - sizeof(char *),
    .tp_itemsize = sizeof(char *),
```

(continúe en la próxima página)

(proviene de la página anterior)

```
};
```

13.5 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.

To create a container type, the *tp_flags* field of the type object must include the *Py_TPFLAGS_HAVE_GC* and provide an implementation of the *tp_traverse* handler. If instances of the type are mutable, a *tp_clear* implementation must also be provided.

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. The memory for the object must be allocated using *PyObject_GC_New* or *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 (*deallocator*) 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**
>
> Si un tipo añade el Py_TPFLAGS_HAVE_GC, entonces *must* implementar al menos un manejado *tp_traverse* o usar explícitamente uno de su subclase o subclases.
>
> 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.

PyObject_GC_New(TYPE, typeobj)

Analogous to *PyObject_New* but for container objects with the *Py_TPFLAGS_HAVE_GC* flag set.

Do not call this directly to allocate memory for an object; call the type's *tp_alloc* slot instead.

When populating a type's *tp_alloc* slot, *PyType_GenericAlloc()* is preferred over a custom function that simply calls this macro.

Memory allocated by this macro must be freed with *PyObject_GC_Del()* (usually called via the object's *tp_free* slot).

> **Ver también**
>
> - *PyObject_GC_Del()*
> - *PyObject_New*
> - *PyType_GenericAlloc()*
> - *tp_alloc*

PyObject_GC_NewVar(TYPE, typeobj, size)

Analogous to *PyObject_NewVar* but for container objects with the *Py_TPFLAGS_HAVE_GC* flag set.

Do not call this directly to allocate memory for an object; call the type's *tp_alloc* slot instead.

When populating a type's *tp_alloc* slot, *PyType_GenericAlloc()* is preferred over a custom function that simply calls this macro.

Memory allocated by this macro must be freed with *PyObject_GC_Del()* (usually called via the object's *tp_free* slot).

Ver también

- *PyObject_GC_Del()*
- *PyObject_NewVar*
- *PyType_GenericAlloc()*
- *tp_alloc*

PyObject ***PyUnstable_Object_GC_NewWithExtraData**(*PyTypeObject* *type, size_t extra_size)

This is *Unstable API*. It may change without warning in minor releases.

Analogous to *PyObject_GC_New* but allocates *extra_size* bytes at the end of the object (at offset *tp_basicsize*). The allocated memory is initialized to zeros, except for the *Python object header*.

The extra data will be deallocated with the object, but otherwise it is not managed by Python.

Memory allocated by this function must be freed with *PyObject_GC_Del()* (usually called via the object's *tp_free* slot).

Advertencia

The function is marked as unstable because the final mechanism for reserving extra data after an instance is not yet decided. For allocating a variable number of fields, prefer using *PyVarObject* and *tp_itemsize* instead.

Added in version 3.12.

PyObject_GC_Resize(TYPE, op, newsize)

Resize an object allocated by *PyObject_NewVar*. Returns the resized object of type `TYPE*` (refers to any C type) or `NULL` on failure.

op must be of type *PyVarObject** and must not be tracked by the collector yet. *newsize* must be of type *Py_ssize_t*.

void **PyObject_GC_Track**(*PyObject* *op)

Part of the Stable ABI. 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)

Part of the Stable ABI *since version 3.9.* 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()`.

Added in version 3.9.

int **PyObject_GC_IsFinalized**(*PyObject* *op)

Part of the Stable ABI *since version 3.9.* 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()`.

Added in version 3.9.

void **PyObject_GC_Del**(void *op)

Part of the Stable ABI. Releases memory allocated to an object using *PyObject_GC_New* or *PyObject_GC_NewVar*.

Do not call this directly to free an object's memory; call the type's *tp_free* slot instead.

Do not use this for memory allocated by *PyObject_New*, *PyObject_NewVar*, or related allocation functions; use *PyObject_Free()* instead.

Ver también

- *PyObject_Free()* is the non-GC equivalent of this function.
- *PyObject_GC_New*
- *PyObject_GC_NewVar*
- *PyType_GenericAlloc()*
- *tp_free*

void **PyObject_GC_UnTrack**(void *op)

Part of the Stable ABI. 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: The `_PyObject_GC_TRACK()` and `_PyObject_GC_UNTRACK()` macros have been removed from the public C API.

El manejador *tp_traverse* acepta un parámetro de función de este tipo:

typedef int (***visitproc**)(*PyObject* *object, void *arg)

Part of the Stable ABI. 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:

typedef int (***traverseproc**)(*PyObject* *self, *visitproc* visit, void *arg)

Part of the Stable ABI. 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*:

Py_VISIT(o)

If the *PyObject** *o* is not NULL, call the *visit* callback, with arguments *o* and *arg*. If *visit* returns a non-zero value, then return it. Using this macro, *tp_traverse* handlers look like:

```
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.

typedef int (***inquiry**)(*PyObject* *self)

Part of the Stable ABI. 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.

13.5.1 Controlar el estado del recolector de basura

La C-API proporciona las siguientes funciones para controlar las ejecuciones de recolección de basura.

Py_ssize_t **PyGC_Collect**(void)

Part of the Stable ABI. Realiza una recolección de basura completa, si el recolector de basura está habilitado. (Tenga en cuenta que `gc.collect()` lo ejecuta incondicionalmente).

Retorna el número de objetos recolectados e inalcanzables que no se pueden recolectar. Si el recolector de basura está deshabilitado o ya está recolectando, retorna `0` inmediatamente. Los errores durante la recolección de basura se pasan a `sys.unraisablehook`. Esta función no genera excepciones.

int **PyGC_Enable**(void)

Part of the Stable ABI *since version 3.10.* Habilita el recolector de basura: similar a `gc.enable()`. Retorna el estado anterior, 0 para deshabilitado y 1 para habilitado.

Added in version 3.10.

int **PyGC_Disable**(void)

Part of the Stable ABI *since version 3.10.* Deshabilita el recolector de basura: similar a `gc.disable()`. Retorna el estado anterior, 0 para deshabilitado y 1 para habilitado.

Added in version 3.10.

int **PyGC_IsEnabled**(void)

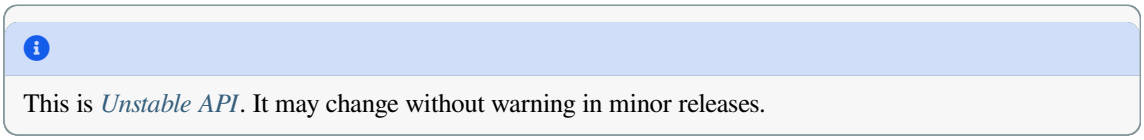
Part of the Stable ABI *since version 3.10.* Consulta el estado del recolector de basura: similar a `gc.isenabled()`. Retorna el estado actual, 0 para deshabilitado y 1 para habilitado.

Added in version 3.10.

13.5.2 Querying Garbage Collector State

The C-API provides the following interface for querying information about the garbage collector.

void **PyUnstable_GC_VisitObjects**(*gcvisitobjects_t* callback, void *arg)



This is *Unstable API*. It may change without warning in minor releases.

Run supplied *callback* on all live GC-capable objects. *arg* is passed through to all invocations of *callback*.

Advertencia

If new objects are (de)allocated by the callback it is undefined if they will be visited.

Garbage collection is disabled during operation. Explicitly running a collection in the callback may lead to undefined behaviour e.g. visiting the same objects multiple times or not at all.

Added in version 3.12.

typedef int (***gcvisitobjects_t**)(*PyObject* *object, void *arg)

Type of the visitor function to be passed to `PyUnstable_GC_VisitObjects()`. *arg* is the same as the *arg* passed to `PyUnstable_GC_VisitObjects`. Return `1` to continue iteration, return `0` to stop iteration. Other return values are reserved for now so behavior on returning anything else is undefined.

Added in version 3.12.

CAPÍTULO 14

Versiones de API y ABI

14.1 Build-time version constants

CPython exposes its version number in the following macros. Note that these correspond to the version code is **built** with. See *Py_Version* for the version used at **run time**.

Consulte *Estabilidad de la API en C* para obtener una discusión sobre la estabilidad de API y ABI en todas las versiones.

PY_MAJOR_VERSION

El `3` en `3.4.1a2`.

PY_MINOR_VERSION

El `4` en `3.4.1a2`.

PY_MICRO_VERSION

El `1` en `3.4.1a2`.

PY_RELEASE_LEVEL

La `a` en `3.4.1a2`. Puede ser `0xA` para la versión alfa, `0xB` para la versión beta, `0xC` para la versión candidata o `0xF` para la versión final.

PY_RELEASE_SERIAL

El `2` en `3.4.1a2`, cero para lanzamientos finales.

PY_VERSION_HEX

The Python version number encoded in a single integer. See *Py_PACK_FULL_VERSION()* for the encoding details.

Use this for numeric comparisons, for example, `#if PY_VERSION_HEX >= ...`.

14.2 Run-time version

const unsigned long **Py_Version**

Part of the Stable ABI *since version 3.11.* The Python runtime version number encoded in a single constant integer. See *Py_PACK_FULL_VERSION()* for the encoding details. This contains the Python version used at run time.

Use this for numeric comparisons, for example, `if (Py_Version >= ...)`.

Added in version 3.11.

14.3 Bit-packing macros

uint32_t **Py_PACK_FULL_VERSION**(int major, int minor, int micro, int release_level, int release_serial)

Part of the Stable ABI *since version 3.14.* Return the given version, encoded as a single 32-bit integer with the following structure:

Argument	No. of bits	Bit mask	Bit shift	Example values	
				`3.4.1a2`	`3.10.0`
major	8	`0xFF000000`	24	`0x03`	`0x03`
minor	8	`0x00FF0000`	16	`0x04`	`0x0A`
micro	8	`0x0000FF00`	8	`0x01`	`0x00`
release_level	4	`0x000000F0`	4	`0xA`	`0xF`
release_serial	4	`0x0000000F`	0	`0x2`	`0x0`

For example:

Version	`Py_PACK_FULL_VERSION` arguments	Encoded version
`3.4.1a2`	`(3, 4, 1, 0xA, 2)`	`0x030401a2`
`3.10.0`	`(3, 10, 0, 0xF, 0)`	`0x030a00f0`

Out-of range bits in the arguments are ignored. That is, the macro can be defined as:

```
#ifndef Py_PACK_FULL_VERSION
#define Py_PACK_FULL_VERSION(X, Y, Z, LEVEL, SERIAL) ( \
   (((X) & 0xff) << 24) |                              \
   (((Y) & 0xff) << 16) |                              \
   (((Z) & 0xff) << 8) |                               \
   (((LEVEL) & 0xf) << 4) |                            \
   (((SERIAL) & 0xf) << 0))
#endif
```

`Py_PACK_FULL_VERSION` is primarily a macro, intended for use in `#if` directives, but it is also available as an exported function.

Added in version 3.14.

uint32_t **Py_PACK_VERSION**(int major, int minor)

Part of the Stable ABI *since version 3.14.* Equivalent to `Py_PACK_FULL_VERSION(major, minor, 0, 0, 0)`. The result does not correspond to any Python release, but is useful in numeric comparisons.

Added in version 3.14.

CAPÍTULO 15

Monitoring C API

Added in version 3.13.

An extension may need to interact with the event monitoring system. Subscribing to events and registering callbacks can be done via the Python API exposed in `sys.monitoring`.

CAPÍTULO 16

Generating Execution Events

The functions below make it possible for an extension to fire monitoring events as it emulates the execution of Python code. Each of these functions accepts a `PyMonitoringState` struct which contains concise information about the activation state of events, as well as the event arguments, which include a `PyObject*` representing the code object, the instruction offset and sometimes additional, event-specific arguments (see `sys.monitoring` for details about the signatures of the different event callbacks). The `codelike` argument should be an instance of `types.CodeType` or of a type that emulates it.

The VM disables tracing when firing an event, so there is no need for user code to do that.

Monitoring functions should not be called with an exception set, except those listed below as working with the current exception.

type **PyMonitoringState**

Representation of the state of an event type. It is allocated by the user while its contents are maintained by the monitoring API functions described below.

All of the functions below return 0 on success and -1 (with an exception set) on error.

See `sys.monitoring` for descriptions of the events.

int **PyMonitoring_FirePyStartEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_START` event.

int **PyMonitoring_FirePyResumeEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_RESUME` event.

int **PyMonitoring_FirePyReturnEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Fire a `PY_RETURN` event.

int **PyMonitoring_FirePyYieldEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Fire a `PY_YIELD` event.

int **PyMonitoring_FireCallEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *callable, *PyObject* *arg0)

Fire a `CALL` event.

int **PyMonitoring_FireLineEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, int lineno)

Fire a `LINE` event.

int **PyMonitoring_FireJumpEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

Fire a `JUMP` event.

int **PyMonitoring_FireBranchLeftEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

Fire a `BRANCH_LEFT` event.

int **PyMonitoring_FireBranchRightEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

Fire a `BRANCH_RIGHT` event.

int **PyMonitoring_FireCReturnEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Fire a `C_RETURN` event.

int **PyMonitoring_FirePyThrowEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_THROW` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireRaiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `RAISE` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireCRaiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `C_RAISE` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireReraiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `RERAISE` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireExceptionHandledEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire an `EXCEPTION_HANDLED` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FirePyUnwindEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_UNWIND` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireStopIterationEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *value)

Fire a `STOP_ITERATION` event. If `value` is an instance of `StopIteration`, it is used. Otherwise, a new `StopIteration` instance is created with `value` as its argument.

16.1 Managing the Monitoring State

Monitoring states can be managed with the help of monitoring scopes. A scope would typically correspond to a python function.

int **PyMonitoring_EnterScope**(*PyMonitoringState* *state_array, uint64_t *version, const uint8_t *event_types, *Py_ssize_t* length)

Enter a monitored scope. `event_types` is an array of the event IDs for events that may be fired from the scope. For example, the ID of a `PY_START` event is the value `PY_MONITORING_EVENT_PY_START`, which is numerically equal to the base-2 logarithm of `sys.monitoring.events.PY_START`. `state_array` is an array with a monitoring state entry for each event in `event_types`, it is allocated by the user but populated by `PyMonitoring_EnterScope()` with information about the activation state of the event. The size of `event_types` (and hence also of `state_array`) is given in `length`.

The `version` argument is a pointer to a value which should be allocated by the user together with `state_array` and initialized to 0, and then set only by `PyMonitoring_EnterScope()` itself. It allows this function to determine whether event states have changed since the previous call, and to return quickly if they have not.

The scopes referred to here are lexical scopes: a function, class or method. `PyMonitoring_EnterScope()` should be called whenever the lexical scope is entered. Scopes can be reentered, reusing the same *state_array* and *version*, in situations like when emulating a recursive Python function. When a code-like's execution is paused, such as when emulating a generator, the scope needs to be exited and re-entered.

The macros for *event_types* are:

Macro	Event
PY_MONITORING_EVENT_BRANCH_LEFT	BRANCH_LEFT
PY_MONITORING_EVENT_BRANCH_RIGHT	BRANCH_RIGHT
PY_MONITORING_EVENT_CALL	CALL
PY_MONITORING_EVENT_C_RAISE	C_RAISE
PY_MONITORING_EVENT_C_RETURN	C_RETURN
PY_MONITORING_EVENT_EXCEPTION_HANDLED	EXCEPTION_HANDLED
PY_MONITORING_EVENT_INSTRUCTION	INSTRUCTION
PY_MONITORING_EVENT_JUMP	JUMP
PY_MONITORING_EVENT_LINE	LINE
PY_MONITORING_EVENT_PY_RESUME	PY_RESUME
PY_MONITORING_EVENT_PY_RETURN	PY_RETURN
PY_MONITORING_EVENT_PY_START	PY_START
PY_MONITORING_EVENT_PY_THROW	PY_THROW
PY_MONITORING_EVENT_PY_UNWIND	PY_UNWIND
PY_MONITORING_EVENT_PY_YIELD	PY_YIELD
PY_MONITORING_EVENT_RAISE	RAISE
PY_MONITORING_EVENT_RERAISE	RERAISE
PY_MONITORING_EVENT_STOP_ITERATION	STOP_ITERATION

int **PyMonitoring_ExitScope**(void)

Exit the last scope that was entered with `PyMonitoring_EnterScope()`.

int **PY_MONITORING_IS_INSTRUMENTED_EVENT**(uint8_t ev)

Return true if the event corresponding to the event ID *ev* is a local event.

Added in version 3.13.

Obsoleto desde la versión 3.14: This function is *soft deprecated*.

APÉNDICE A

Glosario

>>>
: The default Python prompt of the *interactive* shell. Often seen for code examples which can be executed interactively in the interpreter.

...
: Puede referirse a:

  - The default Python prompt of the *interactive* shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.
  - The three dots form of the Ellipsis object.

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

annotate function
: A function that can be called to retrieve the *annotations* of an object. This function is accessible as the `__annotate__` attribute of functions, classes, and modules. Annotate functions are a subset of *evaluate functions*.

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

  Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions can be retrieved by calling `annotationlib.get_annotations()` on modules, classes, and functions, respectively.

  See *variable annotation*, *function annotation*, **PEP 484**, **PEP 526**, and **PEP 649**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

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

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

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

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

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

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

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

administrador asincrónico de contexto
Un objeto que controla el entorno visible en un sentencia `async with` al definir los métodos `__aenter__()` 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
An object created by an *asynchronous generator* function.

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`.

Each `yield` temporarily suspends processing, remembering the execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See **PEP 492** and **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**.

attached thread state
A *thread state* that is active for the current OS thread.

When a *thread state* is attached, the OS thread has access to the full Python C API and can safely invoke the bytecode interpreter.

Unless a function explicitly notes otherwise, attempting to call the C API without an attached thread state will result in a fatal error or undefined behavior. A thread state can be attached and detached explicitly by the

user through the C API, or implicitly by the runtime, including during blocking C calls and by the bytecode interpreter in between calls.

On most builds of Python, having an attached thread state implies that the caller holds the *GIL* for the current interpreter, so only one OS thread can have an attached thread state at a given moment. In *free-threaded* builds of Python, threads can concurrently hold an attached thread state, allowing for true parallelism of the bytecode interpreter.

atributo
Un valor asociado a un objeto al que se suele hacer referencia por su nombre utilizando expresiones punteadas. Por ejemplo, si un objeto *o* tiene un atributo *a* se referenciaría como *o.a*.

Es posible dar a un objeto un atributo cuyo nombre no sea un identificador definido por identifiers, por ejemplo usando `setattr()`, si el objeto lo permite. Dicho atributo no será accesible utilizando una expresión con puntos, y en su lugar deberá ser recuperado con `getattr()`.

a la espera
Un objeto que puede utilizarse en una expresión `await`. Puede ser una *corutina* o un objeto con un método `__await__()`. Véase también **PEP 492**.

