The Python/C API

發🄵 *3.14.0rc3*

Guido van Rossum and the Python development team

10 月 01, 2025

Python Software Foundation
Email: docs@python.org

Contents

1 簡介 **3**
- 1.1 Language version compatibility . . . 3
- 1.2 編寫標准 . . . 3
- 1.3 引入檔案 (include files) . . . 3
- 1.4 有用的巨集 . . . 4
- 1.5 物件、型🄵和參照計數 . . . 6
  - 1.5.1 參照計數 . . . 6
  - 1.5.2 型🄵 . . . 9
- 1.6 例外 . . . 9
- 1.7 嵌入式 Python . . . 11
- 1.8 除錯建置 . . . 11
- 1.9 Recommended third party tools . . . 12

2 C API 穩定性 **13**
- 2.1 不穩定的 C API . . . 13
- 2.2 穩定的應用程式二進位介面 . . . 13
  - 2.2.1 受限 C API . . . 14
  - 2.2.2 穩定 ABI . . . 14
  - 2.2.3 受限 API 範圍和性能 . . . 14
  - 2.2.4 受限 API 注意事項 . . . 14
- 2.3 平台注意事項 . . . 15
- 2.4 受限 API 的🄵容 . . . 15

3 The Very High Level Layer **41**

4 參照計數 **47**

5 例外處理 **51**
- 5.1 Printing and clearing . . . 51
- 5.2 Raising exceptions . . . 52
- 5.3 Issuing warnings . . . 55
- 5.4 Querying the error indicator . . . 55
- 5.5 訊號處理 . . . 59
- 5.6 例外類🄵 . . . 60
- 5.7 例外物件 . . . 60
- 5.8 Unicode 例外物件 . . . 61
- 5.9 遞🄵控制 . . . 62
- 5.10 例外和警告型🄵 . . . 63
  - 5.10.1 例外型🄵 . . . 63
  - 5.10.2 OSError 🄵名 . . . 68
  - 5.10.3 警告型🄵 . . . 69

6 Defining extension modules **71**
6.1 Multiple module instances . . . 71
6.2 Initialization function . . . 72
6.3 Multi-phase initialization . . . 73
6.4 Legacy single-phase initialization . . . 73

7 工具 **75**
7.1 作業系統工具 . . . 75
7.2 系統函式 . . . 78
7.3 行程控制 . . . 80
7.4 引入模組 . . . 80
7.5 資料 marshal 操作的支援 . . . 84
7.6 剖析引數與建置數值 . . . 85
7.6.1 剖析引數 . . . 85
7.6.2 建置數值 . . . 92
7.7 字串轉🄵與格式化 . . . 94
7.8 PyHash API . . . 96
7.9 Reflection . . . 97
7.10 編解碼器🄵🄵表和支援函式 . . . 98
7.10.1 編解碼器查找 API . . . 99
7.10.2 用於 Unicode 編碼錯誤處理程式的🄵🄵 API . . . 99
7.11 PyTime C API . . . 100
7.11.1 型🄵 . . . 100
7.11.2 時鐘函式 . . . 100
7.11.3 原始時鐘函式 . . . 100
7.11.4 轉🄵函式 . . . 101
7.12 對 Perf Map 的支援 . . . 101

8 抽象物件層 (Abstract Objects Layer) **103**
8.1 物件協定 . . . 103
8.2 呼叫協定 (Call Protocol) . . . 113
8.2.1 *tp_call* 協定 . . . 113
8.2.2 Vectorcall 協定 . . . 113
8.2.3 物件呼叫 API . . . 115
8.2.4 呼叫支援 API . . . 117
8.3 數字協定 . . . 117
8.4 序列協定 . . . 120
8.5 對映協定 . . . 122
8.6 🄵代器協定 . . . 124
8.7 緩衝協定 (Buffer Protocol) . . . 125
8.7.1 Buffer structure . . . 125
8.7.2 Buffer request types . . . 127
8.7.3 Complex arrays . . . 129
8.7.4 Buffer-related functions . . . 130

9 具體物件層 **133**
9.1 基礎物件 . . . 133
9.1.1 型🄵物件 . . . 133
9.1.2 `None` 物件 . . . 140
9.2 數值物件 . . . 140
9.2.1 整數物件 . . . 140
9.2.2 Boolean（布林）物件 . . . 150
9.2.3 浮點數（Floating-Point）物件 . . . 150
9.2.4 🄵數物件 . . . 152
9.3 序列物件 . . . 154
9.3.1 位元組物件 (Bytes Objects) . . . 154
9.3.2 位元組陣列物件 (Byte Array Objects) . . . 156
9.3.3 Unicode 物件與編解碼器 . . . 157
9.3.4 Tuple（元組）物件 . . . 179

9.3.5 Struct Sequence Objects . . . 180
9.3.6 List（串列）物件 . . . 181
9.4 容器物件 . . . 183
9.4.1 字典物件 . . . 183
9.4.2 集合物件 . . . 188
9.5 函式物件 . . . 190
9.5.1 函式物件 (Function Objects) . . . 190
9.5.2 實例方法物件 (Instance Method Objects) . . . 192
9.5.3 方法物件 (Method Objects) . . . 192
9.5.4 Cell 物件 . . . 193
9.5.5 程式碼物件 . . . 193
9.5.6 程式碼物件旗標 . . . 196
9.5.7 額外資訊 . . . 198
9.6 其他物件 . . . 199
9.6.1 檔案物件 (File Objects) . . . 199
9.6.2 模組物件 . . . 200
9.6.3 模組定義 . . . 201
9.6.4 Creating extension modules dynamically . . . 203
9.6.5 支援的函式 . . . 204
9.6.6 🄵代器（Iterator）物件 . . . 208
9.6.7 Descriptor（描述器）物件 . . . 208
9.6.8 切片物件 . . . 209
9.6.9 MemoryView 物件 . . . 210
9.6.10 弱參照物件 . . . 211
9.6.11 Capsules . . . 212
9.6.12 Frame 物件 . . . 214
9.6.13 🄵生器 (Generator) 物件 . . . 217
9.6.14 Coroutine（協程）物件 . . . 217
9.6.15 情境變數物件（Context Variables Objects） . . . 217
9.6.16 DateTime 物件 . . . 219
9.6.17 型🄵提示物件 . . . 223

10 初始化、終結化與執行緒 225
10.1 Python 初始化之前 . . . 225
10.2 Global configuration variables . . . 226
10.3 Initializing and finalizing the interpreter . . . 229
10.4 Process-wide parameters . . . 232
10.5 Thread State and the Global Interpreter Lock . . . 235
10.5.1 Detaching the thread state from extension code . . . 236
10.5.2 Non-Python created threads . . . 236
10.5.3 Supporting subinterpreters in non-Python threads . . . 237
10.5.4 Cautions about fork() . . . 237
10.5.5 Cautions regarding runtime finalization . . . 238
10.5.6 高階 API . . . 238
10.5.7 低階 API . . . 241
10.6 Sub-interpreter support . . . 243
10.6.1 直譯器各🄵持有的 GIL . . . 246
10.6.2 Bugs and caveats . . . 246
10.7 Asynchronous Notifications . . . 246
10.8 Profiling and Tracing . . . 247
10.9 Reference tracing . . . 249
10.10 Advanced Debugger Support . . . 249
10.11 Thread Local Storage Support . . . 250
10.11.1 Thread Specific Storage (TSS) API . . . 250
10.11.2 執行緒局部儲存 (Thread Local Storage, TLS) API: . . . 251
10.12 Synchronization Primitives . . . 252
10.12.1 Python Critical Section API . . . 253

11 Python 初始化設定 **255**
11.1 PyInitConfig C API . . . 255
11.1.1 範例 . . . 255
11.1.2 建立設定 . . . 256
11.1.3 錯誤處理 . . . 256
11.1.4 Get Options . . . 257
11.1.5 Set Options . . . 257
11.1.6 模組 . . . 258
11.1.7 初始化 Python . . . 258
11.2 Configuration Options . . . 258
11.3 Runtime Python configuration API . . . 259
11.4 PyConfig C API . . . 261
11.4.1 範例 . . . 261
11.4.2 PyWideStringList . . . 262
11.4.3 PyStatus . . . 262
11.4.4 PyPreConfig . . . 263
11.4.5 Preinitialize Python with PyPreConfig . . . 265
11.4.6 PyConfig . . . 266
11.4.7 Initialization with PyConfig . . . 277
11.4.8 Isolated Configuration . . . 279
11.4.9 Python Configuration . . . 279
11.4.10 Python Path Configuration . . . 279
11.5 Py_GetArgcArgv() . . . 281
11.6 Delaying main module execution . . . 281

12 記憶體管理 **283**
12.1 總覽 . . . 283
12.2 Allocator Domains . . . 284
12.3 Raw Memory Interface . . . 284
12.4 記憶體介面 . . . 285
12.5 Object allocators . . . 286
12.6 Default Memory Allocators . . . 288
12.7 Customize Memory Allocators . . . 288
12.8 Debug hooks on the Python memory allocators . . . 290
12.9 The pymalloc allocator . . . 291
12.9.1 Customize pymalloc Arena Allocator . . . 291
12.10 The mimalloc allocator . . . 292
12.11 tracemalloc C API . . . 292
12.12 範例 . . . 292

13 物件實作支援 **295**
13.1 在 heap 上分配物件 . . . 295
13.2 物件生命🄴期 . . . 297
13.2.1 生命事件 . . . 297
13.2.2 Cyclic Isolate Destruction . . . 300
13.2.3 函式 . . . 301
13.3 通用物件結構 . . . 301
13.3.1 Base object types and macros . . . 301
13.3.2 實作函式與方法 . . . 303
13.3.3 Accessing attributes of extension types . . . 306
13.4 型🄴物件結構 . . . 310
13.4.1 Quick Reference . . . 311
13.4.2 PyTypeObject Definition . . . 315
13.4.3 PyObject Slots . . . 316
13.4.4 PyVarObject Slots . . . 317
13.4.5 PyTypeObject Slots . . . 317
13.4.6 Static Types . . . 340
13.4.7 Heap Types . . . 341

13.4.8 Number Object Structures . . . 341
13.4.9 Mapping Object Structures . . . 343
13.4.10 Sequence Object Structures . . . 343
13.4.11 Buffer Object Structures . . . 344
13.4.12 Async Object Structures . . . 345
13.4.13 Slot Type typedefs . . . 346
13.4.14 範例 . . . 347
13.5 循環垃圾回收的支援 . . . 350
13.5.1 Controlling the Garbage Collector State . . . 353
13.5.2 Querying Garbage Collector State . . . 353

14 API 和 ABI 版本管理 **355**
14.1 Build-time version constants . . . 355
14.2 Run-time version . . . 355
14.3 Bit-packing macros . . . 356

15 監控 C API **357**

16 Generating Execution Events **359**
16.1 Managing the Monitoring State . . . 360

A 術語表 **365**

B 關於這份🄵明文件 **383**
B.1 Python 文件的貢獻者們 . . . 383

C 沿革與授權 **385**
C.1 軟體沿革 . . . 385
C.2 關於存取或以其他方式使用 Python 的合約條款 . . . 386
C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2 . . . 386
C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0 . . . 387
C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1 . . . 387
C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2 . . . 388
C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION . . . 389
C.3 被收🄵軟體的授權與致謝 . . . 389
C.3.1 Mersenne Twister . . . 389
C.3.2 Sockets . . . 390
C.3.3 非同步 socket 服務 . . . 390
C.3.4 Cookie 管理 . . . 391
C.3.5 執行追🄵 . . . 391
C.3.6 UUencode 與 UUdecode 函式 . . . 392
C.3.7 XML 遠端程序呼叫 . . . 392
C.3.8 test_epoll . . . 393
C.3.9 Select kqueue . . . 393
C.3.10 SipHash24 . . . 394
C.3.11 strtod 與 dtoa . . . 394
C.3.12 OpenSSL . . . 394
C.3.13 expat . . . 397
C.3.14 libffi . . . 398
C.3.15 zlib . . . 398
C.3.16 cfuhash . . . 399
C.3.17 libmpdec . . . 399
C.3.18 W3C C14N 測試套件 . . . 400
C.3.19 mimalloc . . . 401
C.3.20 asyncio . . . 401
C.3.21 Global Unbounded Sequences (GUS) . . . 401
C.3.22 Zstandard bindings . . . 402

D 版權宣告 **403**

Bibliography 405

索引 407

對於想要編寫擴充模組或是嵌入 Python 的 C 和 C++ 程式設計師們，這份手🄴記🄴了可使用的 API（應用程式介面）。在 extending-index 中也有相關的🄵容，它描述了編寫擴充的一般原則，但🄵🄵有詳細🄴明 API 函式。

CHAPTER 1

簡介

對於 Python 的應用程式開發介面使得 C 和 C++ 開發者能🄵在各種層級存取 Python 直譯器。該 API 同樣可用於 C++，但🄵簡潔起見，通常將其稱🄵 Python/C API。使用 Python/C API 有兩個不同的原因，第一個是🄵特定目的來編寫擴充模組；這些是擴充 Python 直譯器的 C 模組，這可能是最常見的用法。第二個原因是在更大的應用程式中將 Python 作🄵零件使用；這種技術通常在應用程式中稱🄵 *embedding*（嵌入式）Python。

編寫擴充模組是一個相對容易理解的過程，其中「食譜 (cookbook)」方法很有效。有幾種工具可以在一定程度上自動化該過程，🄵管人們從早期就將 Python 嵌入到其他應用程式中，但嵌入 Python 的過程🄵不像編寫擴充那樣簡單。

不論你是嵌入還是擴充 Python，許多 API 函式都是很有用的；此外，大多數嵌入 Python 的應用程式也需要提供自定義擴充模組，因此在嘗試將 Python 嵌入實際應用程式之前熟悉編寫擴充可能是個好主意。

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 編寫標準

如果你正在編寫要引入於 CPython 中的 C 程式碼，你**必須**遵循 **PEP 7** 中定義的指南和標準。無論你貢獻的 Python 版本如何，這些指南都適用。對於你自己的第三方擴充模組，則不必遵循這些約定，除非你希望最終將它們貢獻給 Python。

1.3 引入檔案 (include files)

使用 Python/C API 所需的所有函式、型🄵和巨集的定義都透過以下這幾行來在你的程式碼中引入：

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

這意味著會引入以下標准標頭：`<stdio.h>`、`<string.h>`、`<errno.h>`、`<limits.h>`、`<assert.h>` 和 `<stdlib.h>`（如果可用）。

> **備🄵**
>
> 由於 Python 可能會定義一些會影響某些系統上標准標頭檔的預處理器 (pre-processor)，因此你必須在引入任何標准標頭檔之前引入 `Python.h`。
>
> 建議在引入 `Python.h` 之前都要定義 `PY_SSIZE_T_CLEAN`。有關此巨集的🄵明，請參🄵剖析引數與建置數值。

所有定義於 Python.h 中且使用者可見的名稱（另外透過標准標頭檔引入的除外）都具有 `Py` 或 `_Py` 前綴。以 `_Py` 開頭的名稱供 Python 實作🄵部使用，擴充編寫者不應使用。結構成員名稱🄵有保留前綴。

> **備🄵**
>
> 使用者程式碼不應定義任何以 `Py` 或 `_Py` 開頭的名稱。這會讓讀者感到困惑，🄵危及使用者程式碼在未來 Python 版本上的可移植性，這些版本可能會定義以這些前綴之一開頭的其他名稱。

標頭檔通常隨 Python 一起安裝。在 Unix 上它們位於目🄵 *prefix*`/include/pythonversion/` 和 *exec_prefix*`/include/pythonversion/`，其中 `prefix` 和 `exec_prefix` 由 Python 的 **`configure`** 🄵本的相應參數定義，*version* 是 `'%d.%d' % sys.version_info[:2]`。在 Windows 上，標頭安裝在 *prefix*`/include` 中，其中 `prefix` 是指定給安裝程式 (installer) 用的安裝目🄵。

要引入標頭，請將兩個（如果不同）目🄵放在編譯器的引入搜尋路徑 (search path) 中。不要將父目🄵放在搜尋路徑上，然後使用 `#include <pythonX.Y/Python.h>`；這會在多平台建置上壞掉，因🄵 `prefix` 下獨立於平台的標頭包括來自 `exec_prefix` 的平台特定標頭。

C++ 使用者應注意，🄵管 API 完全使用 C 來定義，但標頭檔適當地將入口點聲明🄵 `extern "C"`。因此，無需執行任何特殊操作即可使用 C++ 中的 API。

1.4 有用的巨集

Python 標頭檔中定義了幾個有用的巨集，大多被定義在它們有用的地方附近（例如*Py_RETURN_NONE*、*PyMODINIT_FUNC*），其他是更通用的工具程式。以下🄵不一定是完整的列表。

Py_ABS(x)

回傳 `x` 的🄵對值。

在 3.3 版被加入.

Py_ALWAYS_INLINE

要求編譯器總是嵌入🄵態行🄵函式 (static inline function)，編譯器可以忽略它🄵🄵定不嵌入該函式。

在禁用函式嵌入的除錯模式下建置 Python 時，它可用於嵌入有性能要求的🄵態行🄵函式。例如，MSC 在除錯模式下建置時禁用函式嵌入。

盲目地使用 Py_ALWAYS_INLINE 標記🄵態行🄵函式可能會導致更差的性能（例如程式碼大小增加）。在成本/收益分析方面，編譯器通常比開發人員更聰明。

如果 Python 是在除錯模式下建置（如果*Py_DEBUG* 巨集有被定義），*Py_ALWAYS_INLINE* 巨集就什🄵都不會做。

它必須在函式回傳型🄵之前被指定。用法：

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

在 3.11 版被加入.

Py_CHARMASK(c)

引數必須是 [-128, 127] 或 [0, 255] 範圍🄵的字元或整數。這個巨集會將 `c` 轉🄵🄵 `unsigned char` 🄵回傳。

Py_DEPRECATED(version)

將其用於已🄵用的聲明。巨集必須放在符號名稱之前。

範例：

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

在 3.8 版的變更: 新增了 MSVC 支援。

Py_GETENV(s)

類似於 `getenv(s)`，但如果在命令列上傳遞了 `-E` 則回傳 `NULL`（請見*PyConfig.use_environment*）。

Py_MAX(x, y)

回傳 `x` 和 `y` 之間的最大值。

在 3.3 版被加入.

Py_MEMBER_SIZE(type, member)

以位元組🄵單位回傳結構 (`type`) `member` 的大小。

在 3.6 版被加入.

Py_MIN(x, y)

回傳 `x` 和 `y` 之間的最小值。

在 3.3 版被加入.

Py_NO_INLINE

禁用函式的嵌入。例如，它🄵少了 C 堆🄵的消耗：對大量嵌入程式碼的 LTO+PGO 建置很有用（請參🄵 bpo-33720）。

用法：

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

在 3.11 版被加入.

Py_STRINGIFY(x)

將 `x` 轉🄵🄵 C 字串。例如 `Py_STRINGIFY(123)` 會回傳 `"123"`。

在 3.4 版被加入.

Py_UNREACHABLE()

當你的設計中有無法達到的程式碼路徑時，請使用此選項。例如在 `case` 語句已涵蓋了所有可能值的 `switch` 陳述式中的 `default:` 子句。在你可能想要呼叫 `assert(0)` 或 `abort()` 的地方使用它。

在發布模式 (release mode) 下，巨集幫助編譯器最佳化程式碼，🄵避免有關無法存取程式碼的警告。例如該巨集是在發布模式下於 GCC 使用 `__builtin_unreachable()` 來實作。

`Py_UNREACHABLE()` 的一個用途是，在對一個永不回傳但🄵未聲明🄵 `_Py_NO_RETURN` 的函式之呼叫後使用。

如果程式碼路徑是極不可能但在特殊情🄵下可以到達，則不得使用此巨集。例如在低記憶體條件下或系統呼叫回傳了超出預期範圍的值。在這種情🄵下，最好將錯誤回報給呼叫者。如果無法回報錯誤則可以使用*Py_FatalError()*。

在 3.7 版被加入.

Py_UNUSED(arg)

將此用於函式定義中未使用的參數以消除編譯器警告。例如：`int func(int a, int Py_UNUSED(b)) { return a; }`。

在 3.4 版被加入.

PyDoc_STRVAR(name, str)

建立一個名🄵 `name` 的變數，可以在文件字串中使用。如果 Python 是在🄵有文件字串的情🄵下建置，則該值將🄵空。

如 **PEP 7** 中所指明，使用*PyDoc_STRVAR* 作🄵文件字串可以支援在🄵有文件字串的情🄵下建置 Python。

範例：

```
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)

🄵給定的輸入字串建立一個文件字串，如果文件字串被禁用則建立空字串。

如 **PEP 7** 中所指明，使用*PyDoc_STR* 指定文件字串以支援在🄵有文件字串下建置 Python。

範例：

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

1.5 物件、型🄵和參照計數

大多數 Python/C API 函式都有一個或多個引數以及一個型🄵🄵 *PyObject** 的回傳值，此型🄵是一個指標，指向一個表示任意 Python 物件的晦暗 (opaque) 資料型🄵。由於在大多數情🄵下，Python 語言以相同的方式處理所有 Python 物件型🄵（例如賦值、作用域規則和引數傳遞），因此它們應該由單個 C 型🄵來表示。幾乎所有的 Python 物件都存在於堆積 (heap) 中：你永遠不會聲明*PyObject* 型🄵的自動變數或🄵態變數，只能聲明 *PyObject** 型🄵的指標變數。唯一的例外是型🄵物件；由於它們🄵不能被釋放，因此它們通常是🄵態*PyTypeObject* 物件。

所有 Python 物件（甚至是 Python 整數）都有一個型🄵 (*type*) 和一個參照計數 (*reference count*)。一個物件的型🄵🄵定了它是什🄵種類的物件（例如一個整數、一個 list 或一個使用者定義的函式；還有更多型🄵，請見 types）。對於每個🄵所周知的型🄵，都有一個巨集來檢查物件是否屬於該型🄵；例如，若（且唯若）*a* 指向的物件是 Python list 時，`PyList_Check(a)` 🄵真。

1.5.1 參照計數

參照計數很重要，因🄵現今的電腦記憶體大小是有限的（而且通常是非常有限的）；它計算有多少個不同的地方用有了一個物件的🄵參照。這樣的地方可以是另一個物件，或者全域（或🄵態）C 變數，或者某個 C 函式中的本地變數。當一個物件的最後一個🄵參照被釋放時（即其的參照計數變🄵零），該物件將被解除配置 (deallocated)。如果它包含對其他物件的參照，則它們的參照會被釋放。如果這樣的釋放使得再也🄵有任何對於它們的參照，則可以依次🄵那些其他物件解除配置，依此類推。（此處相互參照物件的存在是個明顯的問題；目前，解🄵方案是「就不要那樣做」。）

參照計數總是被明確地操作。正常的方法是使用巨集*Py_INCREF()* 來取得對於物件的參照（即參照計數加一），🄵使用巨集*Py_DECREF()* 來釋放參照（即將參照計數🄵一）。*Py_DECREF()* 巨集比 incref 巨集🄵雜得多，因🄵它必須檢查參照計數是否變🄵零，然後呼叫物件的釋放器 (deallocator)。釋放器是包含在物件型🄵結構中的函式指標。特定型🄵的釋放器，在如果是一個🄵合物件型🄵（例如 list）時負責釋放物件中包含的其他物件的參照，🄵執行任何需要的額外完結步驟。參照計數不可能溢出；至少與🄵擬記憶體中用來保存參照計數的不同記憶體位置數量一樣多的位元會被使用（假設 `sizeof(Py_ssize_t) >= sizeof(void*)`）。因此參照計數增加是一個簡單的操作。

🄵有必要🄵每個包含物件指標的本地變數物件都持有一個🄵參照（即增加參照計數）。理論上，當變數指向它時，物件的參照計數會增加 1，而當變數離開作用域時就會🄵少 1。然而這兩者會相互抵消，所以最後參照計數🄵有改變。使用參照計數的唯一真正原因是防止物件還有變數指向它時被解除配置。如果我們知道至少有一個物件的其他參照生存了至少與我們的變數一樣久，就不需要臨時增加建立新的🄵參照（即增加參照計數）。出現這種情🄵的一個重要情🄵是在從 Python 呼叫的擴充模組中作🄵引數傳遞給 C 函式的物件；呼叫機制保證在呼叫期間保持對每個參數的參照。

然而，一個常見的陷阱是從一個 list 中提取一個物件🄵保留它一段時間而不取得其參照。某些其他操作可能會從列表中🄵除該物件，🄵少其參照計數🄵可能取消分配它。真正的危險是看似無害的操作可能會叫用可以執行此操作的任意 Python 程式碼；有一個程式碼路徑允許控制權從*Py_DECREF()* 回歸使用者，因此幾乎任何操作都有🄵在危險。

一種安全的方法是都使用通用 (generics) 操作（名稱以 `PyObject_`、`PyNumber_`、`PySequence_` 或 `PyMapping_` 開頭的函式）。這些操作總是建立新的對於它們回傳物件的🄵參照（即增加其參照計數）。這讓呼叫者有責任在處理完結果後呼叫*Py_DECREF()*；這就成🄵第二本質。

參照計數詳細資訊

Python/C API 中函式的參照計數行🄵最好用參照的所有權來解釋。所有權附屬於參照而非物件（物件🄵非被擁有，它們總是共享的）。「擁有參照」意味著當不再需要該參照時，負責在其上呼叫 Py_DECREF。所有權也可以轉移，這意味著接收參照所有權的程式碼最終會負責在不需要參照時透過呼叫*Py_DECREF()* 或*Py_XDECREF()* 釋放參照 --- 或者將這個責任再傳遞出去（通常是給它的呼叫者）。當一個函式將參照的所有權傳遞給它的呼叫者時，呼叫者被稱🄵接收到一個新參照。當🄵有所有權轉移時，呼叫者被稱🄵借用參照。如果是借用參照就不需要做任何事情。

相反地，當呼叫的函式傳入物件的參照時，有兩種可能性：函式有竊取 *(steal)* 物件的參照，或者🄵有。竊取參照意味著當你將參照傳遞給函式時，該函式假定它現在擁有該參照，🄵且你不再對它負責。

很少有函式會竊取參照；兩個值得注意的例外是*PyList_SetItem()* 和*PyTuple_SetItem()*，它們竊取了對項目的參照（但不是對項目所在的 tuple 或 list 的參照！）。因🄵有著使用新建立的物件來增加 (populate) tuple 或 list 的習慣，這些函式旨在竊取參照；例如，建立 tuple `(1, 2, "three")` 的程式碼可以如下所示（先暫時忘記錯誤處理；更好的編寫方式如下所示）：

```
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"));
```

這🄵*PyLong_FromLong()* 會回傳一個新的參照，它立即被*PyTuple_SetItem()* 竊取。如果你想繼續使用一個物件，🄵管對它的參照將被竊取，請在呼叫參照竊取函式之前使用*Py_INCREF()* 來取得另一個參照。

附帶地🄵，*PyTuple_SetItem()* 是設定 tuple 項目的唯一方法；*PySequence_SetItem()* 和*PyObject_SetItem()* 拒🄵這樣做，因🄵 tuple 是一種不可變 (immutable) 的資料型🄵。你應該只對你自己建立的 tuple 使用*PyTuple_SetItem()*。

可以使用*PyList_New()* 和*PyList_SetItem()* 編寫用於填充列表的等效程式碼。

但是在實際操作中你很少會使用這些方法來建立和增加 tuple 和 list。有一個通用函式*Py_BuildValue()* 可以從 C 值建立最常見的物件，由 *format string* 引導。例如上面的兩個程式碼可以用以下程式碼替🄵（它還負責了錯誤檢查）：

```
PyObject *tuple, *list;

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

更常見的是以那些借用參照的項目來使用*PyObject_SetItem()* 及其系列函式，比如傳遞給你正在編寫的函式的引數。在那種情🄵下，他們關於參照的行🄵會比較穩健，因🄵你不取得新的一個參照就可以放🄵參照（「讓它被竊取」）。例如，此函式將 list（實際上是任何可變序列）的所有項目設定於給定項目：

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

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

函式回傳值的情🄵略有不同。雖然傳遞對大多數函式的參照不會改變你對該參照的所有權責任，但許多回傳物件參照的函式會給你該參照的所有權。原因很簡單：在很多情🄵下，回傳的物件是即時建立的，你獲得的參照是對該物件的唯一參照。因此回傳物件參照的通用函式，如*PyObject_GetItem()*和*PySequence_GetItem()*，總是回傳一個新的參照（呼叫者成🄵參照的所有者）。

重要的是要意識到你是否擁有一個函式回傳的參照只取🄵於你呼叫哪個函式 --- 羽毛 *(plumage)**（作🄵引數傳遞給函式的物件之型🄵）*不會進入它！因此，如果你使用*PyList_GetItem()* 從 list 中提取一個項目，你不會擁有其參照 --- 但如果你使用*PySequence_GetItem()* 從同一 list 中取得相同的項目（且恰好使用完全相同的引數），你確實會擁有對回傳物件的參照。

以下是一個範例，🄵明如何編寫函式來計算一個整數 list 中項目的總和；一次使用*PyList_GetItem()*，一次使用*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;
```

(繼續下一頁)

(繼續上一頁)

```
    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 型🄵

有少數幾個其他的資料型🄵在 Python/C API 中發揮重要作用；大多數是簡單的 C 型🄵，例如 `int`、`long`、`double` 和 `char*`。一些結構型🄵被用於描述用於列出模組所匯出的函式或新物件型🄵的資料屬性的🄵態表，其他則用於描述🄵數的值。這些將與使用它們的函式一起討論。

type **Py_ssize_t**

> 🄵穩定 ABI 的一部分. 一個帶符號的整數型🄵，使得 `sizeof(Py_ssize_t) == sizeof(size_t)`。C99 🄵有直接定義這樣的東西（size_t 是無符號整數型🄵）。有關詳細資訊，請參🄵 **PEP 353**。`PY_SSIZE_T_MAX` 是*Py_ssize_t* 型🄵的最大正值。

1.6 例外

如果需要特定的錯誤處理，Python 開發者就只需要處理例外；未處理的例外會自動傳遞給呼叫者，然後傳遞給呼叫者的呼叫者，依此類推，直到它們到達頂層直譯器，在那🄵它們透過堆🄵回溯 (stack trace) 回報給使用者。

然而，對於 C 開發者來🄵，錯誤檢查總是必須是顯式的。除非在函式的文件中另有明確聲明，否則 Python/C API 中的所有函式都可以引發例外。通常當一個函式遇到錯誤時，它會設定一個例外，🄵🄵它擁有的任何物件參照，🄵回傳一個錯誤指示器。如果🄵有另外文件記🄵，這個指示器要🄵是 `NULL` 不然就是 `-1`，取🄵於函式的回傳型🄵。有些函式會回傳布林值 true/false 結果，false 表示錯誤。很少有函式不回傳明確的錯誤指示器或者有不明確的回傳值，而需要使用*PyErr_Occurred()* 明確測試錯誤。這些例外都會被明確地記🄵於文件。

例外的狀態會在個🄵執行緒的儲存空間 (per-thread storage) 中維護（這相當於在非執行緒應用程式中使用全域儲存空間）。執行緒可以處於兩種狀態之一：發生例外或未發生例外。函式*PyErr_Occurred()* 可用於檢查這一點：當例外發生時，它回傳對例外型🄵物件的借用參照，否則回傳 `NULL`。設定例外狀態的函式有很多：*PyErr_SetString()* 是最常見的（🄵管不是最通用的）設定例外狀態的函式，而*PyErr_Clear()* 是用來清除例外狀態。

完整的例外狀態由三個（都可以🄵 `NULL` 的）物件組成：例外型🄵、對應的例外值和回溯。這些與 `sys.exc_info()` 的 Python 結果具有相同的含義；但是它們🄵不相同：Python 物件表示由 Python `try ... except` 陳述式處理的最後一個例外，而 C 層級的例外狀態僅在例外在 C 函式間傳遞時存在，直到它到達 Python 位元組碼直譯器的主🄵圈，該🄵圈負責將它傳遞給 `sys.exc_info()` 和其系列函式。

請注意，從 Python 1.5 開始，從 Python 程式碼存取例外狀態的首選且支援執行緒安全的方法是呼叫 `sys.exc_info()` 函式，它回傳 Python 程式碼的個🄵執行緒例外狀態。此外，兩種存取例外狀態方法的

語義都發生了變化，因此捕獲例外的函式將保存和恢復其執行緒的例外狀態，從而保留其呼叫者的例外狀態。這可以防止例外處理程式碼中的常見錯誤，這些錯誤是由看似無辜的函式覆蓋了正在處理的例外而引起的；它還替回溯中被堆🄵框 (stack frame) 參照的物件🄵少了通常不需要的生命🄵期延長。

作🄵一般原則，呼叫另一個函式來執行某些任務的函式應該檢查被呼叫函式是否引發了例外，如果是，則將例外狀態傳遞給它的呼叫者。它應該🄵🄵它擁有的任何物件參照，🄵回傳一個錯誤指示符，但它不應該設定另一個例外 --- 這將覆蓋剛剛引發的例外，🄵🄵失關於錯誤確切原因的重要資訊。

上面的 `sum_sequence()` 範例展示了一個檢測例外🄴將其繼續傳遞的例子。碰巧這個例子在檢測到錯誤時不需要清理任何擁有的參照。以下範例函式展示了一些錯誤清理。首先，🄵了提醒你🄵什🄵喜歡 Python，我們展示了等效的 Python 程式碼：