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

archivo binario
A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

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

referencia prestada
En la API C de Python, una referencia prestada es una referencia a un objeto, donde el código usando el objeto no posee la referencia. Se convierte en un puntero colgante si se destruye el objeto. Por ejemplo, una recolección de basura puede eliminar el último *strong reference* del objeto y así destruirlo.

Se recomienda llamar a *`Py_INCREF()`* en la *referencia prestada* para convertirla en una *referencia fuerte* in situ, excepto cuando el objeto no se puede destruir antes del último uso de la referencia prestada. La función *`Py_NewRef()`* se puede utilizar para crear una nueva *referencia fuerte*.

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 corren en una *virtual machine* que ejecute el código de máquina correspondiente a cada *bytecode*. Note que los *bytecodes* no tienen como requisito trabajar en las diversas máquina virtuales de Python, ni de ser estable entre versiones Python.

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

callable
Un callable es un objeto que puede ser llamado, posiblemente con un conjunto de argumentos (véase *argument*), con la siguiente sintaxis:

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

Una *function*, y por extensión un *method*, es un callable. Una instancia de una clase que implementa el método `__call__()` también es un callable.

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).

closure variable
A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable*:

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

Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

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.

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

- The temporary state or environment established by a *context manager* via a `with` statement.
- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
- A `contextvars.Context` object. Also see *current context*.

context management protocol
The `__enter__()` and `__exit__()` methods called by the `with` statement. See **PEP 343**.

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

variable de contexto
A variable whose value depends on which context is the *current context*. Values are accessed via

`contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contiguo
Un búfer es considerado contiguo con precisión si es *C-contiguo* o *Fortran contiguo*. Los búferes cero dimensionales con C y Fortran contiguos. En los arreglos unidimensionales, los ítems deben ser dispuestos en memoria uno siguiente al otro, ordenados por índices que comienzan en cero. En arreglos unidimensionales C-contiguos, el último índice varía más velozmente en el orden de las direcciones de memoria. Sin embargo, en arreglos Fortran contiguos, el primer índice vería más rápidamente.

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

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

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

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

cyclic isolate
A subgroup of one or more objects that reference each other in a reference cycle, but are not referenced by objects outside the group. The goal of the *cyclic garbage collector* is to identify these groups and break the reference cycles so that the memory can be reclaimed.

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
Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

Para obtener más información sobre los métodos de los descriptores, consulte descriptors o Guía práctica de uso de los descriptores.

diccionario
An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

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
A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

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*.

dunder
An informal short-hand for «double underscore», used when talking about a *special method*. For example, `__init__` is often pronounced «dunder init».

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

evaluate function
A function that can be called to evaluate a lazily evaluated attribute of an object, such as the value of type aliases created with the `type` statement.

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

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

f-string
f-strings
String literals prefixed with `f` or `F` are commonly called «f-strings» which is short for formatted string literals. See also **PEP 498**.

objeto archivo
An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource.

Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

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*.

codificación del sistema de archivos y manejador de errores
Controlador de errores y codificación utilizado por Python para decodificar bytes del sistema operativo y codificar Unicode en el sistema operativo.

La codificación del sistema de archivos debe garantizar la decodificación exitosa de todos los bytes por debajo de 128. Si la codificación del sistema de archivos no proporciona esta garantía, las funciones de API pueden lanzar `UnicodeError`.

Las funciones `sys.getfilesystemencoding()` y `sys.getfilesystemencodeerrors()` se pueden utilizar para obtener la codificación del sistema de archivos y el controlador de errores.

La *codificación del sistema de archivos y el manejador de errores* se configuran al inicio de Python mediante la función *`PyConfig_Read()`*: consulte los miembros *`filesystem_encoding`* y *`filesystem_errors`* de *`PyConfig`*.

Vea también *locale encoding*.

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

There are two types of finder: *meta path finders* for use with `sys.meta_path`, and *path entry finders* for use with `sys.path_hooks`.

See finders-and-loaders and `importlib` for much more detail.

división entera a la baja
Una división matemática que se redondea hacia el entero menor más cercano. El operador de la división entera a la baja 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**.

free threading
A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

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

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.

Consulte *variable annotation* y **PEP 484**, que describen esta funcionalidad. Consulte también annotations-howto para conocer las mejores prácticas sobre cómo trabajar con anotaciones.

__future__
Un future statement, `from __future__ import <feature>`, indica al compilador que compile el módulo actual utilizando una sintaxis o semántica que se convertirá en estándar en una versión futura de Python. El módulo `__future__` documenta los posibles valores de *feature*. Al importar este módulo y evaluar sus variables, puede ver cuándo se agregó por primera vez una nueva característica al lenguaje y cuándo se convertirá (o se convirtió) en la predeterminada:

```
>>> 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*.

Each `yield` temporarily suspends processing, remembering the execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

expresión generadora
An *expression* that returns an *iterator*. It looks like a normal expression followed by a `for` clause defining a loop variable, range, and an optional `if` clause. The combined expression generates values for an enclosing function:

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

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
Un *type* que se puede parametrizar; normalmente un container class como `list` o `dict`. Usado para *type hints* y *annotations*.

Para más detalles, véase generic alias types, **PEP 483**, **PEP 484**, **PEP 585**, y el módulo `typing`.

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.

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

In prior versions of Python's C API, a function might declare that it requires the GIL to be held in order to use it. This refers to having an *attached thread state*.

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
An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

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
Un Entorno Integrado de Desarrollo y Aprendizaje para Python. idle es un editor básico y un entorno de intérprete que se incluye con la distribución estándar de Python.

immortal
Immortal objects are a CPython implementation detail introduced in **PEP 683**.

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

Immortal objects can be identified via `sys._is_immortal()`, or via *`PyUnstable_IsImmortal()`* in the C API.

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

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

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

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

interactivo
Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`). For more on interactive mode, see tut-interac.

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

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

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

iterable
An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *sequence* semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, …). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterador
An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

Puede encontrar más información en typeiter.

Detalles de implementación de CPython: CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

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])`. Además, `operator.attrgetter()`, `operator.itemgetter()` y `operator.methodcaller()` son tres constructores de funciones clave. Consulte Sorting HOW TO para ver ejemplos de cómo crear y utilizar 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.

lexical analyzer
Formal name for the *tokenizer*; see *token*.

lista
A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is *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
An object that loads a module. It must define the `exec_module()` and `create_module()` methods to implement the `Loader` interface. A loader is typically returned by a *finder*. See also:

- finders-and-loaders
- `importlib.abc.Loader`
- **PEP 302**

codificación de la configuración regional
En Unix, es la codificación de la configuración regional LC_CTYPE. Se puede configurar con `locale.setlocale(locale.LC_CTYPE, new_locale)`.

En Windows, es la página de códigos ANSI (por ejemplo, `"cp1252"`).

En Android y VxWorks, Python utiliza `"utf-8"` como codificación regional.

`locale.getencoding()` can be used to get the locale encoding.

Vea también *filesystem encoding and error handler*.

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 `collections.abc.Mapping` o `collections.abc.MutableMapping` abstract base classes. Por ejemplo, `dict`, `collections.defaultdict`, `collections.OrderedDict` y `collections.Counter`.

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

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

metaclase
La clase de una clase. Las definiciones de clases crean nombres de clase, un diccionario de clase, y una lista

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

Más información hallará en metaclasses.

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

orden de resolución de métodos
Method Resolution Order is the order in which base classes are searched for a member during lookup. See python_2.3_mro for details of the algorithm used by the Python interpreter since the 2.3 release.

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`.

See also module-specs.

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
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand, or it can be created by inheriting `typing.NamedTuple`, or with the factory function `collections.namedtuple()`. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

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

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

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

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
Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

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

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

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
A callable on the `sys.path_hooks` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

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

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

paquete provisorio
Vea *provisional API*.

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

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

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

En contraste, un método Pythónico más limpio:

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

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

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

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

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

contador de referencias
The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Some objects are *immortal* and have reference counts that are never modified, and therefore the objects are never deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. Programmers can call the `sys.getrefcount()` function to return the reference count for a particular object.

In *CPython*, reference counts are not considered to be stable or well-defined values; the number of references to an object, and how that number is affected by Python code, may be different between versions.

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

Vea también *namespace package*.

REPL
An acronym for the «read–eval–print loop», another name for the *interactive* interpreter shell.

__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
An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *hashable* keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see Common Sequence Operations.

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 suscripto, `[]` con dos puntos entre los números cuando se ponen varios, como en `nombre_variable[1:3:5]`. La notación con corchete (suscrito) usa internamente objetos `slice`.

soft deprecated
A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See PEP 387: Soft Deprecation.

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.

standard library
The collection of *packages*, *modules* and *extension modules* distributed as a part of the official Python interpreter package. The exact membership of the collection may vary based on platform, available system libraries, or other criteria. Documentation can be found at library-index.

See also `sys.stdlib_module_names` for a list of all possible standard library module names.

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`.

static type checker
An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the `typing` module.

stdlib
An abbreviation of *standard library*.

referencia fuerte
En la API de C de Python, una referencia fuerte es una referencia a un objeto que es propiedad del código que mantiene la referencia. La referencia fuerte se toma llamando a `Py_INCREF()` cuando se crea la referencia y se libera con `Py_DECREF()` cuando se elimina la referencia.

La función `Py_NewRef()` se puede utilizar para crear una referencia fuerte a un objeto. Por lo general, se debe llamar a la función `Py_DECREF()` en la referencia fuerte antes de salir del alcance de la referencia fuerte, para evitar filtrar una referencia.

Consulte también *borrowed reference*.

t-string
t-strings
String literals prefixed with `t` or `T` are commonly called «t-strings» which is short for template string literals.

codificación de texto
Una cadena de caracteres en Python es una secuencia de puntos de código Unicode (en el rango `U+0000`–`U+10FFFF`). Para almacenar o transferir una cadena de caracteres, es necesario serializarla como una secuencia de bytes.

La serialización de una cadena de caracteres en una secuencia de bytes se conoce como «codificación», y la recreación de la cadena de caracteres a partir de la secuencia de bytes se conoce como «decodificación».

Existe una gran variedad de serializaciones de texto codecs, que se denominan colectivamente «codificaciones de texto».

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*.

thread state
The information used by the *CPython* runtime to run in an OS thread. For example, this includes the current exception, if any, and the state of the bytecode interpreter.

Each thread state is bound to a single OS thread, but threads may have many thread states available. At most, one of them may be *attached* at once.

An *attached thread state* is required to call most of Python's C API, unless a function explicitly documents otherwise. The bytecode interpreter only runs under an attached thread state.

Each thread state belongs to a single interpreter, but each interpreter may have many thread states, including multiple for the same OS thread. Thread states from multiple interpreters may be bound to the same thread, but only one can be *attached* in that thread at any given moment.

See *Thread State and the Global Interpreter Lock* for more information.

token
A small unit of source code, generated by the lexical analyzer (also called the *tokenizer*). Names, numbers, strings, operators, newlines and similar are represented by tokens.

The `tokenize` module exposes Python's lexical analyzer. The `token` module contains information on the various types of tokens.

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
The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `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.

Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

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.

Consulte *function annotation*, **PEP 484** y **PEP 526**, que describen esta funcionalidad. Consulte también annotations-howto para conocer las mejores prácticas sobre cómo trabajar con anotaciones.

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*.

walrus operator
: A light-hearted way to refer to the assignment expression operator `:=` because it looks a bit like a walrus if you turn your head.

Zen de Python
: Un listado de los principios de diseño y la filosofía de Python que son útiles para entender y usar el lenguaje. El listado puede encontrarse ingresando «`import this`» en la consola interactiva.

APÉNDICE B

About this documentation

Python's documentation is generated from reStructuredText sources using Sphinx, a documentation generator originally created for Python and now maintained as an independent project.

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., the creator of the original Python documentation toolset and author of much of the content;
- el proyecto Docutils para creación de reStructuredText y la suite Docutils;
- Fredrik Lundh por su proyecto Referencia Alternativa de Python del que Sphinx obtuvo muchas buenas ideas.

B.1 Contributors to the Python documentation

Muchas personas han contribuido para el lenguaje de Python, la librería estándar de Python, y la documentación de Python. Revisa Misc/ACKS la distribución de Python para una lista parcial de contribuidores.

Es solamente con la aportación y contribuciones de la comunidad de Python que Python tiene tan fantástica documentación – Muchas gracias!

APÉNDICE C

Historia y Licencia

C.1 Historia del software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations, which became Zope Corporation. In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation was a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Lanzamiento	Derivado de	Año	Dueño/a	GPL-compatible? (1)
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	yes (2)
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

(1) GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible

licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

(2) According to Richard Stallman, 1.6.1 is not GPL-compatible, because its license has a choice of law clause. According to CNRI, however, Stallman's lawyer has told CNRI's lawyer that 1.6.1 is «not incompatible» with the GPL.

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

Python software and documentation are licensed under the Python Software Foundation License Version 2.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Version 2 and the *Zero-Clause BSD license*.

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 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001 Python Software Foundation; All Rights
   Reserved" are retained in Python alone or in any derivative version
   prepared by Licensee.

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

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

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

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

7. Nothing in this License Agreement shall be deemed to create any relationship
```

(continúe en la próxima página)

(proviene de la página anterior)

```
   of agency, partnership, or joint venture between PSF and Licensee.  This License
   Agreement does not grant permission to use PSF trademarks or trade name in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.

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

C.2.2 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.

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
   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.
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

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

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 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

La extensión C `_random` subyacente al 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)
```

(continúe en la próxima página)

(proviene de la página anterior)

```
or init_by_array(init_key, key_length).

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

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

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

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

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.
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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 `test.support.asynchat` y `test.support.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
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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

The `uu` codec contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
```

(continúe en la próxima página)

(proviene de la página anterior)

```
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 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.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:
```

(continúe en la próxima página)

(proviene de la página anterior)

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

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

C.3.9 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.
```

(continúe en la próxima página)

(proviene de la página anterior)

```
</MIT License>

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

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

C.3.11 strtod y dtoa

El archivo `Python/dtoa.c`, que proporciona las funciones de C dtoa y strtod para la conversión de dobles C hacia y desde cadenas de caracteres, se deriva del archivo del mismo nombre por David M. Gay, actualmente disponible en https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. 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` and `ssl` 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. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

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

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

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

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

(continúe en la próxima página)

(proviene de la página anterior)

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

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

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

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

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

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

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

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

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
```

(continúe en la próxima página)

(proviene de la página anterior)

```
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

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

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

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

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

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

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

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
```

(continúe en la próxima página)

(proviene de la página anterior)

```
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

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

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

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

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

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

END OF TERMS AND CONDITIONS
```

C.3.13 expat

La extensión `pyexpat` se construye usando una copia incluida de las fuentes de expatriados 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 C `_ctypes` subyacente al módulo `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
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
```

(continúe en la próxima página)

(proviene de la página anterior)

```
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

La extension C `_decimal` subyacente al 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

El conjunto de pruebas C14N 2.0 en el paquete `test` (`Lib/test/xmltestdata/c14n-20/`) se recuperó del sitio web de W3C en https://www.w3.org/TR/xml-c14n2-testcases/ y se distribuye bajo la licencia BSD de 3 cláusulas:

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

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

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

(continúe en la próxima página)

(proviene de la página anterior)

```
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT License:

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

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

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

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

C.3.20 asyncio

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license:

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

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

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

(continúe en la próxima página)

(proviene de la página anterior)

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

C.3.21 Global Unbounded Sequences (GUS)

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

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

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   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 "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 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.3.22 Zstandard bindings

Zstandard bindings in `Modules/_zstd` and `Lib/compression/zstd` are based on code from the pyzstd library, copyright Ma Lin and contributors. The pyzstd code is distributed under the 3-Clause BSD License:

```
Copyright (c) 2020-present, Ma Lin and contributors.
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 copyright holder nor the names of its
   contributors may be used to endorse or promote products derived from
```

(continúe en la próxima página)