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

這是相應的 C 程式碼：

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

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

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

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

    if (PyObject_SetItem(dict, key, incremented_item) < 0)
        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 */
}
```

這個例子代表了在 C 語言中對使用 goto 陳述句的認同！它闡述了以*PyErr_ExceptionMatches()* 和*PyErr_Clear()* 來處理特定的例外，以及以*Py_XDECREF()* 來配置其所擁有且可能🄵 NULL 的參照（注意名稱中的 'X'； *Py_DECREF()* 在遇到 NULL 參照時會崩潰）。重要的是，用於保存擁有的參照的變數被初始化🄵 NULL 以使其能🄵順利作用；同樣地，回傳值被初始化🄵 -1（失敗），🄵且僅在最後一次呼叫成功後才設定🄵成功。

1.7 嵌入式 Python

只有 Python 直譯器的嵌入者（而不是擴充編寫者）需要擔心的一項重要任務是 Python 直譯器的初始化與完成階段。直譯器的大部分功能只能在直譯器初始化後使用。

基本的初始化函式是*Py_Initialize()*。這會初始化帶有載入模組的表，🄵建立基礎模組 builtins、__main__ 和 sys。它還會初始化模組搜索路徑 (sys.path)。

Py_Initialize() 不設定「🄵本引數列表 (script argument list)」(sys.argv)。如果稍後將要執行的 Python 程式碼需要此變數，則必須設定*PyConfig.argv* 和*PyConfig.parse_argv*，請見*Python 初始化配置*。

在大多數系統上（特🄵是在 Unix 和 Windows 上，🄵管細節略有不同），*Py_Initialize()* 會假設 Python 函式庫相對於 Python 直譯器可執行檔案的位置固定，🄵根據其對標准 Python 直譯器可執行檔案位置的最佳猜測來計算模組搜尋路徑。或者更詳細地🄵，它會在 shell 命令搜尋路徑（環境變數 PATH）中找到名🄵 python 的可執行檔案，🄵在其父目🄵中查找一個名🄵 lib/python*X.Y* 的目🄵的相對位置。

例如，如果在 /usr/local/bin/python 中找到 Python 可執行檔案，它將假定函式庫位於 /usr/local/lib/python*X.Y* 中。（事實上這個特定的路徑也是「後備 (fallback)」位置，當在 PATH 中找不到名🄵 python 的可執行檔案時使用。）使用者可以透過設定環境變數來覆蓋此行🄵 PYTHONHOME，或者透過設定 PYTHONPATH 在標准路徑前面插入額外的目🄵。

嵌入的應用程式可以透過在呼叫*Py_InitializeFromConfig()* 之前設定*PyConfig.program_name* 來引導搜尋。請注意 PYTHONHOME 仍然覆蓋它🄵且 PYTHONPATH 仍然插入在標准路徑的前面。需要完全控制權的應用程式必須實作自己的*Py_GetPath()*、*Py_GetPrefix()*、*Py_GetExecPrefix()* 和*Py_GetProgramFullPath()*（全部定義在 Modules/getpath.c）。

有時會希望能🄵「取消初始化 (uninitialize)」Python。例如，應用程式可能想要重新開始（再次呼叫*Py_Initialize()*）或者應用程式簡單地完成了對 Python 的使用🄵想要釋放 Python 分配的記憶體。這可以透過呼叫*Py_FinalizeEx()* 來完成。如果 Python 目前處於初始化狀態，函式*Py_IsInitialized()* 會回傳 true。有關這些功能的更多資訊將在後面的章節中給出。請注意*Py_FinalizeEx()* 不會釋放由 Python 直譯器分配的所有記憶體，例如目前無法釋放被擴充模組所分配的記憶體。

1.8 除錯建置

Python 可以在建置時使用多個巨集來🄵用對直譯器和擴充模組的額外檢查，這些檢查往往會在執行環境 (runtime) 增加大量開銷 (overhead)，因此預設情🄵下不🄵用它們。

Python 原始碼發🄵版本中的 Misc/SpecialBuilds.txt 檔案有一份包含多種除錯構置的完整列表，🄵支援追🄵參照計數、🄵記憶體分配器除錯或對主直譯器🄵圈進行低階分析的建置。本節的其余部分將僅描述最常用的建置。

Py_DEBUG

使用定義的 Py_DEBUG 巨集編譯直譯器會生成 Python 的除錯建置。Py_DEBUG 在 Unix 建置中要透過在 ./configure 命令中加入 --with-pydebug 來🄵用。非 Python 限定的 _DEBUG 巨集的存在也暗示了這一點。當 Py_DEBUG 在 Unix 建置中🄵用時，編譯器最佳化會被禁用。

除了下面描述的參照計數除錯之外，還會執行額外的檢查，請參🄵 Python 除錯建置。

定義 Py_TRACE_REFS 來🄵用參照追🄵（參見設定 --with-trace-refs 選項）。當有定義時，透過向每個*PyObject* 新增兩個額外欄位來維護有效物件的循環雙向🄵表 (circular doubly linked list)。全體分配也有被追🄵。退出時將印出所有現行參照。（在交互模式下，這發生在直譯器運行的每個陳述句之後。）

有關更多詳細資訊，請參🄵 Python 原始碼發布版中的 Misc/SpecialBuilds.txt。

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 versions 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.

也參考

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.

CHAPTER 2

C API 穩定性

除非有另外記🄵於文件，Python 的 C API 被包含在向後相容性策略 **PEP 387** 中。大多數改動都是相容於原始碼的（通常只會增加新的 API）。更改現有 API 或🄵除 API 僅在🄵用期後或修復嚴重問題時進行。

CPython 的應用程式二進位介面 (Application Binary Interface, ABI) 在次要版本中是向前和向後相容的（如果它們以相同的方式編譯；請參🄵下面的*平台注意事項*）。因此，🄵 Python 3.10.0 編譯的程式碼將能🄵在 3.10.8 上運行，反之亦然，但 3.9.x 和 3.11.x 就需要分🄴編譯。

C API 有兩層級，有不同的穩定性期望：

- *不穩定 API*，可能會在次要版本中發生變化，而🄵有🄴用階段。會在名稱中以 `PyUnstable` 前綴來標記。
- *受限 API*，在多個次要版本之間相容。當有定義*`Py_LIMITED_API`* 時，只有這個子集會從 `Python.h` 公開。

下面將更詳細地討論這些🄵容。

帶有底🄵前綴的名稱是私有 API (private API)，像是 `_Py_InternalState`，即使在補丁版本 (patch release) 中也可能被更改，不會另行通知。如果你需要使用這個 API，可以聯🄴 CPython 開發者 🄵針對你的使用方法來討論是否新增公開的 API。

2.1 不穩定的 C API

任何以 `PyUnstable` 前綴命名的 API 都會公開 CPython 實作細節，🄵可能在每個次要版本中進行更改（例如從 3.9 到 3.10），而不會出現任何🄵用警告。但是它不會在錯誤修復發布版本中發生變化（例如從 3.10.0 到 3.10.1）。

它通常用於專門的低階工具，例如偵錯器。

使用此 API 的專案應該要遵循 CPython 開發細節，🄵花費額外的力氣來針對這些變動來做調整。

2.2 穩定的應用程式二進位介面

🄵簡單起見，本文件討論*擴充 (extension)*，但受限 API 和穩定 ABI 在所有 API 使用方式中都以相同的方式運作 -- 例如在嵌入式 Python (embedding Python) 中。

2.2.1 受限 C API

Python 3.2 引入了受限 *API (Limited API)*，它是 Python C API 的一個子集。僅使用受限 API 的擴充可以只編譯一次就被載入於多個版本的 Python。受限 API 的🄵容列在下方。

Py_LIMITED_API

在包含 `Python.h` 之前定義此巨集以選擇只使用受限 API，🄵挑選受限 API 版本。

將 `Py_LIMITED_API` 定義🄵對應於你的擴充有支援的最低 Python 版本的 *PY_VERSION_HEX* 值。該擴充與從指定版本開始的所有 Python 3 版本之 ABI 相容，🄵且可以使用過去版本有引入的受限 API。

與其直接使用 `PY_VERSION_HEX` 巨集，不如寫死 (hardcode) 最小次要版本（例如代表 Python 3.10 的 `0x030A0000`），以便在使用未來的 Python 版本進行編譯時仍保持穩定性。

你還可以將 `Py_LIMITED_API` 定義🄵 `3`，這與 `0x03020000`（Python 3.2，引入了受限 API 的版本）相同。

2.2.2 穩定 ABI

🄵了實現它，Python 提供了一個穩定 *ABI (Stable ABI)*：一組將在各個 Python 3.x 版本之間保持 ABI 相容的符號。

備🄵

穩定 ABI 可以防止 ABI 問題，例如由於結構布局或函式簽名變化導致缺少符號或數據損壞的鏈接器 (linker) 錯誤。然而，Python 的其他變化可能會改變 行🄵的擴充。有關詳細信息，請參🄵 Python 的向後相容性政策 (**PEP 387**)。

穩定 ABI 被包含在受限 *API* 中開放的符號，但也包含其他符號 - 例如，支援舊版受限 API 所必需的函式。

在 Windows 上，使用穩定 ABI 的擴充應該連接到 `python3.dll` 而不是特定版本的函式庫，例如 `python39.dll`。

在某些平台上，Python 將查找🄵加載以 `abi3` 標🄵命名的共享函式庫檔案（例如 `mymodule.abi3.so`）。它不檢查此類擴充是否符合穩定的 ABI。確保的責任在使用者（或者打包工具）身上，例如使用 3.10+ 受限 API 建置的擴充不會🄵較低版本的 Python 所安裝。

穩定 ABI 中的所有函式都作🄵函式存在於 Python 的共享函式庫中，而不僅是作🄵巨集。這使得它們可被用於不使用 C 預處理器 (preprocessor) 的語言。

2.2.3 受限 API 范圍和性能

受限 API 的目標是允許使用完整的 C API 進行所有可能的操作，但可能會降低性能。

例如，雖然 *PyList_GetItem()* 可用，但它的「不安全」巨集變體 *PyList_GET_ITEM()* 🄵不可用。巨集運行可以更快，因🄵它可以依賴 list 物件的特定版本實作細節。

如果🄵有定義 `Py_LIMITED_API`，一些 C API 函式將被嵌入或被替🄵🄵巨集。定義 `Py_LIMITED_API` 會禁用嵌入，從而隨著 Python 資料結構的改進而提高穩定性，但可能會降低性能。

透過省略 `Py_LIMITED_API` 定義，可以使用特定版本的 ABI 編譯受限 API 擴充。這可以提高該 Python 版本的性能，但會限制相容性。使用 `Py_LIMITED_API` 編譯將🄵生一個擴充，可以在特定版本的擴充不可用的地方發布—例如，用於即將發布的 Python 版本的預發布版本 (prerelease)。

2.2.4 受限 API 注意事項

請注意，使用 `Py_LIMITED_API` 進行編譯不完全保證程式碼符合受限 *API* 或穩定 *ABI*。`Py_LIMITED_API` 僅涵蓋定義，但 API 還包括其他議題，例如預期的語義 (semantic)。

`Py_LIMITED_API` 無法防範的一個問題是使用在較低 Python 版本中無效的引數來呼叫函式。例如一個開始接受 `NULL` 作🄵引數的函式。在 Python 3.9 中，`NULL` 現在代表選擇預設行🄵，但在 Python 3.8 中，引數將被直接使用，導致 `NULL` 取消參照 (dereference) 且崩潰 (crash)。類似的引數適用於結構 (struct) 的欄位。

另一個問題是，當有定義 `Py_LIMITED_API` 時，一些結構欄位目前不會被隱藏，即使它們是受限 API 的一部分。

出於這些原因，我們建議要以它支援的所有次要 Python 版本來測試擴充，🄵且最好使用最低版本進行建置。

我們也建議要查看所有使用過的 API 的文件，檢查它是否明確屬於受限 API。即使有定義 `Py_LIMITED_API`，一些私有聲明也會因🄵技術原因（或者甚至是無意地，例如臭蟲）而被公開出來。

另請注意，受限 API 不一定是穩定的：在 Python 3.8 中使用 `Py_LIMITED_API` 進行編譯意味著擴充將能以 Python 3.12 運行，但不一定能以 Python 3.12 編譯。特🄵是如果穩定 ABI 保持穩定，部分受限 API 可能會被🄵用和🄵除。

2.3 平台注意事項

ABI 穩定性不僅取🄵於 Python，還取🄵於使用的編譯器、低階函式庫和編譯器選項。出於*穩定 ABI* 的目的，這些細節定義了一個「平台」。它們通常取🄵於作業系統種類和處理器架構

每個特定的 Python 發布者都有責任確保特定平台上的所有 Python 版本都以不破壞穩定 ABI 的方式建置。`python.org` 和許多第三方發布者發布的 Windows 和 macOS 版本就是這種情🄵。

2.4 受限 API 的🄵容

目前，*受限 API* 包括以下項目：

- *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*

CHAPTER 3

The Very High Level Layer

The functions in this chapter will let you execute Python source code given in a file or a buffer, but they will not let you interact in a more detailed way with the interpreter.

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)

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving *closeit* set to `0` and *flags* set to `NULL`.

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

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving the *closeit* argument set to `0`.

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

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving the *flags* argument set to `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)

Executes the Python source code from *command* in the `__main__` module according to the *flags* argument. If `__main__` does not already exist, it is created. Returns `0` on success or `-1` if an exception was raised. If there was an error, there is no way to get the exception information. For the meaning of *flags*, see below.

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)

This is a simplified interface to *`PyRun_SimpleFileExFlags()`* below, leaving *closeit* set to `0` and *flags* set to `NULL`.

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

This is a simplified interface to *`PyRun_SimpleFileExFlags()`* below, leaving *flags* set to `NULL`.

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

Similar to *`PyRun_SimpleStringFlags()`*, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file, it is decoded from *filesystem encoding and error handler*. If *closeit* is true, the file is closed before `PyRun_SimpleFileExFlags()` returns.

> **備🄵**
>
> On Windows, *fp* should be opened as binary mode (e.g. `fopen(filename, "rb")`). Otherwise, Python may not handle script file with LF line ending correctly.

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

This is a simplified interface to *`PyRun_InteractiveOneFlags()`* below, leaving *flags* set to `NULL`.

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

Read and execute a single statement from a file associated with an interactive device according to the *flags* argument. The user will be prompted using `sys.ps1` and `sys.ps2`. *filename* is decoded from the *filesystem encoding and error handler*.

Returns `0` when the input was executed successfully, `-1` if there was an exception, or an error code from the `errcode.h` include file distributed as part of Python if there was a parse error. (Note that `errcode.h` is not included by `Python.h`, so must be included specifically if needed.)

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

This is a simplified interface to *`PyRun_InteractiveLoopFlags()`* below, leaving *flags* set to `NULL`.

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

Read and execute statements from a file associated with an interactive device until EOF is reached. The user will be prompted using `sys.ps1` and `sys.ps2`. *filename* is decoded from the *filesystem encoding and error handler*. Returns `0` at EOF or a negative number upon failure.

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

🄵穩定 ABI 的一部分. Can be set to point to a function with the prototype `int func(void)`. The function will be called when Python's interpreter prompt is about to become idle and wait for user input from the terminal. The return value is ignored. Overriding this hook can be used to integrate the interpreter's prompt with other event loops, as done in the `Modules/_tkinter.c` in the Python source code.

在 3.12 版的變更: This function is only called from the *main interpreter*.

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

Can be set to point to a function with the prototype `char *func(FILE *stdin, FILE *stdout, char *prompt)`, overriding the default function used to read a single line of input at the interpreter's prompt. The function is expected to output the string *prompt* if it's not `NULL`, and then read a line of input from the provided standard input file, returning the resulting string. For example, The `readline` module sets this hook to provide line-editing and tab-completion features.

The result must be a string allocated by *`PyMem_RawMalloc()`* or *`PyMem_RawRealloc()`*, or `NULL` if an error occurred.

在 3.4 版的變更: The result must be allocated by *`PyMem_RawMalloc()`* or *`PyMem_RawRealloc()`*, instead of being allocated by *`PyMem_Malloc()`* or *`PyMem_Realloc()`*.

在 3.12 版的變更: This function is only called from the *main interpreter*.

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

回傳值：新的參照。 This is a simplified interface to `PyRun_StringFlags()` below, leaving *flags* set to `NULL`.

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

回傳值：新的參照。 Execute Python source code from *str* in the context specified by the objects *globals* and *locals* with the compiler flags specified by *flags*. *globals* must be a dictionary; *locals* can be any object that implements the mapping protocol. The parameter *start* specifies the start token that should be used to parse the source code.

Returns the result of executing the code as a Python object, or `NULL` if an exception was raised.

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

回傳值：新的參照。 This is a simplified interface to `PyRun_FileExFlags()` below, leaving *closeit* set to `0` and *flags* set to `NULL`.

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

回傳值：新的參照。 This is a simplified interface to `PyRun_FileExFlags()` below, leaving *flags* set to `NULL`.

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

回傳值：新的參照。 This is a simplified interface to `PyRun_FileExFlags()` below, leaving *closeit* set to `0`.

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

回傳值：新的參照。Similar to `PyRun_StringFlags()`, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file, it is decoded from the *filesystem encoding and error handler*. If *closeit* is true, the file is closed before `PyRun_FileExFlags()` returns.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. This is a simplified interface to `Py_CompileStringFlags()` below, leaving *flags* set to `NULL`.

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

回傳值：新的參照。 This is a simplified interface to `Py_CompileStringExFlags()` below, with *optimize* set to `-1`.

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

回傳值：新的參照。 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.

The integer *optimize* specifies the optimization level of the compiler; a value of `-1` selects the optimization level of the interpreter as given by `-O` options. Explicit levels are `0` (no optimization; `__debug__` is true), `1` (asserts are removed, `__debug__` is false) or `2` (docstrings are removed too).

在 3.4 版被加入.

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

回傳值：新的參照。 Like `Py_CompileStringObject()`, but *filename* is a byte string decoded from the *filesystem encoding and error handler*.

在 3.2 版被加入.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. This is a simplified interface to *PyEval_EvalCodeEx()*, with just the code object, and global and local variables. The other arguments are set to 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Evaluate a precompiled code object, given a particular environment for its evaluation. This environment consists of a dictionary of global variables, a mapping object of local variables, arrays of arguments, keywords and defaults, a dictionary of default values for *keyword-only* arguments and a closure tuple of cells.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. Evaluate an execution frame. This is a simplified interface to *PyEval_EvalFrameEx()*, for backward compatibility.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. This is the main, unvarnished function of Python interpretation. The code object associated with the execution frame *f* is executed, interpreting bytecode and executing calls as needed. The additional *throwflag* parameter can mostly be ignored - if true, then it causes an exception to immediately be thrown; this is used for the `throw()` methods of generator objects.

在 3.4 版的變更: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

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

This function changes the flags of the current evaluation frame, and returns true on success, false on failure.

int **Py_eval_input**

The start symbol from the Python grammar for isolated expressions; for use with *Py_CompileString()*.

int **Py_file_input**

The start symbol from the Python grammar for sequences of statements as read from a file or other source; for use with *Py_CompileString()*. This is the symbol to use when compiling arbitrarily long Python source code.

int **Py_single_input**

The start symbol from the Python grammar for a single statement; for use with *Py_CompileString()*. This is the symbol used for the interactive interpreter loop.

struct **PyCompilerFlags**

This is the structure used to hold compiler flags. In cases where code is only being compiled, it is passed as `int flags`, and in cases where code is being executed, it is passed as `PyCompilerFlags *flags`. In this case, `from __future__ import` can modify *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**

Compiler flags.

int **cf_feature_version**

cf_feature_version is the minor Python version. It should be initialized to `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*.

在 3.8 版的變更: 新增 *cf_feature_version* 欄位。

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 程式碼物件旗標 for a complete list.

CHAPTER 4

參照計數

本節中的函式與巨集用於管理 Python 物件的參照計數。

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

🄵穩定 ABI 的一部分 自 *3.14* 版本開始. 取得物件 *o* 的參照計數。

請注意，回傳的值可能實際上🄵不反映實際保存了多少對該物件的參照。例如，某些物件是「不滅的 *(immortal)*」，🄵且具有非常高的參照計數，不能反映實際的參照數量。因此，除了 0 或 1 以外，不要依賴回傳值的准確性。

使用`Py_SET_REFCNT()` 函式設定物件參照計數。

> **備🄵**
>
> 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()`.

在 3.10 版的變更: `Py_REFCNT()` 更改🄵行🄵🄵態函式 (inline static function)。

在 3.11 版的變更: 參數型🄵不再是 `const PyObject*`。

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

設定物件 *o* 的參照計數。

在🄵用自由執行緒的 Python 建置中，如果 *refcnt* 大於 `UINT32_MAX`，則該物件會被設🄵不滅的 *(immortal)*。

請注意，此函式對不滅的物件🄵有影響。

在 3.9 版被加入.

在 3.12 版的變更: 不滅的物件不會被修改。

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

代表取得對於物件 *o* 的新🄵參照，即它正在使用且不應被銷🄵。

請注意，此函式對不滅的物件🄵有影響。

此函式通常用於將借用參照原地 (in-place) 轉🄵🄵🄵參照。`Py_NewRef()` 函式可用於建立新的🄵參照。

使用完該物件後，透過呼叫*Py_DECREF()* 來釋放它。

該物件不能🄵 NULL；如果你不確定它不是 NULL，請使用*Py_XINCREF()*。

不要期望此函式會以任何方式實際修改 *o*，至少對於**某些物件**來🄵，此函式🄵有任何效果。

在 3.12 版的變更: 不滅的物件不會被修改。

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

與*Py_INCREF()* 類似，但物件 *o* 可以🄵 NULL，在這種情🄵下這就不會有任何效果。

另請見*Py_XNewRef()*。

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

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 建立對物件的新🄵參照：於 *o* 呼叫*Py_INCREF()* 🄵回傳物件 *o*。

當不再需要🄵參照時，應對其呼叫*Py_DECREF()* 以釋放該參照。

物件 *o* 不能🄵 NULL；如果 *o* 可以🄵 NULL，則使用*Py_XNewRef()*。

舉例來🄵:

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

可以寫成:

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

另請參🄵*Py_INCREF()*。

在 3.10 版被加入.

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

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 與*Py_NewRef()* 類似，但物件 *o* 可以🄵 NULL。

如果物件 *o* 🄵 NULL，則該函式僅回傳 NULL。

在 3.10 版被加入.

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

釋放一個對物件 *o* 的🄵參照，代表該參照不會再被使用。

請注意，此函式對不滅的物件🄵有影響。

如果最後一個🄵參照被釋放(即物件的參照計數達到零)，則觸發物件之型🄵的釋放函式 (deallocation function)（不得🄵 NULL)。

此函式通常用於在退出作用域之前🄵除🄵參照。

該物件不能🄵 NULL；如果你不確定它不是 NULL，請改用*Py_XDECREF()*。

不要期望此函式會以任何方式實際修改 *o*，至少對於**某些物件**來🄵，此函式🄵有任何效果。

警告

釋放函式可以導致任意 Python 程式碼被叫用（例如，當釋放具有 `__del__()` 方法的類🄵實例時)。雖然此類程式碼中的例外不會被傳遞出來，但 執行的程式碼可以自由存取所有 Python 全域變數。這意味著在叫用*Py_DECREF()* 之前，可從全域變數存取的任何物件都應處於一致狀態。例如，從 list 中🄵除物件的程式碼應將已🄵除物件的參照🄵🄵到臨時變數中，更新 list 資料結構，然後🄵臨時變數呼叫*Py_DECREF()*。

在 3.12 版的變更: 不滅的物件不會被修改。

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

和*Py_DECREF()* 類似，但該物件可以是 NULL，在這種情🄵下巨集不起作用。在這🄵也會出現與*Py_DECREF()* 相同的警告。

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

釋放對於物件 *o* 的🄵參照。該物件可能是 NULL，在這種情🄵下巨集不起作用；否則，效果與*Py_DECREF()* 相同，除非引數也設定🄵 NULL。*Py_DECREF()* 的警告不適用於傳遞的物件，因🄵巨集在釋放其參照之前小心地使用臨時變數🄵將引數設定🄵 NULL。

每當要釋放垃圾回收 (garbage collection) 期間可能被遍歷到之對於物件的參照時，使用此巨集是個好主意。

在 3.12 版的變更: 巨集引數現在僅會被求值 (evaluate) 一次。如果引數有其他副作用，則不再重🄵作用。

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

🄵穩定 ABI 的一部分. 代表取得對於物件 *o* 的🄵參照。*Py_XINCREF()* 的函式版本。它可用於 Python 的 runtime 動態嵌入。

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

🄵穩定 ABI 的一部分. 釋放對物件 *o* 的🄵參照。*Py_XDECREF()* 的函式版本。它可用於 Python 的 runtime 動態嵌入。

Py_SETREF(dst, src)

巨集安全地釋放對於物件 *dst* 的🄵參照🄵將 *dst* 設定🄵 *src*。

與*Py_CLEAR()* 的情🄵一樣，「明顯的」程式碼可能是致命的：

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

安全的方法是：

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

這會在釋放對 *dst* 舊值的參照之前將 *dst* 設定🄵 *src*，使得因 *dst* 被拆除而觸發的任何副作用 (side-effect) 之程式碼不會相信 *dst* 是指向一個有效物件。

在 3.6 版被加入.

在 3.12 版的變更: 巨集引數現在僅會被求值一次。如果引數有其他副作用，則不再重🄵作用。

Py_XSETREF(dst, src)

Py_SETREF 巨集的變體，請改用*Py_XDECREF()* 而非*Py_DECREF()*。

在 3.6 版被加入.

在 3.12 版的變更: 巨集引數現在僅會被求值一次。如果引數有其他副作用，則不再重🄵作用。

CHAPTER 5

例外處理

The functions described in this chapter will let you handle and raise Python exceptions. It is important to understand some of the basics of Python exception handling. It works somewhat like the POSIX `errno` variable: there is a global indicator (per thread) of the last error that occurred. Most C API functions don't clear this on success, but will set it to indicate the cause of the error on failure. Most C API functions also return an error indicator, usually `NULL` if they are supposed to return a pointer, or `-1` if they return an integer (exception: the `PyArg_*` functions return `1` for success and `0` for failure).

Concretely, the error indicator consists of three object pointers: the exception's type, the exception's value, and the traceback object. Any of those pointers can be `NULL` if non-set (although some combinations are forbidden, for example you can't have a non-`NULL` traceback if the exception type is `NULL`).

When a function must fail because some function it called failed, it generally doesn't set the error indicator; the function it called already set it. It is responsible for either handling the error and clearing the exception or returning after cleaning up any resources it holds (such as object references or memory allocations); it should *not* continue normally if it is not prepared to handle the error. If returning due to an error, it is important to indicate to the caller that an error has been set. If the error is not handled or carefully propagated, additional calls into the Python/C API may not behave as intended and may fail in mysterious ways.

備🄵

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 Printing and clearing

void **`PyErr_Clear`**()

🄵穩定 ABI 的一部分. Clear the error indicator. If the error indicator is not set, there is no effect.

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

🄵穩定 ABI 的一部分. Print a standard traceback to `sys.stderr` and clear the error indicator. **Unless** the error is a `SystemExit`, in that case no traceback is printed and the Python process will exit with the error code specified by the `SystemExit` instance.

Call this function **only** when the error indicator is set. Otherwise it will cause a fatal error!

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.

在 3.12 版的變更: 新增 `sys.last_exc` 設定。

void **PyErr_Print**()

🄵穩定 ABI 的一部分. `PyErr_PrintEx(1)` 的🄵名。

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

🄵穩定 ABI 的一部分. Call `sys.unraisablehook()` using the current exception and *obj* argument.

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.

An exception must be set when calling this function.

在 3.4 版的變更: Print a traceback. Print only traceback if *obj* is `NULL`.

在 3.8 版的變更: 使用 `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.

在 3.13 版被加入.

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

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. Print the standard traceback display of `exc` to `sys.stderr`, including chained exceptions and notes.

在 3.12 版被加入.

5.2 Raising exceptions

These functions help you set the current thread's error indicator. For convenience, some of these functions will always return a `NULL` pointer for use in a `return` statement.

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

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. This function is similar to *PyErr_SetString()* but lets you specify an arbitrary Python object for the "value" of the exception.

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

回傳值: 總是🄵 *NULL*。🄵穩定 ABI 的一部分. This function sets the error indicator and returns `NULL`. *exception* should be a Python exception class. The *format* and subsequent parameters help format the error message; they have the same meaning and values as in *PyUnicode_FromFormat()*. *format* is an ASCII-encoded string.

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

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 自 *3.5* 版本開始. Same as *PyErr_Format()*, but taking a `va_list` argument rather than a variable number of arguments.

在 3.5 版被加入.

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

🄵穩定 ABI 的一部分. This is a shorthand for `PyErr_SetObject(type, Py_None)`.

int **PyErr_BadArgument**()

🄵穩定 ABI 的一部分. This is a shorthand for `PyErr_SetString(PyExc_TypeError, message)`, where *message* indicates that a built-in operation was invoked with an illegal argument. It is mostly for internal use.

PyObject ***PyErr_NoMemory**()

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分. This is a shorthand for `PyErr_SetNone(PyExc_MemoryError)`; it returns `NULL` so an object allocation function can write `return PyErr_NoMemory();` when it runs out of memory.

PyObject ***PyErr_SetFromErrno**(*PyObject* *type)

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分. 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)

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分. 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)

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Similar to *PyErr_SetFromErrnoWithFilenameObject()*, but takes a second filename object, for raising errors when a function that takes two filenames fails.

在 3.4 版被加入.

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

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分. Similar to *PyErr_SetFromErrnoWithFilenameObject()*, but the filename is given as a C string. *filename* is decoded from the *filesystem encoding and error handler*.

PyObject ***PyErr_SetFromWindowsErr**(int ierr)

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 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`.

可用性: Windows.

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

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. Similar to *PyErr_SetFromWindowsErr()*, with an additional parameter specifying the exception type to be raised.

可用性: Windows.

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

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 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.

可用性: Windows.

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

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 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.

可用性: Windows.

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

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. Similar to *PyErr_SetExcFromWindowsErrWithFilenameObject()*, but accepts a second filename object.

可用性: Windows.

在 3.4 版被加入.

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

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. Similar to *PyErr_SetFromWindowsErrWithFilename()*, with an additional parameter specifying the exception type to be raised.

可用性: Windows.

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

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. This is a convenience function to raise `ImportError`. *msg* will be set as the exception's message string. *name* and *path*, both of which can be `NULL`, will be set as the `ImportError`'s respective `name` and `path` attributes.

在 3.3 版被加入.

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

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 自 *3.6* 版本開始. Much like *PyErr_SetImportError()* but this function allows for specifying a subclass of `ImportError` to raise.

在 3.6 版被加入.

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

Set file, line, and offset information for the current exception. If the current exception is not a `SyntaxError`, then it sets additional attributes, which make the exception printing subsystem think the exception is a `SyntaxError`.

在 3.4 版被加入.

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

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Like *PyErr_SyntaxLocationObject()*, but *filename* is a byte string decoded from the *filesystem encoding and error handler*.

在 3.2 版被加入.

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

🄵穩定 ABI 的一部分. Like *PyErr_SyntaxLocationEx()*, but the *col_offset* parameter is omitted.

void **PyErr_BadInternalCall**()

🄵穩定 ABI 的一部分. This is a shorthand for `PyErr_SetString(PyExc_SystemError, message)`, where *message* indicates that an internal operation (e.g. a Python/C API function) was invoked with an illegal argument. It is mostly for internal use.

5.3 Issuing warnings

Use these functions to issue warnings from C code. They mirror similar functions exported by the Python `warnings` module. They normally print a warning message to *sys.stderr*; however, it is also possible that the user has specified that warnings are to be turned into errors, and in that case they will raise an exception. It is also possible that the functions raise an exception because of a problem with the warning machinery. The return value is `0` if no exception is raised, or `-1` if an exception is raised. (It is not possible to determine whether a warning message is actually printed, nor what the reason is for the exception; this is intentional.) If an exception is raised, the caller should do its normal exception handling (for example, *Py_DECREF()* owned references and return an error value).

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

🄵穩定 ABI 的一部分. Issue a warning message. The *category* argument is a warning category (see below) or `NULL`; the *message* argument is a UTF-8 encoded string. *stack_level* is a positive number giving a number of stack frames; the warning will be issued from the currently executing line of code in that stack frame. A *stack_level* of 1 is the function calling *PyErr_WarnEx()*, 2 is the function above that, and so forth.

Warning categories must be subclasses of *PyExc_Warning*; *PyExc_Warning* is a subclass of *PyExc_Exception*; the default warning category is *PyExc_RuntimeWarning*. The standard Python warning categories are available as global variables whose names are enumerated at 警告型🄵.

For information about warning control, see the documentation for the `warnings` module and the `-W` option in the command line documentation. There is no C API for warning control.

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

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

在 3.4 版被加入.

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

🄵穩定 ABI 的一部分. Similar to *PyErr_WarnExplicitObject()* except that *message* and *module* are UTF-8 encoded strings, and *filename* is decoded from the *filesystem encoding and error handler*.

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

🄵穩定 ABI 的一部分. Function similar to *PyErr_WarnEx()*, but use *PyUnicode_FromFormat()* to format the warning message. *format* is an ASCII-encoded string.

在 3.2 版被加入.

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

🄵穩定 ABI 的一部分 自 *3.6* 版本開始. Function similar to *PyErr_WarnFormat()*, but *category* is `ResourceWarning` and it passes *source* to `warnings.WarningMessage`.

在 3.6 版被加入.

5.4 Querying the error indicator

PyObject ***PyErr_Occurred**()

回傳值: 借用參照。🄵穩定 ABI 的一部分. Test whether the error indicator is set. If set, return the exception *type* (the first argument to the last call to one of the `PyErr_Set*` functions or to *PyErr_Restore()*). If not set, return `NULL`. You do not own a reference to the return value, so you do not need to *Py_DECREF()* it.

The caller must have an *attached thread state*.

!!! 備🄵

Do not compare the return value to a specific exception; use *PyErr_ExceptionMatches()* instead, shown below. (The comparison could easily fail since the exception may be an instance instead of a class, in the case of a class exception, or it may be a subclass of the expected exception.)

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

🄵穩定 ABI 的一部分. Equivalent to `PyErr_GivenExceptionMatches(PyErr_Occurred(), exc)`. This should only be called when an exception is actually set; a memory access violation will occur if no exception has been raised.

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

🄵穩定 ABI 的一部分. Return true if the *given* exception matches the exception type in *exc*. If *exc* is a class object, this also returns true when *given* is an instance of a subclass. If *exc* is a tuple, all exception types in the tuple (and recursively in subtuples) are searched for a match.

PyObject ***PyErr_GetRaisedException**(void)

回傳值: 新的參照。🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 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.

舉例來🄵:

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

   /* ... 可能會🄵生其他錯誤的程式碼 ... */

   PyErr_SetRaisedException(exc);
}
```

!!! 也參考

PyErr_GetHandledException(), to save the exception currently being handled.

在 3.12 版被加入.

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

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. Set *exc* as the exception currently being raised, clearing the existing exception if one is set.

!!! 警告

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

在 3.12 版被加入.

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

🄵穩定 ABI 的一部分. 在 3.12 版之後被🄵用: 請改用*PyErr_GetRaisedException()*。

Retrieve the error indicator into three variables whose addresses are passed. If the error indicator is not set, set all three variables to `NULL`. If it is set, it will be cleared and you own a reference to each object retrieved. The value and traceback object may be `NULL` even when the type object is not.

備🄵

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

舉例來🄵:

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

   /* ... 可能會🄵生其他錯誤的程式碼 ... */

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

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

🄵穩定 ABI 的一部分. 在 3.12 版之後被🄵用: 請改用*PyErr_SetRaisedException()*。

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.)

備🄵

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)

🄵穩定 ABI 的一部分. 在 3.12 版之後被🄵用: Use *PyErr_GetRaisedException()* instead, to avoid any possible de-normalization.

Under certain circumstances, the values returned by *PyErr_Fetch()* below can be "unnormalized", meaning that `*exc` is a class object but `*val` is not an instance of the same class. This function can be used to instantiate the class in that case. If the values are already normalized, nothing happens. The delayed normalization is implemented to improve performance.

備🄵

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)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Retrieve the active exception instance, as would be returned by `sys.exception()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. Returns a new reference to the exception or `NULL`. Does not modify the interpreter's exception state.

備🄵

This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use *PyErr_SetHandledException()* to restore or clear the exception state.

在 3.11 版被加入.

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

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Set the active exception, as known from `sys.exception()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. To clear the exception state, pass `NULL`.

備🄵

This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use *PyErr_GetHandledException()* to get the exception state.

在 3.11 版被加入.

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

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Retrieve the old-style representation of the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. Returns new references for the three objects, any of which may be `NULL`. Does not modify the exception info state. This function is kept for backwards compatibility. Prefer using *PyErr_GetHandledException()*.

備🄵

This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use *PyErr_SetExcInfo()* to restore or clear the exception state.

在 3.3 版被加入.

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

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Set the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. This function steals the references of the arguments. To clear the exception state, pass `NULL` for all three arguments. This function is kept for backwards compatibility. Prefer using *PyErr_SetHandledException()*.

備🄵

This function is not normally used by code that wants to handle exceptions. Rather, it can be used when code needs to save and restore the exception state temporarily. Use *PyErr_GetExcInfo()* to read the exception state.

在 3.3 版被加入.

在 3.11 版的變更: The `type` and `traceback` arguments are no longer used and can be NULL. The interpreter now derives them from the exception instance (the `value` argument). The function still steals references of all three arguments.

5.5 訊號處理

int **PyErr_CheckSignals**()

🄵穩定 ABI 的一部分. This function interacts with Python's signal handling.

If the function is called from the main thread and under the main Python interpreter, it checks whether a signal has been sent to the processes and if so, invokes the corresponding signal handler. If the `signal` module is supported, this can invoke a signal handler written in Python.

The function attempts to handle all pending signals, and then returns `0`. However, if a Python signal handler raises an exception, the error indicator is set and the function returns `-1` immediately (such that other pending signals may not have been handled yet: they will be on the next *PyErr_CheckSignals()* invocation).

If the function is called from a non-main thread, or under a non-main Python interpreter, it does nothing and returns `0`.

This function can be called by long-running C code that wants to be interruptible by user requests (such as by pressing Ctrl-C).

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

void **PyErr_SetInterrupt**()

🄵穩定 ABI 的一部分. Simulate the effect of a `SIGINT` signal arriving. This is equivalent to `PyErr_SetInterruptEx(SIGINT)`.

> **備🄵**
>
> 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)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Simulate the effect of a signal arriving. The next time *PyErr_CheckSignals()* is called, the Python signal handler for the given signal number will be called.

This function can be called by C code that sets up its own signal handling and wants Python signal handlers to be invoked as expected when an interruption is requested (for example when the user presses Ctrl-C to interrupt an operation).

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

If *signum* is outside of the allowed range of signal numbers, `-1` is returned. Otherwise, `0` is returned. The error indicator is never changed by this function.

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

在 3.10 版被加入.

int **PySignal_SetWakeupFd**(int fd)

This utility function specifies a file descriptor to which the signal number is written as a single byte whenever a signal is received. *fd* must be non-blocking. It returns the previous such file descriptor.

The value `-1` disables the feature; this is the initial state. This is equivalent to `signal.set_wakeup_fd()` in Python, but without any error checking. *fd* should be a valid file descriptor. The function should only be called from the main thread.

在 3.5 版的變更: On Windows, the function now also supports socket handles.

5.6 例外類🄵

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

回傳值：新的參照。🄵穩定 ABI 的一部分. This utility function creates and returns a new exception class. The *name* argument must be the name of the new exception, a C string of the form `module.classname`. The *base* and *dict* arguments are normally `NULL`. This creates a class object derived from `Exception` (accessible in C as *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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Same as *PyErr_NewException()*, except that the new exception class can easily be given a docstring: If *doc* is non-`NULL`, it will be used as the docstring for the exception class.

在 3.2 版被加入.

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)

🄵穩定 ABI 的一部分 自 *3.8* 版本開始. 回傳例外類🄵 *ob* 的*tp_name*。

5.7 例外物件

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. Set the traceback associated with the exception to *tb*. Use `Py_None` to clear it.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. Set the context associated with the exception to *ctx*. Use `NULL` to clear it. There is no type check to make sure that *ctx* is an exception instance. This steals a reference to *ctx*.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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.

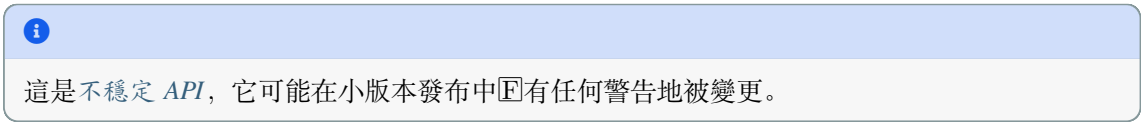
PyObject ***PyException_GetArgs**(*PyObject* *ex)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 回傳例外 *ex* 的 `args`。

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

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 設定例外 *ex* 的 `args` 🄵 *args*。

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



這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

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.

在 3.12 版被加入.

5.8 Unicode 例外物件

The following functions are used to create and modify Unicode exceptions from 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Create a `UnicodeDecodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.

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

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

回傳值：新的參照。🄵穩定 ABI 的一部分. Return the *encoding* attribute of the given exception object.

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

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

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

回傳值：新的參照。🄵穩定 ABI 的一部分. Return the *object* attribute of the given exception object.

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)

🄵穩定 ABI 的一部分. Get the *start* attribute of the given exception object and place it into **start*. *start* must not be `NULL`. Return `0` on success, `-1` on failure.

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

也參考

`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)

🄵穩定 ABI 的一部分. Set the *start* attribute of the given exception object to *start*. Return `0` on success, `-1` on failure.

> **備🄵**
>
> 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)

🄵穩定 ABI 的一部分. Get the *end* attribute of the given exception object and place it into **end*. *end* must not be `NULL`. Return `0` on success, `-1` on failure.

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)

🄵穩定 ABI 的一部分. Set the *end* attribute of the given exception object to *end*. Return `0` on success, `-1` on failure.

> **也參考**
>
> `UnicodeError.end`

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

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

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 回傳給定例外物件的 *reason* 屬性。

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

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

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

🄵穩定 ABI 的一部分. Set the *reason* attribute of the given exception object to *reason*. Return `0` on success, `-1` on failure.

5.9 遞🄵控制

These two functions provide a way to perform safe recursive calls at the C level, both in the core and in extension modules. They are needed if the recursive code does not necessarily invoke Python code (which tracks its recursion depth automatically). They are also not needed for *tp_call* implementations because the *call protocol* takes care of recursion handling.

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

🄵穩定 ABI 的一部分 自 *3.9* 版本開始. Marks a point where a recursive C-level call is about to be performed.

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 should be a UTF-8 encoded string such as `" in instance check"` to be concatenated to the `RecursionError` message caused by the recursion depth limit.

在 3.9 版的變更: This function is now also available in the *limited API*.

void **Py_LeaveRecursiveCall**(void)

🄵穩定 ABI 的一部分 自 *3.9* 版本開始. Ends a *Py_EnterRecursiveCall()*. Must be called once for each *successful* invocation of *Py_EnterRecursiveCall()*.

在 3.9 版的變更: This function is now also available in the *limited API*.

Properly implementing *tp_repr* for container types requires special recursion handling. In addition to protecting the stack, *tp_repr* also needs to track objects to prevent cycles. The following two functions facilitate this functionality. Effectively, these are the C equivalent to `reprlib.recursive_repr()`.

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

🄵穩定 ABI 的一部分. Called at the beginning of the *tp_repr* implementation to detect cycles.

If the object has already been processed, the function returns a positive integer. In that case the *tp_repr* implementation should return a string object indicating a cycle. As examples, `dict` objects return `{...}` and `list` objects return `[...]`.

The function will return a negative integer if the recursion limit is reached. In that case the *tp_repr* implementation should typically return `NULL`.

Otherwise, the function returns zero and the *tp_repr* implementation can continue normally.

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

🄵穩定 ABI 的一部分. Ends a *Py_ReprEnter()*. Must be called once for each invocation of *Py_ReprEnter()* that returns zero.

5.10 例外和警告型🄵

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 例外型🄵

C 名懲	Python 名懲
PyObject ***PyExc_BaseException** 🄵穩定 ABI 的一部分.	`BaseException`
PyObject ***PyExc_BaseExceptionGroup** 🄵穩定 ABI 的一部分 自 *3.11* 版本開始.	`BaseExceptionGroup`
PyObject ***PyExc_Exception** 🄵穩定 ABI 的一部分.	`Exception`
PyObject ***PyExc_ArithmeticError** 🄵穩定 ABI 的一部分.	`ArithmeticError`
PyObject ***PyExc_AssertionError** 🄵穩定 ABI 的一部分.	`AssertionError`

繼續下一頁

表格 1 – 繼續上一頁

C 名懲	Python 名懲
PyObject ***PyExc_AttributeError** 🄵穩定 ABI 的一部分.	`AttributeError`
PyObject ***PyExc_BlockingIOError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`BlockingIOError`
PyObject ***PyExc_BrokenPipeError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`BrokenPipeError`
PyObject ***PyExc_BufferError** 🄵穩定 ABI 的一部分.	`BufferError`
PyObject ***PyExc_ChildProcessError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`ChildProcessError`
PyObject ***PyExc_ConnectionAbortedError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`ConnectionAbortedError`
PyObject ***PyExc_ConnectionError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`ConnectionError`
PyObject ***PyExc_ConnectionRefusedError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`ConnectionRefusedError`
PyObject ***PyExc_ConnectionResetError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`ConnectionResetError`
PyObject ***PyExc_EOFError** 🄵穩定 ABI 的一部分.	`EOFError`
PyObject ***PyExc_FileExistsError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`FileExistsError`
PyObject ***PyExc_FileNotFoundError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`FileNotFoundError`
PyObject ***PyExc_FloatingPointError** 🄵穩定 ABI 的一部分.	`FloatingPointError`

繼續下一頁

表格 1 – 繼續上一頁

C 名懲	Python 名懲
PyObject ***PyExc_GeneratorExit** 🄵穩定 ABI 的一部分.	`GeneratorExit`
PyObject ***PyExc_ImportError** 🄵穩定 ABI 的一部分.	`ImportError`
PyObject ***PyExc_IndentationError** 🄵穩定 ABI 的一部分.	`IndentationError`
PyObject ***PyExc_IndexError** 🄵穩定 ABI 的一部分.	`IndexError`
PyObject ***PyExc_InterruptedError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`InterruptedError`
PyObject ***PyExc_IsADirectoryError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`IsADirectoryError`
PyObject ***PyExc_KeyError** 🄵穩定 ABI 的一部分.	`KeyError`
PyObject ***PyExc_KeyboardInterrupt** 🄵穩定 ABI 的一部分.	`KeyboardInterrupt`
PyObject ***PyExc_LookupError** 🄵穩定 ABI 的一部分.	`LookupError`
PyObject ***PyExc_MemoryError** 🄵穩定 ABI 的一部分.	`MemoryError`
PyObject ***PyExc_ModuleNotFoundError** 🄵穩定 ABI 的一部分 自 *3.6* 版本開始.	`ModuleNotFoundError`
PyObject ***PyExc_NameError** 🄵穩定 ABI 的一部分.	`NameError`
PyObject ***PyExc_NotADirectoryError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`NotADirectoryError`

繼續下一頁

表格 1 – 繼續上一頁

C 名懲	Python 名懲
PyObject ***PyExc_NotImplementedError** 🄵穩定 ABI 的一部分.	`NotImplementedError`
PyObject ***PyExc_OSError** 🄵穩定 ABI 的一部分.	`OSError`
PyObject ***PyExc_OverflowError** 🄵穩定 ABI 的一部分.	`OverflowError`
PyObject ***PyExc_PermissionError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`PermissionError`
PyObject ***PyExc_ProcessLookupError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`ProcessLookupError`
PyObject ***PyExc_PythonFinalizationError**	`PythonFinalizationError`
PyObject ***PyExc_RecursionError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`RecursionError`
PyObject ***PyExc_ReferenceError** 🄵穩定 ABI 的一部分.	`ReferenceError`
PyObject ***PyExc_RuntimeError** 🄵穩定 ABI 的一部分.	`RuntimeError`
PyObject ***PyExc_StopAsyncIteration** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`StopAsyncIteration`
PyObject ***PyExc_StopIteration** 🄵穩定 ABI 的一部分.	`StopIteration`
PyObject ***PyExc_SyntaxError** 🄵穩定 ABI 的一部分.	`SyntaxError`
PyObject ***PyExc_SystemError** 🄵穩定 ABI 的一部分.	`SystemError`
PyObject ***PyExc_SystemExit** 🄵穩定 ABI 的一部分.	`SystemExit`

繼續下一頁

表格 1 – 繼續上一頁

C 名懲	Python 名懲
PyObject ***PyExc_TabError** 🄵穩定 ABI 的一部分.	TabError
PyObject ***PyExc_TimeoutError** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	TimeoutError
PyObject ***PyExc_TypeError** 🄵穩定 ABI 的一部分.	TypeError
PyObject ***PyExc_UnboundLocalError** 🄵穩定 ABI 的一部分.	UnboundLocalError
PyObject ***PyExc_UnicodeDecodeError** 🄵穩定 ABI 的一部分.	UnicodeDecodeError
PyObject ***PyExc_UnicodeEncodeError** 🄵穩定 ABI 的一部分.	UnicodeEncodeError
PyObject ***PyExc_UnicodeError** 🄵穩定 ABI 的一部分.	UnicodeError
PyObject ***PyExc_UnicodeTranslateError** 🄵穩定 ABI 的一部分.	UnicodeTranslateError
PyObject ***PyExc_ValueError** 🄵穩定 ABI 的一部分.	ValueError
PyObject ***PyExc_ZeroDivisionError** 🄵穩定 ABI 的一部分.	ZeroDivisionError

在 3.3 版被加入: *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* 和*PyExc_TimeoutError* 是在 **PEP 3151** 被引入。

在 3.5 版被加入: *PyExc_StopAsyncIteration* 和*PyExc_RecursionError*。

在 3.6 版被加入: *PyExc_ModuleNotFoundError*。

在 3.11 版被加入: *PyExc_BaseExceptionGroup*

5.10.2 OSError 🄵名

The following are a compatibility aliases to *PyExc_OSError*.

在 3.3 版的變更: These aliases used to be separate exception types.

C 名懲	Python 名懲	🄵解
PyObject ***PyExc_EnvironmentError** 🄵穩定 ABI 的一部分.	OSError	
PyObject ***PyExc_IOError** 🄵穩定 ABI 的一部分.	OSError	
PyObject ***PyExc_WindowsError** 🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始.	OSError	[win]

🄵解:

5.10.3 警告型🄵

C 名懲	Python 名懲
PyObject ***PyExc_Warning** 🄵穩定 ABI 的一部分.	`Warning`
PyObject ***PyExc_BytesWarning** 🄵穩定 ABI 的一部分.	`BytesWarning`
PyObject ***PyExc_DeprecationWarning** 🄵穩定 ABI 的一部分.	`DeprecationWarning`
PyObject ***PyExc_EncodingWarning** 🄵穩定 ABI 的一部分 自 *3.10* 版本開始.	`EncodingWarning`
PyObject ***PyExc_FutureWarning** 🄵穩定 ABI 的一部分.	`FutureWarning`
PyObject ***PyExc_ImportWarning** 🄵穩定 ABI 的一部分.	`ImportWarning`
PyObject ***PyExc_PendingDeprecationWarning** 🄵穩定 ABI 的一部分.	`PendingDeprecationWarning`
PyObject ***PyExc_ResourceWarning** 🄵穩定 ABI 的一部分 自 *3.7* 版本開始.	`ResourceWarning`
PyObject ***PyExc_RuntimeWarning** 🄵穩定 ABI 的一部分.	`RuntimeWarning`
PyObject ***PyExc_SyntaxWarning** 🄵穩定 ABI 的一部分.	`SyntaxWarning`
PyObject ***PyExc_UnicodeWarning** 🄵穩定 ABI 的一部分.	`UnicodeWarning`
PyObject ***PyExc_UserWarning** 🄵穩定 ABI 的一部分.	`UserWarning`

在 3.2 版被加入: *PyExc_ResourceWarning*。

在 3.10 版被加入: *PyExc_EncodingWarning*。

CHAPTER 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`.

> **備🄵**
>
> 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.

在 3.5 版的變更: 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)

🄵穩定 ABI 的一部分 自 *3.5* 版本開始. 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.

在 3.5 版被加入.

6.4 Legacy single-phase initialization

⚠ 注意

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()*.

CHAPTER 7

工具

本章中的函式可用來執行各種工具任務，包括幫助 C 程式碼提升跨平臺可🄴性 (portable)、在 C 中使用 Python module（模組）、以及剖析函式引數🄴基於 C 中的值來構建 Python 中的值等。

7.1 作業系統工具

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

回傳值：新的參照。🄴穩定 ABI 的一部分 自 *3.6* 版本開始. 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.

在 3.6 版被加入.

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**()

🄴穩定 ABI 的一部分 *on platforms with fork()* 自 *3.7* 版本開始. Function to prepare some internal state before a process fork. This should be called before calling `fork()` or any similar function that clones the current process. Only available on systems where `fork()` is defined.

> **警告**
>
> The C `fork()` call should only be made from the *"main" thread* (of the *"main" interpreter*). The same is true for `PyOS_BeforeFork()`.

在 3.7 版被加入.

void **PyOS_AfterFork_Parent**()

🄴穩定 ABI 的一部分 *on platforms with fork()* 自 *3.7* 版本開始. Function to update some internal state after a process fork. This should be called from the parent process after calling `fork()` or any similar function that

clones the current process, regardless of whether process cloning was successful. Only available on systems where `fork()` is defined.

> **警告**
>
> The C `fork()` call should only be made from the *"main" thread* (of the *"main" interpreter*). The same is true for `PyOS_AfterFork_Parent()`.

在 3.7 版被加入.

void **PyOS_AfterFork_Child**()

🄵穩定 ABI 的一部分 *on platforms with fork()* 自 *3.7* 版本開始. Function to update internal interpreter state after a process fork. This must be called from the child process after calling `fork()`, or any similar function that clones the current process, if there is any chance the process will call back into the Python interpreter. Only available on systems where `fork()` is defined.

> **警告**
>
> The C `fork()` call should only be made from the *"main" thread* (of the *"main" interpreter*). The same is true for `PyOS_AfterFork_Child()`.

在 3.7 版被加入.

> **也參考**
>
> `os.register_at_fork()` allows registering custom Python functions to be called by *`PyOS_BeforeFork()`*, *`PyOS_AfterFork_Parent()`* and *`PyOS_AfterFork_Child()`*.

void **PyOS_AfterFork**()

🄵穩定 ABI 的一部分 *on platforms with fork()*. Function to update some internal state after a process fork; this should be called in the new process if the Python interpreter will continue to be used. If a new executable is loaded into the new process, this function does not need to be called.

在 3.7 版之後被🄵用: This function is superseded by *`PyOS_AfterFork_Child()`*.

int **PyOS_CheckStack**()

🄵穩定 ABI 的一部分 *on platforms with USE_STACKCHECK* 自 *3.7* 版本開始. 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)

🄵穩定 ABI 的一部分.

PyOS_sighandler_t **PyOS_getsig**(int i)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始.

警告

This function should not be called directly: use the *PyConfig* API with the *PyConfig_SetBytesString()* function which ensures that *Python is preinitialized*.

This function must not be called before *Python is preinitialized* and so that the LC_CTYPE locale is properly configured: see the *Py_PreInitialize()* function.

Decode a byte string from the *filesystem encoding and error handler*. If the error handler is surrogateescape error handler, undecodable bytes are decoded as characters in range U+DC80..U+DCFF; and if a byte sequence can be decoded as a surrogate character, the bytes are escaped using the surrogateescape error handler instead of decoding them.

Return a pointer to a newly allocated wide character string, use *PyMem_RawFree()* to free the memory. If size is not `NULL`, write the number of wide characters excluding the null character into `*size`

Return `NULL` on decoding error or memory allocation error. If *size* is not `NULL`, `*size` is set to `(size_t)-1` on memory error or set to `(size_t)-2` on decoding error.

The *filesystem encoding and error handler* are selected by *PyConfig_Read()*: see *filesystem_encoding* and *filesystem_errors* members of *PyConfig*.

Decoding errors should never happen, unless there is a bug in the C library.

Use the *Py_EncodeLocale()* function to encode the character string back to a byte string.

也參考

The *PyUnicode_DecodeFSDefaultAndSize()* and *PyUnicode_DecodeLocaleAndSize()* functions.

在 3.5 版被加入.

在 3.7 版的變更: The function now uses the UTF-8 encoding in the Python UTF-8 Mode.

在 3.8 版的變更: 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Encode a wide character string to the *filesystem encoding and error handler*. If the error handler is surrogateescape error handler, surrogate characters in the range U+DC80..U+DCFF are converted to bytes 0x80..0xFF.

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

If error_pos is not `NULL`, `*error_pos` is set to `(size_t)-1` on success, or set to the index of the invalid character on encoding error.

The *filesystem encoding and error handler* are selected by *PyConfig_Read()*: see *filesystem_encoding* and *filesystem_errors* members of *PyConfig*.

Use the *Py_DecodeLocale()* function to decode the bytes string back to a wide character string.

警告

This function must not be called before *Python is preinitialized* and so that the LC_CTYPE locale is properly configured: see the *Py_PreInitialize()* function.

> **也參考**
>
> The *PyUnicode_EncodeFSDefault()* and *PyUnicode_EncodeLocale()* functions.

在 3.5 版被加入.

在 3.7 版的變更: The function now uses the UTF-8 encoding in the Python UTF-8 Mode.

在 3.8 版的變更: 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*.

在 3.14 版被加入.

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.

在 3.14 版被加入.

7.2 系統函式

These are utility functions that make functionality from the `sys` module accessible to C code. They all work with the current interpreter thread's `sys` module's dict, which is contained in the internal thread state structure.

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

回傳值：借用參照。🄵穩定 ABI 的一部分. Return the object *name* from the `sys` module or `NULL` if it does not exist, without setting an exception.

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

🄵穩定 ABI 的一部分. Set *name* in the `sys` module to *v* unless *v* is `NULL`, in which case *name* is deleted from the sys module. Returns `0` on success, `-1` on error.

void **PySys_ResetWarnOptions**()

🄵穩定 ABI 的一部分. Reset `sys.warnoptions` to an empty list. This function may be called prior to *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, ...)

🄵穩定 ABI 的一部分. Write the output string described by *format* to `sys.stdout`. No exceptions are raised, even if truncation occurs (see below).

format should limit the total size of the formatted output string to 1000 bytes or less -- after 1000 bytes, the output string is truncated. In particular, this means that no unrestricted "%s" formats should occur; these should be limited using "%.<N>s" where <N> is a decimal number calculated so that <N> plus the maximum size of other formatted text does not exceed 1000 bytes. Also watch out for "%f", which can print hundreds of digits for very large numbers.

If a problem occurs, or `sys.stdout` is unset, the formatted message is written to the real (C level) *stdout*.

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

🄵穩定 ABI 的一部分. As *PySys_WriteStdout()*, but write to `sys.stderr` or *stderr* instead.

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

🄵穩定 ABI 的一部分. Function similar to PySys_WriteStdout() but format the message using *PyUnicode_FromFormatV()* and don't truncate the message to an arbitrary length.

在 3.2 版被加入.

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

🄵穩定 ABI 的一部分. As *PySys_FormatStdout()*, but write to `sys.stderr` or *stderr* instead.

在 3.2 版被加入.

PyObject ***PySys_GetXOptions**()

回傳值：借用參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Return the current dictionary of `-X` options, similarly to `sys._xoptions`. On error, `NULL` is returned and an exception is set.

在 3.2 版被加入.

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

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Raise an auditing event with any active hooks. Return zero for success and non-zero with an exception set on failure.

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.

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

`sys.audit()` performs the same function from Python code.

請參🄵*PySys_AuditTuple()*。

在 3.8 版被加入.

在 3.8.2 版的變更: Require *Py_ssize_t* for `#` format characters. Previously, an unavoidable deprecation warning was raised.

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

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Similar to *PySys_Audit()*, but pass arguments as a Python object. *args* must be a `tuple`. To pass no arguments, *args* can be *NULL*.

在 3.13 版被加入.

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

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

The *userData* pointer is passed into the hook function. Since hook functions may be called from different runtimes, this pointer should not refer directly to Python state.

This function is safe to call before *Py_Initialize()*. When called after runtime initialization, existing audit hooks are notified and may silently abort the operation by raising an error subclassed from `Exception` (other errors will not be silenced).

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

See **PEP 578** for a detailed description of auditing. Functions in the runtime and standard library that raise events are listed in the audit events table. Details are in each function's documentation.

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().

在 3.8 版被加入.

7.3 行程控制

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

🄵穩定 ABI 的一部分. 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.

The `Py_FatalError()` function is replaced with a macro which logs automatically the name of the current function, unless the `Py_LIMITED_API` macro is defined.

在 3.9 版的變更: Log the function name automatically.

void **Py_Exit**(int status)

🄵穩定 ABI 的一部分. Exit the current process. This calls *Py_FinalizeEx()* and then calls the standard C library function `exit(status)`. If *Py_FinalizeEx()* indicates an error, the exit status is set to 120.

在 3.6 版的變更: Errors from finalization no longer ignored.

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

🄵穩定 ABI 的一部分. Register a cleanup function to be called by *Py_FinalizeEx()*. The cleanup function will be called with no arguments and should return no value. At most 32 cleanup functions can be registered. When the registration is successful, *Py_AtExit()* returns `0`; on failure, it returns `-1`. The cleanup function registered last is called first. Each cleanup function will be called at most once. Since Python's internal finalization will have completed before the cleanup function, no Python APIs should be called by *func*.

也參考

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

7.4 引入模組

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

回傳值: 新的參照。🄵穩定 ABI 的一部分. 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)

回傳值: 新的參照。🄵穩定 ABI 的一部分. This function is a deprecated alias of *PyImport_ImportModule()*.

在 3.3 版的變更: This function used to fail immediately when the import lock was held by another thread. In Python 3.3 though, the locking scheme switched to per-module locks for most purposes, so this function's special behaviour isn't needed anymore.

Deprecated since version 3.13, will be removed in version 3.15: 請改用*PyImport_ImportModule()*。

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

回傳值：新的參照。 Import a module. This is best described by referring to the built-in Python function `__import__()`.

The return value is a new reference to the imported module or top-level package, or `NULL` with an exception set on failure. Like for `__import__()`, the return value when a submodule of a package was requested is normally the top-level package, unless a non-empty *fromlist* was given.

Failing imports remove incomplete module objects, like with *PyImport_ImportModule()*.

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

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Import a module. This is best described by referring to the built-in Python function `__import__()`, as the standard `__import__()` function calls this function directly.

The return value is a new reference to the imported module or top-level package, or `NULL` with an exception set on failure. Like for `__import__()`, the return value when a submodule of a package was requested is normally the top-level package, unless a non-empty *fromlist* was given.

在 3.3 版被加入.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. Similar to *PyImport_ImportModuleLevelObject()*, but the name is a UTF-8 encoded string instead of a Unicode object.

在 3.3 版的變更: Negative values for *level* are no longer accepted.

PyObject ***PyImport_Import**(*PyObject* *name)

回傳值：新的參照。🄵穩定 ABI 的一部分. This is a higher-level interface that calls the current "import hook function" (with an explicit *level* of 0, meaning absolute import). It invokes the `__import__()` function from the `__builtins__` of the current globals. This means that the import is done using whatever import hooks are installed in the current environment.

This function always uses absolute imports.

PyObject ***PyImport_ReloadModule**(*PyObject* *m)

回傳值：新的參照。🄵穩定 ABI 的一部分. Reload a module. Return a new reference to the reloaded module, or `NULL` with an exception set on failure (the module still exists in this case).

PyObject ***PyImport_AddModuleRef**(const char *name)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 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.

在 3.13 版被加入.

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

回傳值：借用參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Similar to *PyImport_AddModuleRef()*, but return a *borrowed reference* and *name* is a Python `str` object.

在 3.3 版被加入.

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

回傳值：借用參照。🄵穩定 ABI 的一部分. Similar to *PyImport_AddModuleRef()*, but return a *borrowed reference*.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 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.

This function will reload the module if it was already imported. See *PyImport_ReloadModule()* for the intended way to reload a module.

If *name* points to a dotted name of the form `package.module`, any package structures not already created will still not be created.

See also *PyImport_ExecCodeModuleEx()* and *PyImport_ExecCodeModuleWithPathnames()*.

在 3.12 版的變更: The setting of `__cached__` and `__loader__` is deprecated. See `ModuleSpec` for alternatives.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. Like *PyImport_ExecCodeModule()*, but the `__file__` attribute of the module object is set to *pathname* if it is non-`NULL`.

也請見*PyImport_ExecCodeModuleWithPathnames()*。

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

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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.

在 3.3 版被加入.

在 3.12 版的變更: Setting `__cached__` is deprecated. See `ModuleSpec` for alternatives.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. Like *PyImport_ExecCodeModuleObject()*, but *name*, *pathname* and *cpathname* are UTF-8 encoded strings. Attempts are also made to figure out what the value for *pathname* should be from *cpathname* if the former is set to `NULL`.

在 3.2 版被加入.

在 3.3 版的變更: Uses `imp.source_from_cache()` in calculating the source path if only the bytecode path is provided.

在 3.12 版的變更: 不再使用已被移除的 `imp` 模組。

long **PyImport_GetMagicNumber**()

🄵穩定 ABI 的一部分. Return the magic number for Python bytecode files (a.k.a. `.pyc` file). The magic number should be present in the first four bytes of the bytecode file, in little-endian byte order. Returns `-1` on error.

在 3.3 版的變更: 當失敗時回傳 `-1`。

const char *PyImport_GetMagicTag()

🄵穩定 ABI 的一部分. Return the magic tag string for **PEP 3147** format Python bytecode file names. Keep in mind that the value at `sys.implementation.cache_tag` is authoritative and should be used instead of this function.

在 3.2 版被加入.

PyObject *PyImport_GetModuleDict()

回傳值：借用參照。🄵穩定 ABI 的一部分. Return the dictionary used for the module administration (a.k.a. `sys.modules`). Note that this is a per-interpreter variable.

PyObject *PyImport_GetModule(*PyObject* *name)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.8* 版本開始. Return the already imported module with the given name. If the module has not been imported yet then returns `NULL` but does not set an error. Returns `NULL` and sets an error if the lookup failed.

在 3.7 版被加入.

PyObject *PyImport_GetImporter(*PyObject* *path)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Load a frozen module named *name*. Return `1` for success, `0` if the module is not found, and `-1` with an exception set if the initialization failed. To access the imported module on a successful load, use *PyImport_ImportModule()*. (Note the misnomer --- this function would reload the module if it was already imported.)

在 3.3 版被加入.

在 3.4 版的變更: The `__file__` attribute is no longer set on the module.

int PyImport_ImportFrozenModule(const char *name)

🄵穩定 ABI 的一部分. Similar to *PyImport_ImportFrozenModuleObject()*, but the name is a UTF-8 encoded string instead of a Unicode object.

struct _frozen

This is the structure type definition for frozen module descriptors, as generated by the **freeze** utility (see `Tools/freeze/` in the Python source distribution). Its definition, found in `Include/import.h`, is:

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

在 3.11 版的變更: The new `is_package` field indicates whether the module is a package or not. This replaces setting the `size` field to a negative value.

const struct *_frozen* *PyImport_FrozenModules

This pointer is initialized to point to an array of *_frozen* records, terminated by one whose members are all `NULL` or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

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

🄵穩定 ABI 的一部分. Add a single module to the existing table of built-in modules. This is a convenience wrapper around *PyImport_ExtendInittab()*, returning `-1` if the table could not be extended. 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. This should be called before *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()*.

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

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

回傳值：新的參照。 Import the module *mod_name* and get its attribute *attr_name*.

名稱必須是 Python `str` 物件。

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.

在 3.14 版被加入.

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

回傳值：新的參照。 Similar to *PyImport_ImportModuleAttr()*, but names are UTF-8 encoded strings instead of Python `str` objects.

在 3.14 版被加入.

7.5 資料 marshal 操作的支援

這些例程允許 C 程式碼使用與 `marshal` 模組相同的資料格式來處理序列化物件。有函式可以將資料寫入序列化格式，還有其他函式可以用來讀取回資料。用來儲存 marshal 過的資料的檔案必須以二進位模式開🄵。

數值會以最低有效位元組 (the least significant byte) 優先的方式儲存。

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)

將一個 `long` 整數 *value* marshal 🄵 *file*。這只會寫入 *value* 的最低有效 32 位元；無論原生 `long` 型🄵的大小如何。*version* 表示檔案的格式。

這個函式可能會失敗，這時會設定錯誤指示器。使用*PyErr_Occurred()* 來檢查是否發生錯誤。

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

將一個 Python 物件 *value* marshal 🄵 *file*。*version* 表示檔案的格式。

這個函式可能會失敗，這時會設定錯誤指示器。使用*PyErr_Occurred()* 來檢查是否發生錯誤。

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

回傳值：新的參照。 回傳一個位元組物件，包含 *value* 的 marshal 表示。*version* 🄵檔案的格式。

以下函式允許將 marshal 過後的值讀取回來。

long **PyMarshal_ReadLongFromFile**(FILE *file)

從一個🄵讀取而開🄴的 `FILE*` 中的資料串流回傳一個 C `long`。無論 `long` 的原生大小如何，這個函式只能讀取 32 位元的值。

在錯誤發生時，會設定合適的例外（`EOFError`）🄵回傳 `-1`。

int **PyMarshal_ReadShortFromFile**(FILE *file)

從一個🄵讀取而開🄴的 `FILE*` 中的資料串流回傳一個 C `short`。無論 `short` 的原生大小如何，這個函式只能讀取 16 位元的值。

在錯誤發生時，會設定合適的例外（`EOFError`）🄵回傳 `-1`。

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

回傳值：新的參照。 從一個🄵讀取而開🄴的 `FILE*` 中的資料串流回傳一個 Python 物件。

在錯誤發生時，會設定合適的例外（`EOFError`、`ValueError` 或 `TypeError`）🄵回傳 `NULL`。

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

回傳值：新的參照。 從一個🄵讀取而開🄴的 `FILE*` 中的資料串流回傳一個 Python 物件。與*PyMarshal_ReadObjectFromFile()* 不同，這個函式假設不會從檔案中讀取其他物件，允許它積極地將檔案資料載入記憶體，以便反序列化可以從記憶體中的資料操作，而不是從檔案中逐位元組讀取。只有在確定不會從檔案中讀取其他🄵容時，才使用這個變體。

在錯誤發生時，會設定合適的例外（`EOFError`、`ValueError` 或 `TypeError`）🄵回傳 `NULL`。

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

回傳值：新的參照。 從一個包含 *len* 位元組、被 *data* 指向的位元組緩衝區中的資料串流回傳一個 Python 物件。

在錯誤發生時，會設定合適的例外（`EOFError`、`ValueError` 或 `TypeError`）🄵回傳 `NULL`。

7.6 剖析引數與建置數值

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

The first three of these functions described, *PyArg_ParseTuple()*, *PyArg_ParseTupleAndKeywords()*, and *PyArg_Parse()*, all use *format strings* which are used to tell the function about the expected arguments. The format strings use the same syntax for each of these functions.

7.6.1 剖析引數

A format string consists of zero or more "format units." A format unit describes one Python object; it is usually a single character or a parenthesized sequence of format units. With a few exceptions, a format unit that is not a parenthesized sequence normally corresponds to a single address argument to these functions. In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that matches the format unit; and the entry in [square] brackets is the type of the C variable(s) whose address should be passed.

Strings and buffers

> **備🄵**
>
> 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.

These formats allow accessing an object as a contiguous chunk of memory. You don't have to provide raw storage for the returned unicode or bytes area.

Unless otherwise stated, buffers are not 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 *]
Convert a Unicode object to a C pointer to a character string. A pointer to an existing string is stored in the character pointer variable whose address you pass. The C string is NUL-terminated. The Python string must not contain embedded null code points; if it does, a `ValueError` exception is raised. Unicode objects are converted to C strings using `'utf-8'` encoding. If this conversion fails, a `UnicodeError` is raised.

> **備🄵**
>
> This format does not accept *bytes-like objects*. If you want to accept filesystem paths and convert them to C character strings, it is preferable to use the `O&` format with *PyUnicode_FSConverter()* as *converter*.

在 3.5 版的變更: Previously, `TypeError` was raised when embedded null code points were encountered in the Python string.

`s*` (`str` 或*bytes-like object*) [Py_buffer]
This format accepts Unicode objects as well as bytes-like objects. It fills a *Py_buffer* structure provided by the caller. In this case the resulting C string may contain embedded NUL bytes. Unicode objects are converted to C strings using `'utf-8'` encoding.

`s#` (`str`、唯讀的*bytes-like object*) [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` 或 `None`) [const char *]
Like `s`, but the Python object may also be `None`, in which case the C pointer is set to `NULL`.

`z*` (`str`、*bytes-like object* 或 `None`) [Py_buffer]
Like `s*`, but the Python object may also be `None`, in which case the `buf` member of the *Py_buffer* structure is set to `NULL`.

`z#` (`str`、唯讀的*bytes-like object* 或 `None`) [const char *, *Py_ssize_t*]
Like `s#`, but the Python object may also be `None`, in which case the C pointer is set to `NULL`.

`y` (唯讀的*bytes-like object*) [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.

在 3.5 版的變更: Previously, `TypeError` was raised when embedded null bytes were encountered in the bytes buffer.

`y*` (*bytes-like object*) [Py_buffer]
This variant on `s*` doesn't accept Unicode objects, only bytes-like objects. **This is the recommended way to accept binary data.**

y# (唯讀的*bytes-like object*) **[const char *, *Py_ssize_t*]**
This variant on `s#` doesn't accept Unicode objects, only bytes-like objects.

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* (可讀寫*bytes-like object*) **[Py_buffer]**
This format accepts any object which implements the read-write buffer interface. It fills a *Py_buffer* structure provided by the caller. The buffer may contain embedded null bytes. The caller have to call *PyBuffer_Release()* when it is done with the buffer.

es (**str**) **[const char *encoding, char **buffer]**
This variant on `s` is used for encoding Unicode into a character buffer. It only works for encoded data without embedded NUL bytes.

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() will allocate a buffer of the needed size, copy the encoded data into this buffer and adjust **buffer* to reference the newly allocated storage. The caller is responsible for calling *PyMem_Free()* to free the allocated buffer after use.

et (**str**、**bytes** 或 **bytearray**) **[const char *encoding, char **buffer]**
Same as `es` except that byte string objects are passed through without recoding them. Instead, the implementation assumes that the byte string object uses the encoding passed in as parameter.

es# (**str**) **[const char *encoding, char **buffer, *Py_ssize_t* *buffer_length]**
This variant on `s#` is used for encoding Unicode into a character buffer. Unlike the `es` format, this variant allows input data which contains NUL characters.

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.

有兩個操作模式：

If **buffer* points a `NULL` pointer, the function will allocate a buffer of the needed size, copy the encoded data into this buffer and set **buffer* to reference the newly allocated storage. The caller is responsible for calling *PyMem_Free()* to free the allocated buffer after usage.

If **buffer* points to a non-`NULL` pointer (an already allocated buffer), *PyArg_ParseTuple()* will use this location as the buffer and interpret the initial value of **buffer_length* as the buffer size. It will then copy the encoded data into the buffer and NUL-terminate it. If the buffer is not large enough, a `ValueError` will be set.

In both cases, **buffer_length* is set to the length of the encoded data without the trailing NUL byte.

et# (**str**, **bytes** or **bytearray**) **[const char *encoding, char **buffer, *Py_ssize_t* *buffer_length]**
Same as `es#` except that byte string objects are passed through without recoding them. Instead, the implementation assumes that the byte string object uses the encoding passed in as parameter.

在 3.12 版的變更: `u`, `u#`, `Z`, and `Z#` are removed because they used a legacy `Py_UNICODE*` representation.

數字

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]
將一個 Python 非負整數轉🄵成無符號 tiny integer（小整數），儲存在 C 的 `unsigned`

`B` (`int`) [unsigned char]
將一個 Python 整數轉🄵成 tiny integer，轉🄵過程無溢位檢查，儲存在 C 的 `unsigned char`。

`h` (`int`) [short int]
將一個 Python 整數轉🄵成 C 的 `short int`。

`H` (`int`) [unsigned short int]
將一個 Python 整數轉🄵成 C 的 `unsigned short int`，轉🄵過程無溢位檢查。

`i` (`int`) [int]
將一個 Python 整數轉🄵成 C 的 `int`。

`I` (`int`) [unsigned int]
將一個 Python 整數轉🄵成 C 的 `unsigned int`，轉🄵過程無溢位檢查。

`l` (`int`) [long int]
將一個 Python 整數轉🄵成 C 的 `long int`。

`k` (`int`) [unsigned long]
將一個 Python 整數轉🄵成 C 的 `unsigned long`，轉🄵過程無溢位檢查。

在 3.14 版的變更: Use `__index__()` if available.

`L` (`int`) [long long]
將一個 Python 整數轉🄵成 C 的 `long long`。

`K` (`int`) [unsigned long long]
將一個 Python 整數轉🄵成 C 的 `unsigned long long`，轉🄵過程無溢位檢查。

在 3.14 版的變更: Use `__index__()` if available.

`n` (`int`) [*`Py_ssize_t`*]
將一個 Python 整數轉🄵成 C 的*`Py_ssize_t`*。

`c` (`bytes` 或長度🄵 1 的 `bytearray`) [char]
Convert a Python byte, represented as a `bytes` or `bytearray` object of length 1, to a C `char`.

在 3.3 版的變更: 允許 `bytearray` 物件。

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

`f` (`float`) [float]
將一個 Python 浮點數轉🄵成 C 的:c:type:`float`。

`d` (`float`) [double]
將一個 Python 浮點數轉🄵成 C 的:c:type:`double`。

`D` (`complex`) [Py_complex]
將一個 Python 🄵數轉🄵成 C 的*`Py_complex`* 結構。

其他物件

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()`*.

在 3.1 版的變更: 新增 `Py_CLEANUP_SUPPORTED`。

p (bool) [int]
Tests the value passed in for truth (a boolean **p**redicate) and converts the result to its equivalent C true/false integer value. Sets the int to `1` if the expression was true and `0` if it was false. This accepts any valid Python value. See truth for more information about how Python tests values for truth.

在 3.3 版被加入.

(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.

在 3.14 版的變更: `str` and `bytearray` objects no longer accepted as a sequence.

在 3.14 版之後被🄵用: Non-tuple sequences are deprecated if *items* contains format units which store a borrowed buffer or a borrowed reference.

A few other characters have a meaning in a format string. These may not occur inside nested parentheses. They are:

|
Indicates that the remaining arguments in the Python argument list are optional. The C variables corresponding to optional arguments should be initialized to their default value --- when an optional argument is not specified, *`PyArg_ParseTuple()`* does not touch the contents of the corresponding C variable(s).

$
`PyArg_ParseTupleAndKeywords()` only: Indicates that the remaining arguments in the Python argument list are keyword-only. Currently, all keyword-only arguments must also be optional arguments, so `|` must always be specified before `$` in the format string.

在 3.3 版被加入.

`:`
: The list of format units ends here; the string after the colon is used as the function name in error messages (the "associated value" of the exception that *PyArg_ParseTuple()* raises).

`;`
: The list of format units ends here; the string after the semicolon is used as the error message *instead* of the default error message. `:` and `;` mutually exclude each other.

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)!

Additional arguments passed to these functions must be addresses of variables whose type is determined by the format string; these are used to store values from the input tuple. There are a few cases, as described in the list of format units above, where these parameters are used as input values; they should match what is specified for the corresponding format unit in that case.

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.

API 函式

int **PyArg_ParseTuple**(*PyObject* *args, const char *format, ...)
: 🄵穩定 ABI 的一部分. Parse the parameters of a function that takes only positional parameters into local variables. Returns true on success; on failure, it returns false and raises the appropriate exception.

int **PyArg_VaParse**(*PyObject* *args, const char *format, va_list vargs)
: 🄵穩定 ABI 的一部分. Identical to *PyArg_ParseTuple()*, except that it accepts a va_list rather than a variable number of arguments.

int **PyArg_ParseTupleAndKeywords**(*PyObject* *args, *PyObject* *kw, const char *format, char *const *keywords, ...)
: 🄵穩定 ABI 的一部分. 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.

  > **備🄵**
  >
  > 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.

  在 3.6 版的變更: 新增對僅限位置參數的支援。

  在 3.13 版的變更: 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)
: 🄵穩定 ABI 的一部分. Identical to *PyArg_ParseTupleAndKeywords()*, except that it accepts a va_list rather than a variable number of arguments.

int **PyArg_ValidateKeywordArguments**(*PyObject**)
: 🄵穩定 ABI 的一部分. Ensure that the keys in the keywords argument dictionary are strings. This is only needed if *PyArg_ParseTupleAndKeywords()* is not used, since the latter already does this check.

  在 3.2 版被加入.

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

🄵穩定 ABI 的一部分. 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.

舉例來🄵:

```
// 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, ...)

🄵穩定 ABI 的一部分. 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;
}
```

The call to *PyArg_UnpackTuple()* in this example is entirely equivalent to this call to *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`.

在 3.13 版被加入.

7.6.2 建置數值

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 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()` does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

When memory buffers are passed as parameters to supply data to build objects, as for the `s` and `s#` formats, the required data is copied. Buffers provided by the caller are never referenced by the objects created by *`Py_BuildValue()`*. In other words, if your code invokes `malloc()` and passes the allocated memory to *`Py_BuildValue()`*, your code is responsible for calling `free()` for that memory once *`Py_BuildValue()`* returns.

In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that the format unit will return; and the entry in [square] brackets is the type of the C value(s) to be passed.

The characters space, tab, colon and comma are ignored in format strings (but not within format units such as `s#`). This can be used to make long format strings a tad more readable.

`s` (`str` 或 `None`) [const char *]
Convert a null-terminated C string to a Python `str` object using `'utf-8'` encoding. If the C string pointer is `NULL`, `None` is used.

`s#` (`str` 或 `None`) [const char *, *`Py_ssize_t`*]
Convert a C string and its length to a Python `str` object using `'utf-8'` encoding. If the C string pointer is `NULL`, the length is ignored and `None` is returned.

`y` (`bytes`) [const char *]
This converts a C string to a Python `bytes` object. If the C string pointer is `NULL`, `None` is returned.

`y#` (`bytes`) [const char *, *`Py_ssize_t`*]
This converts a C string and its lengths to a Python object. If the C string pointer is `NULL`, `None` is returned.

`z` (`str` 或 `None`) [const char *]
和 `s` 相同。

`z#` (`str` 或 `None`) [const char *, *`Py_ssize_t`*]
和 `s#` 相同。

`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`*]
Convert a Unicode (UTF-16 or UCS-4) data buffer and its length to a Python Unicode object. If the Unicode buffer pointer is `NULL`, the length is ignored and `None` is returned.

`U` (`str` 或 `None`) [const char *]
和 `s` 相同。

`U#` (`str` 或 `None`) [const char *, *`Py_ssize_t`*]
和 `s#` 相同。

`i` (`int`) [int]
將一個 C 的 `int` 轉🄵成 Python 整數物件。

`b` (`int`) [char]
將一個 C 的 `char` 轉🄵成 Python 整數物件。

h (int) [short int]
將一個 C 的 `short int` 轉🄵成 Python 整數物件。

l (int) [long int]
將一個 C 的 `long int` 轉🄵成 Python 整數物件。

B (int) [unsigned char]
將一個 C 的 `unsigned char` 轉🄵成 Python 整數物件。

H (int) [unsigned short int]
將一個 C 的 `unsigned short int` 轉🄵成 Python 整數物件。

I (int) [unsigned int]
將一個 C 的 `unsigned int` 轉🄵成 Python 整數物件。

k (int) [unsigned long]
將一個 C 的 `unsigned long` 轉🄵成 Python 整數物件。

L (int) [long long]
將一個 C 的 `long long` 轉🄵成 Python 整數物件。

K (int) [unsigned long long]
將一個 C 的 `unsigned long long` 轉🄵成 Python 整數物件。

n (int) [*Py_ssize_t*]
將一個 C 的*Py_ssize_t* 轉🄵成 Python 整數。

p (bool) [int]
將一個 C 的 `int` 轉🄵成 Python `bool` 物件。

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`.

在 3.14 版被加入.

c (長度🄵 1 的 bytes) [char]
將一個 C 中代表一個位元組的 `int` 轉🄵成 Python 中長度🄵一的 `bytes`。

C (長度🄵 1 的 str) [int]
將一個 C 中代表一個字元的 `int` 轉🄵成 Python 中長度🄵一的 `str`。

d (float) [double]
將一個 C 的 `double` 轉🄵成 Python 浮點數。

f (float) [float]
將一個 C 的 `float` 轉🄵成 Python 浮點數。

D (complex) [Py_complex *]
將一個 C 的*Py_complex* 結構轉🄵成 Python 🄵數。

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 *]
和 `O` 相同。

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*]
Convert a sequence of C values to a Python tuple with the same number of items.

[items] (list) [*matching-items*]
Convert a sequence of C values to a Python list with the same number of items.

{items} (dict) [*matching-items*]
Convert a sequence of C values to a Python dictionary. Each pair of consecutive C values adds one item to the dictionary, serving as key and value, respectively.

If there is an error in the format string, the `SystemError` exception is set and `NULL` returned.

PyObject ***Py_VaBuildValue**(const char *format, va_list vargs)
回傳值：新的參照。🄵穩定 ABI 的一部分. Identical to *Py_BuildValue()*, except that it accepts a va_list rather than a variable number of arguments.

7.7 字串轉🄵與格式化

用於數字轉🄵和格式化字串輸出的函式。

int **PyOS_snprintf**(char *str, size_t size, const char *format, ...)
🄵穩定 ABI 的一部分. 根據格式字串 *format* 和額外引數，輸出不超過 *size* 位元組給 *str*。請參🄵 Unix 使用手🄵 *snprintf(3)*。

int **PyOS_vsnprintf**(char *str, size_t size, const char *format, va_list va)
🄵穩定 ABI 的一部分. 根據格式字串 *format* 和變數引數串列 *va*，輸出不超過 *size* 位元組給 *str*。Unix 使用手🄵 *vsnprintf(3)*。

PyOS_snprintf() 和*PyOS_vsnprintf()* 包裝標準 C 函式庫函式 `snprintf()` 和 `vsnprintf()`。它們的目的是確保邊角案例 (corner case) 下的行🄵一致，而標準 C 函式則不然。

包裝器確保回傳時 `str[size-1]` 始終🄵 `'\0'`。他們永遠不會在 str 中寫入超過 *size* 位元組（包括尾隨的 `'\0'`）。這兩個函式都要求 `str != NULL`、`size > 0`、`format != NULL` 和 `size < INT_MAX`。請注意，這表示🄵有與 C99 `n = snprintf(NULL, 0, ...)` 等效的函式來🄵定必要的緩衝區大小。

這些函式的回傳值 (*rv*) 應如下被直譯：

- 當 `0 <= rv < size` 時，輸出轉🄵成功，*rv* 字元被寫入 *str*（不包括 `str[rv]` 處的尾隨 `'\0'` 位元組）。
- 當 `rv >= size` 時，輸出轉🄵被截斷，🄵且需要具有 `rv + 1` 位元組的緩衝區才能成功。在這種情🄵下，`str[size-1]` 是 `'\0'`。
- 當 `rv < 0` 時，代表「有不好的事情發生了」。在這種情🄵下，`str[size-1]` 也是 `'\0'`，但 *str* 的其余部分未定義。錯誤的確切原因取🄵於底層平台。

以下函式提供與區域設定無關 (locale-independent) 的字串到數字的轉🄵。

unsigned long **PyOS_strtoul**(const char *str, char **ptr, int base)
🄵穩定 ABI 的一部分. Convert the initial part of the string in `str` to an `unsigned long` value according to the given `base`, which must be between `2` and `36` inclusive, or be the special value `0`.

Leading white space and case of characters are ignored. If `base` is zero it looks for a leading `0b`, `0o` or `0x` to tell which base. If these are absent it defaults to `10`. Base must be 0 or between 2 and 36 (inclusive). If `ptr` is non-`NULL` it will contain a pointer to the end of the scan.

If the converted value falls out of range of corresponding return type, range error occurs (`errno` is set to `ERANGE`) and `ULONG_MAX` is returned. If no conversion can be performed, `0` is returned.

也請見 Unix 使用手🄵 *strtoul(3)*。

在 3.2 版被加入.

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

🄵穩定 ABI 的一部分. Convert the initial part of the string in `str` to an `long` value according to the given `base`, which must be between `2` and `36` inclusive, or be the special value `0`.

Same as *PyOS_strtoul()*, but return a `long` value instead and `LONG_MAX` on overflows.

也請見 Unix 使用手🄵 *strtol(3)*。

在 3.2 版被加入.

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

🄵穩定 ABI 的一部分. 將字串 `s` 轉🄵🄵 `double`，失敗時引發 Python 例外。接受的字串集合對應於 Python 的 `float()` 建構函式接受的字串集合，但 `s` 不得有前導或尾隨的空格。轉🄵與目前區域設定無關。

如果 `endptr` 🄵 `NULL`，則轉🄵整個字串。如果字串不是浮點數的有效表示，則引發 `ValueError` 🄵回傳 `-1.0`。

如果 endptr 不是 `NULL`，則盡可能轉🄵字串，🄵將 `*endptr` 設定🄵指向第一個未轉🄵的字元。如果字串的初始片段都不是浮點數的有效表示，則設定 `*endptr` 指向字串的開頭，引發 ValueError 🄵回傳 `-1.0`。

如果 `s` 表示的值太大而無法儲存在浮點數中（例如 `"1e500"` 在許多平台上都是這樣的字串），如果 `overflow_exception` 🄵 `NULL` 則回傳 `Py_INFINITY`（會帶有適當的符號）🄵且不設定任何例外。否則，`overflow_exception` 必須指向一個 Python 例外物件；引發該例外🄵回傳 `-1.0`。在這兩種情🄵下，將 `*endptr` 設定🄵指向轉🄵後的值之後的第一個字元。

如果轉🄵期間發生任何其他錯誤（例如記憶體不足的錯誤），請設定適當的 Python 例外🄵回傳 `-1.0`。

在 3.1 版被加入.

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

🄵穩定 ABI 的一部分. 使用提供的 *format_code*、*precision* 和 *flags* 將 `double` *val* 轉🄵🄵字串。

format_code 必須是 `'e'`、`'E'`、`'f'`、`'F'`、`'g'`、`'G'` 或 `'r'` 其中之一。對於 `'r'`，提供的 *precision* 必須🄵 0 🄵會被忽略。`'r'` 格式碼指定標準 `repr()` 格式。

flags 可以是零個或多個值 `Py_DTSF_SIGN`、`Py_DTSF_ADD_DOT_0` 或 `Py_DTSF_ALT`，會被聯集在一起：

- `Py_DTSF_SIGN` 代表總是在回傳的字串前面加上符號字元，即使 *val* 非負數。
- `Py_DTSF_ADD_DOT_0` 代表確保回傳的字串看起來不會像整數。
- `Py_DTSF_ALT` 代表要套用「備用的 (alternate)」格式化規則。有關詳細資訊，請參🄵*PyOS_snprintf()* `'#'` 的文件。

如果 *ptype* 是非 `NULL`，那🄵它指向的值將被設定🄵 `Py_DTST_FINITE`、`Py_DTST_INFINITE` 或 `Py_DTST_NAN` 其中之一，分🄵代表 *val* 是有限數、無限數或非數。

回傳值是指向 *buffer* 的指標，其中包含轉🄵後的字串，如果轉🄵失敗則回傳 `NULL`。呼叫者負責透過呼叫*PyMem_Free()* 來釋放回傳的字串。

在 3.1 版被加入.

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

不區分大小寫的字串比較。函式的作用方式幾乎與 `strcmp()` 相同，只是它忽略大小寫。

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

不區分大小寫的字串比較。函式的作用方式幾乎與 `strncmp()` 相同，只是它忽略大小寫。

7.8 PyHash API

另請參🄵*PyTypeObject.tp_hash* 成員和 numeric-hash。

type **Py_hash_t**

雜🄵值型🄵：有符號整數。

在 3.2 版被加入.

type **Py_uhash_t**

雜🄵值型🄵：無符號整數。

在 3.2 版被加入.

PyHASH_MODULUS

用於數值雜🄵方案的梅森質數 (Mersenne prime) `P = 2**n -1`。

在 3.13 版被加入.

PyHASH_BITS

PyHASH_MODULUS 中 `P` 的指數 `n`。

在 3.13 版被加入.

PyHASH_MULTIPLIER

用於字串和其他各種雜🄵的質數乘數 (prime multiplier)。

在 3.13 版被加入.

PyHASH_INF

正無窮大回傳的雜🄵值。

在 3.13 版被加入.

PyHASH_IMAG

用於🄵數🄵數部分的乘數。

在 3.13 版被加入.

type **PyHash_FuncDef**

PyHash_GetFuncDef() 所使用的雜🄵函式定義。

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

雜🄵函式。

const char ***name**

雜🄵函式名稱（UTF-8 編碼字串）。

const int **hash_bits**

雜🄵值的🄵部大小（以位元🄵單位）。

const int **seed_bits**

Seed 輸入的大小（以位元🄵單位）。

在 3.4 版被加入.

PyHash_FuncDef ***PyHash_GetFuncDef**(void)

取得雜🄵函式定義。

> **也參考**
>
> **PEP 456**「安全且可交替使用的雜🄵演算法 (Secure and interchangeable hash algorithm)」。

在 3.4 版被加入.

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

雜🄵指標值：將指標值作🄵整數處理（在🄵部轉型🄵 `uintptr_t`）。指標不會被取值 (dereference)。

此函式不會失敗：它不會回傳 `-1`。

在 3.13 版被加入.

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 必須大於或等於 `0`。

此函式總是會成功執行。

在 3.14 版被加入.

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

泛用雜🄵函式，旨在放入型🄵物件的 `tp_hash` 插槽中。其結果僅取🄵於物件的識🄵性。

在 CPython 中，它等價於*Py_HashPointer()*。

在 3.13 版被加入.

7.9 Reflection

PyObject ***PyEval_GetBuiltins**(void)

回傳值：借用參照。🄵穩定 ABI 的一部分. 在 3.13 版之後被🄵用: Use *PyEval_GetFrameBuiltins()* instead.

Return a dictionary of the builtins in the current execution frame, or the interpreter of the thread state if no frame is currently executing.

PyObject ***PyEval_GetLocals**(void)

回傳值：借用參照。🄵穩定 ABI 的一部分. 在 3.13 版之後被🄵用: Use either *PyEval_GetFrameLocals()* to obtain the same behaviour as calling `locals()` in Python code, or else call *PyFrame_GetLocals()* on the result of *PyEval_GetFrame()* to access the `f_locals` attribute of the currently executing frame.

Return a mapping providing access to the local variables in the current execution frame, or `NULL` if no frame is currently executing.

Refer to `locals()` for details of the mapping returned at different scopes.

As this function returns a *borrowed reference*, the dictionary returned for *optimized scopes* is cached on the frame object and will remain alive as long as the frame object does. Unlike *PyEval_GetFrameLocals()* and `locals()`, subsequent calls to this function in the same frame will update the contents of the cached dictionary to reflect changes in the state of the local variables rather than returning a new snapshot.

在 3.13 版的變更: As part of **PEP 667**, *PyFrame_GetLocals()*, `locals()`, and `FrameType.f_locals` no longer make use of the shared cache dictionary. Refer to the What's New entry for additional details.

PyObject ***PyEval_GetGlobals**(void)

回傳值：借用參照。🄵穩定 ABI 的一部分. 在 3.13 版之後被🄵用: Use *PyEval_GetFrameGlobals()* instead.

Return a dictionary of the global variables in the current execution frame, or `NULL` if no frame is currently executing.

PyFrameObject ***PyEval_GetFrame**(void)

回傳值：借用參照。🄵穩定 ABI 的一部分. Return the *attached thread state*'s frame, which is `NULL` if no frame is currently executing.

另請見*PyThreadState_GetFrame()*。

PyObject ***PyEval_GetFrameBuiltins**(void)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Return a dictionary of the builtins in the current execution frame, or the interpreter of the thread state if no frame is currently executing.

在 3.13 版被加入.

PyObject ***PyEval_GetFrameLocals**(void)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Return a dictionary of the local variables in the current execution frame, or `NULL` if no frame is currently executing. Equivalent to calling `locals()` in Python code.

To access `f_locals` on the current frame without making an independent snapshot in *optimized scopes*, call *PyFrame_GetLocals()* on the result of *PyEval_GetFrame()*.

在 3.13 版被加入.

PyObject ***PyEval_GetFrameGlobals**(void)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Return a dictionary of the global variables in the current execution frame, or `NULL` if no frame is currently executing. Equivalent to calling `globals()` in Python code.

在 3.13 版被加入.

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

🄵穩定 ABI 的一部分. Return the name of *func* if it is a function, class or instance object, else the name of *func*s type.

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

🄵穩定 ABI 的一部分. Return a description string, depending on the type of *func*. Return values include "()" for functions and methods, " constructor", " instance", and " object". Concatenated with the result of *PyEval_GetFuncName()*, the result will be a description of *func*.

7.10 編解碼器🄵🄵表和支援函式

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

🄵穩定 ABI 的一部分. 🄵🄵一個新的編解碼器搜尋函式。

作🄵副作用 (side effect)，這會嘗試載入 `encodings`（如果尚未完成），以確保它始終位於搜尋函式列表中的第一個。

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

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 取消🄵🄵編解碼器搜尋函式🄵清除🄵🄵表 (registry) 的快取。如果搜尋函式🄵未被🄵🄵，則不執行任何操作。成功回傳 0，發生錯誤時會引發例外🄵回傳 -1。

在 3.10 版被加入.