(proviene de la página anterior)

```
   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 HOLDER 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 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.

Bibliografía

[win] `PyExc_WindowsError` is only defined on Windows; protect code that uses this by testing that the preprocessor macro `MS_WINDOWS` is defined.

Índice

No alfabético

. . ., **375**
    ellipsis literal, 375
>>>, **375**
__all__ (*package variable*), 83
__dict__ (*module attribute*), 205
__doc__ (*module attribute*), 205
__file__ (*module attribute*), 205, 206
__future__, **382**
__import__
    built-in function, 83
__loader__ (*module attribute*), 205
__main__
    module, 12, 235, 251, 252
__name__ (*module attribute*), 205
__package__ (*module attribute*), 205
__PYVENV_LAUNCHER__, 276, 282
__slots__, **390**
_frozen (*C struct*), 85
_inittab (*C struct*), 86
_inittab.initfunc (*C member*), 86
_inittab.name (*C member*), 86
_Py_c_diff (*C function*), 157
_Py_c_neg (*C function*), 157
_Py_c_pow (*C function*), 157
_Py_c_prod (*C function*), 157
_Py_c_quot (*C function*), 157
_Py_c_sum (*C function*), 157
_Py_NoneStruct (*C var*), 305
_PyBytes_Resize (*C function*), 160
_PyCode_GetExtra (*C function*), 203
_PyCode_SetExtra (*C function*), 203
_PyEval_RequestCodeExtraIndex (*C function*), 203
_PyFrameEvalFunction (*C type*), 249
_PyInterpreterFrame (*C struct*), 222
_PyInterpreterState_GetEvalFrameFunc (*C function*), 249
_PyInterpreterState_SetEvalFrameFunc (*C function*), 249
_PyObject_GetDictPtr (*C function*), 111
_PyObject_New (*C function*), 303
_PyObject_NewVar (*C function*), 303
_PyTuple_Resize (*C function*), 184
_thread
    module, 245

A

a la espera, **377**
abort (*C function*), 82
abs
    built-in function, 123
administrador asincrónico de contexto, **376**
administrador de contextos, **378**
alcances anidados, **387**
alias de tipos, **392**
allocfunc (*C type*), 355
annotate function, **375**
anotación, **375**
anotación de función, **381**
anotación de variable, **392**
apagado del intérprete, **384**
API provisional, **388**
archivo binario, **377**
archivo de texto, **391**
argumento, **375**
argumento nombrado, **385**
argumento posicional, **388**
argv (*in module sys*), 240, 275
ascii
    built-in function, 111
atributo, **377**
attached thread state, **376**

B

BDFL, **377**
binaryfunc (*C type*), 356
bloqueo global del intérprete, **382**
buffer interface
    (see buffer protocol), 129
buffer object
    (see buffer protocol), 129
buffer protocol, 129
built-in function
    __import__, 83
    abs, 123
    ascii, 111

bytes, 112
classmethod, 314
compile, 84
divmod, 122
float, 124
hash, 112, 330
int, 124
len, 113, 125, 127, 190, 194
pow, 122, 124
repr, 111, 329
staticmethod, 314
tuple, 126
type, 112
builtins
module, 12, 235, 251, 252
buscador, **381**
buscador basado en ruta, **388**
buscador de entradas de ruta, **388**
bytearray
object, 160
bytecode, **377**
bytes
built-in function, 112
object, 158

C

cadena con triple comilla, **391**
callable, **377**
calloc (*C function*), 291
Cápsula
objeto, 218
cargador, **385**
C-contiguous, 132, 379
clase, **378**
clase base abstracta, **375**
clase de nuevo estilo, **387**
classmethod
built-in function, 314
cleanup functions, 82
close (*in module os*), 252
closure variable, **378**
CO_ASYNC_GENERATOR (*C macro*), 202
CO_COROUTINE (*C macro*), 202
CO_FUTURE_ABSOLUTE_IMPORT (*C macro*), 202
CO_FUTURE_ANNOTATIONS (*C macro*), 202
CO_FUTURE_DIVISION (*C macro*), 202
CO_FUTURE_GENERATOR_STOP (*C macro*), 202
CO_FUTURE_PRINT_FUNCTION (*C macro*), 202
CO_FUTURE_UNICODE_LITERALS (*C macro*), 202
CO_FUTURE_WITH_STATEMENT (*C macro*), 202
CO_GENERATOR (*C macro*), 202
CO_HAS_DOCSTRING (*C macro*), 202
CO_ITERABLE_COROUTINE (*C macro*), 202
CO_METHOD (*C macro*), 202
CO_NESTED (*C macro*), 202
CO_NEWLOCALS (*C macro*), 202
CO_OPTIMIZED (*C macro*), 202
CO_VARARGS (*C macro*), 202
CO_VARKEYWORDS (*C macro*), 202
code object, 198
codificación de la configuración regional, **385**
codificación de texto, **391**
codificación del sistema de archivos y manejador de errores, **381**
Common Vulnerabilities and Exposures
CVE 2008-5983, 241
compile
built-in function, 84
complex number
object, 156
comprensión de conjuntos, **390**
comprensión de diccionarios, **380**
comprensión de listas, **385**
contador de referencias, **389**
context, **378**
context management protocol, **378**
contiguo, **379**
contiguous, 132
copyright (*in module sys*), 240
corrutina, **379**
CPython, **379**
current context, **379**
cyclic isolate, **379**

D

decorador, **379**
descrgetfunc (*C type*), 355
descriptor, **379**
descrsetfunc (*C type*), 355
despacho único, **390**
destructor (*C type*), 355
diccionario, **380**
dictionary
object, 188
división entera a la baja, **381**
divmod
built-in function, 122
docstring, **380**
dunder, **380**

E

EAFP, **380**
entorno virtual, **392**
entrada de ruta, **388**
EOFError (*built-in exception*), 204
espacio de nombres, **386**
especificador de módulo, **386**
evaluate function, **380**
exc_info (*in module sys*), 11
executable (*in module sys*), 239
exit (*C function*), 82
expresión, **380**
expresión generadora, **382**

F

f-string, **380**
f-strings, **380**
file
    object, 204
float
    built-in function, 124
floating-point
    object, 154
Fortran contiguous, 132, 379
free (*C function*), 291
free threading, **381**
free variable, **381**
freefunc (*C type*), 355
freeze utility, 86
frozenset
    object, 193
función, **381**
función clave, **384**
función corrutina, **379**
función genérica, **382**
función integrada
    len, 186
    tupla, 188
function
    object, 195

G

gancho a entrada de ruta, **388**
gcvisitobjects_t (*C type*), 363
generador, **382**
generador asincrónico, **376**
getattrfunc (*C type*), 355
getattrofunc (*C type*), 355
getbufferproc (*C type*), 356
getiterfunc (*C type*), 356
getter (*C type*), 318
GIL, **382**
global interpreter lock, 242

H

hash
    built-in function, 112, 330
hash-based pyc, **383**
hashable, **383**
hashfunc (*C type*), 355

I

IDLE, **383**
immortal, **383**
importador, **383**
importar, **383**
incr_item(), 11, 12
indicador de tipo, **392**
initproc (*C type*), 355
inmutable, **383**
inquiry (*C type*), 362
instancemethod
    object, 197
int
    built-in function, 124
integer
    object, 144
interactivo, **383**
interpretado, **384**
interpreter lock, 242
iterable, **384**
iterable asincrónico, **376**
iterador, **384**
iterador asincrónico, **376**
iterador generador, **382**
iterador generador asincrónico, **376**
iternextfunc (*C type*), 356

K

KeyboardInterrupt (*built-in exception*), 61

L

lambda, **385**
LBYL, **385**
len
    built-in function, 113, 125, 127, 190, 194
    función integrada, 186
lenfunc (*C type*), 356
lexical analyzer, **385**
lista, **385**
    objeto, 186
lock, interpreter, 242
long integer
    object, 144
LONG_MAX (*C macro*), 146

M

magic
    método, 385
main(), 238, 240, 275
malloc (*C function*), 291
mapeado, **385**
mapping
    object, 188
máquina virtual, **393**
memoryview
    object, 215
meta buscadores de ruta, **385**
metaclase, **385**
METH_CLASS (*C macro*), 313
METH_COEXIST (*C macro*), 314
METH_FASTCALL (*C macro*), 313
METH_KEYWORDS (*C macro*), 313
METH_METHOD (*C macro*), 313
METH_NOARGS (*C macro*), 313
METH_O (*C macro*), 313
METH_STATIC (*C macro*), 314
METH_VARARGS (*C macro*), 312
method

object, 197
MethodType (*in module types*), 195, 197
método, **386**
magic, 385
special, 390
método especial, **390**
método mágico, **385**
module
__main__, 12, 235, 251, 252
_thread, 245
builtins, 12, 235, 251, 252
object, 205
search path, 12, 235, 239
signal, 61
sys, 12, 235, 251, 252
modules (*in module sys*), 83, 235
ModuleType (*in module types*), 205
módulo, **386**
módulo de extensión, **380**
MRO, **386**
mutable, **386**

N

newfunc (*C type*), 355
nombre calificado, **389**
None
object, 144
numeric
object, 144
número complejo, **378**

O

object
bytearray, 160
bytes, 158
code, 198
complex number, 156
dictionary, 188
file, 204
floating-point, 154
frozenset, 193
function, 195
instancemethod, 197
integer, 144
long integer, 144
mapping, 188
memoryview, 215
method, 197
module, 205
None, 144
numeric, 144
sequence, 158
set, 193
tuple, 183
type, 7, 137
objeto, **387**
Cápsula, 218
lista, 186
objeto archivo, **380**
objeto tipo ruta, **388**
objetos tipo archivo, **381**
objetos tipo binarios, **377**
objobjargproc (*C type*), 356
objobjproc (*C type*), 356
optimized scope, **387**
orden de resolución de métodos, **386**
OverflowError (*built-in exception*), 146, 147

P

package variable
__all__, 83
paquete, **387**
paquete de espacios de nombres, **387**
paquete provisorio, **389**
paquete regular, **390**
parámetro, **387**
path
module search, 12, 235, 239
PATH, 12
path (*in module sys*), 12, 235, 239
PEP, **388**
platform (*in module sys*), 240
porción, **388**
pow
built-in function, 122, 124
Py_ABS (*C macro*), 4
Py_AddPendingCall (*C function*), 253
Py_ALWAYS_INLINE (*C macro*), 4
Py_ASNATIVEBYTES_ALLOW_INDEX (*C macro*), 150
Py_ASNATIVEBYTES_BIG_ENDIAN (*C macro*), 150
Py_ASNATIVEBYTES_DEFAULTS (*C macro*), 150
Py_ASNATIVEBYTES_LITTLE_ENDIAN (*C macro*), 150
Py_ASNATIVEBYTES_NATIVE_ENDIAN (*C macro*), 150
Py_ASNATIVEBYTES_REJECT_NEGATIVE (*C macro*), 150
Py_ASNATIVEBYTES_UNSIGNED_BUFFER (*C macro*), 150
Py_AtExit (*C function*), 82
Py_AUDIT_READ (*C macro*), 315
Py_AuditHookFunction (*C type*), 82
Py_BEGIN_ALLOW_THREADS (*C macro*), 242, 247
Py_BEGIN_CRITICAL_SECTION (*C macro*), 260
Py_BEGIN_CRITICAL_SECTION2 (*C macro*), 261
Py_BEGIN_CRITICAL_SECTION2_MUTEX (*C macro*), 261
Py_BEGIN_CRITICAL_SECTION_MUTEX (*C macro*), 260
Py_BLOCK_THREADS (*C macro*), 247
Py_buffer (*C type*), 130
Py_buffer.buf (*C member*), 130
Py_buffer.format (*C member*), 131
Py_buffer.internal (*C member*), 131
Py_buffer.itemsize (*C member*), 130
Py_buffer.len (*C member*), 130

Py_buffer.ndim (*C member*), 131
Py_buffer.obj (*C member*), 130
Py_buffer.readonly (*C member*), 130
Py_buffer.shape (*C member*), 131
Py_buffer.strides (*C member*), 131
Py_buffer.suboffsets (*C member*), 131
Py_BuildValue (*C function*), 94
Py_BytesMain (*C function*), 237
Py_BytesWarningFlag (*C var*), 232
Py_CHARMASK (*C macro*), 5
Py_CLEANUP_SUPPORTED (*C macro*), 92
Py_CLEAR (*C function*), 51
Py_CompileString (*C function*), 45, 46
Py_CompileStringExFlags (*C function*), 46
Py_CompileStringFlags (*C function*), 45
Py_CompileStringObject (*C function*), 45
Py_complex (*C type*), 156
Py_complex.imag (*C member*), 156
Py_complex.real (*C member*), 156
Py_CONSTANT_ELLIPSIS (*C macro*), 108
Py_CONSTANT_EMPTY_BYTES (*C macro*), 108
Py_CONSTANT_EMPTY_STR (*C macro*), 108
Py_CONSTANT_EMPTY_TUPLE (*C macro*), 108
Py_CONSTANT_FALSE (*C macro*), 108
Py_CONSTANT_NONE (*C macro*), 108
Py_CONSTANT_NOT_IMPLEMENTED (*C macro*), 108
Py_CONSTANT_ONE (*C macro*), 108
Py_CONSTANT_TRUE (*C macro*), 108
Py_CONSTANT_ZERO (*C macro*), 108
PY_CXX_CONST (*C macro*), 94
Py_DEBUG (*C macro*), 13
Py_DebugFlag (*C var*), 232
Py_DecodeLocale (*C function*), 78
Py_DecRef (*C function*), 51
Py_DECREF (*C function*), 7, 50
Py_DEPRECATED (*C macro*), 5
Py_DontWriteBytecodeFlag (*C var*), 233
Py_Ellipsis (*C var*), 215
Py_EncodeLocale (*C function*), 79
Py_END_ALLOW_THREADS (*C macro*), 242, 247
Py_END_CRITICAL_SECTION (*C macro*), 261
Py_END_CRITICAL_SECTION2 (*C macro*), 261
Py_EndInterpreter (*C function*), 252
Py_EnterRecursiveCall (*C function*), 65
Py_EQ (*C macro*), 340
Py_eval_input (*C var*), 46
Py_Exit (*C function*), 82
Py_ExitStatusException (*C function*), 271
Py_False (*C var*), 154
Py_FatalError (*C function*), 82
Py_FatalError(), 240
Py_fclose (*C function*), 80
Py_FdIsInteractive (*C function*), 77
Py_file_input (*C var*), 46
Py_Finalize (*C function*), 237
Py_FinalizeEx (*C function*), 82, 235, 236, 252
Py_fopen (*C function*), 80
Py_FrozenFlag (*C var*), 233
Py_GE (*C macro*), 340
Py_GenericAlias (*C function*), 229
Py_GenericAliasType (*C var*), 229
Py_GetArgcArgv (*C function*), 289
Py_GetBuildInfo (*C function*), 240
Py_GetCompiler (*C function*), 240
Py_GetConstant (*C function*), 107
Py_GetConstantBorrowed (*C function*), 108
Py_GetCopyright (*C function*), 240
Py_GETENV (*C macro*), 5
Py_GetExecPrefix (*C function*), 12, 239
Py_GetPath (*C function*), 12, 239
Py_GetPath(), 238
Py_GetPlatform (*C function*), 240
Py_GetPrefix (*C function*), 12, 238
Py_GetProgramFullPath (*C function*), 12, 239
Py_GetProgramName (*C function*), 238
Py_GetPythonHome (*C function*), 241
Py_GetVersion (*C function*), 240
Py_GT (*C macro*), 340
Py_hash_t (*C type*), 98
Py_HashBuffer (*C function*), 99
Py_HashPointer (*C function*), 99
Py_HashRandomizationFlag (*C var*), 233
Py_IgnoreEnvironmentFlag (*C var*), 233
Py_IncRef (*C function*), 51
Py_INCREF (*C function*), 7, 49
Py_Initialize (*C function*), 12, 235, 252
Py_Initialize(), 238
Py_InitializeEx (*C function*), 236
Py_InitializeFromConfig (*C function*), 236
Py_InitializeFromInitConfig (*C function*), 266
Py_InspectFlag (*C var*), 233
Py_InteractiveFlag (*C var*), 233
Py_Is (*C function*), 310
Py_IS_TYPE (*C function*), 310
Py_IsFalse (*C function*), 310
Py_IsFinalizing (*C function*), 236
Py_IsInitialized (*C function*), 13, 236
Py_IsNone (*C function*), 310
Py_IsolatedFlag (*C var*), 234
Py_IsTrue (*C function*), 310
Py_LE (*C macro*), 340
Py_LeaveRecursiveCall (*C function*), 65
Py_LegacyWindowsFSEncodingFlag (*C var*), 234
Py_LegacyWindowsStdioFlag (*C var*), 234
Py_LIMITED_API (*C macro*), 16
Py_LT (*C macro*), 340
Py_Main (*C function*), 237
PY_MAJOR_VERSION (*C macro*), 365
Py_MARSHAL_VERSION (*C macro*), 87
Py_MAX (*C macro*), 5
Py_MEMBER_SIZE (*C macro*), 5
PY_MICRO_VERSION (*C macro*), 365
Py_MIN (*C macro*), 5
PY_MINOR_VERSION (*C macro*), 365
Py_mod_create (*C macro*), 208
Py_mod_exec (*C macro*), 208

Py_mod_gil (*C macro*), 208
Py_MOD_GIL_NOT_USED (*C macro*), 208
Py_MOD_GIL_USED (*C macro*), 208
Py_mod_multiple_interpreters (*C macro*), 208
Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED (*C macro*), 208
Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED (*C macro*), 208
Py_MOD_PER_INTERPRETER_GIL_SUPPORTED (*C macro*), 208
PY_MONITORING_EVENT_BRANCH_LEFT (*C macro*), 372
PY_MONITORING_EVENT_BRANCH_RIGHT (*C macro*), 372
PY_MONITORING_EVENT_C_RAISE (*C macro*), 372
PY_MONITORING_EVENT_C_RETURN (*C macro*), 372
PY_MONITORING_EVENT_CALL (*C macro*), 372
PY_MONITORING_EVENT_EXCEPTION_HANDLED (*C macro*), 372
PY_MONITORING_EVENT_INSTRUCTION (*C macro*), 372
PY_MONITORING_EVENT_JUMP (*C macro*), 372
PY_MONITORING_EVENT_LINE (*C macro*), 372
PY_MONITORING_EVENT_PY_RESUME (*C macro*), 372
PY_MONITORING_EVENT_PY_RETURN (*C macro*), 372
PY_MONITORING_EVENT_PY_START (*C macro*), 372