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

🄵穩定 ABI 的一部分. 回傳 `1` 或 `0`，具體取🄵於是否有給定 *encoding* 的已🄵🄵編解碼器。這個函式總會成功。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 基於泛用編解碼器的編碼 API。

object 被傳遞給以給定 *encoding* 所查找到的編碼器函式，🄵使用以 *errors* 定義的錯誤處理方法。*errors* 可以設🄵 `NULL` 來使用編解碼器定義的預設方法。如果找不到編碼器，則引發 `LookupError`。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 基於泛用編解碼器的解碼 API。

object 被傳遞給以給定 *encoding* 所查找到的解碼器函式，🄵使用以 *errors* 定義的錯誤處理方法。*errors* 可以設🄵 `NULL` 來使用編解碼器定義的預設方法。如果找不到編碼器，則引發 `LookupError`。

7.10.1 編解碼器查找 API

在以下函式中，查找的 *encoding* 字串的所有字元將轉🄵🄵小寫，這使得透過此機制查找的編碼可以不區分大小寫而更有效率。如果未找到編解碼器，則會設定 `KeyError` 🄵回傳 `NULL`。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 取得給定 *encoding* 的編碼器函式。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 取得給定 *encoding* 的解碼器函式。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 取得給定 *encoding* 的 `IncrementalEncoder` 物件。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 取得給定 *encoding* 的 `IncrementalDecoder` 物件。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 取得給定 *encoding* 的 `StreamReader` 工廠函式。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 取得給定 *encoding* 的 `StreamWriter` 工廠函式。

7.10.2 用於 Unicode 編碼錯誤處理程式的🄵🄵 API

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

🄵穩定 ABI 的一部分. 在給定的 *name* 下🄵🄵錯誤處理回呼 (callback) 函式 *error*。當編解碼器遇到無法編碼的字元/無法解碼的位元組🄵且 *name* 被指定🄵呼叫編碼/解碼函式時的錯誤參數時，將呼叫此回呼函式。

回呼取得單個引數，即 `UnicodeEncodeError`、`UnicodeDecodeError` 或 `UnicodeTranslateError` 的實例，其中包含關於有問題的字元或位元組序列及其在原始字串中偏移量的資訊（有關取得此資訊的函式，請參🄵*Unicode 例外物件*）。回呼必須引發給定的例外，或者回傳一個包含有問題序列的替🄵的二元組 (two-item tuple)，以及一個代表原始字串中應該被恢復的編碼/解碼偏移量的整數。

成功時回傳 `0`，錯誤時回傳 `-1`。

PyObject ***PyCodec_LookupError**(const char *name)

回傳值：新的參照。🄵穩定 ABI 的一部分. 查找 *name* 下已🄵🄵的錯誤處理回呼函式。作🄵一種特殊情🄵，可以傳遞 `NULL`，在這種情🄵下，將回傳"strict" 的錯誤處理回呼。

PyObject ***PyCodec_StrictErrors**(*PyObject* *exc)

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分. 引發 *exc* 作🄵例外。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 忽略 unicode 錯誤，跳過錯誤的輸入。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 將 unicode 編碼錯誤替🄵🄵 `?` 或 `U+FFFD`。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 將 unicode 編碼錯誤替🄵🄵 XML 字元參照。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 將 unicode 編碼錯誤替🄵🄵反斜🄵跳🄵（`\x`、`\u` 和 `\U`）。

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

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 將 unicode 編碼錯誤替🄵🄵 `\N{...}` 跳🄵。

在 3.5 版被加入.

7.11 PyTime C API

在 3.13 版被加入.

時鐘 C API 提供對系統時鐘的存取。它類似於 Python 的 `time` 模組。

對於與 `datetime` 模組相關的 C API，請參🄵*DateTime 物件*。

7.11.1 型🄵

type **PyTime_t**

以奈秒🄵單位的時間戳記或持續時長，以有符號的 64 位元整數表示。

時間戳記的參照點取🄵於所使用的時鐘。例如*PyTime_Time()* 回傳相對於 UNIX 紀元 (UNIX epoch) 的時間戳記。

支援的範圍約🄵 [-292.3 年； +292.3 年]。以 Unix 紀元 (1970 年 1 月 1 日) 🄵參照，支援的日期範圍約🄵 [1677-09-21; 2262-04-11]。確切的限制是以常數的形式公開出來：

PyTime_t **PyTime_MIN**

PyTime_t 的最小值。

PyTime_t **PyTime_MAX**

PyTime_t 的最大值。

7.11.2 時鐘函式

以下的函式接受一個指向 *PyTime_t* 的指標，🄵將其設定🄵特定時鐘的值。每個時鐘的詳細資訊紀🄵在相對應的 Python 函式🄵明文件中。

函式成功時會回傳 `0` 或在失敗時回傳 `-1`（🄵設定一個例外）。

在整數溢位時，它們會設定*PyExc_OverflowError* 例外，🄵將 `*result` 設定🄵夾在 `[PyTime_MIN; PyTime_MAX]` 範圍🄵的值。（在目前的系統上，整數溢位很可能是由於錯誤設定的系統時間所造成。）

如同任何其他 C API（除非另有指定），必須在具有*attached thread state* 的情🄵下呼叫函式。

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

讀取單調時鐘。請參🄵 `time.monotonic()` 取得此時鐘的重要詳細資訊。

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

讀取效能計數器。請參🄵 `time.perf_counter()` 以取得此時鐘的重要詳細資訊。

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

讀取「🄵上時鐘 (wall clock)」的時間。請參🄵 `time.time()` 以取得詳細資訊。

7.11.3 原始時鐘函式

類似於時鐘函式，但不會在出錯時設定例外，也不需要呼叫者擁有*attached thread state*。

成功時函式會回傳 `0`。

失敗時，它們會將 `*result` 設🄵 `0` 🄵回傳 `-1`，而不設定例外。要取得錯誤原因，請附加一個*thread state*，🄵呼叫常規（非 `Raw`）函式。請注意，常規函式可能會在 `Raw` 的函式失敗後成功。

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

類似於*PyTime_Monotonic()*，但不會在錯誤時設定例外，也不需要擁有*attached thread state*。

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

類似於*PyTime_PerfCounter()*，但不會在錯誤時設定例外，也不需要擁有*attached thread state*。

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

類似於*PyTime_Time()*，但不會在錯誤時設定例外，也不需要擁有*attached thread state*。

7.11.4 轉🄵函式

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

將時間戳記轉🄵🄵 C `double` 的秒數。

此函式不會失敗，但請注意 `double` 對於大數值的精確度有限。

7.12 對 Perf Map 的支援

在支援的平台上（截至撰寫本文時，僅限 Linux），runtime 可以利用 *perf map* 檔案使得外部分析工具（例如 perf）可以見到 Python 函式。正在運行的行程可能會在 `/tmp` 目🄵中建立一個檔案，其中包含可以將一段可執行程式碼對映到名稱的各個條目。此介面在 Linux Perf 工具的文件中有被描述。

在 Python 中，這些輔助 API 可以被依賴於運行期間 (on the fly) 🄵生機器碼的函式庫和功能所使用。

請注意，這些 API 不需要持有*attached thread state*。

int **PyUnstable_PerfMapState_Init**(void)

> 這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

打開 `/tmp/perf-$pid.map` 檔案，除非它已經打開，🄵建立一個鎖以確保執行緒安全地 (thread-safe) 寫入該檔案（前提是寫入是透過*PyUnstable_WritePerfMapEntry()* 完成的）。通常不需要明確地呼叫它；只需使用*PyUnstable_WritePerfMapEntry()* 它就會在首次呼叫時初始化狀態。

建立/打開 perf map 檔案成功時回傳 `0`，失敗時回傳 `-1`，建立鎖時失敗則回傳 `-2`。檢查 `errno` 以取得更多造成失敗的資訊。

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

> 這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

將單一條目寫入 `/tmp/perf-$pid.map` 檔案。此函式是執行緒安全的。以下是一個條目的範例：

```
# 位址         大小   名稱
7f3529fcf759 b     py::bar:/run/t.py
```

如果尚未開🄵 perf map 檔案，將在寫入條目之前呼叫*PyUnstable_PerfMapState_Init()*。成功時回傳 `0`，失敗時回傳與*PyUnstable_PerfMapState_Init()* 失敗時相同的錯誤碼。

void **PyUnstable_PerfMapState_Fini**(void)

> 這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

關閉由*PyUnstable_PerfMapState_Init()* 開🄵的 perf map 檔案，這是在直譯器關閉期間由 runtime 本身呼叫的。一般來🄵，除了處理 forking 等特定場景外，不應該明確地呼叫它。

CHAPTER 8

抽象物件層 (Abstract Objects Layer)

本章中的函式與 Python 物件相互作用，無論其型🄵、或具有廣泛類🄵的物件型🄵（例如所有數值型🄵或所有序列型🄵）。當使用於不適用的物件型🄵時，他們會引發一個 Python 🄵常 (exception)。

這些函式是不可能用於未正確初始化的物件（例如一個由 *PyList_New()* 建立的 list 物件），而其中的項目🄵有被設🄵一些非 NULL 的值。

8.1 物件協定

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

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Get a *strong reference* to a constant.

如果 *constant_id* 無效，則設定一個例外🄴回傳 NULL。

constant_id 必須是這些常數識🄵字之一：

常數識🄵字	數值	回傳物件
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.

在 3.13 版被加入.

在 CPython 中，所有這些常數都是不滅的。

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

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 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.

在 3.13 版被加入.

PyObject ***Py_NotImplemented**

The `NotImplemented` singleton, used to signal that an operation is not implemented for the given type combination.

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)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 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`.

在 3.13 版被加入.

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

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 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**.

在 3.13 版被加入.

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

🄵穩定 ABI 的一部分. Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This function always succeeds.

> **備🄵**
>
> 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)

🄵穩定 ABI 的一部分. 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**.

> **備🄵**
>
> 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Retrieve an attribute named *attr_name* from object *o*. Returns the attribute value on success, or `NULL` on failure. This is the equivalent of the Python expression `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)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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);

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 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`.

在 3.13 版被加入.

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

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 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*`.

在 3.13 版被加入.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. Generic attribute getter function that is meant to be put into a type object's `tp_getattro` slot. It looks for a descriptor in the dictionary of classes in the object's MRO as well as an attribute in the object's `__dict__` (if present). As outlined in descriptors, data descriptors take preference over instance attributes, while non-data descriptors don't. Otherwise, an `AttributeError` is raised.

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

🄵穩定 ABI 的一部分. Set the value of the attribute named *attr_name*, for object *o*, to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o.attr_name = v`.

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

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

🄵穩定 ABI 的一部分. 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*`.

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

🄵穩定 ABI 的一部分. Generic attribute setter and deleter function that is meant to be put into a type object's `tp_setattro` slot. It looks for a data descriptor in the dictionary of classes in the object's MRO, and if found it takes preference over setting or deleting the attribute in the instance dictionary. Otherwise, the attribute is set or deleted in the object's `__dict__` (if present). On success, `0` is returned, otherwise an `AttributeError` is raised and `-1` is returned.

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

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Delete attribute named *attr_name*, for object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o.attr_name`.

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

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.10* 版本開始. A generic implementation for the getter of a `__dict__` descriptor. It creates the dictionary if necessary.

This function may also be called to get the `__dict__` of the object *o*. Pass `NULL` for *context* when calling it. Since this function may need to allocate memory for the dictionary, it may be more efficient to call `PyObject_GetAttr()` when accessing an attribute on the object.

在失敗時，會回傳 `NULL` 🄵設定例外。

在 3.3 版被加入.

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

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. A generic implementation for the setter of a `__dict__` descriptor. This implementation does not allow the dictionary to be deleted.

在 3.3 版被加入.

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

Return a pointer to `__dict__` of the object *obj*. If there is no `__dict__`, return `NULL` without setting an exception.

This function may need to allocate memory for the dictionary, so it may be more efficient to call *PyObject_GetAttr()* when accessing an attribute on the object.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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.

備🄵

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)

🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Compute a string representation of object *o*. Returns the string representation on success, `NULL` on failure. This is the equivalent of the Python expression `repr(o)`. Called by the `repr()` built-in function.

在 3.4 版的變更: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

PyObject ***PyObject_ASCII**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. As *PyObject_Repr()*, compute a string representation of object *o*, but escape the non-ASCII characters in the string returned by *PyObject_Repr()* with `\x`, `\u` or `\U` escapes. This generates a string similar to that returned by *PyObject_Repr()* in Python 2. Called by the `ascii()` built-in function.

PyObject ***PyObject_Str**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. Compute a string representation of object *o*. Returns the string representation on success, `NULL` on failure. This is the equivalent of the Python expression `str(o)`. Called by the `str()` built-in function and, therefore, by the `print()` function.

在 3.4 版的變更: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

PyObject ***PyObject_Bytes**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. Compute a bytes representation of object *o*. `NULL` is returned on failure and a bytes object on success. This is equivalent to the Python expression `bytes(o)`, when *o* is not an integer. Unlike `bytes(o)`, a TypeError is raised when *o* is an integer instead of a zero-initialized bytes object.

int **PyObject_IsSubclass**(*PyObject* *derived, *PyObject* *cls)

🄵穩定 ABI 的一部分. Return `1` if the class *derived* is identical to or derived from the class *cls*, otherwise return `0`. In case of an error, return `-1`.

If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `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)

🄵穩定 ABI 的一部分. Return `1` if *inst* is an instance of the class *cls* or a subclass of *cls*, or `0` if not. On error, returns `-1` and sets an exception.

If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `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)

🄵穩定 ABI 的一部分. Compute and return the hash value of an object *o*. On failure, return `-1`. This is the equivalent of the Python expression `hash(o)`.

在 3.2 版的變更: The return type is now Py_hash_t. This is a signed integer the same size as *Py_ssize_t*.

Py_hash_t **PyObject_HashNotImplemented**(*PyObject* *o)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. Returns `1` if the object *o* is considered to be true, and `0` otherwise. This is equivalent to the Python expression `not not o`. On failure, return `-1`.

int **PyObject_Not**(*PyObject* *o)

🄵穩定 ABI 的一部分. Returns `0` if the object *o* is considered to be true, and `1` otherwise. This is equivalent to the Python expression `not o`. On failure, return `-1`.

PyObject ***PyObject_Type**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

Return non-zero if the object *o* is of type *type* or a subtype of *type*, and `0` otherwise. Both parameters must be non-`NULL`.

Py_ssize_t **PyObject_Size**(*PyObject* *o)

Py_ssize_t **PyObject_Length**(*PyObject* *o)

🄵穩定 ABI 的一部分. Return the length of object *o*. If the object *o* provides either the sequence and mapping protocols, the sequence length is returned. On error, `-1` is returned. This is the equivalent to the Python expression `len(o)`.

Py_ssize_t **PyObject_LengthHint**(*PyObject* *o, *Py_ssize_t* defaultvalue)

Return an estimated length for the object *o*. First try to return its actual length, then an estimate using `__length_hint__()`, and finally return the default value. On error return `-1`. This is the equivalent to the Python expression `operator.length_hint(o, defaultvalue)`.

在 3.4 版被加入.

PyObject ***PyObject_GetItem**(*PyObject* *o, *PyObject* *key)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return element of *o* corresponding to the object *key* or `NULL` on failure. This is the equivalent of the Python expression `o[key]`.

int **PyObject_SetItem**(*PyObject* *o, *PyObject* *key, *PyObject* *v)

🄵穩定 ABI 的一部分. Map the object *key* to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o[key] = v`. This function *does not* steal a reference to *v*.

int **PyObject_DelItem**(*PyObject* *o, *PyObject* *key)

🄵穩定 ABI 的一部分. Remove the mapping for the object *key* from the object *o*. Return `-1` on failure. This is equivalent to the Python statement `del o[key]`.

int **PyObject_DelItemString**(*PyObject* *o, const char *key)

🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. This is equivalent to the Python expression `dir(o)`, returning a (possibly empty) list of strings appropriate for the object argument, or `NULL` if there was an error. If the argument is `NULL`, this is like the Python `dir()`, returning the names of the current locals; in this case, if no execution frame is active then `NULL` is returned but *PyErr_Occurred()* will return false.

PyObject ***PyObject_GetIter**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. This is equivalent to the Python expression `iter(o)`. It returns a new iterator for the object argument, or the object itself if the object is already an iterator. Raises `TypeError` and returns `NULL` if the object cannot be iterated.

PyObject ***PyObject_SelfIter**(*PyObject* *obj)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.10* 版本開始. This is the equivalent to the Python expression `aiter(o)`. Takes an `AsyncIterable` object and returns an `AsyncIterator` for it. This is typically a new iterator but if the argument is an `AsyncIterator`, this returns itself. Raises `TypeError` and returns `NULL` if the object cannot be iterated.

在 3.10 版被加入.

void ***PyObject_GetTypeData**(*PyObject* *o, *PyTypeObject* *cls)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 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.

錯誤時設定一個例外🄵回傳 `NULL`。

在 3.12 版被加入.

Py_ssize_t **PyType_GetTypeDataSize**(*PyTypeObject* *cls)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 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.

錯誤時設定一個例外🄵回傳一個負值。

在 3.12 版被加入.

void ***PyObject_GetItemData**(*PyObject* *o)

Get a pointer to per-item data for a class with `Py_TPFLAGS_ITEMS_AT_END`.

錯誤時設定一個例外🄵回傳 `NULL`。如果 *o* 🄵有設定`Py_TPFLAGS_ITEMS_AT_END`，則會引發 `TypeError`。

在 3.12 版被加入.

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.

在 3.13 版被加入.

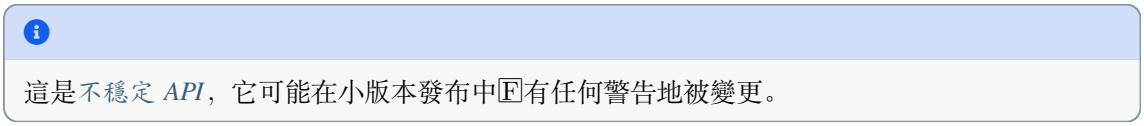
void **PyObject_ClearManagedDict**(*PyObject* *obj)

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.

在 3.13 版被加入.

int **PyUnstable_Object_EnableDeferredRefcount**(*PyObject* *obj)



這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

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.

在 3.14 版被加入.

int **PyUnstable_Object_IsUniqueReferencedTemporary**(*PyObject* *obj)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

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)
```

另請參🄵*`Py_REFCNT()`* 函式。

在 3.14 版被加入.

int **`PyUnstable_IsImmortal`**(*PyObject* *obj)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

This function returns non-zero if *obj* is *immortal*, and zero otherwise. This function cannot fail.

備🄵

Objects that are immortal in one CPython version are not guaranteed to be immortal in another.

在 3.14 版被加入.

int **`PyUnstable_TryIncRef`**(*PyObject* *obj)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

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);
}
```

在 3.14 版被加入.

void **PyUnstable_EnableTryIncRef**(*PyObject* *obj)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

Enables subsequent uses of `PyUnstable_TryIncRef()` on *obj*. The caller must hold a *strong reference* to *obj* when calling this.

在 3.14 版被加入.

int **PyUnstable_Object_IsUniquelyReferenced**(*PyObject* *op)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

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.

在 3.14 版被加入.

8.2 呼叫協定 (Call Protocol)

CPython 支援兩種不同的呼叫協定：*tp_call* 和 vectorcall（向量呼叫）。

8.2.1 *tp_call* 協定

設定`tp_call` 的類🄵之實例都是可呼叫的。該擴充槽 (slot) 的簽章🄵：

```
PyObject *tp_call(PyObject *callable, PyObject *args, PyObject *kwargs);
```

要達成一個呼叫會使用一個 tuple（元組）表示位置引數、一個 dict 表示關鍵字引數，類似於 Python 程式碼中的 `callable(*args, **kwargs)`。*args* 必須不🄵 NULL（如果🄵有引數，會使用一個空 tuple），但如果🄵有關鍵字引數，*kwargs* 可以是 *NULL*。

這個慣例不僅會被 *tp_call* 使用，`tp_new` 和`tp_init` 也這樣傳遞引數。

使用`PyObject_Call()` 或其他呼叫 *API* 來呼叫一個物件。

8.2.2 Vectorcall 協定

在 3.9 版被加入.

Vectorcall 協定是在 **PEP 590** 被引入的，它是使函式呼叫更加有效率的附加協定。

經驗法則上，如果可呼叫物件有支援，CPython 於🄵部呼叫中會更傾向使用 vectorcall。然而，這🄵不是一個硬性規定。此外，有些第三方擴充套件會直接使用 *tp_call*（而不是使用`PyObject_Call()`）。因此，一個支援 vectorcall 的類🄵也必須實作`tp_call`。此外，無論使用哪種協定，可呼叫物件的行🄵都必須是相同的。要達成這個目的的推薦做法是將`tp_call` 設定🄵`PyVectorcall_Call()`。這值得一再提醒：

警告

一個支援 vectorcall 的類🄵**必須**也實作具有相同語義的`tp_call`。

在 3.12 版的變更: 當一個類🄵的 `__call__()` 方法被重新賦值時，*Py_TPFLAGS_HAVE_VECTORCALL* 旗標現在會被移除。(這會在🄵部設定*tp_call*，因此可能會使它與 vectorcall 函式有不同的行🄵。) 在較早的 Python 版本中，vectorcall 應該只與*immutable* 或🄵態型🄵一起使用。

如果一個類🄵的 vectorcall 比 *tp_call* 慢，就不應該實作 vectorcall。例如，如果被呼叫者需要將引數轉🄵🄵 args tuple（引數元組）和 kwargs dict（關鍵字引數字典），那🄵實作 vectorcall 就🄵有意義。

類🄵可以透過🄵用*Py_TPFLAGS_HAVE_VECTORCALL* 旗標🄵將*tp_vectorcall_offset* 設定🄵物件結構中有出現 *vectorcallfunc* 的 offset 來實作 vectorcall 協定。這是一個指向具有以下簽章之函式的指標:

typedef *PyObject* *(***vectorcallfunc**)(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始.

- *callable* 是指被呼叫的物件。
- ***args* 是一個 C 語言陣列 (array)，包含位置引數與後面關鍵字引數的值。如果🄵有引數，這個值可以是 *NULL*。**
- ***nargsf* 是位置引數的數量加上可能會有的*PY_VECTORCALL_ARGUMENTS_OFFSET* 旗標。如果要從 *nargsf* 獲得實際的位置引數數量，請使用*PyVectorcall_NARGS()*。**
- ***kwnames* 是一個包含所有關鍵字引數名稱的 tuple；🄵句話🄵，就是 kwargs 字典的鍵。這些名字必須是字串（`str` 或其子類🄵的實例），🄵且它們必須是不重🄵的。如果🄵有關鍵字引數，那🄵 *kwnames* 可以用 *NULL* 代替。**

PY_VECTORCALL_ARGUMENTS_OFFSET

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 如果在 vectorcall 的 *nargsf* 引數中設定了此旗標，則允許被呼叫者臨時更改 `args[-1]` 的值。🄵句話🄵，*args* 指向向量中的引數 1（不是 0）。被呼叫方必須在回傳之前還原 `args[-1]` 的值。

對於*PyObject_VectorcallMethod()*，這個旗標的改變意味著可能是 `args[0]` 被改變。

當可以以幾乎無代價的方式（無需🄵據額外的記憶體）來達成，那🄵會推薦呼叫者使用*PY_VECTORCALL_ARGUMENTS_OFFSET*。這樣做會讓如 bound method（🄵結方法）之類的可呼叫函式非常有效地繼續向前呼叫（這類函式包含一個在首位的 *self* 引數)。

在 3.8 版被加入.

要呼叫一個實作了 vectorcall 的物件，請就像其他可呼叫物件一樣使用呼叫 *API* 中的函式。*PyObject_Vectorcall()* 通常是最有效率的。

遞🄵控制

在使用 *tp_call* 時，被呼叫者不必擔心遞🄵: CPython 對於使用 *tp_call* 的呼叫會使用*Py_EnterRecursiveCall()* 和*Py_LeaveRecursiveCall()*。

🄵保證效率，這不適用於使用 vectorcall 的呼叫：被呼叫方在需要時應當使用 *Py_EnterRecursiveCall* 和 *Py_LeaveRecursiveCall*。

Vectorcall 支援 API

Py_ssize_t **PyVectorcall_NARGS**(size_t nargsf)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 給定一個 vectorcall *nargsf* 引數，回傳引數的實際數量。目前等同於:

```
(Py_ssize_t)(nargsf & ~PY_VECTORCALL_ARGUMENTS_OFFSET)
```

然而，應使用 `PyVectorcall_NARGS` 函式以便將來需要擴充。

在 3.8 版被加入.

vectorcallfunc **PyVectorcall_Function**(*PyObject* *op)

如果 *op* 不支援 vectorcall 協定（因🄵型🄵不支援或特定實例不支援），就回傳 *NULL*。否則，回傳儲存在 *op* 中的 vectorcall 函式指標。這個函式不會引發例外。

這大多在檢查 *op* 是否支援 vectorcall 時能派上用場，可以透過檢查 `PyVectorcall_Function(op) != NULL` 來達成。

在 3.9 版被加入.

PyObject ***PyVectorcall_Call**(*PyObject* *callable, *PyObject* *tuple, *PyObject* *dict)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 呼叫 *callable* 的*vectorcallfunc*，其位置引數和關鍵字引數分🄵以 tuple 和 dict 格式給定。

這是一個專門函式，其目的是被放入*tp_call* 擴充槽或是用於 `tp_call` 的實作。它不會檢查*Py_TPFLAGS_HAVE_VECTORCALL* 旗標🄵且它不會退回 (fall back) 使用 `tp_call`。

在 3.8 版被加入.

8.2.3 物件呼叫 API

有多個函式可被用來呼叫 Python 物件。各個函式會將其引數轉🄵🄵被呼叫物件所支援的慣用形式–可以是 *tp_call* 或 vectorcall。🄵了🄵可能🄵少轉🄵的進行，請選擇一個適合你所擁有資料格式的函式。

下表總結了可用的函式；請參🄵各個🄵明文件以🄵解詳情。

函式	callable	args	kwargs
PyObject_Call()	`PyObject *`	tuple	dict/`NULL`
PyObject_CallNoArgs()	`PyObject *`	---	---
PyObject_CallOneArg()	`PyObject *`	一個物件	---
PyObject_CallObject()	`PyObject *`	tuple/`NULL`	---
PyObject_CallFunction()	`PyObject *`	format	---
PyObject_CallMethod()	物件 + `char*`	format	---
PyObject_CallFunctionObjArgs()	`PyObject *`	可變引數	---
PyObject_CallMethodObjArgs()	物件 + 名稱	可變引數	---
PyObject_CallMethodNoArgs()	物件 + 名稱	---	---
PyObject_CallMethodOneArg()	物件 + 名稱	一個物件	---
PyObject_Vectorcall()	`PyObject *`	vectorcall	vectorcall
PyObject_VectorcallDict()	`PyObject *`	vectorcall	dict/`NULL`
PyObject_VectorcallMethod()	引數 + 名稱	vectorcall	vectorcall

PyObject ***PyObject_Call**(*PyObject* *callable, *PyObject* *args, *PyObject* *kwargs)

回傳值：新的參照。🄵穩定 ABI 的一部分. 呼叫一個可呼叫的 Python 物件 *callable*，附帶由 tuple *args* 所給定的引數及由字典 *kwargs* 所給定的關鍵字引數。

args 必須不🄵 *NULL*；如果不需要引數，請使用一個空 tuple。如果不需要關鍵字引數，則 *kwargs* 可以🄵 *NULL*。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

這等價於 Python 運算式 `callable(*args, **kwargs)`。

PyObject ***PyObject_CallNoArgs**(*PyObject* *callable)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 呼叫一個可呼叫的 Python 物件 *callable* 🄵不附帶任何引數。這是不帶引數呼叫 Python 可呼叫物件的最有效方式。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

在 3.9 版被加入.

PyObject ***PyObject_CallOneArg**(*PyObject* *callable, *PyObject* *arg)

回傳值：新的參照。 呼叫一個可呼叫的 Python 物件 *callable* 🄵附帶正好一個位置引數 *arg* 而🄵有關鍵字引數。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

在 3.9 版被加入.

PyObject ***PyObject_CallObject**(*PyObject* *callable, *PyObject* *args)

回傳值：新的參照。🄵穩定 ABI 的一部分. 呼叫一個可呼叫的 Python 物件 *callable*，附帶由 tuple *args* 所給定的引數。如果不需要傳入引數，則 *args* 可以🄵 *NULL*。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

這等價於 Python 運算式 `callable(*args)`。

PyObject ***PyObject_CallFunction**(*PyObject* *callable, const char *format, ...)

回傳值：新的參照。🄵穩定 ABI 的一部分. 呼叫一個可呼叫的 Python 物件 *callable*，附帶數量可變的 C 引數。這些 C 引數使用*Py_BuildValue()* 風格的格式字串來描述。格式可以🄵 *NULL*，表示🄵有提供任何引數。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

這等價於 Python 運算式 `callable(*args)`。

注意，如果你只傳入 *PyObject** 引數，則*PyObject_CallFunctionObjArgs()* 是另一個更快速的選擇。

在 3.4 版的變更: 這個 *format* 的型🄵已從 `char *` 更改。

PyObject ***PyObject_CallMethod**(*PyObject* *obj, const char *name, const char *format, ...)

回傳值：新的參照。🄵穩定 ABI 的一部分. 呼叫 *obj* 物件中名🄵 *name* 的 method 🄵附帶數量可變的 C 引數。這些 C 引數由*Py_BuildValue()* 格式字串來描述，🄵應當生成一個 tuple。

格式可以🄵 *NULL*，表示🄵有提供任何引數。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

這等價於 Python 運算式 `obj.name(arg1, arg2, ...)`。

注意，如果你只傳入 *PyObject** 引數，則*PyObject_CallMethodObjArgs()* 是另一個更快速的選擇。

在 3.4 版的變更: *name* 和 *format* 的型🄵已從 `char *` 更改。

PyObject ***PyObject_CallFunctionObjArgs**(*PyObject* *callable, ...)

回傳值：新的參照。🄵穩定 ABI 的一部分. 呼叫一個可呼叫的 Python 物件 *callable*，附帶數量可變的 *PyObject** 引數。這些引數是以位置在 *NULL* 後面、數量可變的參數來提供。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

這等價於 Python 運算式 `callable(arg1, arg2, ...)`。

PyObject ***PyObject_CallMethodObjArgs**(*PyObject* *obj, *PyObject* *name, ...)

回傳值：新的參照。🄵穩定 ABI 的一部分. 呼叫 Python 物件 *obj* 中的一個 method，其中 method 名稱由 *name* 中的 Python 字串物件給定。被呼叫時會附帶數量可變的 *PyObject** 引數。這些引數是以位置在 *NULL* 後面、且數量可變的參數來提供。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

PyObject ***PyObject_CallMethodNoArgs**(*PyObject* *obj, *PyObject* *name)

不附帶任何引數地呼叫 Python 物件 *obj* 中的一個 method，其中 method 名稱由 *name* 中的 Python 字串物件給定。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

在 3.9 版被加入.

PyObject ***PyObject_CallMethodOneArg**(*PyObject* *obj, *PyObject* *name, *PyObject* *arg)

附帶一個位置引數 *arg* 地呼叫 Python 物件 *obj* 中的一個 method，其中 method 名稱由 *name* 中的 Python 字串物件給定。

成功時回傳結果，或在失敗時引發一個例外🄴回傳 *NULL*。

在 3.9 版被加入.

PyObject ***PyObject_Vectorcall**(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 呼叫一個可呼叫的 Python 物件 *callable*。附帶引數與`vectorcallfunc` 的相同。如果 *callable* 支援*vectorcall*，則它會直接呼叫存放在 *callable* 中的 vectorcall 函式。

成功時回傳結果，或在失敗時引發一個例外🄴回傳 *NULL*。

在 3.9 版被加入.

PyObject ***PyObject_VectorcallDict**(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwdict)

附帶與在*vectorcall* 協定中傳入的相同位置引數來呼叫 *callable*，但會加上以字典 *kwdict* 格式傳入的關鍵字引數。*args* 陣列將只包含位置引數。

無論🄴部使用了哪一種協定，都會需要進行引數的轉🄴。因此，此函式應該只有在呼叫方已經擁有一個要作🄴關鍵字引數的字典、但🄴有作🄴位置引數的 tuple 時才被使用。

在 3.9 版被加入.

PyObject ***PyObject_VectorcallMethod**(*PyObject* *name, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 使用 vectorcall 呼叫慣例來呼叫一個 method。method 的名稱以 Python 字串 *name* 的格式給定。被呼叫 method 的物件🄴 *args[0]*，而 *args* 陣列從 *args[1]* 開始的部分則代表呼叫的引數。必須傳入至少一個位置引數。*nargsf* 🄴包括 *args[0]* 在🄴的位置引數的數量，如果 `args[0]` 的值可能被臨時改變則要再加上`PY_VECTORCALL_ARGUMENTS_OFFSET`。關鍵字引數可以像在`PyObject_Vectorcall()` 中一樣被傳入。

如果物件具有`Py_TPFLAGS_METHOD_DESCRIPTOR` 特性，這將以完整的 *args* 向量作🄴引數來呼叫 unbound method（未🄴結方法）物件。

成功時回傳結果，或在失敗時引發一個例外🄴回傳 *NULL*。

在 3.9 版被加入.

8.2.4 呼叫支援 API

int **PyCallable_Check**(*PyObject* *o)

🄵穩定 ABI 的一部分. 判定物件 *o* 是否🄴可呼叫的。如果物件是可呼叫物件則回傳 `1`，其他情🄴回傳 `0`。這個函式不會呼叫失敗。

8.3 數字協定

int **PyNumber_Check**(*PyObject* *o)

🄵穩定 ABI 的一部分. Returns `1` if the object *o* provides numeric protocols, and false otherwise. This function always succeeds.

在 3.8 版的變更: Returns `1` if *o* is an index integer.

PyObject ***PyNumber_Add**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the result of adding *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 + o2`.

PyObject ***PyNumber_Subtract**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the result of subtracting *o2* from *o1*, or `NULL` on failure. This is the equivalent of the Python expression `o1 - o2`.

PyObject ***PyNumber_Multiply**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the result of multiplying *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 * o2`.

PyObject ***PyNumber_MatrixMultiply**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Returns the result of matrix multiplication on *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 @ o2`.

在 3.5 版被加入.

PyObject ***PyNumber_FloorDivide**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return the floor of *o1* divided by *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 // o2`.

PyObject ***PyNumber_TrueDivide**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or `NULL` on failure. The return value is "approximate" because binary floating-point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating-point value when passed two integers. This is the equivalent of the Python expression `o1 / o2`.

PyObject ***PyNumber_Remainder**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the remainder of dividing *o1* by *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 % o2`.

PyObject ***PyNumber_Divmod**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. See the built-in function `divmod()`. Returns `NULL` on failure. This is the equivalent of the Python expression `divmod(o1, o2)`.

PyObject ***PyNumber_Power**(*PyObject* *o1, *PyObject* *o2, *PyObject* *o3)

回傳值：新的參照。🄵穩定 ABI 的一部分. See the built-in function `pow()`. Returns `NULL` on failure. This is the equivalent of the Python expression `pow(o1, o2, o3)`, where *o3* is optional. If *o3* is to be ignored, pass *Py_None* in its place (passing `NULL` for *o3* would cause an illegal memory access).

PyObject ***PyNumber_Negative**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the negation of *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `-o`.

PyObject ***PyNumber_Positive**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `+o`.

PyObject ***PyNumber_Absolute**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the absolute value of *o*, or `NULL` on failure. This is the equivalent of the Python expression `abs(o)`.

PyObject ***PyNumber_Invert**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the bitwise negation of *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `~o`.

PyObject ***PyNumber_Lshift**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the result of left shifting *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 << o2`.

PyObject ***PyNumber_Rshift**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the result of right shifting *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 >> o2`.

PyObject ***PyNumber_And**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the "bitwise and" of *o1* and *o2* on success and `NULL` on failure. This is the equivalent of the Python expression `o1 & o2`.

PyObject ***PyNumber_Xor**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the "bitwise exclusive or" of *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 ^ o2`.

PyObject ***PyNumber_Or**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the "bitwise or" of *o1* and *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 | o2`.

PyObject ***PyNumber_InPlaceAdd**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the result of adding *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 += o2`.

PyObject ***PyNumber_InPlaceSubtract**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the result of subtracting *o2* from *o1*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 -= o2`.

PyObject ***PyNumber_InPlaceMultiply**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the result of multiplying *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 *= o2`.

PyObject ***PyNumber_InPlaceMatrixMultiply**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Returns the result of matrix multiplication on *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 @= o2`.

在 3.5 版被加入.

PyObject ***PyNumber_InPlaceFloorDivide**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the mathematical floor of dividing *o1* by *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 //= o2`.

PyObject ***PyNumber_InPlaceTrueDivide**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or `NULL` on failure. The return value is "approximate" because binary floating-point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating-point value when passed two integers. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 /= o2`.

PyObject ***PyNumber_InPlaceRemainder**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the remainder of dividing *o1* by *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 %= o2`.

PyObject ***PyNumber_InPlacePower**(*PyObject* *o1, *PyObject* *o2, *PyObject* *o3)

回傳值：新的參照。🄵穩定 ABI 的一部分. See the built-in function `pow()`. Returns `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 **= o2` when o3 is *Py_None*, or an in-place variant of `pow(o1, o2, o3)` otherwise. If *o3* is to be ignored, pass *Py_None* in its place (passing `NULL` for *o3* would cause an illegal memory access).

PyObject ***PyNumber_InPlaceLshift**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the result of left shifting *o1* by *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 <<= o2`.

PyObject ***PyNumber_InPlaceRshift**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the result of right shifting *o1* by *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 >>= o2`.

PyObject ***PyNumber_InPlaceAnd**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the "bitwise and" of *o1* and *o2* on success and `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 &= o2`.

PyObject ***PyNumber_InPlaceXor**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the "bitwise exclusive or" of *o1* by *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 ^= o2`.

PyObject ***PyNumber_InPlaceOr**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the "bitwise or" of *o1* and *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 |= o2`.

PyObject ***PyNumber_Long**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the *o* converted to an integer object on success, or `NULL` on failure. This is the equivalent of the Python expression `int(o)`.

PyObject ***PyNumber_Float**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the *o* converted to a float object on success, or `NULL` on failure. This is the equivalent of the Python expression `float(o)`.

PyObject ***PyNumber_Index**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the *o* converted to a Python int on success or `NULL` with a `TypeError` exception raised on failure.

在 3.10 版的變更: The result always has exact type `int`. Previously, the result could have been an instance of a subclass of `int`.

PyObject ***PyNumber_ToBase**(*PyObject* *n, int base)

回傳值：新的參照。🄵穩定 ABI 的一部分. Returns the integer *n* converted to base *base* as a string. The *base* argument must be one of 2, 8, 10, or 16. For base 2, 8, or 16, the returned string is prefixed with a base marker of `'0b'`, `'0o'`, or `'0x'`, respectively. If *n* is not a Python int, it is converted with *PyNumber_Index()* first.

Py_ssize_t **PyNumber_AsSsize_t**(*PyObject* *o, *PyObject* *exc)

🄵穩定 ABI 的一部分. Returns *o* converted to a *Py_ssize_t* value if *o* can be interpreted as an integer. If the call fails, an exception is raised and `-1` is returned.

If *o* can be converted to a Python int but the attempt to convert to a *Py_ssize_t* value would raise an `OverflowError`, then the *exc* argument is the type of exception that will be raised (usually `IndexError` or `OverflowError`). If *exc* is `NULL`, then the exception is cleared and the value is clipped to `PY_SSIZE_T_MIN` for a negative integer or `PY_SSIZE_T_MAX` for a positive integer.

int **PyIndex_Check**(*PyObject* *o)

🄵穩定 ABI 的一部分 自 *3.8* 版本開始. Returns `1` if *o* is an index integer (has the `nb_index` slot of the `tp_as_number` structure filled in), and `0` otherwise. This function always succeeds.

8.4 序列協定

int **PySequence_Check**(*PyObject* *o)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. Returns the number of objects in sequence *o* on success, and `-1` on failure. This is equivalent to the Python expression `len(o)`.

PyObject ***PySequence_Concat**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return the concatenation of *o1* and *o2* on success, and `NULL` on failure. This is the equivalent of the Python expression `o1 + o2`.

PyObject ***PySequence_Repeat**(*PyObject* *o, *Py_ssize_t* count)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return the result of repeating sequence object *o* *count* times, or `NULL` on failure. This is the equivalent of the Python expression `o * count`.

PyObject ***PySequence_InPlaceConcat**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return the concatenation of *o1* and *o2* on success, and `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python expression `o1 += o2`.

PyObject ***PySequence_InPlaceRepeat**(*PyObject* *o, *Py_ssize_t* count)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return the result of repeating sequence object *o* *count* times, or `NULL` on failure. The operation is done *in-place* when *o* supports it. This is the equivalent of the Python expression `o *= count`.

PyObject ***PySequence_GetItem**(*PyObject* *o, *Py_ssize_t* i)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return the *i*th element of *o*, or `NULL` on failure. This is the equivalent of the Python expression `o[i]`.

PyObject ***PySequence_GetSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return the slice of sequence object *o* between *i1* and *i2*, or `NULL` on failure. This is the equivalent of the Python expression `o[i1:i2]`.

int **PySequence_SetItem**(*PyObject* *o, *Py_ssize_t* i, *PyObject* *v)

🄵穩定 ABI 的一部分. Assign object *v* to the *i*th element of *o*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o[i] = v`. This function *does not* steal a reference to *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)

🄵穩定 ABI 的一部分. Delete the *i*th element of object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o[i]`.

int **PySequence_SetSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2, *PyObject* *v)

🄵穩定 ABI 的一部分. Assign the sequence object *v* to the slice in sequence object *o* from *i1* to *i2*. This is the equivalent of the Python statement `o[i1:i2] = v`.

int **PySequence_DelSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

🄵穩定 ABI 的一部分. Delete the slice in sequence object *o* from *i1* to *i2*. Returns `-1` on failure. This is the equivalent of the Python statement `del o[i1:i2]`.

Py_ssize_t **PySequence_Count**(*PyObject* *o, *PyObject* *value)

🄵穩定 ABI 的一部分. Return the number of occurrences of *value* in *o*, that is, return the number of keys for which `o[key] == value`. On failure, return `-1`. This is equivalent to the Python expression `o.count(value)`.

int **PySequence_Contains**(*PyObject* *o, *PyObject* *value)

🄵穩定 ABI 的一部分. Determine if *o* contains *value*. If an item in *o* is equal to *value*, return `1`, otherwise return `0`. On error, return `-1`. This is equivalent to the Python expression `value in o`.

int **PySequence_In**(*PyObject* *o, *PyObject* *value)

🄵穩定 ABI 的一部分. *`PySequence_Contains()`* 的🄵名。

在 3.14 版之後被🄵用: The function is *soft deprecated* and should no longer be used to write new code.

Py_ssize_t **PySequence_Index**(*PyObject* *o, *PyObject* *value)

🄵穩定 ABI 的一部分. Return the first index *i* for which `o[i] == value`. On error, return `-1`. This is equivalent to the Python expression `o.index(value)`.

PyObject ***PySequence_List**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a list object with the same contents as the sequence or iterable *o*, or `NULL` on failure. The returned list is guaranteed to be new. This is equivalent to the Python expression `list(o)`.

PyObject ***PySequence_Tuple**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a tuple object with the same contents as the sequence or iterable *o*, or `NULL` on failure. If *o* is a tuple, a new reference will be returned, otherwise a tuple will be constructed with the appropriate contents. This is equivalent to the Python expression `tuple(o)`.

PyObject ***PySequence_Fast**(*PyObject* *o, const char *m)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return the sequence or iterable *o* as an object usable by the other `PySequence_Fast*` family of functions. If the object is not a sequence or iterable, raises `TypeError` with *m* as the message text. Returns `NULL` on failure.