PY_MONITORING_EVENT_PY_THROW (*C macro*), 372
PY_MONITORING_EVENT_PY_UNWIND (*C macro*), 372
PY_MONITORING_EVENT_PY_YIELD (*C macro*), 372
PY_MONITORING_EVENT_RAISE (*C macro*), 372
PY_MONITORING_EVENT_RERAISE (*C macro*), 372
PY_MONITORING_EVENT_STOP_ITERATION (*C macro*), 372
PY_MONITORING_IS_INSTRUMENTED_EVENT (*C function*), 373
Py_NE (*C macro*), 340
Py_NewInterpreter (*C function*), 252
Py_NewInterpreterFromConfig (*C function*), 251
Py_NewRef (*C function*), 50
Py_NO_INLINE (*C macro*), 5
Py_None (*C var*), 144
Py_NoSiteFlag (*C var*), 234
Py_NotImplemented (*C var*), 108
Py_NoUserSiteDirectory (*C var*), 234
Py_OpenCodeHookFunction (*C type*), 204
Py_OptimizeFlag (*C var*), 235
Py_PACK_FULL_VERSION (*C function*), 366
Py_PACK_VERSION (*C function*), 366
Py_PreInitialize (*C function*), 273
Py_PreInitializeFromArgs (*C function*), 273
Py_PreInitializeFromBytesArgs (*C function*), 273
Py_PRINT_RAW (*C macro*), 108, 205
Py_QuietFlag (*C var*), 235
Py_READONLY (*C macro*), 315
Py_REFCNT (*C function*), 49
Py_RELATIVE_OFFSET (*C macro*), 315
PY_RELEASE_LEVEL (*C macro*), 365
PY_RELEASE_SERIAL (*C macro*), 365
Py_ReprEnter (*C function*), 65
Py_ReprLeave (*C function*), 65
Py_RETURN_FALSE (*C macro*), 154
Py_RETURN_NONE (*C macro*), 144
Py_RETURN_NOTIMPLEMENTED (*C macro*), 108
Py_RETURN_RICHCOMPARE (*C macro*), 340
Py_RETURN_TRUE (*C macro*), 154
Py_RunMain (*C function*), 237
Py_SET_REFCNT (*C function*), 49
Py_SET_SIZE (*C function*), 311
Py_SET_TYPE (*C function*), 311
Py_SetProgramName (*C function*), 238
Py_SetPythonHome (*C function*), 241
Py_SETREF (*C macro*), 51
Py_single_input (*C var*), 46
Py_SIZE (*C function*), 311
Py_ssize_t (*C type*), 10
PY_SSIZE_T_MAX (*C macro*), 147
Py_STRINGIFY (*C macro*), 5
Py_T_BOOL (*C macro*), 317
Py_T_BYTE (*C macro*), 317
Py_T_CHAR (*C macro*), 317
Py_T_DOUBLE (*C macro*), 317
Py_T_FLOAT (*C macro*), 317
Py_T_INT (*C macro*), 317
Py_T_LONG (*C macro*), 317
Py_T_LONGLONG (*C macro*), 317
Py_T_OBJECT_EX (*C macro*), 317
Py_T_PYSSIZET (*C macro*), 317
Py_T_SHORT (*C macro*), 317
Py_T_STRING (*C macro*), 317
Py_T_STRING_INPLACE (*C macro*), 317
Py_T_UBYTE (*C macro*), 317
Py_T_UINT (*C macro*), 317
Py_T_ULONG (*C macro*), 317
Py_T_ULONGLONG (*C macro*), 317
Py_T_USHORT (*C macro*), 317
Py_tp_token (*C macro*), 144
Py_TP_USE_SPEC (*C macro*), 144
Py_TPFLAGS_BASE_EXC_SUBCLASS (*C macro*), 334
Py_TPFLAGS_BASETYPE (*C macro*), 332
Py_TPFLAGS_BYTES_SUBCLASS (*C macro*), 334
Py_TPFLAGS_DEFAULT (*C macro*), 333
Py_TPFLAGS_DICT_SUBCLASS (*C macro*), 334
Py_TPFLAGS_DISALLOW_INSTANTIATION (*C macro*), 335
Py_TPFLAGS_HAVE_FINALIZE (*C macro*), 334
Py_TPFLAGS_HAVE_GC (*C macro*), 333
Py_TPFLAGS_HAVE_VECTORCALL (*C macro*), 334
Py_TPFLAGS_HEAPTYPE (*C macro*), 332
Py_TPFLAGS_IMMUTABLETYPE (*C macro*), 334
Py_TPFLAGS_ITEMS_AT_END (*C macro*), 334
Py_TPFLAGS_LIST_SUBCLASS (*C macro*), 334
Py_TPFLAGS_LONG_SUBCLASS (*C macro*), 334
Py_TPFLAGS_MANAGED_DICT (*C macro*), 333
Py_TPFLAGS_MANAGED_WEAKREF (*C macro*), 333
Py_TPFLAGS_MAPPING (*C macro*), 335

Py_TPFLAGS_METHOD_DESCRIPTOR (*C macro*), 333
Py_TPFLAGS_READY (*C macro*), 332
Py_TPFLAGS_READYING (*C macro*), 332
Py_TPFLAGS_SEQUENCE (*C macro*), 335
Py_TPFLAGS_TUPLE_SUBCLASS (*C macro*), 334
Py_TPFLAGS_TYPE_SUBCLASS (*C macro*), 334
Py_TPFLAGS_UNICODE_SUBCLASS (*C macro*), 334
Py_TPFLAGS_VALID_VERSION_TAG (*C macro*), 336
Py_tracefunc (*C type*), 254
Py_True (*C var*), 154
Py_tss_NEEDS_INIT (*C macro*), 257
Py_tss_t (*C type*), 257
Py_TYPE (*C function*), 310
Py_UCS1 (*C type*), 162
Py_UCS2 (*C type*), 162
Py_UCS4 (*C type*), 162
Py_uhash_t (*C type*), 98
Py_UNBLOCK_THREADS (*C macro*), 247
Py_UnbufferedStdioFlag (*C var*), 235
Py_UNICODE (*C type*), 182
Py_UNICODE_IS_HIGH_SURROGATE (*C function*), 165
Py_UNICODE_IS_LOW_SURROGATE (*C function*), 165
Py_UNICODE_IS_SURROGATE (*C function*), 164
Py_UNICODE_ISALNUM (*C function*), 164
Py_UNICODE_ISALPHA (*C function*), 164
Py_UNICODE_ISDECIMAL (*C function*), 164
Py_UNICODE_ISDIGIT (*C function*), 164
Py_UNICODE_ISLINEBREAK (*C function*), 164
Py_UNICODE_ISLOWER (*C function*), 164
Py_UNICODE_ISNUMERIC (*C function*), 164
Py_UNICODE_ISPRINTABLE (*C function*), 164
Py_UNICODE_ISSPACE (*C function*), 164
Py_UNICODE_ISTITLE (*C function*), 164
Py_UNICODE_ISUPPER (*C function*), 164
Py_UNICODE_JOIN_SURROGATES (*C function*), 165
Py_UNICODE_REPLACEMENT_CHARACTER (*C macro*), 173
Py_UNICODE_TODECIMAL (*C function*), 164
Py_UNICODE_TODIGIT (*C function*), 164
Py_UNICODE_TOLOWER (*C function*), 164
Py_UNICODE_TONUMERIC (*C function*), 164
Py_UNICODE_TOTITLE (*C function*), 164
Py_UNICODE_TOUPPER (*C function*), 164
Py_UNREACHABLE (*C macro*), 6
Py_UNUSED (*C macro*), 6
Py_VaBuildValue (*C function*), 96
PY_VECTORCALL_ARGUMENTS_OFFSET (*C macro*), 118
Py_VerboseFlag (*C var*), 235
Py_Version (*C var*), 365
PY_VERSION_HEX (*C macro*), 365
Py_VISIT (*C macro*), 362
Py_XDECREF (*C function*), 12, 51
Py_XINCREF (*C function*), 50
Py_XNewRef (*C function*), 50
Py_XSETREF (*C macro*), 51
PyAIter_Check (*C function*), 128
PyAnySet_Check (*C function*), 193
PyAnySet_CheckExact (*C function*), 194
PyArg_Parse (*C function*), 93
PyArg_ParseTuple (*C function*), 93
PyArg_ParseTupleAndKeywords (*C function*), 93
PyArg_UnpackTuple (*C function*), 93
PyArg_ValidateKeywordArguments (*C function*), 93
PyArg_VaParse (*C function*), 93
PyArg_VaParseTupleAndKeywords (*C function*), 93
PyASCIIObject (*C type*), 162
PyAsyncMethods (*C type*), 354
PyAsyncMethods.am_aiter (*C member*), 354
PyAsyncMethods.am_anext (*C member*), 354
PyAsyncMethods.am_await (*C member*), 354
PyAsyncMethods.am_send (*C member*), 354
PyBaseObject_Type (*C var*), 310
PyBool_Check (*C function*), 154
PyBool_FromLong (*C function*), 154
PyBool_Type (*C var*), 154
PyBUF_ANY_CONTIGUOUS (*C macro*), 133
PyBUF_C_CONTIGUOUS (*C macro*), 133
PyBUF_CONTIG (*C macro*), 133
PyBUF_CONTIG_RO (*C macro*), 133
PyBUF_F_CONTIGUOUS (*C macro*), 133
PyBUF_FORMAT (*C macro*), 132
PyBUF_FULL (*C macro*), 133
PyBUF_FULL_RO (*C macro*), 133
PyBUF_INDIRECT (*C macro*), 132
PyBUF_MAX_NDIM (*C macro*), 131
PyBUF_ND (*C macro*), 132
PyBUF_READ (*C macro*), 216
PyBUF_RECORDS (*C macro*), 133
PyBUF_RECORDS_RO (*C macro*), 133
PyBUF_SIMPLE (*C macro*), 132
PyBUF_STRIDED (*C macro*), 133
PyBUF_STRIDED_RO (*C macro*), 133
PyBUF_STRIDES (*C macro*), 132
PyBUF_WRITABLE (*C macro*), 132
PyBUF_WRITE (*C macro*), 216
PyBuffer_FillContiguousStrides (*C function*), 136
PyBuffer_FillInfo (*C function*), 136
PyBuffer_FromContiguous (*C function*), 135
PyBuffer_GetPointer (*C function*), 135
PyBuffer_IsContiguous (*C function*), 135
PyBuffer_Release (*C function*), 135
PyBuffer_SizeFromFormat (*C function*), 135
PyBuffer_ToContiguous (*C function*), 135
PyBufferProcs (*C type*), 129, 353
PyBufferProcs.bf_getbuffer (*C member*), 353
PyBufferProcs.bf_releasebuffer (*C member*), 353
PyByteArray_AS_STRING (*C function*), 161
PyByteArray_AsString (*C function*), 161
PyByteArray_Check (*C function*), 160
PyByteArray_CheckExact (*C function*), 160
PyByteArray_Concat (*C function*), 161
PyByteArray_FromObject (*C function*), 161

PyByteArray_FromStringAndSize (*C function*), 161
PyByteArray_GET_SIZE (*C function*), 161
PyByteArray_Resize (*C function*), 161
PyByteArray_Size (*C function*), 161
PyByteArray_Type (*C var*), 160
PyByteArrayObject (*C type*), 160
PyBytes_AS_STRING (*C function*), 159
PyBytes_AsString (*C function*), 159
PyBytes_AsStringAndSize (*C function*), 160
PyBytes_Check (*C function*), 158
PyBytes_CheckExact (*C function*), 158
PyBytes_Concat (*C function*), 160
PyBytes_ConcatAndDel (*C function*), 160
PyBytes_FromFormat (*C function*), 159
PyBytes_FromFormatV (*C function*), 159
PyBytes_FromObject (*C function*), 159
PyBytes_FromString (*C function*), 158
PyBytes_FromStringAndSize (*C function*), 159
PyBytes_GET_SIZE (*C function*), 159
PyBytes_Join (*C function*), 160
PyBytes_Size (*C function*), 159
PyBytes_Type (*C var*), 158
PyBytesObject (*C type*), 158
PyCallable_Check (*C function*), 122
PyCallIter_Check (*C function*), 213
PyCallIter_New (*C function*), 213
PyCallIter_Type (*C var*), 213
PyCapsule (*C type*), 218
PyCapsule_CheckExact (*C function*), 218
PyCapsule_Destructor (*C type*), 218
PyCapsule_GetContext (*C function*), 219
PyCapsule_GetDestructor (*C function*), 219
PyCapsule_GetName (*C function*), 219
PyCapsule_GetPointer (*C function*), 218
PyCapsule_Import (*C function*), 219
PyCapsule_IsValid (*C function*), 219
PyCapsule_New (*C function*), 218
PyCapsule_SetContext (*C function*), 219
PyCapsule_SetDestructor (*C function*), 219
PyCapsule_SetName (*C function*), 220
PyCapsule_SetPointer (*C function*), 220
PyCell_Check (*C function*), 198
PyCell_Get (*C function*), 198
PyCell_GET (*C function*), 198
PyCell_New (*C function*), 198
PyCell_Set (*C function*), 198
PyCell_SET (*C function*), 198
PyCell_Type (*C var*), 198
PyCellObject (*C type*), 198
PyCF_ALLOW_TOP_LEVEL_AWAIT (*C macro*), 47
PyCF_ONLY_AST (*C macro*), 47
PyCF_OPTIMIZED_AST (*C macro*), 47
PyCF_TYPE_COMMENTS (*C macro*), 47
PyCFunction (*C type*), 311
PyCFunction_New (*C function*), 314
PyCFunction_NewEx (*C function*), 314
PyCFunctionFast (*C type*), 312
PyCFunctionFastWithKeywords (*C type*), 312
PyCFunctionWithKeywords (*C type*), 311
PyCMethod (*C type*), 312
PyCMethod_New (*C function*), 314
PyCode_Addr2Line (*C function*), 200
PyCode_Addr2Location (*C function*), 200
PyCode_AddWatcher (*C function*), 201
PyCode_Check (*C function*), 199
PyCode_ClearWatcher (*C function*), 201
PyCode_GetCellvars (*C function*), 200
PyCode_GetCode (*C function*), 200
PyCode_GetFreevars (*C function*), 200
PyCode_GetNumFree (*C function*), 199
PyCode_GetVarnames (*C function*), 200
PyCode_New (*C function*), 199
PyCode_NewEmpty (*C function*), 200
PyCode_NewWithPosOnlyArgs (*C function*), 200
PyCode_Type (*C var*), 199
PyCode_WatchCallback (*C type*), 201
PyCodec_BackslashReplaceErrors (*C function*), 102
PyCodec_Decode (*C function*), 101
PyCodec_Decoder (*C function*), 101
PyCodec_Encode (*C function*), 101
PyCodec_Encoder (*C function*), 101
PyCodec_IgnoreErrors (*C function*), 102
PyCodec_IncrementalDecoder (*C function*), 102
PyCodec_IncrementalEncoder (*C function*), 102
PyCodec_KnownEncoding (*C function*), 101
PyCodec_LookupError (*C function*), 102
PyCodec_NameReplaceErrors (*C function*), 102
PyCodec_Register (*C function*), 101
PyCodec_RegisterError (*C function*), 102
PyCodec_ReplaceErrors (*C function*), 102
PyCodec_StreamReader (*C function*), 102
PyCodec_StreamWriter (*C function*), 102
PyCodec_StrictErrors (*C function*), 102
PyCodec_Unregister (*C function*), 101
PyCodec_XMLCharRefReplaceErrors (*C function*), 102
PyCodeEvent (*C type*), 201
PyCodeObject (*C type*), 199
PyCompactUnicodeObject (*C type*), 162
PyCompilerFlags (*C struct*), 46
PyCompilerFlags.cf_feature_version (*C member*), 47
PyCompilerFlags.cf_flags (*C member*), 46
PyComplex_AsCComplex (*C function*), 158
PyComplex_Check (*C function*), 157
PyComplex_CheckExact (*C function*), 157
PyComplex_FromCComplex (*C function*), 157
PyComplex_FromDoubles (*C function*), 157
PyComplex_ImagAsDouble (*C function*), 158
PyComplex_RealAsDouble (*C function*), 157
PyComplex_Type (*C var*), 157
PyComplexObject (*C type*), 157
PyConfig (*C type*), 274
PyConfig_Clear (*C function*), 275

PyConfig_Get (*C function*), 268
PyConfig_GetInt (*C function*), 268
PyConfig_InitIsolatedConfig (*C function*), 274
PyConfig_InitPythonConfig (*C function*), 274
PyConfig_Names (*C function*), 268
PyConfig._pystats (*C member*), 286
PyConfig_Read (*C function*), 275
PyConfig_Set (*C function*), 268
PyConfig_SetArgv (*C function*), 274
PyConfig_SetBytesArgv (*C function*), 274
PyConfig_SetBytesString (*C function*), 274
PyConfig_SetString (*C function*), 274
PyConfig_SetWideStringList (*C function*), 275
PyConfig.argv (*C member*), 275
PyConfig.base_exec_prefix (*C member*), 276
PyConfig.base_executable (*C member*), 276
PyConfig.base_prefix (*C member*), 276
PyConfig.buffered_stdio (*C member*), 276
PyConfig.bytes_warning (*C member*), 276
PyConfig.check_hash_pycs_mode (*C member*), 277
PyConfig.code_debug_ranges (*C member*), 277
PyConfig.configure_c_stdio (*C member*), 277
PyConfig.cpu_count (*C member*), 280
PyConfig.dev_mode (*C member*), 277
PyConfig.dump_refs (*C member*), 277
PyConfig.dump_refs_file (*C member*), 277
PyConfig.exec_prefix (*C member*), 278
PyConfig.executable (*C member*), 278
PyConfig.faulthandler (*C member*), 278
PyConfig.filesystem_encoding (*C member*), 278
PyConfig.filesystem_errors (*C member*), 278
PyConfig.hash_seed (*C member*), 279
PyConfig.home (*C member*), 279
PyConfig.import_time (*C member*), 279
PyConfig.inspect (*C member*), 279
PyConfig.install_signal_handlers (*C member*), 279
PyConfig.int_max_str_digits (*C member*), 279
PyConfig.interactive (*C member*), 279
PyConfig.isolated (*C member*), 280
PyConfig.legacy_windows_stdio (*C member*), 280
PyConfig.malloc_stats (*C member*), 280
PyConfig.module_search_paths (*C member*), 281
PyConfig.module_search_paths_set (*C member*), 281
PyConfig.optimization_level (*C member*), 281
PyConfig.orig_argv (*C member*), 281
PyConfig.parse_argv (*C member*), 281
PyConfig.parser_debug (*C member*), 282
PyConfig.pathconfig_warnings (*C member*), 282
PyConfig.perf_profiling (*C member*), 284
PyConfig.platlibdir (*C member*), 280
PyConfig.prefix (*C member*), 282
PyConfig.program_name (*C member*), 282
PyConfig.pycache_prefix (*C member*), 282
PyConfig.pythonpath_env (*C member*), 281
PyConfig.quiet (*C member*), 282
PyConfig.run_command (*C member*), 283
PyConfig.run_filename (*C member*), 283
PyConfig.run_module (*C member*), 283
PyConfig.run_presite (*C member*), 283
PyConfig.safe_path (*C member*), 276
PyConfig.show_ref_count (*C member*), 283
PyConfig.site_import (*C member*), 283
PyConfig.skip_source_first_line (*C member*), 283
PyConfig.stdio_encoding (*C member*), 284
PyConfig.stdio_errors (*C member*), 284
PyConfig.stdlib_dir (*C member*), 284
PyConfig.tracemalloc (*C member*), 284
PyConfig.use_environment (*C member*), 285
PyConfig.use_frozen_modules (*C member*), 279
PyConfig.use_hash_seed (*C member*), 279
PyConfig.use_system_logger (*C member*), 285
PyConfig.user_site_directory (*C member*), 285
PyConfig.verbose (*C member*), 285
PyConfig.warn_default_encoding (*C member*), 277
PyConfig.warnoptions (*C member*), 285
PyConfig.write_bytecode (*C member*), 285
PyConfig.xoptions (*C member*), 286
PyContext (*C type*), 223
PyContext_AddWatcher (*C function*), 224
PyContext_CheckExact (*C function*), 224
PyContext_ClearWatcher (*C function*), 224
PyContext_Copy (*C function*), 224
PyContext_CopyCurrent (*C function*), 224
PyContext_Enter (*C function*), 224
PyContext_Exit (*C function*), 224