The `PySequence_Fast*` functions are thus named because they assume *o* is a *PyTupleObject* or a *PyListObject* and access the data fields of *o* directly.

As a CPython implementation detail, if *o* is already a sequence or list, it will be returned.

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)

回傳值：借用參照。 Return the *i*th element of *o*, assuming that *o* was returned by *PySequence_Fast()*, *o* is not `NULL`, and that *i* is within bounds.

PyObject ***PySequence_Fast_ITEMS**(*PyObject* *o)

Return the underlying array of PyObject pointers. Assumes that *o* was returned by *PySequence_Fast()* and *o* is not `NULL`.

Note, if a list gets resized, the reallocation may relocate the items array. So, only use the underlying array pointer in contexts where the sequence cannot change.

PyObject ***PySequence_ITEM**(*PyObject* *o, *Py_ssize_t* i)

回傳值：新的參照。Return the *i*th element of *o* or `NULL` on failure. Faster form of *PySequence_GetItem()* but without checking that *PySequence_Check()* on *o* is true and without adjustment for negative indices.

8.5 對映協定

另請參🄵*PyObject_GetItem()*、*PyObject_SetItem()* 和*PyObject_DelItem()*。

int **PyMapping_Check**(*PyObject* *o)

🄵穩定 ABI 的一部分. 如果物件有提供對映協定或支援切片 (slicing) 則回傳 `1`，否則回傳 `0`。請注意，對於具有 `__getitem__()` 方法的 Python 類🄵，它會回傳 `1`，因🄵通常無法確定該類🄵支援什🄵類型的鍵。這個函式總會是成功的。

Py_ssize_t **PyMapping_Size**(*PyObject* *o)

Py_ssize_t **PyMapping_Length**(*PyObject* *o)

🄵穩定 ABI 的一部分. 成功時回傳物件 *o* 中的鍵數，失敗時回傳 `-1`。這相當於 Python 運算式 `len(o)`。

PyObject ***PyMapping_GetItemString**(*PyObject* *o, const char *key)

回傳值：新的參照。🄵穩定 ABI 的一部分. 這與*PyObject_GetItem()* 相同，但 *key* 被指定🄵 `const char*` UTF-8 編碼位元組字串，而不是 *PyObject**。

int **PyMapping_GetOptionalItem**(*PyObject* *obj, *PyObject* *key, *PyObject* **result)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 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.

在 3.13 版被加入.

int **PyMapping_GetOptionalItemString**(*PyObject* *obj, const char *key, *PyObject* **result)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 這與*PyMapping_GetOptionalItem()* 相同，但 *key* 被指定🄵 const char* UTF-8 編碼位元組字串，而不是 *PyObject**。

在 3.13 版被加入.

int **PyMapping_SetItemString**(*PyObject* *o, const char *key, *PyObject* *v)

🄵穩定 ABI 的一部分. 這與*PyObject_SetItem()* 相同，但 *key* 被指定🄵 const char* UTF-8 編碼位元組字串，而不是 *PyObject**。

int **PyMapping_DelItem**(*PyObject* *o, *PyObject* *key)

這是*PyObject_DelItem()* 的🄵名。

int **PyMapping_DelItemString**(*PyObject* *o, const char *key)

這與*PyObject_DelItem()* 相同，但 *key* 被指定🄵 const char* UTF-8 編碼位元組字串，而不是 *PyObject**。

int **PyMapping_HasKeyWithError**(*PyObject* *o, *PyObject* *key)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 如果對映物件具有鍵 *key* 則回傳 1，否則回傳 0。這相當於 Python 運算式 key in o。失敗時回傳 -1。

在 3.13 版被加入.

int **PyMapping_HasKeyStringWithError**(*PyObject* *o, const char *key)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 這與*PyMapping_HasKeyWithError()* 相同，但 *key* 被指定🄵 const char* UTF-8 編碼位元組字串，而不是 *PyObject**。

在 3.13 版被加入.

int **PyMapping_HasKey**(*PyObject* *o, *PyObject* *key)

🄵穩定 ABI 的一部分. 如果對映物件具有鍵 *key* 則回傳 1，否則回傳 0。這相當於 Python 運算式 key in o。這個函式總會是成功的。

> **備🄵**
>
> 當它呼叫 __getitem__() 方法時發生的例外將被默默忽略。🄵了適當地處理錯誤，請改用*PyMapping_HasKeyWithError()*、*PyMapping_GetOptionalItem()* 或*PyObject_GetItem()*。

int **PyMapping_HasKeyString**(*PyObject* *o, const char *key)

🄵穩定 ABI 的一部分. 這與*PyMapping_HasKey()* 相同，但 *key* 被指定🄵 const char* UTF-8 編碼位元組字串，而不是 *PyObject**。

> **備🄵**
>
> 當它呼叫 __getitem__() 方法或建立臨時 str 物件時發生的例外將被默默忽略。🄵了適當地處理錯誤，請改用*PyMapping_HasKeyStringWithError()*、*PyMapping_GetOptionalItemString()* 或*PyMapping_GetItemString()*。

PyObject ***PyMapping_Keys**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 成功時回傳一個物件 *o* 🄵之鍵的串列，失敗時回傳 NULL。

在 3.7 版的變更: 在以前，該函式會回傳串列或元組。

PyObject ***PyMapping_Values**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 成功時回傳物件 *o* 中值的串列。失敗時回傳 NULL。

在 3.7 版的變更: 在以前，該函式會回傳串列或元組。

PyObject ***PyMapping_Items**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 成功時回傳物件 *o* 🄵之項目的串列，其中每個項目都是包含鍵值對的元組。失敗時回傳 NULL。

在 3.7 版的變更: 在以前，該函式會回傳串列或元組。

8.6 🄵代器協定

有兩個專門用於🄵代器的函式。

int **PyIter_Check**(*PyObject* *o)

🄵穩定 ABI 的一部分 自 *3.8* 版本開始. 如果物件 *o* 可以安全地傳遞給*PyIter_NextItem()* 則回傳非零 (non-zero)，否則回傳 0。這個函式一定會執行成功。

int **PyAIter_Check**(*PyObject* *o)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 如果物件 *o* 有提供 AsyncIterator 協定，則回傳非零，否則回傳 0。這個函式一定會執行成功。

在 3.10 版被加入.

int **PyIter_NextItem**(*PyObject* *iter, *PyObject* **item)

🄵穩定 ABI 的一部分 自 *3.14* 版本開始. 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.

在 3.14 版被加入.

PyObject ***PyIter_Next**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 這是 PyIter_NextItem() 的舊版本，🄵了向後相容而保留。請優先使用*PyIter_NextItem()*。

回傳🄵代器 *o* 的下一個值。根據*PyIter_Check()*，該物件必須是一個🄵代器（由呼叫者檢查）。如果🄵有剩余值，則回傳 NULL 且不設定例外。如果檢索項目時發生錯誤，則回傳 NULL 🄵傳遞例外。

type **PySendResult**

用於表示*PyIter_Send()* 不同結果的列舉 (enum) 值。

在 3.10 版被加入.

PySendResult **PyIter_Send**(*PyObject* *iter, *PyObject* *arg, *PyObject* **presult)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 將 *arg* 值發送到🄵代器 *iter* 中。回傳：

- 如果🄵代器有回傳則🄵 PYGEN_RETURN。回傳值透過 *presult* 回傳。
- 如果🄵代器有🄵生 (yield) 則🄵 PYGEN_NEXT。🄵生值透過 *presult* 回傳。
- 如果🄵代器引發例外則🄵 PYGEN_ERROR。*presult* 被設定🄵 NULL。

在 3.10 版被加入.

8.7 緩沖協定 (Buffer Protocol)

Certain objects available in Python wrap access to an underlying memory array or *buffer*. Such objects include the built-in `bytes` and `bytearray`, and some extension types like `array.array`. Third-party libraries may define their own types for special purposes, such as image processing or numeric analysis.

While each of these types have their own semantics, they share the common characteristic of being backed by a possibly large memory buffer. It is then desirable, in some situations, to access that buffer directly and without intermediate copying.

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 *Buffer Object Structures*; 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`.

Simple objects such as `bytes` and `bytearray` expose their underlying buffer in byte-oriented form. Other forms are possible; for example, the elements exposed by an `array.array` can be multi-byte values.

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.

There are two ways for a consumer of the buffer interface to acquire a buffer over a target object:

- call *`PyObject_GetBuffer()`* with the right parameters;
- call *`PyArg_ParseTuple()`* (or one of its siblings) with one of the `y*`, `w*` or `s*` *format codes*.

In both cases, *`PyBuffer_Release()`* must be called when the buffer isn't needed anymore. Failure to do so could lead to various issues such as resource leaks.

在 3.12 版被加入: The buffer protocol is now accessible in Python, see python-buffer-protocol and `memoryview`.

8.7.1 Buffer structure

Buffer structures (or simply "buffers") are useful as a way to expose the binary data from another object to the Python programmer. They can also be used as a zero-copy slicing mechanism. Using their ability to reference a block of memory, it is possible to expose any data to the Python programmer quite easily. The memory could be a large, constant array in a C extension, it could be a raw block of memory for manipulation before passing to an operating system library, or it could be used to pass around structured data in its native, in-memory format.

Contrary to most data types exposed by the Python interpreter, buffers are not *`PyObject`* pointers but rather simple C structures. This allows them to be created and copied very simply. When a generic wrapper around a buffer is needed, a *memoryview* object can be created.

For short instructions how to write an exporting object, see *Buffer Object Structures*. For obtaining a buffer, see *`PyObject_GetBuffer()`*.

type **`Py_buffer`**

🄵穩定 ABI 的一部分 （包含所有成員）自 *3.11* 版本開始.

void ***`buf`**

A pointer to the start of the logical structure described by the buffer fields. This can be any location within the underlying physical memory block of the exporter. For example, with negative *`strides`* the value may point to the end of the memory block.

For *contiguous* arrays, the value points to the beginning of the memory block.

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.

As a special case, for *temporary* buffers that are wrapped by *PyMemoryView_FromBuffer()* or *PyBuffer_FillInfo()* this field is `NULL`. In general, exporting objects MUST NOT use this scheme.

Py_ssize_t **len**

`product(shape) * itemsize`. For contiguous arrays, this is the length of the underlying memory block. For non-contiguous arrays, it is the length that the logical structure would have if it were copied to a contiguous representation.

Accessing `((char *)buf)[0] up to ((char *)buf)[len-1]` is only valid if the buffer has been obtained by a request that guarantees contiguity. In most cases such a request will be *PyBUF_SIMPLE* or *PyBUF_WRITABLE*.

int **readonly**

An indicator of whether the buffer is read-only. This field is controlled by the *PyBUF_WRITABLE* flag.

Py_ssize_t **itemsize**

Item size in bytes of a single element. Same as the value of `struct.calcsize()` called on non-`NULL` *format* values.

Important exception: If a consumer requests a buffer without the *PyBUF_FORMAT* flag, *format* will be set to `NULL`, but *itemsize* still has the value for the original format.

If *shape* is present, the equality `product(shape) * itemsize == len` still holds and the consumer can use *itemsize* to navigate the buffer.

If *shape* is `NULL` as a result of a *PyBUF_SIMPLE* or a *PyBUF_WRITABLE* request, the consumer must disregard *itemsize* and assume `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.

This field is controlled by the *PyBUF_FORMAT* flag.

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**

An array of *Py_ssize_t* of length *ndim* indicating the shape of the memory as an n-dimensional array. Note that `shape[0] * ... * shape[ndim-1] * itemsize` MUST be equal to *len*.

Shape values are restricted to `shape[n] >= 0`. The case `shape[n] == 0` requires special attention. See *complex arrays* for further information.

The shape array is read-only for the consumer.

Py_ssize_t ***strides**

An array of *Py_ssize_t* of length *ndim* giving the number of bytes to skip to get to a new element in each dimension.

Stride values can be any integer. For regular arrays, strides are usually positive, but a consumer MUST be able to handle the case `strides[n] <= 0`. See *complex arrays* for further information.

The strides array is read-only for the consumer.

Py_ssize_t ***suboffsets**

An array of *Py_ssize_t* of length *ndim*. If `suboffsets[n] >= 0`, the values stored along the nth dimension are pointers and the suboffset value dictates how many bytes to add to each pointer after de-referencing. A suboffset value that is negative indicates that no de-referencing should occur (striding in a contiguous memory block).

If all suboffsets are negative (i.e. no de-referencing is needed), then this field must be `NULL` (the default value).

This type of array representation is used by the Python Imaging Library (PIL). See *complex arrays* for further information how to access elements of such an array.

The suboffsets array is read-only for the consumer.

void ***internal**

This is for use internally by the exporting object. For example, this might be re-cast as an integer by the exporter and used to store flags about whether or not the shape, strides, and suboffsets arrays must be freed when the buffer is released. The consumer MUST NOT alter this value.

常數:

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 Buffer request types

Buffers are usually obtained by sending a buffer request to an exporting object via *PyObject_GetBuffer()*. Since the complexity of the logical structure of the memory can vary drastically, the consumer uses the *flags* argument to specify the exact buffer type it can handle.

All *Py_buffer* fields are unambiguously defined by the request type.

request-independent fields

The following fields are not influenced by *flags* and must always be filled in with the correct values: *obj*, *buf*, *len*, *itemsize*, *ndim*.

readonly, format

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

Controls the *format* field. If set, this field MUST be filled in correctly. Otherwise, this field MUST be `NULL`.

PyBUF_WRITABLE can be |'d to any of the flags in the next section. Since *PyBUF_SIMPLE* is defined as 0, *PyBUF_WRITABLE* can be used as a stand-alone flag to request a simple writable buffer.

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.

shape, strides, suboffsets

The flags that control the logical structure of the memory are listed in decreasing order of complexity. Note that each flag contains all bits of the flags below it.

Request	shape	strides	suboffsets
PyBUF_INDIRECT	yes	yes	if needed
PyBUF_STRIDES	yes	yes	NULL
PyBUF_ND	yes	NULL	NULL
PyBUF_SIMPLE	NULL	NULL	NULL

contiguity requests

C or Fortran *contiguity* can be explicitly requested, with and without stride information. Without stride information, the buffer must be C-contiguous.

Request	shape	strides	suboffsets	contig
PyBUF_C_CONTIGUOUS	yes	yes	NULL	C
PyBUF_F_CONTIGUOUS	yes	yes	NULL	F
PyBUF_ANY_CONTIGUOUS	yes	yes	NULL	C 或 F
PyBUF_ND	yes	NULL	NULL	C

compound requests

All possible requests are fully defined by some combination of the flags in the previous section. For convenience, the buffer protocol provides frequently used combinations as single flags.

In the following table *U* stands for undefined contiguity. The consumer would have to call *PyBuffer_IsContiguous()* to determine contiguity.

Request	shape	strides	suboffsets	contig	readonly	format
PyBUF_FULL	yes	yes	if needed	U	0	yes
PyBUF_FULL_RO	yes	yes	if needed	U	1 或 0	yes
PyBUF_RECORDS	yes	yes	NULL	U	0	yes
PyBUF_RECORDS_RO	yes	yes	NULL	U	1 或 0	yes
PyBUF_STRIDED	yes	yes	NULL	U	0	NULL
PyBUF_STRIDED_RO	yes	yes	NULL	U	1 或 0	NULL
PyBUF_CONTIG	yes	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	yes	NULL	NULL	C	1 或 0	NULL

8.7.3 Complex arrays

NumPy-style: shape and strides

The logical structure of NumPy-style arrays is defined by *itemsize*, *ndim*, *shape* and *strides*.

If `ndim == 0`, the memory location pointed to by *buf* is interpreted as a scalar of size *itemsize*. In that case, both *shape* and *strides* are `NULL`.

If *strides* is `NULL`, the array is interpreted as a standard n-dimensional C-array. Otherwise, the consumer must access an n-dimensional array as follows:

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

As noted above, *buf* can point to any location within the actual memory block. An exporter can check the validity of a buffer with this function:

```
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:
```

(繼續下一頁)

(繼續上一頁)

```
        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
```

PIL-style: shape, strides and suboffsets

In addition to the regular items, PIL-style arrays can contain pointers that must be followed in order to get to the next element in a dimension. For example, the regular three-dimensional C-array `char v[2][2][3]` can also be viewed as an array of 2 pointers to 2 two-dimensional arrays: `char (*v[2])[2][3]`. In suboffsets representation, those two pointers can be embedded at the start of *buf*, pointing to two `char x[2][3]` arrays that can be located anywhere in memory.

Here is a function that returns a pointer to the element in an N-D array pointed to by an N-dimensional index when there are both non-`NULL` strides and suboffsets:

```
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
                       Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >=0 ) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

8.7.4 Buffer-related functions

int **PyObject_CheckBuffer**(*PyObject* *obj)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Return `1` if *obj* supports the buffer interface otherwise `0`. When `1` is returned, it doesn't guarantee that *PyObject_GetBuffer()* will succeed. This function always succeeds.

int **PyObject_GetBuffer**(*PyObject* *exporter, *Py_buffer* *view, int flags)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. 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`.

On success, fill in *view*, set `view->obj` to a new reference to *exporter* and return 0. In the case of chained buffer providers that redirect requests to a single object, `view->obj` MAY refer to this object instead of *exporter* (See *Buffer Object Structures*).

Successful calls to *PyObject_GetBuffer()* must be paired with calls to *PyBuffer_Release()*, similar to `malloc()` and `free()`. Thus, after the consumer is done with the buffer, *PyBuffer_Release()* must be called exactly once.

void **PyBuffer_Release**(*Py_buffer* *view)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. 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.

It is an error to call this function on a buffer that was not obtained via *PyObject_GetBuffer()*.

Py_ssize_t **PyBuffer_SizeFromFormat**(const char *format)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Return the implied `itemsize` from `format`. On error, raise an exception and return -1.

在 3.9 版被加入.

int **PyBuffer_IsContiguous**(const *Py_buffer* *view, char order)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Return `1` if the memory defined by the *view* is C-style (*order* is `'C'`) or Fortran-style (*order* is `'F'`) *contiguous* or either one (*order* is `'A'`). Return `0` otherwise. This function always succeeds.

void *__PyBuffer_GetPointer__(const *Py_buffer* *view, const *Py_ssize_t* *indices)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Get the memory area pointed to by the *indices* inside the given *view*. *indices* must point to an array of `view->ndim` indices.

int **PyBuffer_FromContiguous**(const *Py_buffer* *view, const void *buf, *Py_ssize_t* len, char fort)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Copy contiguous *len* bytes from *buf* to *view*. *fort* can be `'C'` or `'F'` (for C-style or Fortran-style ordering). `0` is returned on success, `-1` on error.

int **PyBuffer_ToContiguous**(void *buf, const *Py_buffer* *src, *Py_ssize_t* len, char order)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Copy *len* bytes from *src* to its contiguous representation in *buf*. *order* can be `'C'` or `'F'` or `'A'` (for C-style or Fortran-style ordering or either one). `0` is returned on success, `-1` on error.

This function fails if *len* != *src->len*.

int **PyObject_CopyData**(*PyObject* *dest, *PyObject* *src)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Copy data from *src* to *dest* buffer. Can convert between C-style and or Fortran-style buffers.

`0` is returned on success, `-1` on error.

void **PyBuffer_FillContiguousStrides**(int ndims, *Py_ssize_t* *shape, *Py_ssize_t* *strides, int itemsize, char order)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Fill the *strides* array with byte-strides of a *contiguous* (C-style if *order* is `'C'` or Fortran-style if *order* is `'F'`) array of the given shape with the given number of bytes per element.

int **PyBuffer_FillInfo**(*Py_buffer* *view, *PyObject* *exporter, void *buf, *Py_ssize_t* len, int readonly, int flags)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Handle buffer requests for an exporter that wants to expose *buf* of size *len* with writability set according to *readonly*. *buf* is interpreted as a sequence of unsigned bytes.

The *flags* argument indicates the request type. This function always fills in *view* as specified by flags, unless *buf* has been designated as read-only and `PyBUF_WRITABLE` is set in *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`;

If this function is used as part of a *getbufferproc*, *exporter* MUST be set to the exporting object and *flags* must be passed unmodified. Otherwise, *exporter* MUST be `NULL`.

CHAPTER 9

具體物件層

此章節列出的函式僅能接受某些特定的 Python 物件型🄵，將錯誤型🄵的物件傳遞給它們🄵不是什🄵好事，如果你從 Python 程式當中接收到一個不確定是否🄵正確型🄵的物件，那🄵請一定要先做型🄵檢查。例如使用*PyDict_Check()* 來確認一個物件是否🄵字典。本章結構類似於 Python 物件型🄵的" 族譜圖 (family tree)"。

> **警告**
>
> 雖然本章所述之函式仔細地檢查了傳入物件的型🄵，但大多🄵無檢查是否🄵 `NULL`。允許 `NULL` 的傳入可能造成記憶體的不合法存取和直譯器的立即中止。

9.1 基礎物件

此段落描述 Python 型🄵物件與單例 (singleton) 物件 `None`。

9.1.1 型🄵物件

type **PyTypeObject**

> 🄵受限 API 的一部分 （做🄵一個不透明結構 *(opaque struct)*）. The C structure of the objects used to describe built-in types.

PyTypeObject **PyType_Type**

> 🄵穩定 ABI 的一部分. This is the type object for type objects; it is the same object as `type` in the Python layer.

int **PyType_Check**(*PyObject* *o)

> Return non-zero if the object *o* is a type object, including instances of types derived from the standard type object. Return 0 in all other cases. This function always succeeds.

int **PyType_CheckExact**(*PyObject* *o)

> Return non-zero if the object *o* is a type object, but not a subtype of the standard type object. Return 0 in all other cases. This function always succeeds.

unsigned int **PyType_ClearCache**()

> 🄵穩定 ABI 的一部分. Clear the internal lookup cache. Return the current version tag.

unsigned long **PyType_GetFlags**(*PyTypeObject* *type)

🄵穩定 ABI 的一部分. 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*.

在 3.2 版被加入.

在 3.4 版的變更: The return type is now `unsigned long` rather than `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.

在 3.12 版被加入.

void **PyType_Modified**(*PyTypeObject* *type)

🄵穩定 ABI 的一部分. Invalidate the internal lookup cache for the type and all of its subtypes. This function must be called after any manual modification of the attributes or base classes of the type.

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).

在 3.12 版被加入.

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()`.

在 3.12 版被加入.

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()`.

在 3.12 版被加入.

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.

在 3.12 版被加入.

int **PyType_HasFeature**(*PyTypeObject* *o, int feature)

Return non-zero if the type object *o* sets the feature *feature*. Type features are denoted by single bit flags.

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)

🄵穩定 ABI 的一部分. Return true if *a* is a subtype of *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)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. Finalize a type object. This should be called on all type objects to finish their initialization. This function is responsible for adding inherited slots from a type's base class. Return `0` on success, or return `-1` and sets an exception on error.

> **備🄵**
>
> 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)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Return the type's name. Equivalent to getting the type's `__name__` attribute.

在 3.11 版被加入.

PyObject ***PyType_GetQualName**(*PyTypeObject* *type)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Return the type's qualified name. Equivalent to getting the type's `__qualname__` attribute.

在 3.11 版被加入.

PyObject ***PyType_GetFullyQualifiedName**(*PyTypeObject* *type)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 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"`.

在 3.13 版被加入.

PyObject ***PyType_GetModuleName**(*PyTypeObject* *type)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Return the type's module name. Equivalent to getting the `type.__module__` attribute.

在 3.13 版被加入.

void ***PyType_GetSlot**(*PyTypeObject* *type, int slot)

🄵穩定 ABI 的一部分 自 *3.4* 版本開始. Return the function pointer stored in the given slot. If the result is `NULL`, this indicates that either the slot is `NULL`, or that the function was called with invalid parameters. Callers will typically cast the result pointer into the appropriate function type.

See `PyType_Slot.slot` for possible values of the *slot* argument.

在 3.4 版被加入.

在 3.10 版的變更: `PyType_GetSlot()` can now accept all types. Previously, it was limited to *heap types*.

PyObject ***PyType_GetModule**(*PyTypeObject* *type)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Return the module object associated with the given type when the type was created using `PyType_FromModuleAndSpec()`.

If no module is associated with the given type, sets `TypeError` and returns `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.

在 3.9 版被加入.

void ***PyType_GetModuleState**(*PyTypeObject* *type)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Return the state of the module object associated with the given type. This is a shortcut for calling `PyModule_GetState()` on the result of `PyType_GetModule()`.

If no module is associated with the given type, sets `TypeError` and returns `NULL`.

If the *type* has an associated module but its state is `NULL`, returns `NULL` without setting an exception.

在 3.9 版被加入.

PyObject ***PyType_GetModuleByDef**(*PyTypeObject* *type, struct *PyModuleDef* *def)

回傳值: 借用參照。🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Find the first superclass whose module was created from the given `PyModuleDef` *def*, and return that module.

If no module is found, raises a `TypeError` and returns `NULL`.

This function is intended to be used together with `PyModule_GetState()` to get module state from slot methods (such as `tp_init` or `nb_add`) and other places where a method's defining class cannot be passed using the `PyCMethod` calling convention.

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.

在 3.11 版被加入.

int **PyType_GetBaseByToken**(*PyTypeObject* *type, void *token, *PyTypeObject* **result)

🄵穩定 ABI 的一部分 自 *3.14* 版本開始. 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`.
- 如果🄵找到，則將 **result* 設🄵 `NULL` 🄵回傳 `0`。
- 在發生錯誤時，將 **result* 設🄵 `NULL` 🄵回傳 `-1`，同時設定例外。

如果 *result* 引數🄵 `NULL`，則不會設定 **result*。如果你只需要回傳值，請使用此選項。

token 引數不可🄵 `NULL`。

在 3.14 版被加入.

int **PyUnstable_Type_AssignVersionTag**(*PyTypeObject* *type)

這是不穩定 *API*，它可能在小版本發布中🄴有任何警告地被變更。

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.

在 3.12 版被加入.

Creating Heap-Allocated Types

The following functions and structs are used to create *heap types*.

PyObject ***PyType_FromMetaclass**(*PyTypeObject* *metaclass, *PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 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`.

The *bases* argument can be used to specify base classes; it can either be only one class or a tuple of classes. If *bases* is `NULL`, the *Py_tp_bases* slot is used instead. If that also is `NULL`, the *Py_tp_base* slot is used instead. If that also is `NULL`, the new type derives from `object`.

The *module* argument can be used to record the module in which the new class is defined. It must be a module object or `NULL`. If not `NULL`, the module is associated with the new type and can later be retrieved with `PyType_GetModule()`. The associated module is not inherited by subclasses; it must be specified for each class individually.

This function calls `PyType_Ready()` on the new type.

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.

在 3.12 版被加入.

PyObject ***PyType_FromModuleAndSpec**(*PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

回傳值: 新的參照。🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 等價於 `PyType_FromMetaclass(NULL, module, spec, bases)`。

在 3.9 版被加入.

在 3.10 版的變更: The function now accepts a single class as the *bases* argument and `NULL` as the `tp_doc` slot.

在 3.12 版的變更: 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.

在 3.14 版的變更: Creating classes whose metaclass overrides *tp_new* is no longer allowed.

PyObject ***PyType_FromSpecWithBases**(*PyType_Spec* *spec, *PyObject* *bases)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.3* 版本開始. 等價於 `PyType_FromMetaclass(NULL, NULL, spec, bases)`。

在 3.3 版被加入.

在 3.12 版的變更: 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.

在 3.14 版的變更: Creating classes whose metaclass overrides *tp_new* is no longer allowed.

PyObject ***PyType_FromSpec**(*PyType_Spec* *spec)

回傳值：新的參照。🄵穩定 ABI 的一部分. 等價於 `PyType_FromMetaclass(NULL, NULL, spec, NULL)`。

在 3.12 版的變更: 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.

在 3.14 版的變更: Creating classes whose metaclass overrides *tp_new* is no longer allowed.

int **PyType_Freeze**(*PyTypeObject* *type)

🄵穩定 ABI 的一部分 自 *3.14* 版本開始. 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.

在 3.14 版被加入.

type **PyType_Spec**

🄵穩定 ABI 的一部分 （包含所有成員）. Structure defining a type's behavior.

const char ***name**

Name of the type, used to set *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.

在 3.12 版的變更: 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**

Type flags, used to set *PyTypeObject.tp_flags*.

If the `Py_TPFLAGS_HEAPTYPE` flag is not set, *PyType_FromSpecWithBases()* sets it automatically.

PyType_Slot ***slots**

Array of *PyType_Slot* structures. Terminated by the special slot value `{0, NULL}`.

Each slot ID should be specified at most once.

type **PyType_Slot**

🄵穩定 ABI 的一部分 （包含所有成員）. Structure defining optional functionality of a type, containing a slot ID and a value pointer.

int **slot**

A slot ID.

Slot IDs are named like the field names of the structures *PyTypeObject*, *PyNumberMethods*, *PySequenceMethods*, *PyMappingMethods* and *PyAsyncMethods* with an added `Py_` prefix. For example, use:

- `Py_tp_dealloc` to set *PyTypeObject.tp_dealloc*
- `Py_nb_add` to set *PyNumberMethods.nb_add*
- `Py_sq_length` to set *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*（如果可能，請改用*Py_TPFLAGS_MANAGED_DICT*）
- *tp_vectorcall_offset*（請用*PyMemberDef* 中的 `"__vectorcalloffset__"`）

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.

在 3.9 版的變更: Slots in *PyBufferProcs* may be set in the unlimited API.

在 3.11 版的變更: *bf_getbuffer* and *bf_releasebuffer* are now available under the *limited API*.

在 3.14 版的變更: The field *tp_vectorcall* can now set using `Py_tp_vectorcall`. See the field's documentation for details.

void ***pfunc**

The desired value of the slot. In most cases, this is a pointer to a function.

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.

在 3.14 版被加入.

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.

在 3.14 版被加入.

9.1.2 None 物件

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*.

在 3.12 版的變更: *Py_None* 🄵不滅的 *(immortal)*。

Py_RETURN_NONE

Return *Py_None* from a function.

9.2 數值物件

9.2.1 整數物件

All integers are implemented as "long" integer objects of arbitrary size.

On error, most PyLong_As* APIs return (return type)-1 which cannot be distinguished from a number. Use *PyErr_Occurred()* to disambiguate.

type **PyLongObject**

🄵受限 API 的一部分 （做🄵一個不透明結構 *(opaque struct)*）. This subtype of *PyObject* represents a Python integer object.

PyTypeObject **PyLong_Type**

🄵穩定 ABI 的一部分. This instance of *PyTypeObject* represents the Python integer type. This is the same object as int in the Python layer.

int **PyLong_Check**(*PyObject* *p)

Return true if its argument is a *PyLongObject* or a subtype of *PyLongObject*. This function always succeeds.

int **PyLong_CheckExact**(*PyObject* *p)

Return true if its argument is a *PyLongObject*, but not a subtype of *PyLongObject*. This function always succeeds.

PyObject ***PyLong_FromLong**(long v)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new *PyLongObject* object from *v*, or NULL on failure.

CPython 實作細節： 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new *PyLongObject* object from a C unsigned long, or NULL on failure.

PyObject ***PyLong_FromSsize_t**(*Py_ssize_t* v)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new *PyLongObject* object from a C *Py_ssize_t*, or NULL on failure.

PyObject ***PyLong_FromSize_t**(size_t v)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new *PyLongObject* object from a C size_t, or NULL on failure.

PyObject ***PyLong_FromLongLong**(long long v)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.14* 版本開始. Return a new *PyLongObject* object from a signed C int32_t or int64_t, or NULL with an exception set on failure.

在 3.14 版被加入.

PyObject ***PyLong_FromUnsignedLongLong**(unsigned long long v)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.14* 版本開始. Return a new *PyLongObject* object from an unsigned C uint32_t or uint64_t, or NULL with an exception set on failure.

在 3.14 版被加入.

PyObject ***PyLong_FromDouble**(double v)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new *PyLongObject* object from the integer part of *v*, or NULL on failure.

PyObject ***PyLong_FromString**(const char *str, char **pend, int base)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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.

> **也參考**
>
> `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)

回傳值：新的參照。 Convert a sequence of Unicode digits in the string *u* to a Python integer value.

在 3.3 版被加入.

PyObject ***PyLong_FromVoidPtr**(void *p)

回傳值：新的參照。🄵穩定 ABI 的一部分. Create a Python integer from the pointer *p*. The pointer value can be retrieved from the resulting value using `PyLong_AsVoidPtr()`.

PyObject ***PyLong_FromNativeBytes**(const void *buffer, size_t n_bytes, int flags)

🄵穩定 ABI 的一部分 自 *3.14* 版本開始. 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.

在 3.13 版被加入.

PyObject ***PyLong_FromUnsignedNativeBytes**(const void *buffer, size_t n_bytes, int flags)

🄵穩定 ABI 的一部分 自 *3.14* 版本開始. 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.

在 3.13 版被加入.

long **PyLong_AsLong**(*PyObject* *obj)

🄵穩定 ABI 的一部分. 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`.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

在 3.8 版的變更: Use `__index__()` if available.

在 3.10 版的變更: 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.

在 3.14 版之後被🄵用: The function is soft deprecated.

int **PyLong_AsInt**(*PyObject* *obj)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Similar to `PyLong_AsLong()`, but store the result in a C `int` instead of a C `long`.

在 3.13 版被加入.

long **PyLong_AsLongAndOverflow**(*PyObject* *obj, int *overflow)

🄵穩定 ABI 的一部分. 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.

Returns `-1` on error. Use *PyErr_Occurred()* to disambiguate.

在 3.8 版的變更: Use `__index__()` if available.

在 3.10 版的變更: This function will no longer use `__int__()`.

long long **PyLong_AsLongLong**(*PyObject* *obj)

🄵穩定 ABI 的一部分. 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`.

Returns `-1` on error. Use *PyErr_Occurred()* to disambiguate.

在 3.8 版的變更: Use `__index__()` if available.

在 3.10 版的變更: This function will no longer use `__int__()`.

long long **PyLong_AsLongLongAndOverflow**(*PyObject* *obj, int *overflow)

🄵穩定 ABI 的一部分. 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.

Returns `-1` on error. Use *PyErr_Occurred()* to disambiguate.

在 3.2 版被加入.

在 3.8 版的變更: Use `__index__()` if available.

在 3.10 版的變更: This function will no longer use `__int__()`.

Py_ssize_t **PyLong_AsSsize_t**(*PyObject* *pylong)

🄵穩定 ABI 的一部分. Return a C *Py_ssize_t* representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a *Py_ssize_t*.

Returns `-1` on error. Use *PyErr_Occurred()* to disambiguate.

unsigned long **PyLong_AsUnsignedLong**(*PyObject* *pylong)

🄵穩定 ABI 的一部分. 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`.

Returns `(unsigned long)-1` on error. Use *PyErr_Occurred()* to disambiguate.

size_t **PyLong_AsSize_t**(*PyObject* *pylong)

🄵穩定 ABI 的一部分. Return a C `size_t` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a `size_t`.

Returns `(size_t)-1` on error. Use *PyErr_Occurred()* to disambiguate.

unsigned long long **PyLong_AsUnsignedLongLong**(*PyObject* *pylong)

🄵穩定 ABI 的一部分. 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`.

Returns `(unsigned long long)-1` on error. Use *PyErr_Occurred()* to disambiguate.

在 3.1 版的變更: A negative *pylong* now raises `OverflowError`, not `TypeError`.

unsigned long **PyLong_AsUnsignedLongMask**(*PyObject* *obj)

🄵穩定 ABI 的一部分. 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`.

Returns `(unsigned long)-1` on error. Use *PyErr_Occurred()* to disambiguate.

在 3.8 版的變更: Use `__index__()` if available.

在 3.10 版的變更: This function will no longer use `__int__()`.

unsigned long long **PyLong_AsUnsignedLongLongMask**(*PyObject* *obj)

🄵穩定 ABI 的一部分. 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`.

Returns `(unsigned long long)-1` on error. Use *PyErr_Occurred()* to disambiguate.

在 3.8 版的變更: Use `__index__()` if available.

在 3.10 版的變更: This function will no longer use `__int__()`.

int **PyLong_AsInt32**(*PyObject* *obj, int32_t *value)

int **PyLong_AsInt64**(*PyObject* *obj, int64_t *value)

🄵穩定 ABI 的一部分 自 *3.14* 版本開始. 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`.

在 3.14 版被加入.

int **PyLong_AsUInt32**(*PyObject* *obj, uint32_t *value)

int **PyLong_AsUInt64**(*PyObject* *obj, uint64_t *value)

🄵穩定 ABI 的一部分 自 *3.14* 版本開始. 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`.

在 3.14 版被加入.

double **PyLong_AsDouble**(*PyObject* *pylong)

🄵穩定 ABI 的一部分. 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`.

Returns `-1.0` on error. Use *PyErr_Occurred()* to disambiguate.

void ***PyLong_AsVoidPtr**(*PyObject* *pylong)

🄵穩定 ABI 的一部分. 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()*.

Returns `NULL` on error. Use *PyErr_Occurred()* to disambiguate.

Py_ssize_t **PyLong_AsNativeBytes**(*PyObject* *pylong, void *buffer, *Py_ssize_t* n_bytes, int flags)

🄵穩定 ABI 的一部分 自 *3.14* 版本開始. 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.

> **備🄵**
>
> Overflow is not considered an error. If the returned value is larger than *n_bytes*, most significant bits were discarded.

`0` 將永不被回傳。

Values are always copied as two's-complement.

使用範例:

```
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`.

> **備🄵**
>
> 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) {
    // Failed. A Python exception was set with the reason.
```

(繼續下一頁)

(繼續上一頁)

```
    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`.

旗標	數值
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 calcula-

tions. 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`.

備🄵

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.

在 3.13 版被加入.

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.

在 3.14 版被加入.

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`.

在 3.14 版被加入.

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`.

在 3.14 版被加入.

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`.

在 3.14 版被加入.

PyObject ***PyLong_GetInfo**(void)

🄵穩定 ABI 的一部分. 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.

在失敗時，會回傳 `NULL` 🄵設定例外。

在 3.1 版被加入.

int **PyUnstable_Long_IsCompact**(const *PyLongObject* *op)

> 這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

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.

在 3.12 版被加入.

Py_ssize_t **PyUnstable_Long_CompactValue**(const *PyLongObject* *op)

> 這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

If *op* is compact, as determined by `PyUnstable_Long_IsCompact()`, return its value.

Otherwise, the return value is undefined.

在 3.12 版被加入.

Export API

在 3.14 版被加入.

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.

CPython 實作細節： 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()*.

CPython 實作細節： Calling *PyLong_FreeExport()* is optional if *export_long->digits* is `NULL`.

PyLongWriter API

The *PyLongWriter* API can be used to import an integer.

在 3.14 版被加入.

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)

回傳值：新的參照。 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 Boolean（布林）物件

Python 中的 boolean 是以整數子類🄵化來實現的。只有*Py_False* 和*Py_True* 兩個 boolean。因此一般的建立和🄵除函式🄵不適用於 boolean。但下列巨集 (macro) 是可用的。

PyTypeObject **PyBool_Type**

🄵穩定 ABI 的一部分. 此*PyTypeObject* 實例代表 Python 的布林型🄵；它與 Python 層級中的 `bool` 是同一個物件。

int **PyBool_Check**(*PyObject* *o)

如果 *o* 的型🄵🄵*PyBool_Type* 則回傳真值。此函式總是會成功執行。

PyObject ***Py_False**

Python 的 `False` 物件。此物件🄵有任何方法且🄵不滅的 *(immortal)*。

在 3.12 版的變更: *Py_False* 🄵不滅的。

PyObject ***Py_True**

Python 的 `True` 物件。此物件🄵有任何方法且🄵不滅的。

在 3.12 版的變更: *Py_True* 🄵不滅的。

Py_RETURN_FALSE

從函式回傳*Py_False*。

Py_RETURN_TRUE

從函式回傳*Py_True*。

PyObject ***PyBool_FromLong**(long v)

回傳值：新的參照。🄵穩定 ABI 的一部分. 根據 *v* 的實際值來回傳*Py_True* 或者*Py_False*。

9.2.3 浮點數（Floating-Point）物件

type **PyFloatObject**

This subtype of *PyObject* represents a Python floating-point object.

PyTypeObject **PyFloat_Type**

🄵穩定 ABI 的一部分. 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)

Return true if its argument is a *PyFloatObject* or a subtype of *PyFloatObject*. This function always succeeds.

int **PyFloat_CheckExact**(*PyObject* *p)

Return true if its argument is a *PyFloatObject*, but not a subtype of *PyFloatObject*. This function always succeeds.

PyObject ***PyFloat_FromString**(*PyObject* *str)

回傳值：新的參照。🄵穩定 ABI 的一部分. Create a *PyFloatObject* object based on the string value in *str*, or `NULL` on failure.

PyObject ***PyFloat_FromDouble**(double v)

回傳值：新的參照。🄵穩定 ABI 的一部分. Create a *PyFloatObject* object from *v*, or `NULL` on failure.

double **PyFloat_AsDouble**(*PyObject* *pyfloat)

🄵穩定 ABI 的一部分. 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.

在 3.8 版的變更: Use `__index__()` if available.

double **PyFloat_AS_DOUBLE**(*PyObject* *pyfloat)

Return a C `double` representation of the contents of *pyfloat*, but without error checking.

PyObject ***PyFloat_GetInfo**(void)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a structseq instance which contains information about the precision, minimum and maximum values of a float. It's a thin wrapper around the header file `float.h`.

double **PyFloat_GetMax**()

🄵穩定 ABI 的一部分. Return the maximum representable finite float *DBL_MAX* as C `double`.

double **PyFloat_GetMin**()

🄵穩定 ABI 的一部分. Return the minimum normalized positive float *DBL_MIN* as C `double`.

Pack and Unpack functions

The pack and unpack functions provide an efficient platform-independent way to store floating-point values as byte strings. The Pack routines produce a bytes string from a C `double`, and the Unpack routines produce a C `double` from such a bytes string. The suffix (2, 4 or 8) specifies the number of bytes in the bytes string.

On platforms that appear to use IEEE 754 formats these functions work by copying bits. On other platforms, the 2-byte format is identical to the IEEE 754 binary16 half-precision format, the 4-byte format (32-bit) is identical to the IEEE 754 binary32 single precision format, and the 8-byte format to the IEEE 754 binary64 double precision format, although the packing of INFs and NaNs (if such things exist on the platform) isn't handled correctly, and attempting to unpack a bytes string containing an IEEE INF or NaN will raise an exception.

Note that NaNs type may not be preserved on IEEE platforms (silent NaN become quiet), for example on x86 systems in 32-bit mode.

On non-IEEE platforms with more precision, or larger dynamic range, than IEEE 754 supports, not all values can be packed; on non-IEEE platforms with less precision, or smaller dynamic range, not all values can be unpacked. What happens in such cases is partly accidental (alas).

在 3.11 版被加入.

Pack functions

The pack routines write 2, 4 or 8 bytes, starting at *p*. *le* is an `int` argument, non-zero if you want the bytes string in little-endian format (exponent last, at `p+1`, `p+3`, or `p+6` `p+7`), zero if you want big-endian format (exponent first, at *p*). The `PY_BIG_ENDIAN` constant can be used to use the native endian: it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value: `0` if all is OK, `-1` if error (and an exception is set, most likely `OverflowError`).

There are two problems on non-IEEE platforms:

- What this does is undefined if *x* is a NaN or infinity.
- `-0.0` 和 `+0.0` 會🄵生同樣的位元組字串。

int **PyFloat_Pack2**(double x, char *p, int le)

Pack a C double as the IEEE 754 binary16 half-precision format.

int **PyFloat_Pack4**(double x, char *p, int le)

Pack a C double as the IEEE 754 binary32 single precision format.

int **PyFloat_Pack8**(double x, char *p, int le)

Pack a C double as the IEEE 754 binary64 double precision format.

Unpack functions

The unpack routines read 2, 4 or 8 bytes, starting at *p*. *le* is an `int` argument, non-zero if the bytes string is in little-endian format (exponent last, at `p+1`, `p+3` or `p+6` and `p+7`), zero if big-endian (exponent first, at *p*). The `PY_BIG_ENDIAN` constant can be used to use the native endian: it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value: The unpacked double. On error, this is `-1.0` and *`PyErr_Occurred()`* is true (and an exception is set, most likely `OverflowError`).

Note that on a non-IEEE platform this will refuse to unpack a bytes string that represents a NaN or infinity.

double **PyFloat_Unpack2**(const char *p, int le)

Unpack the IEEE 754 binary16 half-precision format as a C double.

double **PyFloat_Unpack4**(const char *p, int le)

Unpack the IEEE 754 binary32 single precision format as a C double.

double **PyFloat_Unpack8**(const char *p, int le)

Unpack the IEEE 754 binary64 double precision format as a C double.

9.2.4 🄵數物件

從 C API 來看，Python 的🄵數物件被實作🄵兩種不同的型🄵：一種是公開給 Python 程式的 Python 物件，另一種是表示實際🄵數值的 C 結構。API 提供了與兩者一起作用的函式。

作🄵 C 結構的🄵數

請注意，接受這些結構作🄵參數🄵將它們作🄵結果回傳的函式是*按值 (by value)* 執行的，而不是透過指標取消參照 (dereference) 它們。這在整個 API 中都是一致的。

type **Py_complex**

相對於 Python 🄵數物件之數值部分的 C 結構。大多數處理🄵數物件的函式根據需求會使用這種型🄵的結構作🄵輸入或輸出值。

double **real**

double **imag**

該結構被定義🄵：

```
typedef struct {
    double real;
    double imag;
} Py_complex;
```

Py_complex **_Py_c_sum**(*Py_complex* left, *Py_complex* right)

以 C 的*Py_complex* 表示形式來回傳兩個🄵數之和。

Py_complex **_Py_c_diff**(*Py_complex* left, *Py_complex* right)

以 C 的*Py_complex* 表示形式來回傳兩個🄵數間的差。

Py_complex **_Py_c_neg**(*Py_complex* num)

以 C 的*Py_complex* 表示形式來回傳🄵數 *num* 的相反數 (negation)。

Py_complex **_Py_c_prod**(*Py_complex* left, *Py_complex* right)

以 C 的*Py_complex* 表示形式來回傳兩個🄵數的乘積。

Py_complex **_Py_c_quot**(*Py_complex* dividend, *Py_complex* divisor)

以 C 的*Py_complex* 表示形式來回傳兩個🄵數的商。

如果 *divisor* 🄵 null，則此方法會回傳零🄵將 `errno` 設定🄵 `EDOM`。

Py_complex **_Py_c_pow**(*Py_complex* num, *Py_complex* exp)

以 C 的*Py_complex* 表示形式來回傳 *num* 的 *exp* 次方的結果。

如果 *num* 🄵 null 且 *exp* 不是正實數，則此方法會回傳零🄵將 `errno` 設定🄵 `EDOM`。

Set `errno` to `ERANGE` on overflows.

作🄵 Python 物件的🄵數

type **PyComplexObject**

這個*PyObject* 的子型🄵代表一個 Python 🄵數物件。

PyTypeObject **PyComplex_Type**

🄵穩定 ABI 的一部分. 這個*PyTypeObject* 的實例代表 Python 🄵數型🄵。它與 Python 層中的 `complex` 是同一個物件。

int **PyComplex_Check**(*PyObject* *p)

如果其引數是一個*PyComplexObject* 或者是*PyComplexObject* 的子型🄵，則會回傳 true。這個函式不會失敗。

int **PyComplex_CheckExact**(*PyObject* *p)

如果其引數是一個*PyComplexObject*，但不是*PyComplexObject* 的子型🄵，則會回傳 true。這個函式不會失敗。

PyObject ***PyComplex_FromCComplex**(*Py_complex* v)

回傳值：新的參照。 從 C 的*Py_complex* 值建立一個新的 Python 🄵數物件。在錯誤時回傳 `NULL` 🄵設定例外。

PyObject ***PyComplex_FromDoubles**(double real, double imag)

回傳值：新的參照。🄵穩定 ABI 的一部分. 從 *real* 和 *imag* 回傳一個新的*PyComplexObject* 物件。在錯誤時回傳 `NULL` 🄵設定例外。

double **PyComplex_RealAsDouble**(*PyObject* *op)

🄵穩定 ABI 的一部分. 以 C 的 `double` 形式回傳 *op* 的實部。

如果 *op* 不是 Python 🄵數物件，但有一個 `__complex__()` 方法，則首先會呼叫該方法將 *op* 轉🄵🄵 Python 🄵數物件。如果 `__complex__()` 🄵未定義，那🄵它會回退到呼叫 `PyFloat_AsDouble()` 🄵回傳其結果。

失敗時，此方法回傳 `-1.0` 🄵設定例外，因此應該呼叫*PyErr_Occurred()* 來檢查錯誤。

在 3.13 版的變更: 如果可用則使用 `__complex__()`。

double **PyComplex_ImagAsDouble**(*PyObject* *op)

🄵穩定 ABI 的一部分. 將 *op* 的🄵部作🄵 C 的 `double` 回傳。

如果 *op* 不是 Python 🄵數物件，但有一個 `__complex__()` 方法，則首先會呼叫該方法將 *op* 轉🄵🄵 Python 🄵數物件。如果 `__complex__()` 🄵未定義，那🄵它會回退到呼叫 `PyFloat_AsDouble()` 🄵於成功時回傳 `0.0`。

失敗時，此方法回傳 `-1.0` 🄵設定例外，因此應該呼叫*PyErr_Occurred()* 來檢查錯誤。

在 3.13 版的變更: 如果可用則使用 `__complex__()`。

Py_complex **PyComplex_AsCComplex**(*PyObject* *op)

回傳🄵數 *op* 的*Py_complex* 值。

如果 *op* 不是 Python 🄵數物件，但有一個 `__complex__()` 方法，則首先會呼叫該方法將 *op* 轉🄵🄵 Python 🄵數物件。如果 `__complex__()` 🄵未定義，那🄵它會回退到 `__float__()`。如果 `__float__()` 未定義，則它將繼續回退🄵 `__index__()`。

失敗時，此方法回傳*Py_complex* 🄵將*real* 設🄵 `-1.0`，🄵設定例外，因此應該呼叫*PyErr_Occurred()* 來檢查錯誤。

在 3.8 版的變更: 如果可用則使用 `__index__()`。

9.3 序列物件

序列物件的一般操作在前一章節討論過了；此段落將討論 Python 語言特有的特定型🄵序列物件。

9.3.1 位元組物件 (Bytes Objects)

These functions raise `TypeError` when expecting a bytes parameter and called with a non-bytes parameter.

type **PyBytesObject**

This subtype of *PyObject* represents a Python bytes object.

PyTypeObject **PyBytes_Type**

🄵穩定 ABI 的一部分. This instance of *PyTypeObject* represents the Python bytes type; it is the same object as `bytes` in the Python layer.

int **PyBytes_Check**(*PyObject* *o)

Return true if the object *o* is a bytes object or an instance of a subtype of the bytes type. This function always succeeds.

int **PyBytes_CheckExact**(*PyObject* *o)

Return true if the object *o* is a bytes object, but not an instance of a subtype of the bytes type. This function always succeeds.

PyObject ***PyBytes_FromString**(const char *v)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new bytes object with a copy of the string *v* as value on success, and `NULL` on failure. The parameter *v* must not be `NULL`; it will not be checked.

PyObject ***PyBytes_FromStringAndSize**(const char *v, *Py_ssize_t* len)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new bytes object with a copy of the string *v* as value and length *len* on success, and `NULL` on failure. If *v* is `NULL`, the contents of the bytes object are uninitialized.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python bytes object and return a bytes object with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* string. The following format characters are allowed:

Format Characters	Type	Comment
`%%`	*n/a*	The literal % character.
`%c`	int	A single byte, represented as a C int.
`%d`	int	等價於 `printf("%d")`.[1]
`%u`	unsigned int	等價於 `printf("%u")`.[1]
`%ld`	long	等價於 `printf("%ld")`.[1]
`%lu`	unsigned long	等價於 `printf("%lu")`.[1]
`%zd`	*Py_ssize_t*	等價於 `printf("%zd")`.[1]
`%zu`	size_t	等價於 `printf("%zu")`.[1]
`%i`	int	等價於 `printf("%i")`.[1]
`%x`	int	等價於 `printf("%x")`.[1]
`%s`	const char*	A null-terminated C character array.
`%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.

An unrecognized format character causes all the rest of the format string to be copied as-is to the result object, and any extra arguments discarded.

PyObject ***PyBytes_FromFormatV**(const char *format, va_list vargs)

回傳值：新的參照。🄵穩定 ABI 的一部分. Identical to *PyBytes_FromFormat()* except that it takes exactly two arguments.

PyObject ***PyBytes_FromObject**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return the bytes representation of object *o* that implements the buffer protocol.

Py_ssize_t **PyBytes_Size**(*PyObject* *o)

🄵穩定 ABI 的一部分. Return the length of the bytes in bytes object *o*.

Py_ssize_t **PyBytes_GET_SIZE**(*PyObject* *o)

和*PyBytes_Size()* 類似，但不進行錯誤檢查。

char ***PyBytes_AsString**(*PyObject* *o)

🄵穩定 ABI 的一部分. Return a pointer to the contents of *o*. The pointer refers to the internal buffer of *o*, which consists of `len(o) + 1` bytes. The last byte in the buffer is always null, regardless of whether there are any other null bytes. The data must not be modified in any way, unless the object was just created using `PyBytes_FromStringAndSize(NULL, size)`. It must not be deallocated. If *o* is not a bytes object at all, *PyBytes_AsString()* returns `NULL` and raises `TypeError`.

char ***PyBytes_AS_STRING**(*PyObject* *string)

和*PyBytes_AsString()* 類似，但不進行錯誤檢查。

int **PyBytes_AsStringAndSize**(*PyObject* *obj, char **buffer, *Py_ssize_t* *length)

🄵穩定 ABI 的一部分. Return the null-terminated contents of the object *obj* through the output variables *buffer* and *length*. Returns `0` on success.

If *length* is `NULL`, the bytes object may not contain embedded null bytes; if it does, the function returns `-1` and a `ValueError` is raised.

The buffer refers to an internal buffer of *obj*, which includes an additional null byte at the end (not counted in *length*). The data must not be modified in any way, unless the object was just created using `PyBytes_FromStringAndSize(NULL, size)`. It must not be deallocated. If *obj* is not a bytes object at all, *PyBytes_AsStringAndSize()* returns `-1` and raises `TypeError`.

在 3.5 版的變更: Previously, `TypeError` was raised when embedded null bytes were encountered in the bytes object.

[1] For integer specifiers (d, u, ld, lu, zd, zu, i, x): the 0-conversion flag has effect even when a precision is given.

void **PyBytes_Concat**(*PyObject* **bytes, *PyObject* *newpart)

🄵穩定 ABI 的一部分. Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*; the caller will own the new reference. The reference to the old value of *bytes* will be stolen. If the new object cannot be created, the old reference to *bytes* will still be discarded and the value of **bytes* will be set to `NULL`; the appropriate exception will be set.

void **PyBytes_ConcatAndDel**(*PyObject* **bytes, *PyObject* *newpart)

🄵穩定 ABI 的一部分. 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)

類似 Python 中的 `sep.join(iterable)`。

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`.

在 3.14 版被加入.

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 位元組陣列物件 (Byte Array Objects)

type **PyByteArrayObject**

這個*`PyObject`* 的子型🄵代表了 Python 的位元組陣列物件。

PyTypeObject **PyByteArray_Type**

🄵穩定 ABI 的一部分. 這個*`PyTypeObject`* 的實例代表了 Python 的位元組陣列型🄵；在 Python 層中的 `bytearray` 🄵同一個物件。

型🄵檢查巨集

int **PyByteArray_Check**(*PyObject* *o)

如果物件 *o* 是一個位元組陣列物件，或者是位元組陣列型🄵的子型🄵的實例，則回傳真值。此函式總是會成功執行。

int **PyByteArray_CheckExact**(*PyObject* *o)

如果物件 *o* 是一個位元組陣列物件，但不是位元組陣列型🄵的子型🄵的實例，則回傳真值。此函式總是會成功執行。

直接 API 函式

PyObject ***PyByteArray_FromObject**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 由任意物件 *o* 回傳一個新的位元組陣列物件，🄵實作了緩衝協定 *(buffer protocol)*。

在失敗時，會回傳 `NULL` 🄵設定例外。

PyObject ***PyByteArray_FromStringAndSize**(const char *string, *Py_ssize_t* len)

回傳值：新的參照。🄵穩定 ABI 的一部分. 從 *string* 及其長度 *len* 建立一個新的位元組陣列物件。

在失敗時，會回傳 `NULL` 🄵設定例外。

PyObject ***PyByteArray_Concat**(*PyObject* *a, *PyObject* *b)

回傳值：新的參照。🄵穩定 ABI 的一部分. 連接位元組陣列 *a* 和 *b*，🄵回傳一個包含結果的新位元組陣列。

在失敗時，會回傳 `NULL` 🄵設定例外。

Py_ssize_t **PyByteArray_Size**(*PyObject* *bytearray)

🄵穩定 ABI 的一部分. 在檢查🄵 `NULL` 指標後，回傳 *bytearray* 的大小。

char ***PyByteArray_AsString**(*PyObject* *bytearray)

🄵穩定 ABI 的一部分. 在檢查是否🄵 `NULL` 指標後，將 *bytearray* 的🄵容回傳🄵字元陣列。回傳的陣列總是會多附加一個空位元組。

int **PyByteArray_Resize**(*PyObject* *bytearray, *Py_ssize_t* len)

🄵穩定 ABI 的一部分. 將 *bytearray* 的🄵部緩衝區大小調整🄵 *len*。失敗時會回傳 `-1` 🄵設定例外。

在 3.14 版的變更: 負的 *len* 現在會設定例外🄵回傳 -1。

巨集

這些巨集犧牲了安全性以🄵取速度，🄵且它們不會檢查指標。

char ***PyByteArray_AS_STRING**(*PyObject* *bytearray)

與*PyByteArray_AsString()* 類似，但🄵有錯誤檢查。

Py_ssize_t **PyByteArray_GET_SIZE**(*PyObject* *bytearray)

與*PyByteArray_Size()* 類似，但🄵有錯誤檢查。

9.3.3 Unicode 物件與編解碼器

Unicode 物件

Since the implementation of **PEP 393** in Python 3.3, Unicode objects internally use a variety of representations, in order to allow handling the complete range of Unicode characters while staying memory efficient. There are special cases for strings where all code points are below 128, 256, or 65536; otherwise, code points must be below 1114112 (which is the full Unicode range).

UTF-8 表示法會在需要時建立🄵快取在 Unicode 物件中。

> **備🄵**
>
> 自 Python 3.12 起，已移除*Py_UNICODE* 表示法，🄵標示🄵已🄵用的 API。更多資訊請參🄵 **PEP 623**。

Unicode 型🄵

這些是 Python 中用於 Unicode 實作的基本 Unicode 物件型🄵:

PyTypeObject **PyUnicode_Type**

🄵穩定 ABI 的一部分. This instance of *PyTypeObject* represents the Python Unicode type. It is exposed to Python code as `str`.

PyTypeObject **PyUnicodeIter_Type**

🄵穩定 ABI 的一部分. 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**

🄵穩定 ABI 的一部分. These types are typedefs for unsigned integer types wide enough to contain characters of 32 bits, 16 bits and 8 bits, respectively. When dealing with single Unicode characters, use *Py_UCS4*.

在 3.3 版被加入.

type **PyASCIIObject**

type **PyCompactUnicodeObject**

type **PyUnicodeObject**

These subtypes of *PyObject* represent a Python Unicode object. In almost all cases, they shouldn't be used directly, since all API functions that deal with Unicode objects take and return *PyObject* pointers.

在 3.3 版被加入.

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).

在 3.3 版被加入.

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.

在 3.3 版被加入.

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND

PyUnicode_4BYTE_KIND

Return values of the *PyUnicode_KIND()* macro.

在 3.3 版被加入.

在 3.12 版的變更: `PyUnicode_WCHAR_KIND` 已被移除。

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).

在 3.3 版被加入.

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).

在 3.3 版被加入.

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.

在 3.3 版被加入.

Py_UCS4 **PyUnicode_READ**(int kind, void *data, *Py_ssize_t* index)

Read a code point from a canonical representation *data* (as obtained with *PyUnicode_DATA()*). No checks or ready calls are performed.

在 3.3 版被加入.

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.

在 3.3 版被加入.

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.

在 3.3 版被加入.

int **PyUnicode_IsIdentifier**(*PyObject* *unicode)

🄵穩定 ABI 的一部分. Return 1 if the string is a valid identifier according to the language definition, section identifiers. Return 0 otherwise.

在 3.9 版的變更: The function does not call *Py_FatalError()* anymore if the string is not ready.

unsigned int **PyUnicode_IS_ASCII**(*PyObject* *unicode)

Return true if the string only contains ASCII characters. Equivalent to str.isascii().

在 3.2 版被加入.

Unicode Character Properties

Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration.

int **Py_UNICODE_ISSPACE**(*Py_UCS4* ch)

根據 *ch* 是否🄵空白字元來回傳 1 或 0。

int **Py_UNICODE_ISLOWER**(*Py_UCS4* ch)

根據 *ch* 是否🄵小寫字元來回傳 1 或 0。

int **Py_UNICODE_ISUPPER**(*Py_UCS4* ch)

根據 *ch* 是否🄵大寫字元來回傳 1 或 0。

int **Py_UNICODE_ISTITLE**(*Py_UCS4* ch)

根據 *ch* 是否🄵首字大寫字元來回傳 1 或 0。

int **Py_UNICODE_ISLINEBREAK**(*Py_UCS4* ch)

根據 *ch* 是否🄵🄵行字元來回傳 1 或 0。

int **Py_UNICODE_ISDECIMAL**(*Py_UCS4* ch)

根據 *ch* 是否🄵十進位字元來回傳 1 或 0。

int **Py_UNICODE_ISDIGIT**(*Py_UCS4* ch)
根據 *ch* 是否🄵數字 (digit) 字元來回傳 `1` 或 `0`。

int **Py_UNICODE_ISNUMERIC**(*Py_UCS4* ch)
根據 *ch* 是否🄵數值 (numeric) 字元來回傳 `1` 或 `0`。

int **Py_UNICODE_ISALPHA**(*Py_UCS4* ch)
根據 *ch* 是否🄵字母字元來回傳 `1` 或 `0`。

int **Py_UNICODE_ISALNUM**(*Py_UCS4* ch)
根據 *ch* 是否🄵字母數字 (alphanumeric) 字元來回傳 `1` 或 `0`。

int **Py_UNICODE_ISPRINTABLE**(*Py_UCS4* ch)
根據 *ch* 是否🄵可列印字元（如 `str.isprintable()` 所定義）來回傳 `1` 或 `0`。

這些 API 可用於快速直接字元轉🄵:

Py_UCS4 **Py_UNICODE_TOLOWER**(*Py_UCS4* ch)
回傳轉🄵🄵小寫的 *ch* 字元。

Py_UCS4 **Py_UNICODE_TOUPPER**(*Py_UCS4* ch)
回傳轉🄵🄵大寫的 *ch* 字元。

Py_UCS4 **Py_UNICODE_TOTITLE**(*Py_UCS4* ch)
回傳轉🄵🄵首字大寫的 *ch* 字元。

int **Py_UNICODE_TODECIMAL**(*Py_UCS4* ch)
回傳轉🄵🄵十進位正整數的 *ch* 字元，若無法轉🄵則回傳 `-1`。此函式不會引發例外。

int **Py_UNICODE_TODIGIT**(*Py_UCS4* ch)
回傳轉🄵🄵單一數字整數的 *ch* 字元，若無法轉🄵則回傳 `-1`。此函式不會引發例外。

double **Py_UNICODE_TONUMERIC**(*Py_UCS4* ch)
回傳轉🄵🄵雙精度浮點數 (double) 的 *ch* 字元，若無法轉🄵則回傳 `-1.0`。此函式不會引發例外。

這些 API 可用於處理代理字元:

int **Py_UNICODE_IS_SURROGATE**(*Py_UCS4* ch)
檢查 *ch* 是否🄵代理字元 (surrogate, `0xD800 <= ch <= 0xDFFF`)。

int **Py_UNICODE_IS_HIGH_SURROGATE**(*Py_UCS4* ch)
檢查 *ch* 是否🄵高代理字元 (high surrogate, `0xD800 <= ch <= 0xDBFF`)。

int **Py_UNICODE_IS_LOW_SURROGATE**(*Py_UCS4* ch)
檢查 *ch* 是否🄵低代理字元 (low surrogate, `0xDC00 <= 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].

Creating and accessing Unicode strings

To create Unicode objects and access their basic sequence properties, use these APIs:

PyObject ***PyUnicode_New**(*Py_ssize_t* size, *Py_UCS4* maxchar)
回傳值：新的參照。 Create a new Unicode object. *maxchar* should be the true maximum code point to be placed in the string. As an approximation, it can be rounded up to the nearest value in the sequence 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.

在 3.3 版被加入.

PyObject ***PyUnicode_FromKindAndData**(int kind, const void *buffer, *Py_ssize_t* size)

回傳值：新的參照。 Create a new Unicode object with the given *kind* (possible values are *PyUnicode_1BYTE_KIND* etc., as returned by *PyUnicode_KIND()*). The *buffer* must point to an array of *size* units of 1, 2 or 4 bytes per character, as given by the kind.

If necessary, the input *buffer* is copied and transformed into the canonical representation. For example, if the *buffer* is a UCS4 string (*PyUnicode_4BYTE_KIND*) and it consists only of codepoints in the UCS1 range, it will be transformed into UCS1 (*PyUnicode_1BYTE_KIND*).

在 3.3 版被加入.

PyObject ***PyUnicode_FromStringAndSize**(const char *str, *Py_ssize_t* size)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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.

此函式在以下情🄵下會引發 `SystemError`：

- *size* < 0,
- *str* 🄵 `NULL` 且 *size* > 0

在 3.12 版的變更: *str* == `NULL` 且 *size* > 0 不再被允許。

PyObject ***PyUnicode_FromString**(const char *str)

回傳值：新的參照。🄵穩定 ABI 的一部分. Create a Unicode object from a UTF-8 encoded null-terminated char buffer *str*.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 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:

旗標	含義
`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` 或 `unsigned long`
`ll`	`long long` 或 `unsigned long long`
`j`	`intmax_t` 或 `uintmax_t`
`z`	`size_t` 或 `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	Type	Comment
`%`	*n/a*	字面 `%` 字元。
`d`, `i`	Specified by the length modifier	一個有符號 C 整數的十進位表示法。
`u`	Specified by the length modifier	一個無符號 C 整數的十進位表示法。
`o`	Specified by the length modifier	一個無符號 C 整數的八進位表示法。
`x`	Specified by the length modifier	一個無符號 C 整數的十六進位表示法（小寫）。
`X`	Specified by the length modifier	一個無符號 C 整數的十六進位表示法（大寫）。
`c`	`int`	一個單一字元。
`s`	`const char*` 或 `const wchar_t*`	一個以 null 結尾的 C 字元陣列。
`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**	呼叫 `ascii()` 的結果。
`U`	*PyObject**	一個 Unicode 物件。
`V`	*PyObject**、`const char*` 或 `const wchar_t*`	A Unicode object (which may be `NULL`) and a null-terminated C character array as a second parameter (which will be used, if the first parameter is `NULL`).
`S`	*PyObject**	呼叫*PyObject_Str()* 的結果。
`R`	*PyObject**	呼叫*PyObject_Repr()* 的結果。
`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.

備🄵

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`).

備🄵

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`).

在 3.2 版的變更: 新增對 `"%lld"` 和 `"%llu"` 的支援。

在 3.3 版的變更: 新增對 `"%li"`、`"%lli"` 和 `"%zi"` 的支援。

在 3.4 版的變更: 新增對 `"%s"`、`"%A"`、`"%U"`、`"%V"`、`"%S"`、`"%R"` 的寬度和精確度格式化支援。

在 3.12 版的變更: 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.

在 3.13 版的變更: 新增對 `%T`、`%#T`、`%N` 和 `%#N` 格式的支援。

PyObject ***PyUnicode_FromFormatV**(const char *format, va_list vargs)

回傳值：新的參照。🄵穩定 ABI 的一部分. Identical to *`PyUnicode_FromFormat()`* except that it takes exactly two arguments.

PyObject ***PyUnicode_FromObject**(*PyObject* *obj)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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.

Objects other than Unicode or its subtypes will cause a `TypeError`.

PyObject ***PyUnicode_FromOrdinal**(int ordinal)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Decode an encoded object *obj* to a Unicode object.

`bytes`, `bytearray` and other *bytes-like objects* are decoded according to the given *encoding* and using the error handling defined by *errors*. Both can be `NULL` to have the interface use the default values (see 🄵建編解碼器 for details).

All other objects, including Unicode objects, cause a `TypeError` to be set.

The API returns `NULL` if there was an error. The caller is responsible for decref'ing the returned objects.

void **PyUnicode_Append**(*PyObject* **p_left, *PyObject* *right)

🄵穩定 ABI 的一部分. 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.

於錯誤發生時，將 **p_left* 設🄵 `NULL` 🄵設定例外。

On success, set **p_left* to a new strong reference to the result.

void **PyUnicode_AppendAndDel**(*PyObject* **p_left, *PyObject* *right)

🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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.

在 3.2 版被加入.

const char ***PyUnicode_GetDefaultEncoding**(void)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Return the length of the Unicode object, in code points.

發生錯誤時，設定例外🄵回傳 `-1`。

在 3.3 版被加入.

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.

在 3.3 版被加入.

int **PyUnicode_Resize**(*PyObject* **unicode, *Py_ssize_t* length);

🄵穩定 ABI 的一部分. 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)

Fill a string with a character: write *fill_char* into `unicode[start:start+length]`.

Fail if *fill_char* is bigger than the string maximum character, or if the string has more than 1 reference.

The string must not have been “used” yet. See *PyUnicode_New()* for details.

Return the number of written character, or return `-1` and raise an exception on error.

在 3.3 版被加入.

int **PyUnicode_WriteChar**(*PyObject* *unicode, *Py_ssize_t* index, *Py_UCS4* character)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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.

在 3.3 版被加入.

Py_UCS4 **PyUnicode_ReadChar**(*PyObject* *unicode, *Py_ssize_t* index)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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.

成功時回傳字元，發生錯誤時設定例外🄵回傳 `-1`。

在 3.3 版被加入.

PyObject ***PyUnicode_Substring**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* end)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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`.

在 3.3 版被加入.

Py_UCS4 ***PyUnicode_AsUCS4**(*PyObject* *unicode, *Py_UCS4* *buffer, *Py_ssize_t* buflen, int copy_null)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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.

在 3.3 版被加入.

Py_UCS4 ***PyUnicode_AsUCS4Copy**(*PyObject* *unicode)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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.

在 3.3 版被加入.

Locale Encoding

The current locale encoding can be used to decode text from the operating system.

PyObject ***PyUnicode_DecodeLocaleAndSize**(const char *str, *Py_ssize_t* length, const char *errors)

回傳值: 新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Decode a string from UTF-8 on Android and VxWorks, or from the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The decoder uses `"strict"` error handler if *errors* is `NULL`. *str* must end with a null character but cannot contain embedded null characters.

Use *PyUnicode_DecodeFSDefaultAndSize()* to decode a string from the *filesystem encoding and error handler*.

此函式會忽略 Python UTF-8 模式。

也參考

Py_DecodeLocale() 函式。

在 3.3 版被加入.

在 3.7 版的變更: The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, *Py_DecodeLocale()* was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

PyObject ***PyUnicode_DecodeLocale**(const char *str, const char *errors)

回傳值: 新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 類似於*PyUnicode_DecodeLocaleAndSize()*, 但使用 `strlen()` 計算字串長度。

在 3.3 版被加入.

PyObject ***PyUnicode_EncodeLocale**(*PyObject* *unicode, const char *errors)

回傳值: 新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Encode a Unicode object to UTF-8 on Android and VxWorks, or to the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The encoder uses `"strict"` error handler if *errors* is `NULL`. Return a `bytes` object. *unicode* cannot contain embedded null characters.

Use *PyUnicode_EncodeFSDefault()* to encode a string to the *filesystem encoding and error handler*.

此函式會忽略 Python UTF-8 模式。

也參考

Py_EncodeLocale() 函式。

在 3.3 版被加入.

在 3.7 版的變更: The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, *Py_EncodeLocale()* was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

檔案系統編碼

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)

🄵穩定 ABI 的一部分. *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`.

在 3.1 版被加入.

在 3.6 版的變更: 接受一個*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)

🄵穩定 ABI 的一部分. *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`.

在 3.2 版被加入.

在 3.6 版的變更: 接受一個*path-like object*。

PyObject ***PyUnicode_DecodeFSDefaultAndSize**(const char *str, *Py_ssize_t* size)

回傳值：新的參照。🄵穩定 ABI 的一部分. Decode a string from the *filesystem encoding and error handler*.

If you need to decode a string from the current locale encoding, use *PyUnicode_DecodeLocaleAndSize()*.

也參考

Py_DecodeLocale() 函式。

在 3.6 版的變更: The *filesystem error handler* is now used.

PyObject ***PyUnicode_DecodeFSDefault**(const char *str)

回傳值：新的參照。🄵穩定 ABI 的一部分. Decode a null-terminated string from the *filesystem encoding and error handler*.

如果字串長度已知，請使用*PyUnicode_DecodeFSDefaultAndSize()*。

在 3.6 版的變更: The *filesystem error handler* is now used.

PyObject ***PyUnicode_EncodeFSDefault**(*PyObject* *unicode)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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()*.

也參考

Py_EncodeLocale() 函式。

在 3.2 版被加入.

在 3.6 版的變更: The *filesystem error handler* is now used.

wchar_t 支援

對支援 `wchar_t` 的平台提供支援：

PyObject ***PyUnicode_FromWideChar**(const wchar_t *wstr, *Py_ssize_t* size)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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.

在 3.2 版被加入.

在 3.7 版的變更: Raises a `ValueError` if *size* is `NULL` and the `wchar_t*` string contains null characters.

🄵建編解碼器

Python provides a set of built-in codecs which are written in C for speed. All of these codecs are directly usable via the following functions.

Many of the following APIs take two arguments encoding and errors, and they have the same semantics as the ones of the built-in `str()` string object constructor.

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.

Error handling is set by errors which may also be set to `NULL` meaning to use the default handling defined for the codec. Default error handling for all built-in codecs is "strict" (`ValueError` is raised).

The codecs all use a similar interface. Only deviations from the following generic ones are documented for simplicity.

泛用編解碼器

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".

這些是泛用編解碼器的 API：

PyObject ***PyUnicode_Decode**(const char *str, *Py_ssize_t* size, const char *encoding, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Encode a Unicode object and return the result as Python bytes object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode `encode()` method. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

UTF-8 編解碼器

這些是 UTF-8 編解碼器的 API：

PyObject ***PyUnicode_DecodeUTF8**(const char *str, *Py_ssize_t* size, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. If *consumed* is `NULL`, behave like *PyUnicode_DecodeUTF8()*. If *consumed* is not `NULL`, trailing incomplete UTF-8 byte sequences will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF8String**(*PyObject* *unicode)

回傳值：新的參照。🄵穩定 ABI 的一部分. Encode a Unicode object using UTF-8 and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

The function fails if the string contains surrogate code points (`U+D800` - `U+DFFF`).

const char ***PyUnicode_AsUTF8AndSize**(*PyObject* *unicode, *Py_ssize_t* *size)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Return a pointer to the UTF-8 encoding of the Unicode object, and store the size of the encoded representation (in bytes) in *size*. The *size* argument can be `NULL`; in this case no size will be stored. The returned buffer always has an extra null byte appended (not included in *size*), regardless of whether there are any other null code points.

發生錯誤時，設定例外🄵將 *size* 設🄵 `-1`（如果不是 NULL），🄵回傳 `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.

在 3.3 版被加入.

在 3.7 版的變更: 回傳型🄵現在是 `const char *` 而不是 `char *`。

在 3.10 版的變更: This function is a part of the *limited API*.

const char ***PyUnicode_AsUTF8**(*PyObject* *unicode)

與*PyUnicode_AsUTF8AndSize()* 類似，但不儲存大小。

> **⚠ 警告**
>
> 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.

在 3.3 版被加入.

在 3.7 版的變更: 回傳型🄵現在是 `const char *` 而不是 `char *`。

UTF-32 編解碼器

這些是 UTF-32 編解碼器的 API:

PyObject ***PyUnicode_DecodeUTF32**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

回傳值: 新的參照。🄵穩定 ABI 的一部分. Decode *size* bytes from a UTF-32 encoded buffer string and return the corresponding Unicode object. *errors* (if non-`NULL`) defines the error handling. It defaults to "strict".

If *byteorder* is non-`NULL`, the decoder starts decoding using the given byte order:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

If `*byteorder` is zero, and the first four bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output.

After completion, **byteorder* is set to the current byte order at the end of input data.

If *byteorder* is `NULL`, the codec starts in native order mode.

如果編解碼器引發例外則回傳 `NULL`。

PyObject ***PyUnicode_DecodeUTF32Stateful**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

回傳值: 新的參照。🄵穩定 ABI 的一部分. If *consumed* is `NULL`, behave like *PyUnicode_DecodeUTF32()*. If *consumed* is not `NULL`, *PyUnicode_DecodeUTF32Stateful()* will not treat trailing incomplete UTF-32 byte sequences (such as a number of bytes not divisible by four) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF32String**(*PyObject* *unicode)

回傳值: 新的參照。🄵穩定 ABI 的一部分. Return a Python byte string using the UTF-32 encoding in native byte order. The string always starts with a BOM mark. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

UTF-16 編解碼器

這些是 UTF-16 編解碼器的 API:

PyObject ***PyUnicode_DecodeUTF16**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

回傳值：新的參照。🄵穩定 ABI 的一部分. Decode *size* bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. *errors* (if non-`NULL`) defines the error handling. It defaults to "strict".

If *byteorder* is non-`NULL`, the decoder starts decoding using the given byte order:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

If `*byteorder` is zero, and the first two bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output (where it will result in either a `\ufeff` or a `\ufffe` character).

After completion, `*byteorder` is set to the current byte order at the end of input data.

If *byteorder* is `NULL`, the codec starts in native order mode.

如果編解碼器引發例外則回傳 `NULL`。

PyObject ***PyUnicode_DecodeUTF16Stateful**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

回傳值：新的參照。🄵穩定 ABI 的一部分. If *consumed* is `NULL`, behave like *PyUnicode_DecodeUTF16()*. If *consumed* is not `NULL`, *PyUnicode_DecodeUTF16Stateful()* will not treat trailing incomplete UTF-16 byte sequences (such as an odd number of bytes or a split surrogate pair) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF16String**(*PyObject* *unicode)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a Python byte string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

UTF-7 編解碼器

這些是 UTF-7 編解碼器的 API：

PyObject ***PyUnicode_DecodeUTF7**(const char *str, *Py_ssize_t* size, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. If *consumed* is `NULL`, behave like *PyUnicode_DecodeUTF7()*. If *consumed* is not `NULL`, trailing incomplete UTF-7 base-64 sections will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

Unicode-Escape Codecs

These are the "Unicode Escape" codec APIs:

PyObject ***PyUnicode_DecodeUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Encode a Unicode object using Unicode-Escape and return the result as a bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

Raw-Unicode-Escape Codecs

These are the "Raw Unicode Escape" codec APIs:

PyObject ***PyUnicode_DecodeRawUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Encode a Unicode object using Raw-Unicode-Escape and return the result as a bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

Latin-1 編解碼器

These are the Latin-1 codec APIs: Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

PyObject ***PyUnicode_DecodeLatin1**(const char *str, *Py_ssize_t* size, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Encode a Unicode object using Latin-1 and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

ASCII 編解碼器

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

PyObject ***PyUnicode_DecodeASCII**(const char *str, *Py_ssize_t* size, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Encode a Unicode object using ASCII and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

Character Map Codecs

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.

這些是對映編解碼器的 API：

PyObject ***PyUnicode_DecodeCharmap**(const char *str, *Py_ssize_t* length, *PyObject* *mapping, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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.

If *mapping* is `NULL`, Latin-1 decoding will be applied. Else *mapping* must map bytes ordinals (integers in the range from 0 to 255) to Unicode strings, integers (which are then interpreted as Unicode ordinals) or `None`. Unmapped data bytes -- ones which cause a `LookupError`, as well as ones which get mapped to `None`, `0xFFFE` or `'\ufffe'`, are treated as undefined mappings and cause an error.

PyObject ***PyUnicode_AsCharmapString**(*PyObject* *unicode, *PyObject* *mapping)

回傳值：新的參照。🄵穩定 ABI 的一部分. Encode a Unicode object using the given *mapping* object and return the result as a bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

The *mapping* object must map Unicode ordinal integers to bytes objects, integers in the range from 0 to 255 or `None`. Unmapped character ordinals (ones which cause a `LookupError`) as well as mapped to `None` are treated as "undefined mapping" and cause an error.

The following codec API is special in that maps Unicode to Unicode.

PyObject ***PyUnicode_Translate**(*PyObject* *unicode, *PyObject* *table, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分. Translate a string by applying a character mapping table to it and return the resulting Unicode object. Return `NULL` if an exception was raised by the codec.

The mapping table must map Unicode ordinal integers to Unicode ordinal integers or `None` (causing deletion of the character).

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 has the usual meaning for codecs. It may be `NULL` which indicates to use the default error handling.

Windows 的 MBCS 編解碼器

These are the MBCS codec APIs. They are currently only available on Windows and use the Win32 MBCS converters to implement the conversions. Note that MBCS (or DBCS) is a class of encodings, not just one. The target encoding is defined by the user settings on the machine running the codec.

PyObject ***PyUnicode_DecodeMBCS**(const char *str, *Py_ssize_t* size, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. If *consumed* is `NULL`, behave like *PyUnicode_DecodeMBCS()*. If *consumed* is not `NULL`, *PyUnicode_DecodeMBCSStateful()* will not decode trailing lead byte and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_DecodeCodePageStateful**(int code_page, const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

回傳值：新的參照。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. Similar to *PyUnicode_DecodeMBCSStateful()*, except uses the code page specified by *code_page*.

PyObject ***PyUnicode_AsMBCSString**(*PyObject* *unicode)

回傳值：新的參照。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. Encode a Unicode object using MBCS and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_EncodeCodePage**(int code_page, *PyObject* *unicode, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 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.

在 3.3 版被加入.

Methods and Slot Functions

The following APIs are capable of handling Unicode objects and strings on input (we refer to them as strings in the descriptions) and return Unicode objects or integers as appropriate.

如果發生例外，則回傳 `NULL` 或 `-1`。

PyObject ***PyUnicode_Concat**(*PyObject* *left, *PyObject* *right)

回傳值：新的參照。🄵穩定 ABI 的一部分. Concat two strings giving a new Unicode string.

PyObject ***PyUnicode_Split**(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

回傳值：新的參照。🄵穩定 ABI 的一部分. Split a string giving a list of Unicode strings. If *sep* is `NULL`, splitting will be done at all whitespace substrings. Otherwise, splits occur at the given separator. At most *maxsplit* splits will be done. If negative, no limit is set. Separators are not included in the resulting list.

於錯誤發生時回傳 `NULL` 🄵設定例外。

等價於 `str.split()`。

PyObject ***PyUnicode_RSplit**(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

回傳值：新的參照。🄵穩定 ABI 的一部分. Similar to *PyUnicode_Split()*, but splitting will be done beginning at the end of the string.

於錯誤發生時回傳 `NULL` 🄵設定例外。

等價於 `str.rsplit()`。

PyObject ***PyUnicode_Splitlines**(*PyObject* *unicode, int keepends)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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 不得🄵空。

於錯誤發生時回傳 `NULL` 🄵設定例外。

等價於 `str.partition()`。

PyObject ***PyUnicode_RPartition**(*PyObject* *unicode, *PyObject* *sep)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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 不得🄵空。

於錯誤發生時回傳 `NULL` 🄵設定例外。

等價於 `str.rpartition()`。

PyObject ***PyUnicode_Join**(*PyObject* *separator, *PyObject* *seq)

回傳值：新的參照。🄵穩定 ABI 的一部分. Join a sequence of strings using the given *separator* and return the resulting Unicode string.

Py_ssize_t **PyUnicode_Tailmatch**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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.

在 3.3 版被加入.

在 3.7 版的變更: *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)

🄵穩定 ABI 的一部分. 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)

回傳值: 新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. Compare two strings and return `-1`, `0`, `1` for less than, equal, and greater than, respectively.

This function returns `-1` upon failure, so one should call *`PyErr_Occurred()`* to check for errors.

也參考

`PyUnicode_Equal()` 函式。

int **PyUnicode_Equal**(*PyObject* *a, *PyObject* *b)

🄵穩定 ABI 的一部分 自 *3.14* 版本開始. 測試兩個字串是否相等:

- 如果 *a* 等於 *b*，則回傳 `1`。
- 如果 *a* 不等於 *b*，則回傳 `0`。
- 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.

也參考

`PyUnicode_Compare()` 函式。

在 3.14 版被加入.

int **PyUnicode_EqualToUTF8AndSize**(*PyObject* *unicode, const char *string, *Py_ssize_t* size)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 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.

此函式不會引發例外。

在 3.13 版被加入.

int **PyUnicode_EqualToUTF8**(*PyObject* *unicode, const char *string)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Similar to *`PyUnicode_EqualToUTF8AndSize()`*, but compute *string* length using `strlen()`. If the Unicode object contains null characters, false (`0`) is returned.

在 3.13 版被加入.

int **PyUnicode_CompareWithASCIIString**(*PyObject* *unicode, const char *string)

🄵穩定 ABI 的一部分. 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.

此函式不會引發例外。

PyObject ***PyUnicode_RichCompare**(*PyObject* *left, *PyObject* *right, int op)

回傳值：新的參照。🄵穩定 ABI 的一部分. Rich compare two Unicode strings and return one of the following:

- `NULL` in case an exception was raised
- *`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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new string object from *format* and *args*; this is analogous to `format % args`.

int **PyUnicode_Contains**(*PyObject* *unicode, *PyObject* *substr)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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.

CPython 實作細節： 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.

在 3.14 版被加入.

type **PyUnicodeWriter**

一個 Unicode 寫入器實例。

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.

發生錯誤時，設定例外🄵回傳 `NULL`。

PyObject ***PyUnicodeWriter_Finish**(*PyUnicodeWriter* *writer)

Return the final Python `str` object and destroy the writer instance.

發生錯誤時，設定例外🄵回傳 `NULL`。

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*.

成功時回傳 `0`，發生錯誤時設定例外、保持寫入器不變，🄵回傳 `-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.

成功時回傳 `0`，發生錯誤時設定例外、保持寫入器不變，🄵回傳 `-1`。

另請參🄵*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.

成功時回傳 `0`，發生錯誤時設定例外、保持寫入器不變，🄵回傳 `-1`。

在 3.14 版被加入.

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.

成功時回傳 `0`，發生錯誤時設定例外、保持寫入器不變，🄵回傳 `-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.

成功時回傳 `0`，發生錯誤時設定例外、保持寫入器不變，🄵回傳 `-1`。

int **PyUnicodeWriter_WriteStr**(*PyUnicodeWriter* *writer, *PyObject* *obj)

Call *PyObject_Str()* on *obj* and write the output into *writer*.

成功時回傳 0，發生錯誤時設定例外、保持寫入器不變，🄵回傳 -1。

int **PyUnicodeWriter_WriteRepr**(*PyUnicodeWriter* *writer, *PyObject* *obj)

Call *PyObject_Repr()* on *obj* and write the output into *writer*.