PyContext_New (*C function*), 224
PyContext_Type (*C var*), 224
PyContext_WatchCallback (*C type*), 225
PyContextEvent (*C type*), 224
PyContextToken (*C type*), 223
PyContextToken_CheckExact (*C function*), 224
PyContextToken_Type (*C var*), 224
PyContextVar (*C type*), 223
PyContextVar_CheckExact (*C function*), 224
PyContextVar_Get (*C function*), 225
PyContextVar_New (*C function*), 225
PyContextVar_Reset (*C function*), 225
PyContextVar_Set (*C function*), 225
PyContextVar_Type (*C var*), 224
PyCoro_CheckExact (*C function*), 223
PyCoro_New (*C function*), 223
PyCoro_Type (*C var*), 223
PyCoroObject (*C type*), 223
PyDate_Check (*C function*), 226
PyDate_CheckExact (*C function*), 226
PyDate_FromDate (*C function*), 227
PyDate_FromTimestamp (*C function*), 229
PyDateTime_Check (*C function*), 226
PyDateTime_CheckExact (*C function*), 226
PyDateTime_Date (*C type*), 225

PyDateTime_DATE_GET_FOLD (*C function*), 228
PyDateTime_DATE_GET_HOUR (*C function*), 228
PyDateTime_DATE_GET_MICROSECOND (*C function*), 228
PyDateTime_DATE_GET_MINUTE (*C function*), 228
PyDateTime_DATE_GET_SECOND (*C function*), 228
PyDateTime_DATE_GET_TZINFO (*C function*), 228
PyDateTime_DateTime (*C type*), 225
PyDateTime_DateTimeType (*C var*), 226
PyDateTime_DateType (*C var*), 226
PyDateTime_Delta (*C type*), 226
PyDateTime_DELTA_GET_DAYS (*C function*), 228
PyDateTime_DELTA_GET_MICROSECONDS (*C function*), 229
PyDateTime_DELTA_GET_SECONDS (*C function*), 228
PyDateTime_DeltaType (*C var*), 226
PyDateTime_FromDateAndTime (*C function*), 227
PyDateTime_FromDateAndTimeAndFold (*C function*), 227
PyDateTime_FromTimestamp (*C function*), 229
PyDateTime_GET_DAY (*C function*), 228
PyDateTime_GET_MONTH (*C function*), 227
PyDateTime_GET_YEAR (*C function*), 227
PyDateTime_Time (*C type*), 225
PyDateTime_TIME_GET_FOLD (*C function*), 228
PyDateTime_TIME_GET_HOUR (*C function*), 228
PyDateTime_TIME_GET_MICROSECOND (*C function*), 228
PyDateTime_TIME_GET_MINUTE (*C function*), 228
PyDateTime_TIME_GET_SECOND (*C function*), 228
PyDateTime_TIME_GET_TZINFO (*C function*), 228
PyDateTime_TimeType (*C var*), 226
PyDateTime_TimeZone_UTC (*C var*), 226
PyDateTime_TZInfoType (*C var*), 226
PyDelta_Check (*C function*), 226
PyDelta_CheckExact (*C function*), 226
PyDelta_FromDSU (*C function*), 227
PyDescr_IsData (*C function*), 214
PyDescr_NewClassMethod (*C function*), 214
PyDescr_NewGetSet (*C function*), 214
PyDescr_NewMember (*C function*), 214
PyDescr_NewMethod (*C function*), 214
PyDescr_NewWrapper (*C function*), 214
PyDict_AddWatcher (*C function*), 192
PyDict_Check (*C function*), 188
PyDict_CheckExact (*C function*), 188
PyDict_Clear (*C function*), 188
PyDict_ClearWatcher (*C function*), 192
PyDict_Contains (*C function*), 188
PyDict_ContainsString (*C function*), 188
PyDict_Copy (*C function*), 188
PyDict_DelItem (*C function*), 189
PyDict_DelItemString (*C function*), 189
PyDict_GetItem (*C function*), 189
PyDict_GetItemRef (*C function*), 189
PyDict_GetItemString (*C function*), 189
PyDict_GetItemStringRef (*C function*), 189
PyDict_GetItemWithError (*C function*), 189
PyDict_Items (*C function*), 190
PyDict_Keys (*C function*), 190
PyDict_Merge (*C function*), 191
PyDict_MergeFromSeq2 (*C function*), 192
PyDict_New (*C function*), 188
PyDict_Next (*C function*), 190
PyDict_Pop (*C function*), 190
PyDict_PopString (*C function*), 190
PyDict_SetDefault (*C function*), 190
PyDict_SetDefaultRef (*C function*), 190
PyDict_SetItem (*C function*), 188
PyDict_SetItemString (*C function*), 189
PyDict_Size (*C function*), 190
PyDict_Type (*C var*), 188
PyDict_Unwatch (*C function*), 192
PyDict_Update (*C function*), 192
PyDict_Values (*C function*), 190
PyDict_Watch (*C function*), 192
PyDict_WatchCallback (*C type*), 192
PyDict_WatchEvent (*C type*), 192
PyDictObject (*C type*), 188
PyDictProxy_New (*C function*), 188
PyDoc_STR (*C macro*), 6
PyDoc_STRVAR (*C macro*), 6
PyEllipsis_Type (*C var*), 215
PyErr_BadArgument (*C function*), 55
PyErr_BadInternalCall (*C function*), 56
PyErr_CheckSignals (*C function*), 61
PyErr_Clear (*C function*), 10, 12, 53
PyErr_DisplayException (*C function*), 54
PyErr_ExceptionMatches (*C function*), 12, 58
PyErr_Fetch (*C function*), 59
PyErr_Format (*C function*), 54
PyErr_FormatUnraisable (*C function*), 54
PyErr_FormatV (*C function*), 54
PyErr_GetExcInfo (*C function*), 60
PyErr_GetHandledException (*C function*), 60
PyErr_GetRaisedException (*C function*), 58
PyErr_GivenExceptionMatches (*C function*), 58
PyErr_NewException (*C function*), 62
PyErr_NewExceptionWithDoc (*C function*), 62
PyErr_NoMemory (*C function*), 55
PyErr_NormalizeException (*C function*), 59
PyErr_Occurred (*C function*), 10, 58
PyErr_Print (*C function*), 54
PyErr_PrintEx (*C function*), 53
PyErr_ResourceWarning (*C function*), 57
PyErr_Restore (*C function*), 59
PyErr_SetExcFromWindowsErr (*C function*), 55
PyErr_SetExcFromWindowsErrWithFilename (*C function*), 56
PyErr_SetExcFromWindowsErrWithFilenameObject (*C function*), 56
PyErr_SetExcFromWindowsErrWithFilenameObjects (*C function*), 56
PyErr_SetExcInfo (*C function*), 60
PyErr_SetFromErrno (*C function*), 55

PyErr_SetFromErrnoWithFilename (*C function*), 55
PyErr_SetFromErrnoWithFilenameObject (*C function*), 55
PyErr_SetFromErrnoWithFilenameObjects (*C function*), 55
PyErr_SetFromWindowsErr (*C function*), 55
PyErr_SetFromWindowsErrWithFilename (*C function*), 55
PyErr_SetHandledException (*C function*), 60
PyErr_SetImportError (*C function*), 56
PyErr_SetImportErrorSubclass (*C function*), 56
PyErr_SetInterrupt (*C function*), 61
PyErr_SetInterruptEx (*C function*), 61
PyErr_SetNone (*C function*), 55
PyErr_SetObject (*C function*), 54
PyErr_SetRaisedException (*C function*), 58
PyErr_SetString (*C function*), 10, 54
PyErr_SyntaxLocation (*C function*), 56
PyErr_SyntaxLocationEx (*C function*), 56
PyErr_SyntaxLocationObject (*C function*), 56
PyErr_WarnEx (*C function*), 57
PyErr_WarnExplicit (*C function*), 57
PyErr_WarnExplicitObject (*C function*), 57
PyErr_WarnFormat (*C function*), 57
PyErr_WriteUnraisable (*C function*), 54
PyEval_AcquireThread (*C function*), 249
PyEval_AcquireThread(), 245
PyEval_EvalCode (*C function*), 46
PyEval_EvalCodeEx (*C function*), 46
PyEval_EvalFrame (*C function*), 46
PyEval_EvalFrameEx (*C function*), 46
PyEval_GetBuiltins (*C function*), 100
PyEval_GetFrame (*C function*), 100
PyEval_GetFrameBuiltins (*C function*), 100
PyEval_GetFrameGlobals (*C function*), 101
PyEval_GetFrameLocals (*C function*), 100
PyEval_GetFuncDesc (*C function*), 101
PyEval_GetFuncName (*C function*), 101
PyEval_GetGlobals (*C function*), 100
PyEval_GetLocals (*C function*), 100
PyEval_InitThreads (*C function*), 245
PyEval_InitThreads(), 235
PyEval_MergeCompilerFlags (*C function*), 46
PyEval_ReleaseThread (*C function*), 250
PyEval_ReleaseThread(), 245
PyEval_RestoreThread (*C function*), 243, 245
PyEval_RestoreThread(), 245
PyEval_SaveThread (*C function*), 243, 245
PyEval_SaveThread(), 245
PyEval_SetProfile (*C function*), 255
PyEval_SetProfileAllThreads (*C function*), 255
PyEval_SetTrace (*C function*), 255
PyEval_SetTraceAllThreads (*C function*), 255
PyExc_ArithmeticError (*C var*), 66
PyExc_AssertionError (*C var*), 66
PyExc_AttributeError (*C var*), 66
PyExc_BaseException (*C var*), 65
PyExc_BaseExceptionGroup (*C var*), 66
PyExc_BlockingIOError (*C var*), 66
PyExc_BrokenPipeError (*C var*), 66
PyExc_BufferError (*C var*), 66
PyExc_BytesWarning (*C var*), 71
PyExc_ChildProcessError (*C var*), 66
PyExc_ConnectionAbortedError (*C var*), 66
PyExc_ConnectionError (*C var*), 66
PyExc_ConnectionRefusedError (*C var*), 66
PyExc_ConnectionResetError (*C var*), 66
PyExc_DeprecationWarning (*C var*), 71
PyExc_EncodingWarning (*C var*), 71
PyExc_EnvironmentError (*C var*), 70
PyExc_EOFError (*C var*), 67
PyExc_Exception (*C var*), 66
PyExc_FileExistsError (*C var*), 67
PyExc_FileNotFoundError (*C var*), 67
PyExc_FloatingPointError (*C var*), 67
PyExc_FutureWarning (*C var*), 71
PyExc_GeneratorExit (*C var*), 67
PyExc_ImportError (*C var*), 67
PyExc_ImportWarning (*C var*), 71
PyExc_IndentationError (*C var*), 67
PyExc_IndexError (*C var*), 67
PyExc_InterruptedError (*C var*), 67
PyExc_IOError (*C var*), 70
PyExc_IsADirectoryError (*C var*), 67
PyExc_KeyboardInterrupt (*C var*), 67
PyExc_KeyError (*C var*), 67
PyExc_LookupError (*C var*), 67
PyExc_MemoryError (*C var*), 68
PyExc_ModuleNotFoundError (*C var*), 68
PyExc_NameError (*C var*), 68
PyExc_NotADirectoryError (*C var*), 68
PyExc_NotImplementedError (*C var*), 68
PyExc_OSError (*C var*), 68
PyExc_OverflowError (*C var*), 68
PyExc_PendingDeprecationWarning (*C var*), 71
PyExc_PermissionError (*C var*), 68
PyExc_ProcessLookupError (*C var*), 68
PyExc_PythonFinalizationError (*C var*), 68
PyExc_RecursionError (*C var*), 68
PyExc_ReferenceError (*C var*), 68
PyExc_ResourceWarning (*C var*), 71
PyExc_RuntimeError (*C var*), 68
PyExc_RuntimeWarning (*C var*), 71
PyExc_StopAsyncIteration (*C var*), 68
PyExc_StopIteration (*C var*), 69
PyExc_SyntaxError (*C var*), 69
PyExc_SyntaxWarning (*C var*), 71
PyExc_SystemError (*C var*), 69
PyExc_SystemExit (*C var*), 69
PyExc_TabError (*C var*), 69
PyExc_TimeoutError (*C var*), 69
PyExc_TypeError (*C var*), 69
PyExc_UnboundLocalError (*C var*), 69
PyExc_UnicodeDecodeError (*C var*), 69
PyExc_UnicodeEncodeError (*C var*), 69

PyExc_UnicodeError (C var), 69
PyExc_UnicodeTranslateError (C var), 69
PyExc_UnicodeWarning (C var), 71
PyExc_UserWarning (C var), 71
PyExc_ValueError (C var), 69
PyExc_Warning (C var), 71
PyExc_WindowsError (C var), 70
PyExc_ZeroDivisionError (C var), 70
PyException_GetArgs (C function), 63
PyException_GetCause (C function), 63
PyException_GetContext (C function), 62
PyException_GetTraceback (C function), 62
PyException_SetArgs (C function), 63
PyException_SetCause (C function), 63
PyException_SetContext (C function), 63
PyException_SetTraceback (C function), 62
PyExceptionClass_Check (C function), 62
PyExceptionClass_Name (C function), 62
PyFile_FromFd (C function), 204
PyFile_GetLine (C function), 204
PyFile_SetOpenCodeHook (C function), 204
PyFile_WriteObject (C function), 205
PyFile_WriteString (C function), 205
PyFloat_AS_DOUBLE (C function), 155
PyFloat_AsDouble (C function), 155
PyFloat_Check (C function), 155
PyFloat_CheckExact (C function), 155
PyFloat_FromDouble (C function), 155
PyFloat_FromString (C function), 155
PyFloat_GetInfo (C function), 155
PyFloat_GetMax (C function), 155
PyFloat_GetMin (C function), 155
PyFloat_Pack2 (C function), 156
PyFloat_Pack4 (C function), 156
PyFloat_Pack8 (C function), 156
PyFloat_Type (C var), 154
PyFloat_Unpack2 (C function), 156
PyFloat_Unpack4 (C function), 156
PyFloat_Unpack8 (C function), 156
PyFloatObject (C type), 154
PyFrame_Check (C function), 220
PyFrame_GetBack (C function), 220
PyFrame_GetBuiltins (C function), 220
PyFrame_GetCode (C function), 220
PyFrame_GetGenerator (C function), 220
PyFrame_GetGlobals (C function), 221
PyFrame_GetLasti (C function), 221
PyFrame_GetLineNumber (C function), 221
PyFrame_GetLocals (C function), 221
PyFrame_GetVar (C function), 221
PyFrame_GetVarString (C function), 221
PyFrame_Type (C var), 220
PyFrameLocalsProxy_Check (C function), 221
PyFrameLocalsProxy_Type (C var), 221
PyFrameObject (C type), 220
PyFrozenSet_Check (C function), 193
PyFrozenSet_CheckExact (C function), 194
PyFrozenSet_New (C function), 194
PyFrozenSet_Type (C var), 193
PyFunction_AddWatcher (C function), 196
PyFunction_Check (C function), 195
PyFunction_ClearWatcher (C function), 196
PyFunction_GET_ANNOTATIONS (C function), 196
PyFunction_GET_CLOSURE (C function), 196
PyFunction_GET_CODE (C function), 196
PyFunction_GET_DEFAULTS (C function), 196
PyFunction_GET_GLOBALS (C function), 196
PyFunction_GET_KW_DEFAULTS (C function), 196
PyFunction_GET_MODULE (C function), 196
PyFunction_GetAnnotations (C function), 196
PyFunction_GetClosure (C function), 195
PyFunction_GetCode (C function), 195
PyFunction_GetDefaults (C function), 195
PyFunction_GetGlobals (C function), 195
PyFunction_GetKwDefaults (C function), 195
PyFunction_GetModule (C function), 195
PyFunction_New (C function), 195
PyFunction_NewWithQualName (C function), 195
PyFunction_SetAnnotations (C function), 196
PyFunction_SetClosure (C function), 196
PyFunction_SetDefaults (C function), 195
PyFunction_SetVectorcall (C function), 195
PyFunction_Type (C var), 195
PyFunction_WatchCallback (C type), 196
PyFunction_WatchEvent (C type), 196
PyFunctionObject (C type), 195
PyGC_Collect (C function), 362
PyGC_Disable (C function), 362
PyGC_Enable (C function), 362
PyGC_IsEnabled (C function), 362
PyGen_Check (C function), 222
PyGen_CheckExact (C function), 222
PyGen_New (C function), 223
PyGen_NewWithQualName (C function), 223
PyGen_Type (C var), 222
PyGenObject (C type), 222
PyGetSetDef (C type), 318
PyGetSetDef.closure (C member), 318
PyGetSetDef.doc (C member), 318
PyGetSetDef.get (C member), 318
PyGetSetDef.name (C member), 318
PyGetSetDef.set (C member), 318
PyGILState_Check (C function), 247
PyGILState_Ensure (C function), 246
PyGILState_GetThisThreadState (C function), 246
PyGILState_Release (C function), 246
PyHASH_BITS (C macro), 98
PyHash_FuncDef (C type), 99
PyHash_FuncDef.hash (C member), 99
PyHash_FuncDef.hash_bits (C member), 99
PyHash_FuncDef.name (C member), 99
PyHash_FuncDef.seed_bits (C member), 99
PyHash_GetFuncDef (C function), 99
PyHASH_IMAG (C macro), 99
PyHASH_INF (C macro), 99

PyHASH_MODULUS (*C macro*), 98
PyHASH_MULTIPLIER (*C macro*), 99
PyImport_AddModule (*C function*), 84
PyImport_AddModuleObject (*C function*), 84
PyImport_AddModuleRef (*C function*), 83
PyImport_AppendInittab (*C function*), 86
PyImport_ExecCodeModule (*C function*), 84
PyImport_ExecCodeModuleEx (*C function*), 84
PyImport_ExecCodeModuleObject (*C function*), 84
PyImport_ExecCodeModuleWithPathnames (*C function*), 85
PyImport_ExtendInittab (*C function*), 86
PyImport_FrozenModules (*C var*), 86
PyImport_GetImporter (*C function*), 85
PyImport_GetMagicNumber (*C function*), 85
PyImport_GetMagicTag (*C function*), 85
PyImport_GetModule (*C function*), 85
PyImport_GetModuleDict (*C function*), 85
PyImport_Import (*C function*), 83
PyImport_ImportFrozenModule (*C function*), 85
PyImport_ImportFrozenModuleObject (*C function*), 85
PyImport_ImportModule (*C function*), 83
PyImport_ImportModuleAttr (*C function*), 86
PyImport_ImportModuleAttrString (*C function*), 86
PyImport_ImportModuleEx (*C function*), 83
PyImport_ImportModuleLevel (*C function*), 83
PyImport_ImportModuleLevelObject (*C function*), 83
PyImport_ImportModuleNoBlock (*C function*), 83
PyImport_ReloadModule (*C function*), 83
PyIndex_Check (*C function*), 125
PyInit_modulename (*C function*), 74
PyInitConfig (*C struct*), 264