成功時回傳 0，發生錯誤時設定例外、保持寫入器不變，🄵回傳 -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.

成功時回傳 0，發生錯誤時設定例外、保持寫入器不變，🄵回傳 -1。

int **PyUnicodeWriter_Format**(*PyUnicodeWriter* *writer, const char *format, ...)

類似於*PyUnicode_FromFormat()*，但是直接將輸出寫入 *writer*。

成功時回傳 0，發生錯誤時設定例外、保持寫入器不變，🄵回傳 -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.

成功時回傳 0，發生錯誤時設定例外、保持寫入器不變，🄵回傳 -1。

另請參🄵*PyUnicodeWriter_WriteUTF8()*。

已🄵用的 API

以下 API 已🄵用。

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.

在 3.3 版的變更: In previous versions, this was a 16-bit type or a 32-bit type depending on whether you selected a "narrow" or "wide" Unicode version of Python at build time.

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.

在 3.3 版被加入.

在 3.10 版之後被🄵用: 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.

在 3.3 版被加入.

在 3.14 版之後被🄵用: This API does nothing since Python 3.12. Previously, this could be called to check if *PyUnicode_READY()* is necessary.

9.3.4 Tuple（元組）物件

type **PyTupleObject**

This subtype of *PyObject* represents a Python tuple object.

PyTypeObject **PyTuple_Type**

🄵穩定 ABI 的一部分. This instance of *PyTypeObject* represents the Python tuple type; it is the same object as `tuple` in the Python layer.

int **PyTuple_Check**(*PyObject* *p)

Return true if *p* is a tuple object or an instance of a subtype of the tuple type. This function always succeeds.

int **PyTuple_CheckExact**(*PyObject* *p)

Return true if *p* is a tuple object, but not an instance of a subtype of the tuple type. This function always succeeds.

PyObject ***PyTuple_New**(*Py_ssize_t* len)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new tuple object of size *len*, or `NULL` with an exception set on failure.

PyObject ***PyTuple_Pack**(*Py_ssize_t* n, ...)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

回傳值：借用參照。🄵穩定 ABI 的一部分. Return the object at position *pos* in the tuple pointed to by *p*. If *pos* is negative or out of bounds, return `NULL` and set an `IndexError` exception.

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)

回傳值：借用參照。 Like *PyTuple_GetItem()*, but does no checking of its arguments.

PyObject ***PyTuple_GetSlice**(*PyObject* *p, *Py_ssize_t* low, *Py_ssize_t* high)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. Insert a reference to object *o* at position *pos* of the tuple pointed to by *p*. Return `0` on success. If *pos* is out of bounds, return `-1` and set an `IndexError` exception.

> **備🄵**
>
> This function ”steals” a reference to *o* and discards a reference to an item already in the tuple at the affected position.

void **PyTuple_SET_ITEM**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Like `PyTuple_SetItem()`, but does no error checking, and should *only* be used to fill in brand new tuples.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

> **備🄵**
>
> This function "steals" a reference to *o*, and, unlike `PyTuple_SetItem()`, does *not* discard a reference to any item that is being replaced; any reference in the tuple at position *pos* will be leaked.

> **警告**
>
> 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)

Can be used to resize a tuple. *newsize* will be the new length of the tuple. Because tuples are *supposed* to be immutable, this should only be used if there is only one reference to the object. Do *not* use this if the tuple may already be known to some other part of the code. The tuple will always grow or shrink at the end. Think of this as destroying the old tuple and creating a new one, only more efficiently. Returns `0` on success. Client code should never assume that the resulting value of `*p` will be the same as before calling this function. If the object referenced by `*p` is replaced, the original `*p` is destroyed. On failure, returns `-1` and sets `*p` to `NULL`, and raises `MemoryError` or `SystemError`.

9.3.5 Struct Sequence Objects

Struct sequence objects are the C equivalent of `namedtuple()` objects, i.e. a sequence whose items can also be accessed through attributes. To create a struct sequence, you first have to create a specific struct sequence type.

PyTypeObject ***PyStructSequence_NewType**(*PyStructSequence_Desc* *desc)

回傳值：新的參照。🄵穩定 ABI 的一部分. Create a new struct sequence type from the data in *desc*, described below. Instances of the resulting type can be created with `PyStructSequence_New()`.

失敗時回傳 `NULL` 🄵設定例外。

void **PyStructSequence_InitType**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

Initializes a struct sequence type *type* from *desc* in place.

int **PyStructSequence_InitType2**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

Like `PyStructSequence_InitType()`, but returns `0` on success and `-1` with an exception set on failure.

在 3.4 版被加入.

type **PyStructSequence_Desc**

🄵穩定 ABI 的一部分（包含所有成員）. Contains the meta information of a struct sequence type to create.

const char ***name**

Fully qualified name of the type; null-terminated UTF-8 encoded. The name must contain the module name.

const char ***doc**

Pointer to docstring for the type or `NULL` to omit.

PyStructSequence_Field ***fields**

Pointer to `NULL`-terminated array with field names of the new type.

int **n_in_sequence**

Number of fields visible to the Python side (if used as tuple).

type **PyStructSequence_Field**

🄵穩定 ABI 的一部分 （包含所有成員）. Describes a field of a struct sequence. As a struct sequence is modeled as a tuple, all fields are typed as *PyObject**. The index in the *fields* array of the *PyStructSequence_Desc* determines which field of the struct sequence is described.

const char ***name**

Name for the field or `NULL` to end the list of named fields, set to *PyStructSequence_UnnamedField* to leave unnamed.

const char ***doc**

Field docstring or `NULL` to omit.

const char *const **PyStructSequence_UnnamedField**

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Special value for a field name to leave it unnamed.

在 3.9 版的變更: The type was changed from `char *`.

PyObject ***PyStructSequence_New**(*PyTypeObject* *type)

回傳值: 新的參照。🄵穩定 ABI 的一部分. Creates an instance of *type*, which must have been created with *PyStructSequence_NewType()*.

失敗時回傳 `NULL` 🄵設定例外。

PyObject ***PyStructSequence_GetItem**(*PyObject* *p, *Py_ssize_t* pos)

回傳值: 借用參照。🄵穩定 ABI 的一部分. 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)

回傳值: 借用參照。 Alias to *PyStructSequence_GetItem()*.

在 3.13 版的變更: Now implemented as an alias to *PyStructSequence_GetItem()*.

void **PyStructSequence_SetItem**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

🄵穩定 ABI 的一部分. Sets the field at index *pos* of the struct sequence *p* to value *o*. Like *PyTuple_SET_ITEM()*, this should only be used to fill in brand new instances.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

> **備🄵**
>
> This function ”steals” a reference to *o*.

void **PyStructSequence_SET_ITEM**(*PyObject* *p, *Py_ssize_t* *pos, *PyObject* *o)

Alias to *PyStructSequence_SetItem()*.

在 3.13 版的變更: Now implemented as an alias to *PyStructSequence_SetItem()*.

9.3.6 List（串列）物件

type **PyListObject**

PyObject 的這個子型🄵表示 Python 的 list（串列）物件。

PyTypeObject **PyList_Type**

🄵穩定 ABI 的一部分. 此*PyTypeObject* 實例表示 Python 的 list 型🄵。這與 Python 層中的 `list` 是同一個物件。

int **PyList_Check**(*PyObject* *p)
如果 *p* 是一個 list 物件或者是 list 型🄵之子型🄵的實例，就回傳 true。這個函式永遠會成功執行。

int **PyList_CheckExact**(*PyObject* *p)
如果 *p* 是一個 list 物件但不是 list 型🄵的子型🄵的實例，就回傳 true。這個函式永遠會成功執行。

PyObject ***PyList_New**(*Py_ssize_t* len)
回傳值：新的參照。🄵穩定 ABI 的一部分. 成功時回傳長度🄵 *len* 的新串列，失敗時回傳 NULL。

> **備🄵**
>
> 如果 *len* 大於零，則回傳的串列物件之項目將被設定🄵 NULL。因此，在使用*PyList_SetItem()* 或*PyList_SET_ITEM()* 來將所有項目設定🄵一個真實物件前，你無法使用像是*PySequence_SetItem()* 的抽象 API 函式，也不能將物件暴露 (expose) 給 Python 程式碼。以下 API 在串列完全初始化之前是安全的：*PyList_SetItem()* 和*PyList_SET_ITEM()*。

Py_ssize_t **PyList_Size**(*PyObject* *list)
🄵穩定 ABI 的一部分. 回傳 *list* 串列物件的長度；這相當於串列物件的 `len(list)`。

Py_ssize_t **PyList_GET_SIZE**(*PyObject* *list)
與*PyList_Size()* 類似，但🄵有錯誤檢查。

PyObject ***PyList_GetItemRef**(*PyObject* *list, *Py_ssize_t* index)
回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 回傳 *list* 指向的串列中位於 *index* 位置的物件。該位置不可🄵負數；🄵不支援從串列尾末開始索引。如果 *index* 超出邊界範圍 (`<0 or >=len(list)`) 則回傳 NULL 🄵設定 `IndexError` 例外。

在 3.13 版被加入.

PyObject ***PyList_GetItem**(*PyObject* *list, *Py_ssize_t* index)
回傳值：借用參照。🄵穩定 ABI 的一部分. Like *PyList_GetItemRef()*, but returns a *borrowed reference* instead of a *strong reference*.

PyObject ***PyList_GET_ITEM**(*PyObject* *list, *Py_ssize_t* i)
回傳值：借用參照。與*PyList_GetItem()* 類似，但🄵有錯誤檢查。

int **PyList_SetItem**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)
🄵穩定 ABI 的一部分. 將串列中索引 *index* 處的項目設定🄵 *item*。成功時回傳 `0`。如果 *index* 超出邊界範圍則回傳 `-1` 🄵設定一個 `IndexError` 例外。

> **備🄵**
>
> 此函式「竊取」對 *item* 的參照，🄵🄵🄵對串列中受影響位置上已存在項目的參照。

void **PyList_SET_ITEM**(*PyObject* *list, *Py_ssize_t* i, *PyObject* *o)
PyList_SetItem() 的巨集形式，🄵有錯誤檢查。這通常僅用於填充🄵有已存在🄵容的新串列。

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

> **備🄵**
>
> 該巨集「竊取」對 *item* 的參照，🄵且與*PyList_SetItem()* 不同的是，它不會🄵🄵對任意被替🄵項目的參照；*list* 中位置 *i* 的任何參照都將被🄵漏 (leak)。

int **PyList_Insert**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)
🄵穩定 ABI 的一部分. 將項目 *item* 插入串列 *list* 中索引 *index* 的位置之前。如果成功則回傳 `0`；如果失敗則回傳 `-1` 🄵設定例外。類似於 `list.insert(index, item)`。

int **PyList_Append**(*PyObject* *list, *PyObject* *item)

🄵穩定 ABI 的一部分. 將物件 *item* 附加到串列 *list* 的最後面。如果成功則回傳 `0`；如果不成功，則回傳 `-1` 🄴設定例外。類似於 `list.append(item)`。

PyObject ***PyList_GetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high)

回傳值：新的參照。🄵穩定 ABI 的一部分. 回傳 *list* 中的物件串列，其中包含 *low* 和 *high* 之間的物件。如果🄴有成功則回傳 `NULL` 🄴設定例外。類似於 `list[low:high]`。不支援從串列尾末開始索引。

int **PyList_SetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high, *PyObject* *itemlist)

🄵穩定 ABI 的一部分. 將 *low* 和 *high* 之間的 *list* 切片設定🄵 *itemlist* 的🄴容。類似於 `list[low:high] = itemlist`。*itemlist* 可能🄵 `NULL`，表示分配一個空串列（切片🄴除）。成功時回傳 `0`，失敗時則回傳 `-1`。不支援從串列尾末開始索引。

int **PyList_Extend**(*PyObject* *list, *PyObject* *iterable)

Extend *list* with the contents of *iterable*. This is the same as `PyList_SetSlice(list, PY_SSIZE_T_MAX, PY_SSIZE_T_MAX, iterable)` and analogous to `list.extend(iterable)` or `list += iterable`.

Raise an exception and return `-1` if *list* is not a `list` object. Return 0 on success.

在 3.13 版被加入.

int **PyList_Clear**(*PyObject* *list)

Remove all items from *list*. This is the same as `PyList_SetSlice(list, 0, PY_SSIZE_T_MAX, NULL)` and analogous to `list.clear()` or `del list[:]`.

Raise an exception and return `-1` if *list* is not a `list` object. Return 0 on success.

在 3.13 版被加入.

int **PyList_Sort**(*PyObject* *list)

🄵穩定 ABI 的一部分. 對 *list* 的項目進行原地 (in place) 排序。成功時回傳 `0`，失敗時回傳 `-1`。這相當於 `list.sort()`。

int **PyList_Reverse**(*PyObject* *list)

🄵穩定 ABI 的一部分. 原地反轉 *list* 的項目。成功時回傳 `0`，失敗時回傳 `-1`。這相當於 `list.reverse()`。

PyObject ***PyList_AsTuple**(*PyObject* *list)

回傳值：新的參照。🄵穩定 ABI 的一部分. 回傳一個新的 tuple（元組）物件，其中包含 *list* 的🄴容；相當於 `tuple(list)`。

9.4 容器物件

9.4.1 字典物件

type **PyDictObject**

PyObject 子型態代表一個 Python 字典物件。

PyTypeObject **PyDict_Type**

🄵穩定 ABI 的一部分. *PyTypeObject* 實例代表一個 Python 字典型態。此與 Python 層中的 `dict` 🄵同一個物件。

int **PyDict_Check**(*PyObject* *p)

若 *p* 是一個字典物件或字典的子型態實例則會回傳 true。此函式每次都會執行成功。

int **PyDict_CheckExact**(*PyObject* *p)

若 *p* 是一個字典物件但🄵不是一個字典子型態的實例，則回傳 true。此函式每次都會執行成功。

PyObject ***PyDict_New**()

回傳值：新的參照。🄵穩定 ABI 的一部分. 回傳一個新的空字典，或在失敗時回傳 `NULL`。

PyObject ***PyDictProxy_New**(*PyObject* *mapping)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a `types.MappingProxyType` object for a mapping which enforces read-only behavior. This is normally used to create a view to prevent modification of the dictionary for non-dynamic class types.

void **PyDict_Clear**(*PyObject* *p)

🄵穩定 ABI 的一部分. 清空現有字典中的所有鍵值對。

int **PyDict_Contains**(*PyObject* *p, *PyObject* *key)

🄵穩定 ABI 的一部分. Determine if dictionary *p* contains *key*. If an item in *p* is matches *key*, return `1`, otherwise return `0`. On error, return `-1`. This is equivalent to the Python expression `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**.

在 3.13 版被加入.

PyObject ***PyDict_Copy**(*PyObject* *p)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new dictionary that contains the same key-value pairs as *p*.

int **PyDict_SetItem**(*PyObject* *p, *PyObject* *key, *PyObject* *val)

🄵穩定 ABI 的一部分. Insert *val* into the dictionary *p* with a key of *key*. *key* must be *hashable*; if it isn't, `TypeError` will be raised. Return `0` on success or `-1` on failure. This function *does not* steal a reference to *val*.

int **PyDict_SetItemString**(*PyObject* *p, const char *key, *PyObject* *val)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 *自 3.13 版本開始*. 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`.
- 錯誤發生時，引發一個例外🄵回傳 `-1`。

在 3.13 版被加入.

另見*PyObject_GetItem()* 函式。

PyObject ***PyDict_GetItem**(*PyObject* *p, *PyObject* *key)

回傳值：借用參照。🄵穩定 ABI 的一部分. 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.

備🄵

Exceptions that occur while this calls `__hash__()` and `__eq__()` methods are silently ignored. Prefer the *PyDict_GetItemWithError()* function instead.

在 3.10 版的變更: 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)

回傳值：借用參照。🄵穩定 ABI 的一部分. Variant of *PyDict_GetItem()* that does not suppress exceptions. Return `NULL` **with** an exception set if an exception occurred. Return `NULL` **without** an exception set if the key wasn't present.

PyObject ***PyDict_GetItemString**(*PyObject* *p, const char *key)

回傳值：借用參照。🄵穩定 ABI 的一部分. This is the same as *PyDict_GetItem()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

> **備🄵**
>
> 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)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Similar to *PyDict_GetItemRef()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

在 3.13 版被加入.

PyObject ***PyDict_SetDefault**(*PyObject* *p, *PyObject* *key, *PyObject* *defaultobj)

回傳值：借用參照。 This is the same as the Python-level `dict.setdefault()`. If present, it returns the value corresponding to *key* from the dictionary *p*. If the key is not in the dict, it is inserted with value *defaultobj* and *defaultobj* is returned. This function evaluates the hash function of *key* only once, instead of evaluating it independently for the lookup and the insertion.

在 3.4 版被加入.

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.

在 3.13 版被加入.

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`.
- 錯誤發生時，引發一個例外🄵回傳 `-1`。

Similar to `dict.pop()`, but without the default value and not raising `KeyError` if the key missing.

在 3.13 版被加入.

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**.

在 3.13 版被加入.

PyObject ***PyDict_Items**(*PyObject* *p)

回傳值：新的參照。🄵穩定 ABI 的一部分. 回傳一個包含字典中所有項目的*PyListObject*。

PyObject ***PyDict_Keys**(*PyObject* *p)

回傳值：新的參照。🄵穩定 ABI 的一部分. 回傳一個包含字典中所有鍵的*PyListObject*。

PyObject ***PyDict_Values**(*PyObject* *p)

回傳值：新的參照。🄵穩定 ABI 的一部分. 回傳一個包含字典 *p* 中所有值的*PyListObject*。

Py_ssize_t **PyDict_Size**(*PyObject* *p)

🄵穩定 ABI 的一部分. 回傳字典中項目的數量。此與於字典呼叫 `len(p)` 等效。

int **PyDict_Next**(*PyObject* *p, *Py_ssize_t* *ppos, *PyObject* **pkey, *PyObject* **pvalue)

🄵穩定 ABI 的一部分. 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.

舉例來🄵:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}
```

The dictionary *p* should not be mutated during iteration. It is safe to modify the values of the keys as you iterate over the dictionary, but only so long as the set of keys does not change. For example:

```
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();
```

> **備🄵**
>
> 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)

🄵穩定 ABI 的一部分. Iterate over mapping object *b* adding key-value pairs to dictionary *a*. *b* may be a dictionary, or any object supporting `PyMapping_Keys()` and `PyObject_GetItem()`. If *override* is true, existing pairs in *a* will be replaced if a matching key is found in *b*, otherwise pairs will only be added if there is not a matching key in *a*. Return `0` on success or `-1` if an exception was raised.

int **PyDict_Update**(*PyObject* *a, *PyObject* *b)

🄵穩定 ABI 的一部分. This is the same as `PyDict_Merge(a, b, 1)` in C, and is similar to `a.update(b)` in Python except that `PyDict_Update()` doesn't fall back to the iterating over a sequence of key value pairs if the second argument has no "keys" attribute. Return `0` on success or `-1` if an exception was raised.

int **PyDict_MergeFromSeq2**(*PyObject* *a, *PyObject* *seq2, int override)

🄵穩定 ABI 的一部分. Update or merge into dictionary *a*, from the key-value pairs in *seq2*. *seq2* must be an iterable object producing iterable objects of length 2, viewed as key-value pairs. In case of duplicate keys, the last wins if *override* is true, else the first wins. Return `0` on success or `-1` if an exception was raised. Equivalent Python (except for the return value):

```
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.

在 3.12 版被加入.

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.)

在 3.12 版被加入.

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.

在 3.12 版被加入.

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.

在 3.12 版被加入.

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`.

在 3.12 版被加入.

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.

在 3.12 版被加入.

9.4.2 集合物件

This section details the public API for `set` and `frozenset` objects. Any functionality not listed below is best accessed using either the abstract object protocol (including *PyObject_CallMethod()*, *PyObject_RichCompareBool()*, *PyObject_Hash()*, *PyObject_Repr()*, *PyObject_IsTrue()*, *PyObject_Print()*, and *PyObject_GetIter()*) or the abstract number protocol (including *PyNumber_And()*, *PyNumber_Subtract()*, *PyNumber_Or()*, *PyNumber_Xor()*, *PyNumber_InPlaceAnd()*, *PyNumber_InPlaceSubtract()*, *PyNumber_InPlaceOr()*, and *PyNumber_InPlaceXor()*).

type **PySetObject**

This subtype of *PyObject* is used to hold the internal data for both `set` and `frozenset` objects. It is like a *PyDictObject* in that it is a fixed size for small sets (much like tuple storage) and will point to a separate, variable sized block of memory for medium and large sized sets (much like list storage). None of the fields of this structure should be considered public and all are subject to change. All access should be done through the documented API rather than by manipulating the values in the structure.

PyTypeObject **PySet_Type**

🄵穩定 ABI 的一部分. This is an instance of *PyTypeObject* representing the Python `set` type.

PyTypeObject **PyFrozenSet_Type**

🄵穩定 ABI 的一部分. This is an instance of *PyTypeObject* representing the Python `frozenset` type.

The following type check macros work on pointers to any Python object. Likewise, the constructor functions work with any iterable Python object.

int **PySet_Check**(*PyObject* *p)

Return true if *p* is a `set` object or an instance of a subtype. This function always succeeds.

int **PyFrozenSet_Check**(*PyObject* *p)

Return true if *p* is a `frozenset` object or an instance of a subtype. This function always succeeds.

int **PyAnySet_Check**(*PyObject* *p)

Return true if *p* is a `set` object, a `frozenset` object, or an instance of a subtype. This function always succeeds.

int **PySet_CheckExact**(*PyObject* *p)

Return true if *p* is a `set` object but not an instance of a subtype. This function always succeeds.

在 3.10 版被加入.

int **PyAnySet_CheckExact**(*PyObject* *p)

Return true if *p* is a `set` object or a `frozenset` object but not an instance of a subtype. This function always succeeds.

int **PyFrozenSet_CheckExact**(*PyObject* *p)

Return true if *p* is a `frozenset` object but not an instance of a subtype. This function always succeeds.

PyObject ***PySet_New**(*PyObject* *iterable)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new `set` containing objects returned by the *iterable*. The *iterable* may be `NULL` to create a new empty set. Return the new set on success or `NULL` on failure. Raise `TypeError` if *iterable* is not actually iterable. The constructor is also useful for copying a set (`c=set(s)`).

PyObject ***PyFrozenSet_New**(*PyObject* *iterable)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new `frozenset` containing objects returned by the *iterable*. The *iterable* may be `NULL` to create a new empty frozenset. Return the new set on success or `NULL` on failure. Raise `TypeError` if *iterable* is not actually iterable.

The following functions and macros are available for instances of `set` or `frozenset` or instances of their subtypes.

Py_ssize_t **PySet_Size**(*PyObject* *anyset)

🄵穩定 ABI 的一部分. 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)

Macro form of *PySet_Size()* without error checking.

int **PySet_Contains**(*PyObject* *anyset, *PyObject* *key)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. Add *key* to a `set` instance. Also works with `frozenset` instances (like *PyTuple_SetItem()* it can be used to fill in the values of brand new frozensets before they are exposed to other code). Return `0` on success or `-1` on failure. Raise a `TypeError` if the *key* is unhashable. Raise a `MemoryError` if there is no room to grow. Raise a `SystemError` if *set* is not an instance of `set` or its subtype.

The following functions are available for instances of `set` or its subtypes but not for instances of `frozenset` or its subtypes.

int **PySet_Discard**(*PyObject* *set, *PyObject* *key)

🄵穩定 ABI 的一部分. 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new reference to an arbitrary object in the *set*, and removes the object from the *set*. Return `NULL` on failure. Raise `KeyError` if the set is empty. Raise a `SystemError` if *set* is not an instance of `set` or its subtype.

int **PySet_Clear**(*PyObject* *set)

🄵穩定 ABI 的一部分. 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 函式物件

9.5.1 函式物件 (Function Objects)

這有一些特用於 Python 函式的函式。

type **PyFunctionObject**

用於函式的 C 結構。

PyTypeObject **PyFunction_Type**

這是個*PyTypeObject* 的實例，且代表了 Python 函式型🄵，Python 程式設計者可透過 `types.FunctionType` 使用它。

int **PyFunction_Check**(*PyObject* *o)

如果 *o* 是個函式物件（擁有*PyFunction_Type* 的型🄵）則回傳 true。參數必須不🄵 `NULL`。此函式必能成功執行。

PyObject ***PyFunction_New**(*PyObject* *code, *PyObject* *globals)

回傳值：新的參照。 回傳一個與程式碼物件 *code* 相關聯的函式物件。*globals* 必須是一個帶有函式能🄵存取的全域變數的字典。

函式的文件字串 (docstring) 和名稱是從程式碼物件所取得，`__module__` 是自 *globals* 所取得。引數預設值、標🄵 (annotation) 和閉包 (closure) 被設🄵 `NULL`，`__qualname__` 被設🄵和程式碼物件 `co_qualname` 欄位相同的值。

PyObject ***PyFunction_NewWithQualName**(*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)

回傳值：新的參照。 和*PyFunction_New()* 相似，但也允許函式物件 `__qualname__` 屬性的設定，*qualname* 應🄵一個 unicode 物件或是 `NULL`；如🄵 `NULL`，`__qualname__` 屬性會被設🄵與程式碼物件 `co_qualname` 欄位相同的值。

在 3.3 版被加入.

PyObject ***PyFunction_GetCode**(*PyObject* *op)

回傳值：借用參照。 回傳與程式碼物件相關的函式物件 *op*。

PyObject ***PyFunction_GetGlobals**(*PyObject* *op)

回傳值：借用參照。 回傳與全域函式字典相關的函式物件 *op*。

PyObject ***PyFunction_GetModule**(*PyObject* *op)

回傳值：借用參照。回傳一個函式物件 *op* 之 `__module__` 屬性的*borrowed reference*，它可以是 *NULL*。

這通常是個包含模組名稱的字串，但可以被 Python 程式設🄵任何其他物件。

PyObject ***PyFunction_GetDefaults**(*PyObject* *op)

回傳值：借用參照。 回傳函式物件 *op* 的引數預設值，這可以是一個含有多個引數的 tuple（元組）或 `NULL`。

int **PyFunction_SetDefaults**(*PyObject* *op, *PyObject* *defaults)

設定函式物件 *op* 的引數預設值。*defaults* 必須是 `Py_None` 或一個 tuple。

引發 `SystemError` 且在失敗時回傳 `-1`。

void **PyFunction_SetVectorcall**(*PyFunctionObject* *func, *vectorcallfunc* vectorcall)

🄵一個給定的函式物件 *func* 設定 vectorcall 欄位。

警告：使用此 API 的擴充套件必須保留未更改（預設）vectorcall 函式的行🄵!

在 3.12 版被加入.

PyObject ***PyFunction_GetKwDefaults**(*PyObject* *op)

回傳值：借用參照。 回傳函式物件 *op* 的僅限關鍵字引數預設值，這可以是一個含有多個引數的字典或 `NULL`。

PyObject ***PyFunction_GetClosure**(*PyObject* *op)

回傳值：借用參照。 回傳與函式物件 *op* 相關聯的閉包，這可以是個 `NULL` 或是一個包含 cell 物件的 tuple。

int **PyFunction_SetClosure**(*PyObject* *op, *PyObject* *closure)

設定與函式物件 *op* 相關聯的閉包，*closure* 必須是 `Py_None` 或是一個包含 cell 物件的 tuple。

引發 `SystemError` 且在失敗時回傳 `-1`。

PyObject ***PyFunction_GetAnnotations**(*PyObject* *op)

回傳值：借用參照。 回傳函式物件 *op* 的標🄵，這可以是一個可變動的 (mutable) 字典或 `NULL`。

int **PyFunction_SetAnnotations**(*PyObject* *op, *PyObject* *annotations)

設定函式物件 *op* 的標🄵，*annotations* 必須是一個字典或 `Py_None`。

引發 `SystemError` 且在失敗時回傳 `-1`。

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)

回傳值：借用參照。 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.

在 3.12 版被加入.

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.)

在 3.12 版被加入.

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`

在 3.12 版被加入.

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.

在 3.12 版被加入.

9.5.2 實例方法物件 (Instance Method Objects)

實例方法是*`PyCFunction`* 的包裝器 (wrapper)，也是將*`PyCFunction`* 🄵結 (bind) 到類🄵物件的一種新方式。它替代了原先對 `PyMethod_New(func, NULL, class)` 的呼叫。

PyTypeObject **PyInstanceMethod_Type**

`PyTypeObject` 的實例代表 Python 實例方法型🄵。它不會公開 (expose) 給 Python 程式。

int **PyInstanceMethod_Check**(*PyObject* *o)

如果 *o* 是一個實例方法物件（型🄵🄵*`PyInstanceMethod_Type`*）則回傳 true。參數必須不🄵 `NULL`。此函式總是會成功執行。

PyObject ***PyInstanceMethod_New**(*PyObject* *func)

回傳值：新的參照。 回傳一個新的實例方法物件，*func* 🄵任意可呼叫物件，在實例方法被呼叫時 *func* 函式也會被呼叫。

PyObject ***PyInstanceMethod_Function**(*PyObject* *im)

回傳值：借用參照。 回傳關聯到實例方法 *im* 的函式物件。

PyObject ***PyInstanceMethod_GET_FUNCTION**(*PyObject* *im)

回傳值：借用參照。 巨集 (macro) 版本的*`PyInstanceMethod_Function()`*，忽略了錯誤檢查。

9.5.3 方法物件 (Method Objects)

方法🄵🄵結函式 (bound function) 物件。方法總是會被🄵結到一個使用者定義類🄵的實例。未🄵結方法（🄵結到一個類🄵的方法）已不可用。

PyTypeObject **PyMethod_Type**

這個*`PyTypeObject`* 實例代表 Python 方法型🄵。它作🄵 `types.MethodType` 公開給 Python 程式。

int **PyMethod_Check**(*PyObject* *o)

如果 *o* 是一個方法物件（型🄵🄵*PyMethod_Type*）則回傳 true。參數必須不🄵 NULL。此函式總是會成功執行。

PyObject ***PyMethod_New**(*PyObject* *func, *PyObject* *self)

回傳值：新的參照。 回傳一個新的方法物件，*func* 應🄵任意可呼叫物件，*self* 🄵該方法應🄵結的實例。在方法被呼叫時，*func* 函式也會被呼叫。*self* 必須不🄵 NULL。

PyObject ***PyMethod_Function**(*PyObject* *meth)

回傳值：借用參照。 回傳關聯到方法 *meth* 的函式物件。

PyObject ***PyMethod_GET_FUNCTION**(*PyObject* *meth)

回傳值：借用參照。 巨集版本的*PyMethod_Function()*，忽略了錯誤檢查。

PyObject ***PyMethod_Self**(*PyObject* *meth)

回傳值：借用參照。 回傳關聯到方法 *meth* 的實例。

PyObject ***PyMethod_GET_SELF**(*PyObject* *meth)

回傳值：借用參照。 巨集版本的*PyMethod_Self()*，忽略了錯誤檢查。

9.5.4 Cell 物件

”Cell” 物件用於實現被多個作用域所參照 (reference) 的變數。對於每個這樣的變數，都會有個 cell 物件🄵了儲存該值而被建立；參照該值的每個 stack frame 中的區域性變數包含外部作用域的 cell 參照，它同樣使用了該變數。存取該值時，將使用 cell 中包含的值而不是 cell 物件本身。這種對 cell 物件的去除參照 (de-reference) 需要生成的位元組碼 (byte-code) 有支援；存取時不會自動去除參照。cell 物件在其他地方可能不太有用。

type **PyCellObject**

Cell 物件所用之 C 結構。

PyTypeObject **PyCell_Type**

對應 cell 物件的物件型🄵。

int **PyCell_Check**(*PyObject* *ob)

如果 *ob* 是一個 cell 物件則回傳真值；*ob* 必須不🄵 NULL。此函式總是會成功執行。

PyObject ***PyCell_New**(*PyObject* *ob)

回傳值：新的參照。 建立🄵回傳一個包含 *ob* 的新 cell 物件。參數可以🄵 NULL。

PyObject ***PyCell_Get**(*PyObject* *cell)

回傳值：新的參照。 回傳 cell 物件 *cell* 的🄵容，其可能🄵 NULL。如果 *cell* 不是一個 cell 物件，則將回傳 NULL 🄵設定例外。

PyObject ***PyCell_GET**(*PyObject* *cell)

回傳值：借用參照。 回傳 cell 物件 *cell* 的🄵容，但是不檢查 *cell* 是否非 NULL 🄵且🄵一個 cell 物件。

int **PyCell_Set**(*PyObject* *cell, *PyObject* *value)

將 cell 物件 *cell* 的🄵容設🄵 *value*。這將釋放任何對 cell 物件目前🄵容的參照。*value* 可以🄵 NULL。*cell* 必須不🄵 NULL。

在成功時回傳 0。如果 *cell* 不是一個 cell 物件，則將設定例外🄵回傳 -1。

void **PyCell_SET**(*PyObject* *cell, *PyObject* *value)

將 cell 物件 *cell* 的值設🄵 *value*。不會調整參照計數，🄵且不會進行任何安全檢查；*cell* 必須🄵非 NULL 🄵且🄵一個 cell 物件。

9.5.5 程式碼物件

Code objects are a low-level detail of the CPython implementation. Each one represents a chunk of executable code that hasn't yet been bound into a function.

type **PyCodeObject**

The C structure of the objects used to describe code objects. The fields of this type are subject to change at any time.

PyTypeObject **PyCode_Type**

This is an instance of `PyTypeObject` representing the Python code object.

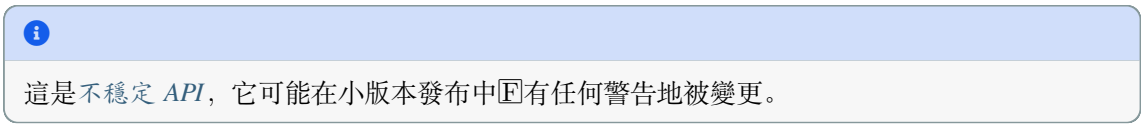
int **PyCode_Check**(*PyObject* *co)

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)

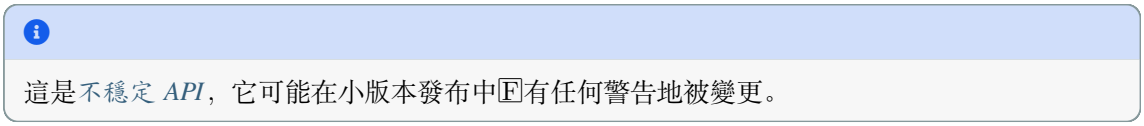


這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

Return the position of the first *free (closure) variable* in a code object.

在 3.13 版的變更: Renamed from `PyCode_GetFirstFree` as part of 不穩定的 *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)



這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

Return a new code object. If you need a dummy code object to create a frame, use `PyCode_NewEmpty()` instead.

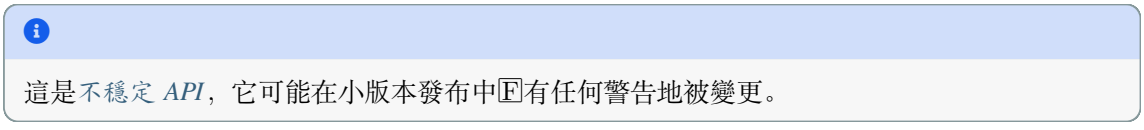
Since the definition of the bytecode changes often, calling `PyUnstable_Code_New()` directly can bind you to a precise Python version.

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.

在 3.11 版的變更: 新增 `qualname` 和 `exceptiontable` 參數。

在 3.12 版的變更: Renamed from `PyCode_New` as part of 不穩定的 *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)



這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

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.

在 3.8 版被加入: as `PyCode_NewWithPosOnlyArgs`

在 3.11 版的變更: 新增 `qualname` 和 `exceptiontable` 參數。

在 3.12 版的變更: 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)

回傳值：新的參照。 Return a new empty code object with the specified filename, function name, and first line number. The resulting code object will raise an `Exception` if executed.

int **PyCode_Addr2Line**(*PyCodeObject* *co, int byte_offset)

Return the line number of the instruction that occurs on or before `byte_offset` and ends after it. If you just need the line number of a frame, use *PyFrame_GetLineNumber()* instead.

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)

Sets the passed `int` pointers to the source code line and column numbers for the instruction at `byte_offset`. Sets the value to `0` when information is not available for any particular element.

如果函式成功則回傳 `1`，否則回傳 `0`。

在 3.11 版被加入.

PyObject ***PyCode_GetCode**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_code')`. Returns a strong reference to a *PyBytesObject* representing the bytecode in a code object. On error, `NULL` is returned and an exception is raised.

This `PyBytesObject` may be created on-demand by the interpreter and does not necessarily represent the bytecode actually executed by CPython. The primary use case for this function is debuggers and profilers.

在 3.11 版被加入.

PyObject ***PyCode_GetVarnames**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_varnames')`. Returns a new reference to a *PyTupleObject* containing the names of the local variables. On error, `NULL` is returned and an exception is raised.

在 3.11 版被加入.

PyObject ***PyCode_GetCellvars**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_cellvars')`. Returns a new reference to a *PyTupleObject* containing the names of the local variables that are referenced by nested functions. On error, `NULL` is returned and an exception is raised.

在 3.11 版被加入.

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.

在 3.11 版被加入.

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.

在 3.12 版被加入.

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.)

在 3.12 版被加入.

type **PyCodeEvent**

Enumeration of possible code object watcher events: - `PY_CODE_EVENT_CREATE` - `PY_CODE_EVENT_DESTROY`

在 3.12 版被加入.

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.

在 3.12 版被加入.

9.5.6 程式碼物件旗標

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.

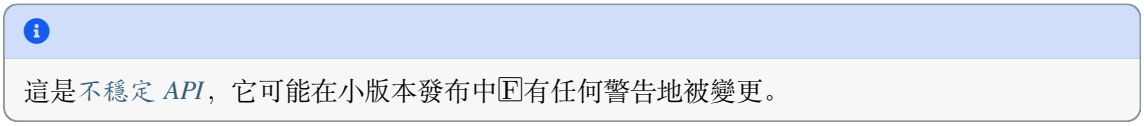
旗標	意義
CO_OPTIMIZED	inspect.CO_OPTIMIZED
CO_NEWLOCALS	inspect.CO_NEWLOCALS
CO_VARARGS	inspect.CO_VARARGS
CO_VARKEYWORDS	inspect.CO_VARKEYWORDS
CO_NESTED	inspect.CO_NESTED
CO_GENERATOR	inspect.CO_GENERATOR
CO_COROUTINE	inspect.CO_COROUTINE
CO_ITERABLE_COROUTINE	inspect.CO_ITERABLE_COROUTINE
CO_ASYNC_GENERATOR	inspect.CO_ASYNC_GENERATOR
CO_HAS_DOCSTRING	inspect.CO_HAS_DOCSTRING
CO_METHOD	inspect.CO_METHOD
CO_FUTURE_DIVISION	無效果 (__future__.division)
CO_FUTURE_ABSOLUTE_IMPORT	無效果 (__future__.absolute_import)
CO_FUTURE_WITH_STATEMENT	無效果 (__future__.with_statement)
CO_FUTURE_PRINT_FUNCTION	無效果 (__future__.print_function)
CO_FUTURE_UNICODE_LITERALS	無效果 (__future__.unicode_literals)
CO_FUTURE_GENERATOR_STOP	無效果 (__future__.generator_stop)
CO_FUTURE_ANNOTATIONS	__future__.annotations

9.5.7 額外資訊

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)



這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

Return a new an opaque index value used to adding data to code objects.

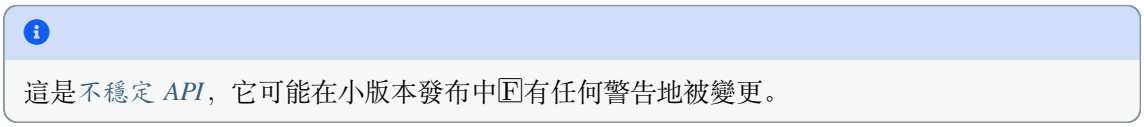
You generally call this function once (per interpreter) and use the result with `PyCode_GetExtra` and `PyCode_SetExtra` to manipulate data on individual code objects.

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`*.

在 3.6 版被加入: as `_PyEval_RequestCodeExtraIndex`

在 3.12 版的變更: 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)



這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

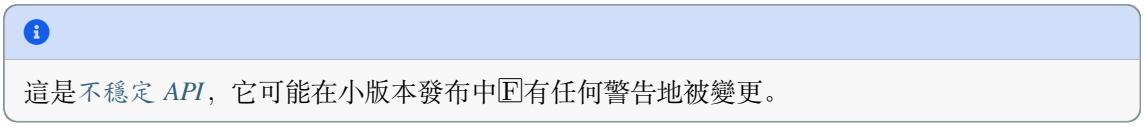
Set *extra* to the extra data stored under the given index. Return 0 on success. Set an exception and return -1 on failure.

If no data was set under the index, set *extra* to `NULL` and return 0 without setting an exception.

在 3.6 版被加入: as `_PyCode_GetExtra`

在 3.12 版的變更: 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)



這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

Set the extra data stored under the given index to *extra*. Return 0 on success. Set an exception and return -1 on failure.

在 3.6 版被加入: as `_PyCode_SetExtra`

在 3.12 版的變更: Renamed to `PyUnstable_Code_SetExtra`. The old private name is deprecated, but will be available until the API changes.