PyInitConfig_AddModule (*C function*), 266
PyInitConfig_Create (*C function*), 264
PyInitConfig_Free (*C function*), 264
PyInitConfig_FreeStrList (*C function*), 265
PyInitConfig_GetError (*C function*), 264
PyInitConfig_GetExitCode (*C function*), 264
PyInitConfig_GetInt (*C function*), 265
PyInitConfig_GetStr (*C function*), 265
PyInitConfig_GetStrList (*C function*), 265
PyInitConfig_HasOption (*C function*), 265
PyInitConfig_SetInt (*C function*), 265
PyInitConfig_SetStr (*C function*), 265
PyInitConfig_SetStrList (*C function*), 265
PyInstanceMethod_Check (*C function*), 197
PyInstanceMethod_Function (*C function*), 197
PyInstanceMethod_GET_FUNCTION (*C function*), 197
PyInstanceMethod_New (*C function*), 197
PyInstanceMethod_Type (*C var*), 197
PyInterpreterConfig (*C type*), 250
PyInterpreterConfig_DEFAULT_GIL (*C macro*), 251
PyInterpreterConfig_OWN_GIL (*C macro*), 251
PyInterpreterConfig_SHARED_GIL (*C macro*), 251
PyInterpreterConfig.allow_daemon_threads (*C member*), 251
PyInterpreterConfig.allow_exec (*C member*), 251
PyInterpreterConfig.allow_fork (*C member*), 250
PyInterpreterConfig.allow_threads (*C member*), 251
PyInterpreterConfig.check_multi_interp_extensions (*C member*), 251
PyInterpreterConfig.gil (*C member*), 251
PyInterpreterConfig.use_main_obmalloc (*C member*), 250
PyInterpreterState (*C type*), 245
PyInterpreterState_Clear (*C function*), 247
PyInterpreterState_Delete (*C function*), 247
PyInterpreterState_Get (*C function*), 248
PyInterpreterState_GetDict (*C function*), 249
PyInterpreterState_GetID (*C function*), 249
PyInterpreterState_Head (*C function*), 256
PyInterpreterState_Main (*C function*), 256
PyInterpreterState_New (*C function*), 247
PyInterpreterState_Next (*C function*), 256
PyInterpreterState_ThreadHead (*C function*), 256
PyIter_Check (*C function*), 128
PyIter_Next (*C function*), 129
PyIter_NextItem (*C function*), 129
PyIter_Send (*C function*), 129
PyList_Append (*C function*), 187
PyList_AsTuple (*C function*), 188
PyList_Check (*C function*), 186
PyList_CheckExact (*C function*), 186
PyList_Clear (*C function*), 187
PyList_Extend (*C function*), 187
PyList_GET_ITEM (*C function*), 186
PyList_GET_SIZE (*C function*), 186
PyList_GetItem (*C function*), 9, 186
PyList_GetItemRef (*C function*), 186
PyList_GetSlice (*C function*), 187
PyList_Insert (*C function*), 187
PyList_New (*C function*), 186
PyList_Reverse (*C function*), 188
PyList_SET_ITEM (*C function*), 187
PyList_SetItem (*C function*), 8, 187
PyList_SetSlice (*C function*), 187
PyList_Size (*C function*), 186
PyList_Sort (*C function*), 187
PyList_Type (*C var*), 186
PyListObject (*C type*), 186
PyLong_AS_LONG (*C function*), 146
PyLong_AsDouble (*C function*), 148
PyLong_AsInt (*C function*), 146
PyLong_AsInt32 (*C function*), 148
PyLong_AsInt64 (*C function*), 148
PyLong_AsLong (*C function*), 146

PyLong_AsLongAndOverflow (*C function*), 146
PyLong_AsLongLong (*C function*), 147
PyLong_AsLongLongAndOverflow (*C function*), 147
PyLong_AsNativeBytes (*C function*), 149
PyLong_AsSize_t (*C function*), 147
PyLong_AsSsize_t (*C function*), 147
PyLong_AsUInt32 (*C function*), 148
PyLong_AsUInt64 (*C function*), 148
PyLong_AsUnsignedLong (*C function*), 147
PyLong_AsUnsignedLongLong (*C function*), 147
PyLong_AsUnsignedLongLongMask (*C function*), 148
PyLong_AsUnsignedLongMask (*C function*), 148
PyLong_AsVoidPtr (*C function*), 148
PyLong_Check (*C function*), 144
PyLong_CheckExact (*C function*), 145
PyLong_Export (*C function*), 153
PyLong_FreeExport (*C function*), 153
PyLong_FromDouble (*C function*), 145
PyLong_FromInt32 (*C function*), 145
PyLong_FromInt64 (*C function*), 145
PyLong_FromLong (*C function*), 145
PyLong_FromLongLong (*C function*), 145
PyLong_FromNativeBytes (*C function*), 146
PyLong_FromSize_t (*C function*), 145
PyLong_FromSsize_t (*C function*), 145
PyLong_FromString (*C function*), 145
PyLong_FromUInt32 (*C function*), 145
PyLong_FromUInt64 (*C function*), 145
PyLong_FromUnicodeObject (*C function*), 146
PyLong_FromUnsignedLong (*C function*), 145
PyLong_FromUnsignedLongLong (*C function*), 145
PyLong_FromUnsignedNativeBytes (*C function*), 146
PyLong_FromVoidPtr (*C function*), 146
PyLong_GetInfo (*C function*), 151
PyLong_GetNativeLayout (*C function*), 152
PyLong_GetSign (*C function*), 151
PyLong_IsNegative (*C function*), 151
PyLong_IsPositive (*C function*), 151
PyLong_IsZero (*C function*), 151
PyLong_Type (*C var*), 144
PyLongExport (*C struct*), 153
PyLongExport.digits (*C member*), 153
PyLongExport.ndigits (*C member*), 153
PyLongExport.negative (*C member*), 153
PyLongExport.value (*C member*), 153
PyLongLayout (*C struct*), 152
PyLongLayout.bits_per_digit (*C member*), 152
PyLongLayout.digit_endianness (*C member*), 152
PyLongLayout.digit_size (*C member*), 152
PyLongLayout.digits_order (*C member*), 152
PyLongObject (*C type*), 144
PyLongWriter (*C struct*), 153
PyLongWriter_Create (*C function*), 153
PyLongWriter_Discard (*C function*), 154
PyLongWriter_Finish (*C function*), 154
PyMapping_Check (*C function*), 127
PyMapping_DelItem (*C function*), 127
PyMapping_DelItemString (*C function*), 127
PyMapping_GetItemString (*C function*), 127
PyMapping_GetOptionalItem (*C function*), 127
PyMapping_GetOptionalItemString (*C function*), 127
PyMapping_HasKey (*C function*), 128
PyMapping_HasKeyString (*C function*), 128
PyMapping_HasKeyStringWithError (*C function*), 128
PyMapping_HasKeyWithError (*C function*), 127
PyMapping_Items (*C function*), 128
PyMapping_Keys (*C function*), 128
PyMapping_Length (*C function*), 127
PyMapping_SetItemString (*C function*), 127
PyMapping_Size (*C function*), 127
PyMapping_Values (*C function*), 128
PyMappingMethods (*C type*), 352
PyMappingMethods.mp_ass_subscript (*C member*), 352
PyMappingMethods.mp_length (*C member*), 352
PyMappingMethods.mp_subscript (*C member*), 352
PyMarshal_ReadLastObjectFromFile (*C function*), 87
PyMarshal_ReadLongFromFile (*C function*), 87
PyMarshal_ReadObjectFromFile (*C function*), 87
PyMarshal_ReadObjectFromString (*C function*), 87
PyMarshal_ReadShortFromFile (*C function*), 87
PyMarshal_WriteLongToFile (*C function*), 87
PyMarshal_WriteObjectToFile (*C function*), 87
PyMarshal_WriteObjectToString (*C function*), 87
PyMem_Calloc (*C function*), 294
PyMem_Del (*C function*), 294
PYMEM_DOMAIN_MEM (*C macro*), 297
PYMEM_DOMAIN_OBJ (*C macro*), 297
PYMEM_DOMAIN_RAW (*C macro*), 297
PyMem_Free (*C function*), 294
PyMem_GetAllocator (*C function*), 297
PyMem_Malloc (*C function*), 293
PyMem_New (*C macro*), 294
PyMem_RawCalloc (*C function*), 293
PyMem_RawFree (*C function*), 293
PyMem_RawMalloc (*C function*), 293
PyMem_RawRealloc (*C function*), 293
PyMem_Realloc (*C function*), 294
PyMem_Resize (*C macro*), 294
PyMem_SetAllocator (*C function*), 297
PyMem_SetupDebugHooks (*C function*), 298
PyMemAllocatorDomain (*C type*), 297
PyMemAllocatorEx (*C type*), 296
PyMember_GetOne (*C function*), 315
PyMember_SetOne (*C function*), 315
PyMemberDef (*C type*), 314
PyMemberDef.doc (*C member*), 315
PyMemberDef.flags (*C member*), 315

PyMemberDef.name (*C member*), 314
PyMemberDef.offset (*C member*), 315
PyMemberDef.type (*C member*), 315
PyMemoryView_Check (*C function*), 216
PyMemoryView_FromBuffer (*C function*), 216
PyMemoryView_FromMemory (*C function*), 216
PyMemoryView_FromObject (*C function*), 216
PyMemoryView_GET_BASE (*C function*), 216
PyMemoryView_GET_BUFFER (*C function*), 216
PyMemoryView_GetContiguous (*C function*), 216
PyMethod_Check (*C function*), 197
PyMethod_Function (*C function*), 198
PyMethod_GET_FUNCTION (*C function*), 198
PyMethod_GET_SELF (*C function*), 198
PyMethod_New (*C function*), 197
PyMethod_Self (*C function*), 198
PyMethod_Type (*C var*), 197
PyMethodDef (*C type*), 312
PyMethodDef.ml_doc (*C member*), 312
PyMethodDef.ml_flags (*C member*), 312
PyMethodDef.ml_meth (*C member*), 312
PyMethodDef.ml_name (*C member*), 312
PyMODINIT_FUNC (*C macro*), 74
PyModule_Add (*C function*), 211
PyModule_AddFunctions (*C function*), 212
PyModule_AddIntConstant (*C function*), 211
PyModule_AddIntMacro (*C macro*), 211
PyModule_AddObject (*C function*), 211
PyModule_AddObjectRef (*C function*), 210
PyModule_AddStringConstant (*C function*), 211
PyModule_AddStringMacro (*C macro*), 212
PyModule_AddType (*C function*), 212
PyModule_Check (*C function*), 205
PyModule_CheckExact (*C function*), 205
PyModule_Create (*C function*), 209
PyModule_Create2 (*C function*), 209
PyModule_ExecDef (*C function*), 209
PyModule_FromDefAndSpec (*C function*), 209
PyModule_FromDefAndSpec2 (*C function*), 209
PyModule_GetDef (*C function*), 206
PyModule_GetDict (*C function*), 205
PyModule_GetFilename (*C function*), 206
PyModule_GetFilenameObject (*C function*), 206
PyModule_GetName (*C function*), 205
PyModule_GetNameObject (*C function*), 205
PyModule_GetState (*C function*), 205
PyModule_New (*C function*), 205
PyModule_NewObject (*C function*), 205
PyModule_SetDocString (*C function*), 212
PyModule_Type (*C var*), 205
PyModuleDef (*C type*), 206
PyModuleDef_Init (*C function*), 75
PyModuleDef_Slot (*C type*), 207
PyModuleDef_Slot.slot (*C member*), 207
PyModuleDef_Slot.value (*C member*), 207
PyModuleDef.m_base (*C member*), 206
PyModuleDef.m_clear (*C member*), 207
PyModuleDef.m_doc (*C member*), 206
PyModuleDef.m_free (*C member*), 207
PyModuleDef.m_methods (*C member*), 207
PyModuleDef.m_name (*C member*), 206
PyModuleDef.m_size (*C member*), 206
PyModuleDef.m_slots (*C member*), 207
PyModuleDef.m_slots.m_reload (*C member*), 207
PyModuleDef.m_traverse (*C member*), 207
PyMonitoring_EnterScope (*C function*), 370
PyMonitoring_ExitScope (*C function*), 372
PyMonitoring_FireBranchLeftEvent (*C function*), 370
PyMonitoring_FireBranchRightEvent (*C function*), 370
PyMonitoring_FireCallEvent (*C function*), 369
PyMonitoring_FireCRaiseEvent (*C function*), 370
PyMonitoring_FireCReturnEvent (*C function*), 370
PyMonitoring_FireExceptionHandledEvent (*C function*), 370
PyMonitoring_FireJumpEvent (*C function*), 369
PyMonitoring_FireLineEvent (*C function*), 369
PyMonitoring_FirePyResumeEvent (*C function*), 369
PyMonitoring_FirePyReturnEvent (*C function*), 369
PyMonitoring_FirePyStartEvent (*C function*), 369
PyMonitoring_FirePyThrowEvent (*C function*), 370
PyMonitoring_FirePyUnwindEvent (*C function*), 370
PyMonitoring_FirePyYieldEvent (*C function*), 369
PyMonitoring_FireRaiseEvent (*C function*), 370
PyMonitoring_FireReraiseEvent (*C function*), 370
PyMonitoring_FireStopIterationEvent (*C function*), 370
PyMonitoringState (*C type*), 369
PyMutex (*C type*), 259
PyMutex_IsLocked (*C function*), 259
PyMutex_Lock (*C function*), 259
PyMutex_Unlock (*C function*), 259
PyNumber_Absolute (*C function*), 123
PyNumber_Add (*C function*), 122
PyNumber_And (*C function*), 123
PyNumber_AsSsize_t (*C function*), 125
PyNumber_Check (*C function*), 122
PyNumber_Divmod (*C function*), 122
PyNumber_Float (*C function*), 124
PyNumber_FloorDivide (*C function*), 122
PyNumber_Index (*C function*), 124
PyNumber_InPlaceAdd (*C function*), 123
PyNumber_InPlaceAnd (*C function*), 124
PyNumber_InPlaceFloorDivide (*C function*), 123
PyNumber_InPlaceLshift (*C function*), 124
PyNumber_InPlaceMatrixMultiply (*C function*), 123

PyNumber_InPlaceMultiply (*C function*), 123
PyNumber_InPlaceOr (*C function*), 124
PyNumber_InPlacePower (*C function*), 124
PyNumber_InPlaceRemainder (*C function*), 124
PyNumber_InPlaceRshift (*C function*), 124
PyNumber_InPlaceSubtract (*C function*), 123
PyNumber_InPlaceTrueDivide (*C function*), 123
PyNumber_InPlaceXor (*C function*), 124
PyNumber_Invert (*C function*), 123
PyNumber_Long (*C function*), 124
PyNumber_Lshift (*C function*), 123
PyNumber_MatrixMultiply (*C function*), 122
PyNumber_Multiply (*C function*), 122
PyNumber_Negative (*C function*), 122
PyNumber_Or (*C function*), 123
PyNumber_Positive (*C function*), 122
PyNumber_Power (*C function*), 122
PyNumber_Remainder (*C function*), 122
PyNumber_Rshift (*C function*), 123
PyNumber_Subtract (*C function*), 122
PyNumber_ToBase (*C function*), 124
PyNumber_TrueDivide (*C function*), 122
PyNumber_Xor (*C function*), 123
PyNumberMethods (*C type*), 349
PyNumberMethods.nb_absolute (*C member*), 351
PyNumberMethods.nb_add (*C member*), 351
PyNumberMethods.nb_and (*C member*), 351
PyNumberMethods.nb_bool (*C member*), 351
PyNumberMethods.nb_divmod (*C member*), 351
PyNumberMethods.nb_float (*C member*), 351
PyNumberMethods.nb_floor_divide (*C member*), 351
PyNumberMethods.nb_index (*C member*), 352
PyNumberMethods.nb_inplace_add (*C member*), 351
PyNumberMethods.nb_inplace_and (*C member*), 351
PyNumberMethods.nb_inplace_floor_divide (*C member*), 351
PyNumberMethods.nb_inplace_lshift (*C member*), 351
PyNumberMethods.nb_inplace_matrix_multiply (*C member*), 352
PyNumberMethods.nb_inplace_multiply (*C member*), 351
PyNumberMethods.nb_inplace_or (*C member*), 351
PyNumberMethods.nb_inplace_power (*C member*), 351
PyNumberMethods.nb_inplace_remainder (*C member*), 351
PyNumberMethods.nb_inplace_rshift (*C member*), 351
PyNumberMethods.nb_inplace_subtract (*C member*), 351
PyNumberMethods.nb_inplace_true_divide (*C member*), 352
PyNumberMethods.nb_inplace_xor (*C member*), 351
PyNumberMethods.nb_int (*C member*), 351
PyNumberMethods.nb_invert (*C member*), 351
PyNumberMethods.nb_lshift (*C member*), 351
PyNumberMethods.nb_matrix_multiply (*C member*), 352
PyNumberMethods.nb_multiply (*C member*), 351
PyNumberMethods.nb_negative (*C member*), 351
PyNumberMethods.nb_or (*C member*), 351
PyNumberMethods.nb_positive (*C member*), 351
PyNumberMethods.nb_power (*C member*), 351
PyNumberMethods.nb_remainder (*C member*), 351
PyNumberMethods.nb_reserved (*C member*), 351
PyNumberMethods.nb_rshift (*C member*), 351
PyNumberMethods.nb_subtract (*C member*), 351
PyNumberMethods.nb_true_divide (*C member*), 351
PyNumberMethods.nb_xor (*C member*), 351
PyObject (*C type*), 309
PyObject_ASCII (*C function*), 111
PyObject_AsFileDescriptor (*C function*), 204
PyObject_Bytes (*C function*), 112
PyObject_Call (*C function*), 119
PyObject_CallFinalizer (*C function*), 309
PyObject_CallFinalizerFromDealloc (*C function*), 309
PyObject_CallFunction (*C function*), 120
PyObject_CallFunctionObjArgs (*C function*), 120
PyObject_CallMethod (*C function*), 120
PyObject_CallMethodNoArgs (*C function*), 121
PyObject_CallMethodObjArgs (*C function*), 120
PyObject_CallMethodOneArg (*C function*), 121
PyObject_CallNoArgs (*C function*), 120
PyObject_CallObject (*C function*), 120
PyObject_Calloc (*C function*), 295
PyObject_CallOneArg (*C function*), 120
PyObject_CheckBuffer (*C function*), 135
PyObject_ClearManagedDict (*C function*), 114
PyObject_ClearWeakRefs (*C function*), 217
PyObject_CopyData (*C function*), 135
PyObject_Del (*C function*), 305
PyObject_DelAttr (*C function*), 110
PyObject_DelAttrString (*C function*), 110
PyObject_DelItem (*C function*), 113
PyObject_DelItemString (*C function*), 113
PyObject_Dir (*C function*), 113
PyObject_Format (*C function*), 111
PyObject_Free (*C function*), 295
PyObject_GC_Del (*C function*), 361
PyObject_GC_IsFinalized (*C function*), 361
PyObject_GC_IsTracked (*C function*), 361
PyObject_GC_New (*C macro*), 359
PyObject_GC_NewVar (*C macro*), 359
PyObject_GC_Resize (*C macro*), 360
PyObject_GC_Track (*C function*), 360
PyObject_GC_UnTrack (*C function*), 361
PyObject_GenericGetAttr (*C function*), 110

PyObject_GenericGetDict (*C function*), 110
PyObject_GenericHash (*C function*), 99
PyObject_GenericSetAttr (*C function*), 110
PyObject_GenericSetDict (*C function*), 111
PyObject_GetAIter (*C function*), 113
PyObject_GetArenaAllocator (*C function*), 300
PyObject_GetAttr (*C function*), 109
PyObject_GetAttrString (*C function*), 109
PyObject_GetBuffer (*C function*), 135
PyObject_GetItem (*C function*), 113
PyObject_GetItemData (*C function*), 114
PyObject_GetIter (*C function*), 113
PyObject_GetOptionalAttr (*C function*), 109
PyObject_GetOptionalAttrString (*C function*), 110
PyObject_GetTypeData (*C function*), 113
PyObject_HasAttr (*C function*), 109
PyObject_HasAttrString (*C function*), 109
PyObject_HasAttrStringWithError (*C function*), 109
PyObject_HasAttrWithError (*C function*), 109
PyObject_Hash (*C function*), 112
PyObject_HashNotImplemented (*C function*), 112
PyObject_HEAD (*C macro*), 310
PyObject_HEAD_INIT (*C macro*), 311
PyObject_Init (*C function*), 303
PyObject_InitVar (*C function*), 303
PyObject_IS_GC (*C function*), 360
PyObject_IsInstance (*C function*), 112
PyObject_IsSubclass (*C function*), 112
PyObject_IsTrue (*C function*), 112
PyObject_Length (*C function*), 113
PyObject_LengthHint (*C function*), 113
PyObject_Malloc (*C function*), 295
PyObject_New (*C macro*), 303
PyObject_NewVar (*C macro*), 304
PyObject_Not (*C function*), 112
PyObject_Print (*C function*), 108
PyObject_Realloc (*C function*), 295
PyObject_Repr (*C function*), 111
PyObject_RichCompare (*C function*), 111
PyObject_RichCompareBool (*C function*), 111
PyObject_SelfIter (*C function*), 113
PyObject_SetArenaAllocator (*C function*), 300
PyObject_SetAttr (*C function*), 110
PyObject_SetAttrString (*C function*), 110
PyObject_SetItem (*C function*), 113
PyObject_Size (*C function*), 113
PyObject_Str (*C function*), 111
PyObject_Type (*C function*), 112
PyObject_TypeCheck (*C function*), 112
PyObject_VAR_HEAD (*C macro*), 310
PyObject_Vectorcall (*C function*), 121
PyObject_VectorcallDict (*C function*), 121
PyObject_VectorcallMethod (*C function*), 121
PyObject_VisitManagedDict (*C function*), 114
PyObjectArenaAllocator (*C type*), 300
PyObject.ob_refcnt (*C member*), 309
PyObject.ob_type (*C member*), 309
PyOS_AfterFork (*C function*), 78
PyOS_AfterFork_Child (*C function*), 78
PyOS_AfterFork_Parent (*C function*), 77
PyOS_BeforeFork (*C function*), 77
PyOS_CheckStack (*C function*), 78
PyOS_double_to_string (*C function*), 98
PyOS_FSPath (*C function*), 77
PyOS_getsig (*C function*), 78
PyOS_InputHook (*C var*), 44
PyOS_ReadlineFunctionPointer (*C var*), 44
PyOS_setsig (*C function*), 78
PyOS_sighandler_t (*C type*), 78
PyOS_snprintf (*C function*), 96
PyOS_stricmp (*C function*), 98
PyOS_string_to_double (*C function*), 97
PyOS_strnicmp (*C function*), 98
PyOS_strtol (*C function*), 97
PyOS_strtoul (*C function*), 97
PyOS_vsnprintf (*C function*), 96
PyPreConfig (*C type*), 271
PyPreConfig_InitIsolatedConfig (*C function*), 271
PyPreConfig_InitPythonConfig (*C function*), 271
PyPreConfig.allocator (*C member*), 271
PyPreConfig.coerce_c_locale (*C member*), 272
PyPreConfig.coerce_c_locale_warn (*C member*), 272
PyPreConfig.configure_locale (*C member*), 272
PyPreConfig.dev_mode (*C member*), 272
PyPreConfig.isolated (*C member*), 272
PyPreConfig.legacy_windows_fs_encoding (*C member*), 272
PyPreConfig.parse_argv (*C member*), 273
PyPreConfig.use_environment (*C member*), 273
PyPreConfig.utf8_mode (*C member*), 273
PyProperty_Type (*C var*), 213
PyRefTracer (*C type*), 256
PyRefTracer_CREATE (*C var*), 256
PyRefTracer_DESTROY (*C var*), 256
PyRefTracer_GetTracer (*C function*), 256
PyRefTracer_SetTracer (*C function*), 256
PyRun_AnyFile (*C function*), 43
PyRun_AnyFileEx (*C function*), 43
PyRun_AnyFileExFlags (*C function*), 43
PyRun_AnyFileFlags (*C function*), 43
PyRun_File (*C function*), 45
PyRun_FileEx (*C function*), 45
PyRun_FileExFlags (*C function*), 45
PyRun_FileFlags (*C function*), 45
PyRun_InteractiveLoop (*C function*), 44
PyRun_InteractiveLoopFlags (*C function*), 44
PyRun_InteractiveOne (*C function*), 44
PyRun_InteractiveOneFlags (*C function*), 44
PyRun_SimpleFile (*C function*), 44
PyRun_SimpleFileEx (*C function*), 44
PyRun_SimpleFileExFlags (*C function*), 44
PyRun_SimpleString (*C function*), 43

PyRun_SimpleStringFlags (*C function*), 43
PyRun_String (*C function*), 45
PyRun_StringFlags (*C function*), 45
PySendResult (*C type*), 129
PySeqIter_Check (*C function*), 213
PySeqIter_New (*C function*), 213
PySeqIter_Type (*C var*), 213
PySequence_Check (*C function*), 125
PySequence_Concat (*C function*), 125
PySequence_Contains (*C function*), 126
PySequence_Count (*C function*), 126
PySequence_DelItem (*C function*), 126
PySequence_DelSlice (*C function*), 126
PySequence_Fast (*C function*), 126
PySequence_Fast_GET_ITEM (*C function*), 126
PySequence_Fast_GET_SIZE (*C function*), 126
PySequence_Fast_ITEMS (*C function*), 127
PySequence_GetItem (*C function*), 9, 125
PySequence_GetSlice (*C function*), 125
PySequence_In (*C function*), 126
PySequence_Index (*C function*), 126
PySequence_InPlaceConcat (*C function*), 125
PySequence_InPlaceRepeat (*C function*), 125
PySequence_ITEM (*C function*), 127
PySequence_Length (*C function*), 125
PySequence_List (*C function*), 126
PySequence_Repeat (*C function*), 125
PySequence_SetItem (*C function*), 125
PySequence_SetSlice (*C function*), 126
PySequence_Size (*C function*), 125
PySequence_Tuple (*C function*), 126
PySequenceMethods (*C type*), 352
PySequenceMethods.sq_ass_item (*C member*), 352
PySequenceMethods.sq_concat (*C member*), 352
PySequenceMethods.sq_contains (*C member*), 353
PySequenceMethods.sq_inplace_concat (*C member*), 353
PySequenceMethods.sq_inplace_repeat (*C member*), 353
PySequenceMethods.sq_item (*C member*), 352
PySequenceMethods.sq_length (*C member*), 352
PySequenceMethods.sq_repeat (*C member*), 352
PySet_Add (*C function*), 194
PySet_Check (*C function*), 193
PySet_CheckExact (*C function*), 193
PySet_Clear (*C function*), 194
PySet_Contains (*C function*), 194
PySet_Discard (*C function*), 194
PySet_GET_SIZE (*C function*), 194
PySet_New (*C function*), 194
PySet_Pop (*C function*), 194
PySet_Size (*C function*), 194
PySet_Type (*C var*), 193
PySetObject (*C type*), 193
PySignal_SetWakeupFd (*C function*), 62
PySlice_AdjustIndices (*C function*), 215
PySlice_Check (*C function*), 214
PySlice_GetIndices (*C function*), 214
PySlice_GetIndicesEx (*C function*), 214
PySlice_New (*C function*), 214
PySlice_Type (*C var*), 214
PySlice_Unpack (*C function*), 215
PyState_AddModule (*C function*), 213
PyState_FindModule (*C function*), 212
PyState_RemoveModule (*C function*), 213
PyStatus (*C type*), 270
PyStatus_Error (*C function*), 270
PyStatus_Exception (*C function*), 270
PyStatus_Exit (*C function*), 270
PyStatus_IsError (*C function*), 271
PyStatus_IsExit (*C function*), 271
PyStatus_NoMemory (*C function*), 270
PyStatus_Ok (*C function*), 270
PyStatus.err_msg (*C member*), 270
PyStatus.exitcode (*C member*), 270
PyStatus.func (*C member*), 270
PyStructSequence_Desc (*C type*), 185
PyStructSequence_Desc.doc (*C member*), 185
PyStructSequence_Desc.fields (*C member*), 185
PyStructSequence_Desc.n_in_sequence (*C member*), 185
PyStructSequence_Desc.name (*C member*), 185
PyStructSequence_Field (*C type*), 185
PyStructSequence_Field.doc (*C member*), 185
PyStructSequence_Field.name (*C member*), 185
PyStructSequence_GET_ITEM (*C function*), 185
PyStructSequence_GetItem (*C function*), 185
PyStructSequence_InitType (*C function*), 184
PyStructSequence_InitType2 (*C function*), 184
PyStructSequence_New (*C function*), 185
PyStructSequence_NewType (*C function*), 184
PyStructSequence_SET_ITEM (*C function*), 186
PyStructSequence_SetItem (*C function*), 185
PyStructSequence_UnnamedField (*C var*), 185
PySys_AddAuditHook (*C function*), 81
PySys_Audit (*C function*), 81
PySys_AuditTuple (*C function*), 81
PySys_FormatStderr (*C function*), 81
PySys_FormatStdout (*C function*), 81
PySys_GetObject (*C function*), 80
PySys_GetXOptions (*C function*), 81
PySys_ResetWarnOptions (*C function*), 80
PySys_SetArgv (*C function*), 241
PySys_SetArgvEx (*C function*), 240
PySys_SetObject (*C function*), 80
PySys_WriteStderr (*C function*), 81
PySys_WriteStdout (*C function*), 80
Python 3000, **389**
Python Enhancement Proposals
    PEP 1, 388
    PEP 7, 3, 6
    PEP 238, 381
    PEP 278, 392
    PEP 302, 385

PEP 343, 378
PEP 353, 10
PEP 362, 376, 388
PEP 383, 170, 171
PEP 387, 15, 16
PEP 393, 161
PEP 411, 389
PEP 420, 387, 388
PEP 442, 308, 348
PEP 443, 382
PEP 446, 80
PEP 451, 208
PEP 456, 99
PEP 483, 382
PEP 484, 375, 382, 392
PEP 489, 73, 74, 251
PEP 492, 376, 377, 379
PEP 498, 380
PEP 519, 388
PEP 523, 222, 249
PEP 525, 376
PEP 526, 375, 392
PEP 528, 234, 280
PEP 529, 171, 234
PEP 538, 288
PEP 539, 257
PEP 540, 288
PEP 552, 277
PEP 554, 252
PEP 578, 82
PEP 585, 382
PEP 587, 269
PEP 590, 117
PEP 623, 162
PEP 0626#out-of-process-debuggers-and-profilers, 200
PEP 634, 335, 336
PEP 649, 375
PEP 667, 100, 221
PEP 0683, 50, 383
PEP 684, 245, 252
PEP 703, 381, 383
PEP 741, 263
PEP 3116, 392
PEP 3119, 112
PEP 3121, 207
PEP 3147, 85
PEP 3151, 70
PEP 3155, 389
PYTHON_ABI_VERSION (*C macro*), 210
PYTHON_API_VERSION (*C macro*), 210
PYTHON_CPU_COUNT, 280
PYTHON_FROZEN_MODULES, 279
PYTHON_GIL, 383
PYTHON_PERF_JIT_SUPPORT, 284
PYTHON_PRESITE, 283
PYTHONCOERCECLOCALE, 288
PYTHONDEBUG, 233, 282
PYTHONDEVMODE, 277
PYTHONDONTWRITEBYTECODE, 233, 285
PYTHONDUMPREFS, 277
PYTHONDUMPREFSFILE, 278
PYTHONEXECUTABLE, 282
PYTHONFAULTHANDLER, 278
PYTHONHASHSEED, 233, 279
PYTHONHOME, 12, 233, 241, 242, 279
Pythónico, **389**
PYTHONINSPECT, 233, 279
PYTHONINTMAXSTRDIGITS, 280
PYTHONIOENCODING, 284
PYTHONLEGACYWINDOWSFSENCODING, 234, 273
PYTHONLEGACYWINDOWSSTDIO, 234, 280
PYTHONMALLOC, 292, 296, 298, 300
PYTHONMALLOCSTATS, 280, 292
PYTHONNODEBUGRANGES, 277
PYTHONNOUSERSITE, 235, 285
PYTHONOPTIMIZE, 235, 281
PYTHONPATH, 12, 233, 281
PYTHONPERFSUPPORT, 284
PYTHONPLATLIBDIR, 281
PYTHONPROFILEIMPORTTIME, 279
PYTHONPYCACHEPREFIX, 282
PYTHONSAFEPATH, 276
PYTHONTRACEMALLOC, 284
PYTHONUNBUFFERED, 235, 276
PYTHONUTF8, 273, 288
PYTHONVERBOSE, 235, 285
PYTHONWARNINGS, 285
PyThread_create_key (*C function*), 258
PyThread_delete_key (*C function*), 258
PyThread_delete_key_value (*C function*), 259
PyThread_get_key_value (*C function*), 258
PyThread_ReInitTLS (*C function*), 259
PyThread_set_key_value (*C function*), 258
PyThread_tss_alloc (*C function*), 257
PyThread_tss_create (*C function*), 258
PyThread_tss_delete (*C function*), 258
PyThread_tss_free (*C function*), 257
PyThread_tss_get (*C function*), 258
PyThread_tss_is_created (*C function*), 258
PyThread_tss_set (*C function*), 258
PyThreadState (*C type*), 242, 245
PyThreadState_Clear (*C function*), 248
PyThreadState_Delete (*C function*), 248
PyThreadState_DeleteCurrent (*C function*), 248
PyThreadState_EnterTracing (*C function*), 248
PyThreadState_Get (*C function*), 245
PyThreadState_GetDict (*C function*), 249
PyThreadState_GetFrame (*C function*), 248
PyThreadState_GetID (*C function*), 248
PyThreadState_GetInterpreter (*C function*), 248
PyThreadState_GetUnchecked (*C function*), 245
PyThreadState_LeaveTracing (*C function*), 248
PyThreadState_New (*C function*), 248
PyThreadState_Next (*C function*), 256
PyThreadState_SetAsyncExc (*C function*), 249

PyThreadState_Swap (*C function*), 246
PyThreadState.interp (*C member*), 245
PyTime_AsSecondsDouble (*C function*), 104
PyTime_Check (*C function*), 226
PyTime_CheckExact (*C function*), 226
PyTime_FromTime (*C function*), 227
PyTime_FromTimeAndFold (*C function*), 227
PyTime_MAX (*C var*), 103
PyTime_MIN (*C var*), 103
PyTime_Monotonic (*C function*), 103
PyTime_MonotonicRaw (*C function*), 103
PyTime_PerfCounter (*C function*), 103
PyTime_PerfCounterRaw (*C function*), 104
PyTime_t (*C type*), 103
PyTime_Time (*C function*), 103
PyTime_TimeRaw (*C function*), 104
PyTimeZone_FromOffset (*C function*), 227
PyTimeZone_FromOffsetAndName (*C function*), 227
PyTrace_C_CALL (*C var*), 255
PyTrace_C_EXCEPTION (*C var*), 255
PyTrace_C_RETURN (*C var*), 255
PyTrace_CALL (*C var*), 254
PyTrace_EXCEPTION (*C var*), 254
PyTrace_LINE (*C var*), 254
PyTrace_OPCODE (*C var*), 255
PyTrace_RETURN (*C var*), 255
PyTraceMalloc_Track (*C function*), 300
PyTraceMalloc_Untrack (*C function*), 300
PyTuple_Check (*C function*), 183
PyTuple_CheckExact (*C function*), 183
PyTuple_GET_ITEM (*C function*), 183
PyTuple_GET_SIZE (*C function*), 183
PyTuple_GetItem (*C function*), 183
PyTuple_GetSlice (*C function*), 183
PyTuple_New (*C function*), 183
PyTuple_Pack (*C function*), 183
PyTuple_SET_ITEM (*C function*), 184
PyTuple_SetItem (*C function*), 8, 184
PyTuple_Size (*C function*), 183
PyTuple_Type (*C var*), 183
PyTupleObject (*C type*), 183
PyType_AddWatcher (*C function*), 138
PyType_Check (*C function*), 137
PyType_CheckExact (*C function*), 137
PyType_ClearCache (*C function*), 137
PyType_ClearWatcher (*C function*), 138
PyType_Freeze (*C function*), 142
PyType_FromMetaclass (*C function*), 141
PyType_FromModuleAndSpec (*C function*), 141
PyType_FromSpec (*C function*), 142
PyType_FromSpecWithBases (*C function*), 142
PyType_GenericAlloc (*C function*), 139
PyType_GenericNew (*C function*), 139
PyType_GetBaseByToken (*C function*), 140
PyType_GetDict (*C function*), 138
PyType_GetFlags (*C function*), 137
PyType_GetFullyQualifiedName (*C function*), 139
PyType_GetModule (*C function*), 140
PyType_GetModuleByDef (*C function*), 140
PyType_GetModuleName (*C function*), 139
PyType_GetModuleState (*C function*), 140
PyType_GetName (*C function*), 139
PyType_GetQualName (*C function*), 139
PyType_GetSlot (*C function*), 140
PyType_GetTypeDataSize (*C function*), 114
PyType_HasFeature (*C function*), 138
PyType_IS_GC (*C function*), 138
PyType_IsSubtype (*C function*), 139
PyType_Modified (*C function*), 138
PyType_Ready (*C function*), 139
PyType_Slot (*C type*), 143
PyType_Slot.pfunc (*C member*), 143
PyType_Slot.slot (*C member*), 143
PyType_Spec (*C type*), 142
PyType_Spec.basicsize (*C member*), 142
PyType_Spec.flags (*C member*), 143
PyType_Spec.itemsize (*C member*), 142
PyType_Spec.name (*C member*), 142
PyType_Spec.slots (*C member*), 143
PyType_Type (*C var*), 137
PyType_Watch (*C function*), 138
PyType_WatchCallback (*C type*), 138
PyTypeObject (*C type*), 137
PyTypeObject.tp_alloc (*C member*), 344
PyTypeObject.tp_as_async (*C member*), 329
PyTypeObject.tp_as_buffer (*C member*), 332
PyTypeObject.tp_as_mapping (*C member*), 330
PyTypeObject.tp_as_number (*C member*), 330
PyTypeObject.tp_as_sequence (*C member*), 330
PyTypeObject.tp_base (*C member*), 342
PyTypeObject.tp_bases (*C member*), 345
PyTypeObject.tp_basicsize (*C member*), 325
PyTypeObject.tp_cache (*C member*), 346
PyTypeObject.tp_call (*C member*), 330
PyTypeObject.tp_clear (*C member*), 337
PyTypeObject.tp_dealloc (*C member*), 326
PyTypeObject.tp_del (*C member*), 346
PyTypeObject.tp_descr_get (*C member*), 342
PyTypeObject.tp_descr_set (*C member*), 343
PyTypeObject.tp_dict (*C member*), 342
PyTypeObject.tp_dictoffset (*C member*), 343
PyTypeObject.tp_doc (*C member*), 336
PyTypeObject.tp_finalize (*C member*), 346
PyTypeObject.tp_flags (*C member*), 332
PyTypeObject.tp_free (*C member*), 344
PyTypeObject.tp_getattr (*C member*), 329
PyTypeObject.tp_getattro (*C member*), 331
PyTypeObject.tp_getset (*C member*), 341
PyTypeObject.tp_hash (*C member*), 330
PyTypeObject.tp_init (*C member*), 343
PyTypeObject.tp_is_gc (*C member*), 345
PyTypeObject.tp_itemsize (*C member*), 325
PyTypeObject.tp_iter (*C member*), 341
PyTypeObject.tp_iternext (*C member*), 341
PyTypeObject.tp_members (*C member*), 341
PyTypeObject.tp_methods (*C member*), 341

PyTypeObject.tp_mro (*C member*), 345
PyTypeObject.tp_name (*C member*), 325
PyTypeObject.tp_new (*C member*), 344
PyTypeObject.tp_repr (*C member*), 329
PyTypeObject.tp_richcompare (*C member*), 339
PyTypeObject.tp_setattr (*C member*), 329
PyTypeObject.tp_setattro (*C member*), 331
PyTypeObject.tp_str (*C member*), 331
PyTypeObject.tp_subclasses (*C member*), 346
PyTypeObject.tp_traverse (*C member*), 336
PyTypeObject.tp_vectorcall (*C member*), 348
PyTypeObject.tp_vectorcall_offset (*C member*), 328
PyTypeObject.tp_version_tag (*C member*), 346
PyTypeObject.tp_watched (*C member*), 349
PyTypeObject.tp_weaklist (*C member*), 346
PyTypeObject.tp_weaklistoffset (*C member*), 340
PyTZInfo_Check (*C function*), 227
PyTZInfo_CheckExact (*C function*), 227
PyUnicode_1BYTE_DATA (*C function*), 162
PyUnicode_1BYTE_KIND (*C macro*), 162
PyUnicode_2BYTE_DATA (*C function*), 162
PyUnicode_2BYTE_KIND (*C macro*), 162
PyUnicode_4BYTE_DATA (*C function*), 162
PyUnicode_4BYTE_KIND (*C macro*), 162
PyUnicode_Append (*C function*), 168
PyUnicode_AppendAndDel (*C function*), 168
PyUnicode_AsASCIIString (*C function*), 176
PyUnicode_AsCharmapString (*C function*), 177
PyUnicode_AsEncodedString (*C function*), 173
PyUnicode_AsLatin1String (*C function*), 176
PyUnicode_AsMBCSString (*C function*), 177
PyUnicode_AsRawUnicodeEscapeString (*C function*), 176
PyUnicode_AsUCS4 (*C function*), 169
PyUnicode_AsUCS4Copy (*C function*), 170
PyUnicode_AsUnicodeEscapeString (*C function*), 175
PyUnicode_AsUTF8 (*C function*), 174
PyUnicode_AsUTF8AndSize (*C function*), 173
PyUnicode_AsUTF8String (*C function*), 173
PyUnicode_AsUTF16String (*C function*), 175
PyUnicode_AsUTF32String (*C function*), 174
PyUnicode_AsWideChar (*C function*), 172
PyUnicode_AsWideCharString (*C function*), 172
PyUnicode_BuildEncodingMap (*C function*), 168
PyUnicode_Check (*C function*), 162
PyUnicode_CHECK_INTERNED (*C function*), 180
PyUnicode_CheckExact (*C function*), 162
PyUnicode_Compare (*C function*), 179
PyUnicode_CompareWithASCIIString (*C function*), 180
PyUnicode_Concat (*C function*), 178
PyUnicode_Contains (*C function*), 180
PyUnicode_CopyCharacters (*C function*), 169
PyUnicode_Count (*C function*), 179
PyUnicode_DATA (*C function*), 163
PyUnicode_Decode (*C function*), 173
PyUnicode_DecodeASCII (*C function*), 176
PyUnicode_DecodeCharmap (*C function*), 176
PyUnicode_DecodeCodePageStateful (*C function*), 177
PyUnicode_DecodeFSDefault (*C function*), 171
PyUnicode_DecodeFSDefaultAndSize (*C function*), 171
PyUnicode_DecodeLatin1 (*C function*), 176
PyUnicode_DecodeLocale (*C function*), 170
PyUnicode_DecodeLocaleAndSize (*C function*), 170
PyUnicode_DecodeMBCS (*C function*), 177
PyUnicode_DecodeMBCSStateful (*C function*), 177
PyUnicode_DecodeRawUnicodeEscape (*C function*), 176
PyUnicode_DecodeUnicodeEscape (*C function*), 175
PyUnicode_DecodeUTF7 (*C function*), 175
PyUnicode_DecodeUTF7Stateful (*C function*), 175
PyUnicode_DecodeUTF8 (*C function*), 173
PyUnicode_DecodeUTF8Stateful (*C function*), 173
PyUnicode_DecodeUTF16 (*C function*), 175
PyUnicode_DecodeUTF16Stateful (*C function*), 175
PyUnicode_DecodeUTF32 (*C function*), 174
PyUnicode_DecodeUTF32Stateful (*C function*), 174
PyUnicode_EncodeCodePage (*C function*), 177
PyUnicode_EncodeFSDefault (*C function*), 171
PyUnicode_EncodeLocale (*C function*), 170
PyUnicode_Equal (*C function*), 179
PyUnicode_EqualToUTF8 (*C function*), 180
PyUnicode_EqualToUTF8AndSize (*C function*), 179
PyUnicode_Fill (*C function*), 169
PyUnicode_Find (*C function*), 178
PyUnicode_FindChar (*C function*), 179
PyUnicode_Format (*C function*), 180
PyUnicode_FromEncodedObject (*C function*), 168
PyUnicode_FromFormat (*C function*), 166
PyUnicode_FromFormatV (*C function*), 168
PyUnicode_FromKindAndData (*C function*), 165
PyUnicode_FromObject (*C function*), 168
PyUnicode_FromOrdinal (*C function*), 168
PyUnicode_FromString (*C function*), 166
PyUnicode_FromStringAndSize (*C function*), 165
PyUnicode_FromWideChar (*C function*), 172
PyUnicode_FSConverter (*C function*), 171
PyUnicode_FSDecoder (*C function*), 171
PyUnicode_GET_LENGTH (*C function*), 162
PyUnicode_GetDefaultEncoding (*C function*), 168
PyUnicode_GetLength (*C function*), 168
PyUnicode_InternFromString (*C function*), 180
PyUnicode_InternInPlace (*C function*), 180
PyUnicode_IS_ASCII (*C function*), 163
PyUnicode_IS_READY (*C function*), 183
PyUnicode_IsIdentifier (*C function*), 163
PyUnicode_Join (*C function*), 178

PyUnicode_KIND (*C function*), 163
PyUnicode_MAX_CHAR_VALUE (*C function*), 163
PyUnicode_New (*C function*), 165
PyUnicode_Partition (*C function*), 178
PyUnicode_READ (*C function*), 163
PyUnicode_READ_CHAR (*C function*), 163
PyUnicode_ReadChar (*C function*), 169
PyUnicode_READY (*C function*), 182
PyUnicode_Replace (*C function*), 179
PyUnicode_Resize (*C function*), 169
PyUnicode_RichCompare (*C function*), 180
PyUnicode_RPartition (*C function*), 178
PyUnicode_RSplit (*C function*), 178
PyUnicode_Split (*C function*), 178
PyUnicode_Splitlines (*C function*), 178
PyUnicode_Substring (*C function*), 169
PyUnicode_Tailmatch (*C function*), 178
PyUnicode_Translate (*C function*), 177
PyUnicode_Type (*C var*), 162
PyUnicode_WRITE (*C function*), 163
PyUnicode_WriteChar (*C function*), 169
PyUnicodeDecodeError_Create (*C function*), 63
PyUnicodeDecodeError_GetEncoding (*C function*), 63
PyUnicodeDecodeError_GetEnd (*C function*), 64
PyUnicodeDecodeError_GetObject (*C function*), 63
PyUnicodeDecodeError_GetReason (*C function*), 64
PyUnicodeDecodeError_GetStart (*C function*), 63
PyUnicodeDecodeError_SetEnd (*C function*), 64
PyUnicodeDecodeError_SetReason (*C function*), 64
PyUnicodeDecodeError_SetStart (*C function*), 64
PyUnicodeEncodeError_GetEncoding (*C function*), 63
PyUnicodeEncodeError_GetEnd (*C function*), 64
PyUnicodeEncodeError_GetObject (*C function*), 63
PyUnicodeEncodeError_GetReason (*C function*), 64
PyUnicodeEncodeError_GetStart (*C function*), 63
PyUnicodeEncodeError_SetEnd (*C function*), 64
PyUnicodeEncodeError_SetReason (*C function*), 64
PyUnicodeEncodeError_SetStart (*C function*), 64
PyUnicodeIter_Type (*C var*), 162
PyUnicodeObject (*C type*), 162
PyUnicodeTranslateError_GetEnd (*C function*), 64
PyUnicodeTranslateError_GetObject (*C function*), 63
PyUnicodeTranslateError_GetReason (*C function*), 64
PyUnicodeTranslateError_GetStart (*C function*), 63
PyUnicodeTranslateError_SetEnd (*C function*), 64
PyUnicodeTranslateError_SetReason (*C function*), 64
PyUnicodeTranslateError_SetStart (*C function*), 64
PyUnicodeWriter (*C type*), 181
PyUnicodeWriter_Create (*C function*), 181
PyUnicodeWriter_DecodeUTF8Stateful (*C function*), 182
PyUnicodeWriter_Discard (*C function*), 181
PyUnicodeWriter_Finish (*C function*), 181
PyUnicodeWriter_Format (*C function*), 182
PyUnicodeWriter_WriteASCII (*C function*), 181
PyUnicodeWriter_WriteChar (*C function*), 181
PyUnicodeWriter_WriteRepr (*C function*), 182
PyUnicodeWriter_WriteStr (*C function*), 182
PyUnicodeWriter_WriteSubstring (*C function*), 182
PyUnicodeWriter_WriteUCS4 (*C function*), 182
PyUnicodeWriter_WriteUTF8 (*C function*), 181
PyUnicodeWriter_WriteWideChar (*C function*), 181
PyUnstable, 15
PyUnstable_AtExit (*C function*), 238
PyUnstable_Code_GetExtra (*C function*), 203
PyUnstable_Code_GetFirstFree (*C function*), 199
PyUnstable_Code_New (*C function*), 199
PyUnstable_Code_NewWithPosOnlyArgs (*C function*), 199
PyUnstable_Code_SetExtra (*C function*), 203
PyUnstable_EnableTryIncRef (*C function*), 116
PyUnstable_Eval_RequestCodeExtraIndex (*C function*), 203
PyUnstable_Exc_PrepReraiseStar (*C function*), 63
PyUnstable_GC_VisitObjects (*C function*), 362
PyUnstable_InterpreterFrame_GetCode (*C function*), 222
PyUnstable_InterpreterFrame_GetLasti (*C function*), 222
PyUnstable_InterpreterFrame_GetLine (*C function*), 222
PyUnstable_IsImmortal (*C function*), 115
PyUnstable_Long_CompactValue (*C function*), 152
PyUnstable_Long_IsCompact (*C function*), 152
PyUnstable_Module_SetGIL (*C function*), 212
PyUnstable_Object_ClearWeakRefsNoCallbacks (*C function*), 217
PyUnstable_Object_EnableDeferredRefcount (*C function*), 114
PyUnstable_Object_GC_NewWithExtraData (*C function*), 360
PyUnstable_Object_IsUniquelyReferenced (*C function*), 117
PyUnstable_Object_IsUniqueReferencedTemporary (*C function*), 115
PyUnstable_PerfMapState_Fini (*C function*), 105
PyUnstable_PerfMapState_Init (*C function*), 104
PyUnstable_TryIncRef (*C function*), 115

PyUnstable_Type_AssignVersionTag (*C function*), 140
PyUnstable_WritePerfMapEntry (*C function*), 104
PyVarObject (*C type*), 309
PyVarObject_HEAD_INIT (*C macro*), 311
PyVarObject.ob_size (*C member*), 309
PyVectorcall_Call (*C function*), 119
PyVectorcall_Function (*C function*), 119
PyVectorcall_NARGS (*C function*), 119
PyWeakref_Check (*C function*), 216
PyWeakref_CheckProxy (*C function*), 216
PyWeakref_CheckRef (*C function*), 216
PyWeakref_GET_OBJECT (*C function*), 217
PyWeakref_GetObject (*C function*), 217
PyWeakref_GetRef (*C function*), 217
PyWeakref_IsDead (*C function*), 217
PyWeakref_NewProxy (*C function*), 217
PyWeakref_NewRef (*C function*), 217
PyWideStringList (*C type*), 270
PyWideStringList_Append (*C function*), 270
PyWideStringList_Insert (*C function*), 270
PyWideStringList.items (*C member*), 270
PyWideStringList.length (*C member*), 270
PyWrapper_New (*C function*), 214

R

READ_RESTRICTED (*C macro*), 316
READONLY (*C macro*), 316
realloc (*C function*), 291
rebanada, **390**
recolección de basura, **382**
referencia fuerte, **391**
referencia prestada, **377**
releasebufferproc (*C type*), 356
REPL, **390**
repr
    built-in function, 111, 329
reprfunc (*C type*), 355
RESTRICTED (*C macro*), 316
retrollamada, **378**
richcmpfunc (*C type*), 355
ruta de importación, **383**

S

saltos de líneas universales, **392**
search
    path, module, 12, 235, 239
secuencia, **390**
sendfunc (*C type*), 356
sentencia, **390**
sequence
    object, 158
set
    object, 193
set_all(), 9
setattrfunc (*C type*), 355
setattrofunc (*C type*), 355
setswitchinterval (*in module sys*), 242
setter (*C type*), 318
SIGINT (*C macro*), 61
signal
    module, 61
SIZE_MAX (*C macro*), 147
soft deprecated, **390**
special
    método, 390
ssizeargfunc (*C type*), 356
ssizeobjargproc (*C type*), 356
standard library, **390**
static type checker, **391**
staticmethod
    built-in function, 314
stderr (*in module sys*), 251, 252
stdin (*in module sys*), 251, 252
stdlib, **391**
stdout (*in module sys*), 251, 252
strerror (*C function*), 55
string
    PyObject_Str (*C function*), 111
structmember.h, 318
sum_list(), 9
sum_sequence(), 10, 11
sys
    module, 12, 235, 251, 252
SystemError (*built-in exception*), 205, 206

T

t-string, **391**
t-strings, **391**
T_BOOL (*C macro*), 318
T_BYTE (*C macro*), 318
T_CHAR (*C macro*), 318
T_DOUBLE (*C macro*), 318
T_FLOAT (*C macro*), 318
T_INT (*C macro*), 318
T_LONG (*C macro*), 318
T_LONGLONG (*C macro*), 318
T_NONE (*C macro*), 318
T_OBJECT (*C macro*), 318
T_OBJECT_EX (*C macro*), 318
T_PYSSIZET (*C macro*), 318
T_SHORT (*C macro*), 318
T_STRING (*C macro*), 318
T_STRING_INPLACE (*C macro*), 318
T_UBYTE (*C macro*), 318
T_UINT (*C macro*), 318
T_ULONG (*C macro*), 318
T_ULONGULONG (*C macro*), 318
T_USHORT (*C macro*), 318
ternaryfunc (*C type*), 356
thread state, **391**
tipado de pato, **380**
tipo, **392**
tipos genéricos, **382**
token, **391**
traverseproc (*C type*), 361

tupla
    función integrada, 188
tupla nombrada, **386**
tuple
    built-in function, 126
    object, 183
type
    built-in function, 112
    object, 7, 137

U

ULONG_MAX (*C macro*), 147
unaryfunc (*C type*), 356
USE_STACKCHECK (*C macro*), 78

V

variable de clase, **378**
variable de contexto, **378**
variables de entorno
    __PYVENV_LAUNCHER__, 276, 282
    PATH, 12
    PYTHON_CPU_COUNT, 280
    PYTHON_FROZEN_MODULES, 279
    PYTHON_GIL, 383
    PYTHON_PERF_JIT_SUPPORT, 284
    PYTHON_PRESITE, 283
    PYTHONCOERCECLOCALE, 288
    PYTHONDEBUG, 233, 282
    PYTHONDEVMODE, 277
    PYTHONDONTWRITEBYTECODE, 233, 285
    PYTHONDUMPREFS, 277
    PYTHONDUMPREFSFILE, 278
    PYTHONEXECUTABLE, 282
    PYTHONFAULTHANDLER, 278
    PYTHONHASHSEED, 233, 279
    PYTHONHOME, 12, 233, 241, 242, 279
    PYTHONINSPECT, 233, 279
    PYTHONINTMAXSTRDIGITS, 280
    PYTHONIOENCODING, 284
    PYTHONLEGACYWINDOWSFSENCODING, 234, 273
    PYTHONLEGACYWINDOWSSTDIO, 234, 280
    PYTHONMALLOC, 292, 296, 298, 300
    PYTHONMALLOCSTATS, 280, 292
    PYTHONNODEBUGRANGES, 277
    PYTHONNOUSERSITE, 235, 285
    PYTHONOPTIMIZE, 235, 281
    PYTHONPATH, 12, 233, 281
    PYTHONPERFSUPPORT, 284
    PYTHONPLATLIBDIR, 281
    PYTHONPROFILEIMPORTTIME, 279
    PYTHONPYCACHEPREFIX, 282
    PYTHONSAFEPATH, 276
    PYTHONTRACEMALLOC, 284
    PYTHONUNBUFFERED, 235, 276
    PYTHONUTF8, 273, 288
    PYTHONVERBOSE, 235, 285
    PYTHONWARNINGS, 285
vectorcallfunc (*C type*), 118
version (*in module sys*), 240
visitproc (*C type*), 361
vista de diccionario, **380**

W

walrus operator, **393**
WRITE_RESTRICTED (*C macro*), 316

Z

Zen de Python, **393**