9.6 其他物件

9.6.1 檔案物件 (File Objects)

這些 API 是用於🄵建檔案物件的 Python 2 C API 的最小模擬 (minimal emulation)，它過去依賴於 C 標准函式庫對於緩衝 I/O (`FILE*`) 的支援。在 Python 3 中，檔案和串流使用新的 `io` 模組，它在作業系統的低階無緩衝 I/O 上定義了多個層級。下面描述的函式是這些新 API 的便捷 C 包裝器，主要用於直譯器中的🄵部錯誤報告；建議第三方程式碼改🄵存取 `io` API。

PyObject ***PyFile_FromFd**(int fd, const char *name, const char *mode, int buffering, const char *encoding, const char *errors, const char *newline, int closefd)

回傳值：新的參照。🄵穩定 ABI 的一部分. 從已打開檔案 *fd* 的檔案描述器建立一個 Python 檔案物件。引數 *name*、*encoding*、*errors* 和 *newline* 可以🄵 `NULL` 以使用預設值；*buffering* 可以是 *-1* 以使用預設值。*name* 被忽略🄵保留以實作向後相容性。失敗時回傳 `NULL`。有關引數的更全面描述，請參🄵 `io.open()` 函式文件。

> **⚠ 警告**
>
> 由於 Python 串流有自己的緩衝層，將它們與作業系統層級檔案描述器混合使用會🄵生各種問題（例如資料的排序不符合預期）。

在 3.2 版的變更: 忽略 *name* 屬性。

int **PyObject_AsFileDescriptor**(*PyObject* *p)

🄵穩定 ABI 的一部分. 回傳與 *p* 關聯的檔案描述器作🄵 `int`。如果物件是整數，則回傳其值。如果不是整數，則呼叫物件的 `fileno()` 方法（如果存在）；該方法必須回傳一個整數，它作🄵檔案描述器值回傳。設定例外🄵在失敗時回傳 `-1`。

PyObject ***PyFile_GetLine**(*PyObject* *p, int n)

回傳值：新的參照。🄵穩定 ABI 的一部分. 相當於 `p.readline([n])`，這個函式從物件 *p* 中讀取一行。*p* 可以是檔案物件或任何具有 `readline()` 方法的物件。如果 *n* 🄵 `0`，則不管該行的長度都只會讀取一行。如果 *n* 大於 `0`，則不會從檔案中讀取超過 *n* 個位元組；可以回傳該行的一部分。在這兩種情🄵下，如果立即到達檔案末尾，則回傳一個空字串。但是，如果 *n* 小於 `0`，無論長度如何，都會讀取一行，但如果立即到達檔案末尾，則會引發 `EOFError`。

int **PyFile_SetOpenCodeHook**(*Py_OpenCodeHookFunction* handler)

覆蓋 `io.open_code()` 的正常行🄵以透過提供的處理程式 (handler) 傳遞其參數。

The *handler* is a function of type:

typedef *PyObject* *(***Py_OpenCodeHookFunction**)(*PyObject**, void*)

相當於 *PyObject* *(*)(PyObject *path, void *userData)*，其中 *path* 保證🄵*PyUnicodeObject*。

userData 指標被傳遞到🄵🄵函式 (hook function) 中。由於可能會從不同的執行環境 (runtime) 呼叫🄵🄵函式，因此該指標不應直接指向 Python 狀態。

由於此🄵🄵函式是在導入期間有意使用的，因此請避免在其執行期間導入新模組，除非它們已知有被凍結或在 `sys.modules` 中可用。

一旦設定了一個🄵🄵函式，它就不能被🄵除或替🄵，且後續對*PyFile_SetOpenCodeHook()* 的呼叫將會失敗。失敗時，函式回傳 -1 且若直譯器已初始化便會設定例外。

在*Py_Initialize()* 之前呼叫此函式是安全的。

不帶引數地引發一個稽核事件 (auditing event) `setopencodehook`。

在 3.8 版被加入.

int **PyFile_WriteObject**(*PyObject* *obj, *PyObject* *p, int flags)

🄵穩定 ABI 的一部分. 將物件 *obj* 寫入檔案物件 *p*。*flags* 唯一支援的旗標是*Py_PRINT_RAW*；如果有給定，則寫入物件的 `str()` 而不是 `repr()`。在成功回傳 `0` 或在失敗回傳 `-1`；將設定適當的例外。

int **PyFile_WriteString**(const char *s, *PyObject* *p)

🄵穩定 ABI 的一部分. 寫入字串 *s* 到檔案物件 *p*。當成功時回傳 0，而當失敗時回傳 -1，🄵會設定合適的例外狀🄵。

9.6.2 模組物件

PyTypeObject **PyModule_Type**

🄵穩定 ABI 的一部分. This instance of *PyTypeObject* represents the Python module type. This is exposed to Python programs as `types.ModuleType`.

int **PyModule_Check**(*PyObject* *p)

Return true if *p* is a module object, or a subtype of a module object. This function always succeeds.

int **PyModule_CheckExact**(*PyObject* *p)

Return true if *p* is a module object, but not a subtype of *PyModule_Type*. This function always succeeds.

PyObject ***PyModule_NewObject**(*PyObject* *name)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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.

在失敗時回傳 `NULL` 🄵設定例外。

在 3.3 版被加入.

在 3.4 版的變更: `__package__` 和 `__loader__` 現在被設🄵 `None`。

PyObject ***PyModule_New**(const char *name)

回傳值：新的參照。🄵穩定 ABI 的一部分. Similar to *PyModule_NewObject()*, but the name is a UTF-8 encoded string instead of a Unicode object.

PyObject ***PyModule_GetDict**(*PyObject* *module)

回傳值：借用參照。🄵穩定 ABI 的一部分. Return the dictionary object that implements *module*'s namespace; this object is the same as the `__dict__` attribute of the module object. If *module* is not a module object (or a subtype of a module object), `SystemError` is raised and `NULL` is returned.

It is recommended extensions use other `PyModule_*` and `PyObject_*` functions rather than directly manipulate a module's `__dict__`.

PyObject ***PyModule_GetNameObject**(*PyObject* *module)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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.

在 3.3 版被加入.

const char ***PyModule_GetName**(*PyObject* *module)

🄵穩定 ABI 的一部分. Similar to *PyModule_GetNameObject()* but return the name encoded to `'utf-8'`.

void ***PyModule_GetState**(*PyObject* *module)

🄵穩定 ABI 的一部分. Return the "state" of the module, that is, a pointer to the block of memory allocated at module creation time, or `NULL`. See *PyModuleDef.m_size*.

PyModuleDef ***PyModule_GetDef**(*PyObject* *module)

🄵穩定 ABI 的一部分. Return a pointer to the *PyModuleDef* struct from which the module was created, or `NULL` if the module wasn't created from a definition.

PyObject ***PyModule_GetFilenameObject**(*PyObject* *module)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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.

在 3.2 版被加入.

const char *PyModule_GetFilename(*PyObject* *module)

🄵穩定 ABI 的一部分. Similar to *PyModule_GetFilenameObject()* but return the filename encoded to 'utf-8'.

在 3.2 版之後被🄵用: *PyModule_GetFilename()* raises `UnicodeEncodeError` on unencodable filenames, use *PyModule_GetFilenameObject()* instead.

9.6.3 模組定義

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**

🄵穩定 ABI 的一部分（包含所有成員）. 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**

Name for the new module.

const char ***m_doc**

Docstring for the module; usually a docstring variable created with *PyDoc_STRVAR* is used.

Py_ssize_t **m_size**

Module state may be kept in a per-module memory area that can be retrieved with *PyModule_GetState()*, rather than in static globals. This makes modules safe for use in multiple sub-interpreters.

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*.

更多詳情請見 **PEP 3121**。

PyMethodDef ***m_methods**

A pointer to a table of module-level functions, described by *PyMethodDef* values. Can be `NULL` if no functions are present.

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`.

在 3.5 版的變更: Prior to version 3.5, this member was always set to `NULL`, and was defined as:

inquiry **m_reload**

traverseproc **m_traverse**

A traversal function to call during GC traversal of the module object, or `NULL` if not needed.

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`.

在 3.9 版的變更: No longer called before the module state is allocated.

inquiry **m_clear**

A clear function to call during GC clearing of the module object, or `NULL` if not needed.

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`.

Like *PyTypeObject.tp_clear*, this function is not *always* called before a module is deallocated. For example, when reference counting is enough to determine that an object is no longer used, the cyclic garbage collector is not involved and *m_free* is called directly.

在 3.9 版的變更: No longer called before the module state is allocated.

freefunc **m_free**

A function to call during deallocation of the module object, or `NULL` if not needed.

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`.

在 3.9 版的變更: No longer called before the module state is allocated.

模組槽 (Module slots)

type **PyModuleDef_Slot**

int **slot**

A slot ID, chosen from the available values explained below.

void ***value**

Value of the slot, whose meaning depends on the slot ID.

在 3.5 版被加入.

The available slot types are:

Py_mod_create

Specifies a function that is called to create the module object itself. The *value* pointer of this slot must point to a function of the signature:

PyObject ***create_module**(*PyObject* *spec, *PyModuleDef* *def)

The function receives a `ModuleSpec` instance, as defined in **PEP 451**, and the module definition. It should return a new module object, or set an error and return `NULL`.

This function should be kept minimal. In particular, it should not call arbitrary Python code, as trying to import the same module again may result in an infinite loop.

Multiple `Py_mod_create` slots may not be specified in one module definition.

If `Py_mod_create` is not specified, the import machinery will create a normal module object using *PyModule_New()*. The name is taken from *spec*, not the definition, to allow extension modules to dynamically adjust to their place in the module hierarchy and be imported under different names through symlinks, all while sharing a single module definition.

There is no requirement for the returned object to be an instance of *PyModule_Type*. Any type can be used, as long as it supports setting and getting import-related attributes. However, only `PyModule_Type` instances may be returned if the `PyModuleDef` has non-`NULL` `m_traverse`, `m_clear`, `m_free`; non-zero `m_size`; or slots other than `Py_mod_create`.

Py_mod_exec

Specifies a function that is called to *execute* the module. This is equivalent to executing the code of a Python module: typically, this function adds classes and constants to the module. The signature of the function is:

int **exec_module**(*PyObject* *module)

If multiple `Py_mod_exec` slots are specified, they are processed in the order they appear in the *m_slots* array.

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`.

在 3.12 版被加入.

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`.

在 3.13 版被加入.

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)

回傳值：新的參照。 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)

回傳值：新的參照。🄵穩定 ABI 的一部分. Create a new module object, given the definition in *def*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

在失敗時回傳 `NULL` 🄵設定例外。

This function does not support slots. The *m_slots* member of *def* must be `NULL`.

> **備🄵**
>
> Most uses of this function should be using *PyModule_Create()* instead; only use this if you are sure you need it.

PyObject ***PyModule_FromDefAndSpec**(*PyModuleDef* *def, *PyObject* *spec)

回傳值：新的參照。 This macro calls *PyModule_FromDefAndSpec2()* with *module_api_version* set to *PYTHON_API_VERSION*, or to *PYTHON_ABI_VERSION* if using the *limited API*.

在 3.5 版被加入.

PyObject ***PyModule_FromDefAndSpec2**(*PyModuleDef* *def, *PyObject* *spec, int module_api_version)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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.

在失敗時回傳 `NULL` 🄵設定例外。

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.

> **備🄵**
>
> Most uses of this function should be using *PyModule_FromDefAndSpec()* instead; only use this if you are sure you need it.

在 3.5 版被加入.

int **PyModule_ExecDef**(*PyObject* *module, *PyModuleDef* *def)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Process any execution slots (*Py_mod_exec*) given in *def*.

在 3.5 版被加入.

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 支援的函式

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)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Add an object to *module* as *name*. This is a convenience function which can be used from the module's initialization function.

On success, return `0`. On error, raise an exception and return `-1`.

用法範例:

```
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.

The example can also be written without checking explicitly if *obj* is `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 that `Py_XDECREF()` should be used instead of `Py_DECREF()` in this case, since *obj* can be `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.

在 3.10 版被加入.

int **PyModule_Add**(*PyObject* *module, const char *name, *PyObject* *value)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 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.

用法範例:

```
if (PyModule_Add(module, "spam", PyBytes_FromString(value)) < 0) {
    goto error;
}
```

在 3.13 版被加入.

int **PyModule_AddObject**(*PyObject* *module, const char *name, *PyObject* *value)

🄵穩定 ABI 的一部分. Similar to *PyModule_AddObjectRef()*, but steals a reference to *value* on success (if it returns `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.

!!! 備🄵

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.

用法範例：

```
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.
```

在 3.13 版之後被🄵用: *`PyModule_AddObject()`* is *soft deprecated*.

int **`PyModule_AddIntConstant`**(*PyObject* *module, const char *name, long value)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

將字串常數加入到 *module* 中。

int **`PyModule_AddType`**(*PyObject* *module, *PyTypeObject* *type)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 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.

在 3.9 版被加入.

int **`PyModule_AddFunctions`**(*PyObject* *module, *PyMethodDef* *functions)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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.

在 3.5 版被加入.

int **PyModule_SetDocString**(*PyObject* *module, const char *docstring)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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`).

在 3.5 版被加入.

int **PyUnstable_Module_SetGIL**(*PyObject* *module, void *gil)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

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.

在 3.13 版被加入.

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.

These functions will not work on modules created using multi-phase initialization, since multiple such modules can be created from a single definition.

PyObject ***PyState_FindModule**(*PyModuleDef* *def)

回傳值：借用參照。🄵穩定 ABI 的一部分. Returns the module object that was created from *def* for the current interpreter. This method requires that the module object has been attached to the interpreter state with `PyState_AddModule()` beforehand. In case the corresponding module object is not found or has not been attached to the interpreter state yet, it returns `NULL`.

int **PyState_AddModule**(*PyObject* *module, *PyModuleDef* *def)

🄵穩定 ABI 的一部分 自 *3.3* 版本開始. Attaches the module object passed to the function to the interpreter state. This allows the module object to be accessible via `PyState_FindModule()`.

Only effective on modules created using single-phase initialization.

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*.

成功時回傳 `0`，在失敗時回傳 `-1` 🄵設定例外。

在 3.3 版被加入.

int **PyState_RemoveModule**(*PyModuleDef* *def)

🄵穩定 ABI 的一部分 自 *3.3* 版本開始. 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*.

在 3.3 版被加入.

9.6.6 🄵代器（Iterator）物件

Python 提供了兩種通用的🄵代器 (iterator) 物件，第一種是序列🄵代器 (sequence iterator)，適用於支援 `__getitem__()` 方法的任意序列，第二種是與可呼叫 (callable) 物件和哨兵值 (sentinel value) 一起使用，會呼叫序列中的每個可呼叫物件，當回傳哨兵值時就結束🄵代。

PyTypeObject **PySeqIter_Type**

🄵穩定 ABI 的一部分. 此型🄵物件用於由 *PySeqIter_New()* 所回傳的🄵代器物件以及用於🄵建序列型🄵的🄵建函式 `iter()` 的單引數形式。

int **PySeqIter_Check**(*PyObject* *op)

Return true if the type of *op* is *PySeqIter_Type*. This function always succeeds.

PyObject ***PySeqIter_New**(*PyObject* *seq)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return an iterator that works with a general sequence object, *seq*. The iteration ends when the sequence raises `IndexError` for the subscripting operation.

PyTypeObject **PyCallIter_Type**

🄵穩定 ABI 的一部分. Type object for iterator objects returned by *PyCallIter_New()* and the two-argument form of the `iter()` built-in function.

int **PyCallIter_Check**(*PyObject* *op)

Return true if the type of *op* is *PyCallIter_Type*. This function always succeeds.

PyObject ***PyCallIter_New**(*PyObject* *callable, *PyObject* *sentinel)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new iterator. The first parameter, *callable*, can be any Python callable object that can be called with no parameters; each call to it should return the next item in the iteration. When *callable* returns a value equal to *sentinel*, the iteration will be terminated.

9.6.7 Descriptor（描述器）物件

"Descriptor" 是描述物件某些屬性的物件，它們存在於型🄵物件的 dictionary（字典）中。

PyTypeObject **PyProperty_Type**

🄵穩定 ABI 的一部分. 🄵建 descriptor 型🄵的型🄵物件。

PyObject ***PyDescr_NewGetSet**(*PyTypeObject* *type, struct *PyGetSetDef* *getset)

回傳值：新的參照。🄵穩定 ABI 的一部分.

PyObject ***PyDescr_NewMember**(*PyTypeObject* *type, struct *PyMemberDef* *meth)

回傳值：新的參照。🄵穩定 ABI 的一部分.

PyObject ***PyDescr_NewMethod**(*PyTypeObject* *type, struct *PyMethodDef* *meth)

回傳值：新的參照。🄵穩定 ABI 的一部分.

PyObject ***PyDescr_NewWrapper**(*PyTypeObject* *type, struct wrapperbase *wrapper, void *wrapped)

回傳值：新的參照。

PyObject ***PyDescr_NewClassMethod**(*PyTypeObject* *type, *PyMethodDef* *method)

回傳值：新的參照。🄵穩定 ABI 的一部分.

int **PyDescr_IsData**(*PyObject* *descr)

如果 descriptor 物件 *descr* 描述的是一個資料屬性則回傳非零值，或者如果它描述的是一個方法則返回 `0`。*descr* 必須🄵一個 descriptor 物件；🄵有錯誤檢查。

PyObject ***PyWrapper_New**(*PyObject**, *PyObject**)

回傳值：新的參照。🄵穩定 ABI 的一部分.

9.6.8 切片物件

PyTypeObject **PySlice_Type**

🄵穩定 ABI 的一部分. The type object for slice objects. This is the same as `slice` in the Python layer.

int **PySlice_Check**(*PyObject* *ob)

Return true if *ob* is a slice object; *ob* must not be `NULL`. This function always succeeds.

PyObject ***PySlice_New**(*PyObject* *start, *PyObject* *stop, *PyObject* *step)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a new slice object with the given values. The *start*, *stop*, and *step* parameters are used as the values of the slice object attributes of the same names. Any of the values may be `NULL`, in which case the `None` will be used for the corresponding attribute.

Return `NULL` with an exception set if the new object could not be allocated.

int **PySlice_GetIndices**(*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)

🄵穩定 ABI 的一部分. Retrieve the start, stop and step indices from the slice object *slice*, assuming a sequence of length *length*. Treats indices greater than *length* as errors.

Returns `0` on success and `-1` on error with no exception set (unless one of the indices was not `None` and failed to be converted to an integer, in which case `-1` is returned with an exception set).

You probably do not want to use this function.

在 3.2 版的變更: The parameter type for the *slice* parameter was `PySliceObject*` before.

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)

🄵穩定 ABI 的一部分. Usable replacement for *PySlice_GetIndices()*. Retrieve the start, stop, and step indices from the slice object *slice* assuming a sequence of length *length*, and store the length of the slice in *slicelength*. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

成功時回傳 `0`，在失敗時回傳 `-1` 🄵設定例外。

備🄵

This function is considered not safe for resizable sequences. Its invocation should be replaced by a combination of *PySlice_Unpack()* and *PySlice_AdjustIndices()* where

```
if (PySlice_GetIndicesEx(slice, length, &start, &stop, &step, &slicelength) < 0) {
    // return error
}
```

is replaced by

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {
    // return error
}
slicelength = PySlice_AdjustIndices(length, &start, &stop, step);
```

在 3.2 版的變更: The parameter type for the *slice* parameter was `PySliceObject*` before.

在 3.6.1 版的變更: If `Py_LIMITED_API` is not set or set to the value between `0x03050400` and `0x03060000` (not including) or `0x03060100` or higher `PySlice_GetIndicesEx()` is implemented as a macro using `PySlice_Unpack()` and `PySlice_AdjustIndices()`. Arguments *start*, *stop* and *step* are evaluated more than once.

在 3.6.1 版之後被🄵用: If `Py_LIMITED_API` is set to the value less than `0x03050400` or between `0x03060000` and `0x03060100` (not including) `PySlice_GetIndicesEx()` is a deprecated function.

int **PySlice_Unpack**(*PyObject* *slice, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Extract the start, stop and step data members from a slice object as C integers. Silently reduce values larger than `PY_SSIZE_T_MAX` to `PY_SSIZE_T_MAX`, silently boost the start and stop values less than `PY_SSIZE_T_MIN` to `PY_SSIZE_T_MIN`, and silently boost the step values less than `-PY_SSIZE_T_MAX` to `-PY_SSIZE_T_MAX`.

成功時回傳 `0`，在失敗時回傳 `-1` 🄵設定例外。

在 3.6.1 版被加入.

Py_ssize_t **PySlice_AdjustIndices**(*Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* step)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Adjust start/end slice indices assuming a sequence of the specified length. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Return the length of the slice. Always successful. Doesn't call Python code.

在 3.6.1 版被加入.

Ellipsis 物件

PyTypeObject **PyEllipsis_Type**

🄵穩定 ABI 的一部分. The type of Python `Ellipsis` object. Same as `types.EllipsisType` in the Python layer.

PyObject ***Py_Ellipsis**

The Python `Ellipsis` object. This object has no methods. Like *Py_None*, it is an *immortal* singleton object.

在 3.12 版的變更: *Py_Ellipsis* 🄵不滅的 (immortal)。

9.6.9 MemoryView 物件

A `memoryview` object exposes the C level *buffer interface* as a Python object which can then be passed around like any other object.

PyObject ***PyMemoryView_FromObject**(*PyObject* *obj)

回傳值：新的參照。🄵穩定 ABI 的一部分. Create a memoryview object from an object that provides the buffer interface. If *obj* supports writable buffer exports, the memoryview object will be read/write, otherwise it may be either read-only or read/write at the discretion of the exporter.

PyBUF_READ

Flag to request a readonly buffer.

PyBUF_WRITE

Flag to request a writable buffer.

PyObject ***PyMemoryView_FromMemory**(char *mem, *Py_ssize_t* size, int flags)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Create a memoryview object using *mem* as the underlying buffer. *flags* can be one of *PyBUF_READ* or *PyBUF_WRITE*.

在 3.3 版被加入.

PyObject ***PyMemoryView_FromBuffer**(const *Py_buffer* *view)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.11* 版本開始. Create a memoryview object wrapping the given buffer structure *view*. For simple byte buffers, *PyMemoryView_FromMemory()* is the preferred function.

PyObject ***PyMemoryView_GetContiguous**(*PyObject* *obj, int buffertype, char order)

回傳值：新的參照。🄵穩定 ABI 的一部分. Create a memoryview object to a *contiguous* chunk of memory (in either 'C' or 'F'ortran *order*) from an object that defines the buffer interface. If memory is contiguous, the memoryview object points to the original memory. Otherwise, a copy is made and the memoryview points to a new bytes object.

buffertype can be one of *PyBUF_READ* or *PyBUF_WRITE*.

int **PyMemoryView_Check**(*PyObject* *obj)

Return true if the object *obj* is a memoryview object. It is not currently allowed to create subclasses of `memoryview`. This function always succeeds.

Py_buffer ***PyMemoryView_GET_BUFFER**(*PyObject* *mview)

Return a pointer to the memoryview's private copy of the exporter's buffer. *mview* **must** be a memoryview instance; this macro doesn't check its type, you must do it yourself or you will risk crashes.

PyObject ***PyMemoryView_GET_BASE**(*PyObject* *mview)

Return either a pointer to the exporting object that the memoryview is based on or `NULL` if the memoryview has been created by one of the functions *`PyMemoryView_FromMemory()`* or *`PyMemoryView_FromBuffer()`*. *mview* **must** be a memoryview instance.

9.6.10 弱參照物件

Python supports *weak references* as first-class objects. There are two specific object types which directly implement weak references. The first is a simple reference object, and the second acts as a proxy for the original object as much as it can.

int **PyWeakref_Check**(*PyObject* *ob)

Return non-zero if *ob* is either a reference or proxy object. This function always succeeds.

int **PyWeakref_CheckRef**(*PyObject* *ob)

Return non-zero if *ob* is a reference object. This function always succeeds.

int **PyWeakref_CheckProxy**(*PyObject* *ob)

Return non-zero if *ob* is a proxy object. This function always succeeds.

PyObject ***PyWeakref_NewRef**(*PyObject* *ob, *PyObject* *callback)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a weak reference object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object; an existing reference object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be `None` or `NULL`. If *ob* is not a weakly referenceable object, or if *callback* is not callable, `None`, or `NULL`, this will return `NULL` and raise `TypeError`.

PyObject ***PyWeakref_NewProxy**(*PyObject* *ob, *PyObject* *callback)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return a weak reference proxy object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object; an existing proxy object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be `None` or `NULL`. If *ob* is not a weakly referenceable object, or if *callback* is not callable, `None`, or `NULL`, this will return `NULL` and raise `TypeError`.

int **PyWeakref_GetRef**(*PyObject* *ref, *PyObject* **pobj)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Get a *strong reference* to the referenced object from a weak reference, *ref*, into **pobj*.

- On success, set **pobj* to a new *strong reference* to the referenced object and return 1.
- If the reference is dead, set **pobj* to `NULL` and return 0.
- On error, raise an exception and return -1.

在 3.13 版被加入.

PyObject ***PyWeakref_GetObject**(*PyObject* *ref)

回傳值：借用參照。🄵穩定 ABI 的一部分. Return a *borrowed reference* to the referenced object from a weak reference, *ref*. If the referent is no longer live, returns `Py_None`.

> **備🄵**
>
> This function returns a *borrowed reference* to the referenced object. This means that you should always call `Py_INCREF()` on the object except when it cannot be destroyed before the last usage of the borrowed reference.

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

PyObject ***PyWeakref_GET_OBJECT**(*PyObject* *ref)

回傳值：借用參照。 Similar to `PyWeakref_GetObject()`, but does no error checking.

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

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.

在 3.14 版被加入.

void **PyObject_ClearWeakRefs**(*PyObject* *object)

🄵穩定 ABI 的一部分. This function is called by the `tp_dealloc` handler to clear weak references.

This iterates through the weak references for *object* and calls callbacks for those references which have one. It returns when all callbacks have been attempted.

void **PyUnstable_Object_ClearWeakRefsNoCallbacks**(*PyObject* *object)

> 這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

Clears the weakrefs for *object* without calling the callbacks.

This function is called by the `tp_dealloc` handler for types with finalizers (i.e., `__del__()`). The handler for those objects first calls `PyObject_ClearWeakRefs()` to clear weakrefs and call their callbacks, then the finalizer, and finally this function to clear any weakrefs that may have been created by the finalizer.

In most circumstances, it's more appropriate to use `PyObject_ClearWeakRefs()` to clear weakrefs instead of this function.

在 3.13 版被加入.

9.6.11 Capsules

Refer to using-capsules for more information on using these objects.

在 3.1 版被加入.

type **PyCapsule**

This subtype of `PyObject` represents an opaque value, useful for C extension modules who need to pass an opaque value (as a `void*` pointer) through Python code to other C code. It is often used to make a C function pointer defined in one module available to other modules, so the regular import mechanism can be used to access C APIs defined in dynamically loaded modules.

type **PyCapsule_Destructor**

🄵穩定 ABI 的一部分. The type of a destructor callback for a capsule. Defined as:

```
typedef void (*PyCapsule_Destructor)(PyObject *);
```

See `PyCapsule_New()` for the semantics of PyCapsule_Destructor callbacks.

int **PyCapsule_CheckExact**(*PyObject* *p)

Return true if its argument is a `PyCapsule`. This function always succeeds.

PyObject ***PyCapsule_New**(void *pointer, const char *name, *PyCapsule_Destructor* destructor)

回傳值：新的參照。🄵穩定 ABI 的一部分. Create a `PyCapsule` encapsulating the *pointer*. The *pointer* argument may not be `NULL`.

On failure, set an exception and return `NULL`.

The *name* string may either be `NULL` or a pointer to a valid C string. If non-`NULL`, this string must outlive the capsule. (Though it is permitted to free it inside the *destructor*.)

If the *destructor* argument is not `NULL`, it will be called with the capsule as its argument when it is destroyed.

If this capsule will be stored as an attribute of a module, the *name* should be specified as `modulename.attributename`. This will enable other modules to import the capsule using `PyCapsule_Import()`.

void ***PyCapsule_GetPointer**(*PyObject* *capsule, const char *name)

🄵穩定 ABI 的一部分. Retrieve the *pointer* stored in the capsule. On failure, set an exception and return `NULL`.

The *name* parameter must compare exactly to the name stored in the capsule. If the name stored in the capsule is `NULL`, the *name* passed in must also be `NULL`. Python uses the C function `strcmp()` to compare capsule names.

PyCapsule_Destructor ***PyCapsule_GetDestructor**(*PyObject* *capsule)

🄵穩定 ABI 的一部分. Return the current destructor stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` destructor. This makes a `NULL` return code somewhat ambiguous; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

void ***PyCapsule_GetContext**(*PyObject* *capsule)

🄵穩定 ABI 的一部分. Return the current context stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` context. This makes a `NULL` return code somewhat ambiguous; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

const char ***PyCapsule_GetName**(*PyObject* *capsule)

🄵穩定 ABI 的一部分. Return the current name stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` name. This makes a `NULL` return code somewhat ambiguous; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

void ***PyCapsule_Import**(const char *name, int no_block)

🄵穩定 ABI 的一部分. Import a pointer to a C object from a capsule attribute in a module. The *name* parameter should specify the full name to the attribute, as in `module.attribute`. The *name* stored in the capsule must match this string exactly.

This function splits *name* on the `.` character, and imports the first element. It then processes further elements using attribute lookups.

Return the capsule's internal *pointer* on success. On failure, set an exception and return `NULL`.

備🄵

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.

在 3.3 版的變更: *no_block* has no effect anymore.

int **PyCapsule_IsValid**(*PyObject* *capsule, const char *name)

🄵穩定 ABI 的一部分. Determines whether or not *capsule* is a valid capsule. A valid capsule is non-NULL, passes *PyCapsule_CheckExact()*, has a non-NULL pointer stored in it, and its internal name matches the *name* parameter. (See *PyCapsule_GetPointer()* for information on how capsule names are compared.)

In other words, if *PyCapsule_IsValid()* returns a true value, calls to any of the accessors (any function starting with PyCapsule_Get) are guaranteed to succeed.

Return a nonzero value if the object is valid and matches the name passed in. Return 0 otherwise. This function will not fail.

int **PyCapsule_SetContext**(*PyObject* *capsule, void *context)

🄵穩定 ABI 的一部分. Set the context pointer inside *capsule* to *context*.

Return 0 on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetDestructor**(*PyObject* *capsule, *PyCapsule_Destructor* destructor)

🄵穩定 ABI 的一部分. Set the destructor inside *capsule* to *destructor*.

Return 0 on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetName**(*PyObject* *capsule, const char *name)

🄵穩定 ABI 的一部分. Set the name inside *capsule* to *name*. If non-NULL, the name must outlive the capsule. If the previous *name* stored in the capsule was not NULL, no attempt is made to free it.

Return 0 on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetPointer**(*PyObject* *capsule, void *pointer)

🄵穩定 ABI 的一部分. Set the void pointer inside *capsule* to *pointer*. The pointer may not be NULL.

Return 0 on success. Return nonzero and set an exception on failure.

9.6.12 Frame 物件

type **PyFrameObject**

🄵受限 API 的一部分 （做🄵一個不透明結構 *(opaque struct)*）. 用來描述 frame 物件的 C 結構。

在這個結構中🄵有公開的成員。

在 3.11 版的變更: The members of this structure were removed from the public C API. Refer to the What's New entry for details.

The *PyEval_GetFrame()* and *PyThreadState_GetFrame()* functions can be used to get a frame object.

另請參🄵*Reflection*。

PyTypeObject **PyFrame_Type**

The type of frame objects. It is the same object as types.FrameType in the Python layer.

在 3.11 版的變更: Previously, this type was only available after including <frameobject.h>.

int **PyFrame_Check**(*PyObject* *obj)

Return non-zero if *obj* is a frame object.

在 3.11 版的變更: Previously, this function was only available after including <frameobject.h>.

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

回傳值: 新的參照。 Get the *frame* next outer frame.

Return a *strong reference*, or NULL if *frame* has no outer frame.

在 3.9 版被加入.

PyObject ***PyFrame_GetBuiltins**(*PyFrameObject* *frame)

回傳值：新的參照。 取得 *frame* 的 `f_builtins` 屬性。

回傳*strong reference*。結果不能🄵 `NULL`。

在 3.11 版被加入.

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

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Get the *frame* code.

回傳*strong reference*。

The result (frame code) cannot be `NULL`.

在 3.9 版被加入.

PyObject ***PyFrame_GetGenerator**(*PyFrameObject* *frame)

回傳值：新的參照。 Get the generator, coroutine, or async generator that owns this frame, or `NULL` if this frame is not owned by a generator. Does not raise an exception, even if the return value is `NULL`.

回傳*strong reference* 或 `NULL`。

在 3.11 版被加入.

PyObject ***PyFrame_GetGlobals**(*PyFrameObject* *frame)

回傳值：新的參照。 取得 *frame* 的 `f_globals` 屬性。

回傳*strong reference*。結果不能🄵 `NULL`。

在 3.11 版被加入.

int **PyFrame_GetLasti**(*PyFrameObject* *frame)

取得 *frame* 的 `f_lasti` 屬性。

如果 `frame.f_lasti` 是 `None` 則回傳 -1。

在 3.11 版被加入.

PyObject ***PyFrame_GetVar**(*PyFrameObject* *frame, *PyObject* *name)

回傳值：新的參照。 取得 *frame* 的變數 *name*。

- 在成功時回傳變數值的*strong reference*。
- 如果變數不存在，則引發 `NameError` 🄵回傳 `NULL`。
- 在錯誤時引發例外🄵回傳 `NULL`。

name 的型🄵必須是 `str`。

在 3.12 版被加入.

PyObject ***PyFrame_GetVarString**(*PyFrameObject* *frame, const char *name)

回傳值：新的參照。Similar to *`PyFrame_GetVar()`*, but the variable name is a C string encoded in UTF-8.

在 3.12 版被加入.

PyObject ***PyFrame_GetLocals**(*PyFrameObject* *frame)

回傳值：新的參照。 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()`).

回傳*strong reference*。

在 3.11 版被加入.

在 3.13 版的變更: As part of **PEP 667**, return an instance of *`PyFrameLocalsProxy_Type`*.

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

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Return the line number that *frame* is currently executing.

Frame Locals Proxies

在 3.13 版被加入.

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

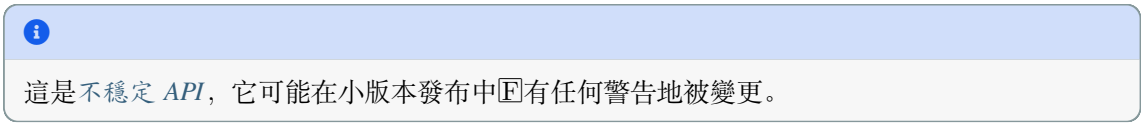
Unless using **PEP 523**, you will not need this.

struct **_PyInterpreterFrame**

The interpreter's internal frame representation.

在 3.11 版被加入.

PyObject ***PyUnstable_InterpreterFrame_GetCode**(struct *_PyInterpreterFrame* *frame);

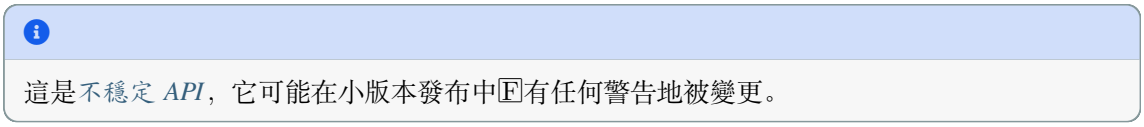


這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

Return a *strong reference* to the code object for the frame.

在 3.12 版被加入.

int **PyUnstable_InterpreterFrame_GetLasti**(struct *_PyInterpreterFrame* *frame);

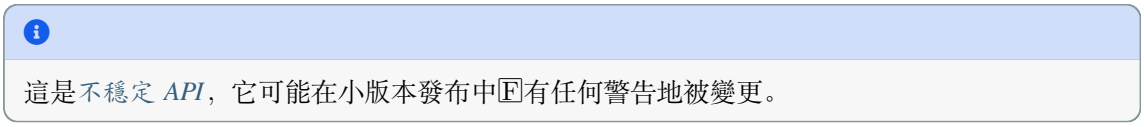


這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

Return the byte offset into the last executed instruction.

在 3.12 版被加入.

int **PyUnstable_InterpreterFrame_GetLine**(struct *_PyInterpreterFrame* *frame);



這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

Return the currently executing line number, or -1 if there is no line number.

在 3.12 版被加入.

9.6.13 🄵生器 (Generator) 物件

🄵生器物件是 Python 用來實現🄵生器🄵代器 (generator iterator) 的物件。它們通常透過🄵代會🄵生值的函式來建立，而不是顯式呼叫*PyGen_New()* 或*PyGen_NewWithQualName()*。

type **PyGenObject**

用於🄵生器物件的 C 結構。

PyTypeObject **PyGen_Type**

與🄵生器物件對應的型🄵物件。

int **PyGen_Check**(*PyObject* *ob)

如果 *ob* 是一個🄵生器 (generator) 物件則回傳真值；*ob* 必須不🄵 NULL。此函式總是會成功執行。

int **PyGen_CheckExact**(*PyObject* *ob)

如果 *ob* 的型🄵是*PyGen_Type* 則回傳真值；*ob* 必須不🄵 NULL。此函式總是會成功執行。

PyObject ***PyGen_New**(*PyFrameObject* *frame)

回傳值：新的參照。 基於 *frame* 物件建立🄵回傳一個新的🄵生器物件。此函式會取走一個對 *frame* 的參照 (reference)。引數必須不🄵 NULL。

PyObject ***PyGen_NewWithQualName**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

回傳值：新的參照。 基於 *frame* 物件建立🄵回傳一個新的🄵生器物件，其中 __name__ 和 __qualname__ 設🄵 *name* 和 *qualname*。此函式會取走一個對 *frame* 的參照。*frame* 引數必須不🄵 NULL。

9.6.14 Coroutine（協程）物件

在 3.5 版被加入.

Coroutine 物件是那些以 `async` 關鍵字來宣告的函式所回傳的物件。

type **PyCoroObject**

用於 coroutine 物件的 C 結構。

PyTypeObject **PyCoro_Type**

與 coroutine 物件對應的型🄵物件。

int **PyCoro_CheckExact**(*PyObject* *ob)

如果 *ob* 的型🄵是*PyCoro_Type* 則回傳真值；*ob* 必須不🄵 NULL。此函式總是會執行成功。

PyObject ***PyCoro_New**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

回傳值：新的參照。 基於 *frame* 物件來建立🄵回傳一個新的 coroutine 物件，其中 __name__ 和 __qualname__ 被設🄵 *name* 和 *qualname*。此函式會取得一個對 *frame* 的參照 (reference)。*frame* 引數必須不🄵 NULL。

9.6.15 情境變數物件（Context Variables Objects）

在 3.7 版被加入.

在 3.7.1 版的變更:

備🄵

在 Python 3.7.1 中所有情境變數 C API 的簽名都被**改**🄵使用*PyObject* 指標，而不是*PyContext*、*PyContextVar* 或*PyContextToken*，例如:

```
// 在 3.7.0:
PyContext *PyContext_New(void);

// 在 3.7.1+:
PyObject *PyContext_New(void);
```

更多細節請見 bpo-34762。

本節詳述 `contextvars` 模組的公開 C API。

type **PyContext**
用來代表 `contextvars.Context` 物件的 C 結構。

type **PyContextVar**
用來代表 `contextvars.ContextVar` 物件的 C 結構。

type **PyContextToken**
用來代表 `contextvars.Token` 物件的 C 結構。

PyTypeObject **PyContext_Type**
代表 *context* 型🄵的型🄵物件。

PyTypeObject **PyContextVar_Type**
代表 情境變數型🄵的型🄵物件。

PyTypeObject **PyContextToken_Type**
代表 情境變數 *token* 型🄵的型🄵物件。

型🄵檢查巨集：

int **PyContext_CheckExact**(*PyObject* *o)
如果 *o* 的型🄵🄵*PyContext_Type* 則回傳 true。*o* 不得🄵 `NULL`。此函式一定會成功回傳。

int **PyContextVar_CheckExact**(*PyObject* *o)
如果 *o* 的類型🄵*PyContextVar_Type* 則回傳 true。*o* 不得🄵 `NULL`。此函式一定會成功回傳。

int **PyContextToken_CheckExact**(*PyObject* *o)
如果 *o* 的類型🄵*PyContextToken_Type* 則回傳 true。*o* 不得🄵 `NULL`。此函式一定會成功回傳。

情境物件管理函式：

PyObject ***PyContext_New**(void)
回傳值：新的參照。 建立一個新的空的情境物件。如果發生錯誤，則回傳 `NULL`。

PyObject ***PyContext_Copy**(*PyObject* *ctx)
回傳值：新的參照。 建立傳入的 *ctx* 情境物件的淺層🄵🄵 (shallow copy)。如果發生錯誤，則回傳 `NULL`。

PyObject ***PyContext_CopyCurrent**(void)
回傳值：新的參照。 建立目前執行緒上的情境的淺層🄵🄵。如果發生錯誤，則回傳 `NULL`。

int **PyContext_Enter**(*PyObject* *ctx)
設定 *ctx* 🄵目前執行緒上的目前情境。成功時回傳 `0`，錯誤時回傳 `-1`。

int **PyContext_Exit**(*PyObject* *ctx)
關閉 *ctx* 情境，🄵將目前執行緒的目前情境設回之前的情境。成功時回傳 `0`，錯誤時回傳 `-1`。

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.

在 3.14 版被加入.

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.)

在 3.14 版被加入.

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.

在 3.14 版被加入.

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.

在 3.14 版被加入.

情境變數函式：

PyObject ***PyContextVar_New**(const char *name, *PyObject* *def)

回傳值：新的參照。 建立一個新的 `ContextVar` 物件。*name* 參數用於自我檢查（introspection）和除錯目的。*def* 參數指定情境變數的預設值，亦可用 `NULL` 表示🄵有預設值。如果發生錯誤，此函式會回傳 `NULL`。

int **PyContextVar_Get**(*PyObject* *var, *PyObject* *default_value, *PyObject* **value)

取得情境變數的值。如果在查找過程中發生錯誤則回傳 `-1`，如果🄵有發生錯誤則無論是否找到值都會回傳 `0`。

如果找到情境變數，*value* 將會是指向它的指標。如果 🄵有找到情境變數，*value* 將會指到：

- *default_value* 預設值，但前提是預設值不是 `NULL`；
- *var* 的預設值，但前提是預設值不是 `NULL`；
- `NULL`

除了 `NULL` 之外，此函式會回傳一個新的參照（reference）。

PyObject ***PyContextVar_Set**(*PyObject* *var, *PyObject* *value)

回傳值：新的參照。 在目前的情境中將 *var* 的值設🄵 *value*。會回傳一個用來代表此變更的新 token 物件，如果發生錯誤則回傳 `NULL`。

int **PyContextVar_Reset**(*PyObject* *var, *PyObject* *token)

將 *var* 情境變數的狀態設回之前的狀態，亦即上一次回傳 token 的*PyContextVar_Set()* 被呼叫前的狀態。此函式在成功時回傳 `0`，錯誤時回傳 `-1`。

9.6.16 DateTime 物件

`datetime` 模組提供各種日期和時間物件。在使用任何這些函式之前，必須將標頭檔 `datetime.h` 引入於原始碼中（請注意，`Python.h` 🄵無引入該標頭檔），且巨集 `PyDateTime_IMPORT` 必須被叫用，而這通常作🄵模組初始化函式的一部分。該巨集將指向 C 結構的指標放入🄵態變數 `PyDateTimeAPI` 中，該變數會被以下巨集使用。

type **PyDateTime_Date**

This subtype of *PyObject* represents a Python date object.

type **PyDateTime_DateTime**

This subtype of *PyObject* represents a Python datetime object.

type **PyDateTime_Time**

This subtype of *PyObject* represents a Python time object.

type **PyDateTime_Delta**

This subtype of *PyObject* represents the difference between two datetime values.

PyTypeObject **PyDateTime_DateType**

This instance of *PyTypeObject* represents the Python date type; it is the same object as `datetime.date` in the Python layer.

PyTypeObject **PyDateTime_DateTimeType**

This instance of *PyTypeObject* represents the Python datetime type; it is the same object as `datetime.datetime` in the Python layer.

PyTypeObject **PyDateTime_TimeType**

This instance of *PyTypeObject* represents the Python time type; it is the same object as `datetime.time` in the Python layer.

PyTypeObject **PyDateTime_DeltaType**

This instance of *PyTypeObject* represents Python type for the difference between two datetime values; it is the same object as `datetime.timedelta` in the Python layer.

PyTypeObject **PyDateTime_TZInfoType**

This instance of *PyTypeObject* represents the Python time zone info type; it is the same object as `datetime.tzinfo` in the Python layer.

用於存取 UTC 單例 (singleton) 的巨集：

PyObject ***PyDateTime_TimeZone_UTC**

回傳表示 UTC 的時區單例，是與 `datetime.timezone.utc` 相同的物件。

在 3.7 版被加入.

型🄵檢查巨集：

int **PyDate_Check**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_DateType* 或 `PyDateTime_DateType` 的子型🄵，則回傳 true。*ob* 不得🄵 `NULL`。這個函式一定會執行成功。

int **PyDate_CheckExact**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_DateType*，則回傳 true。*ob* 不得🄵 `NULL`。這個函式一定會執行成功。

int **PyDateTime_Check**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_DateTimeType* 或 `PyDateTime_DateTimeType` 的子型🄵，則回傳 true。*ob* 不得🄵 `NULL`。這個函式一定會執行成功。

int **PyDateTime_CheckExact**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_DateTimeType*，則回傳 true。*ob* 不得🄵 `NULL`。這個函式一定會執行成功。

int **PyTime_Check**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_TimeType* 或 `PyDateTime_TimeType` 的子型🄵，則回傳 true。*ob* 不得🄵 `NULL`。這個函式一定會執行成功。

int **PyTime_CheckExact**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_TimeType*，則回傳 true。*ob* 不得🄵 `NULL`。這個函式一定會執行成功。

int **PyDelta_Check**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_DeltaType* 或 `PyDateTime_DeltaType` 的子型🄵，則回傳 true。*ob* 不得🄵 `NULL`。這個函式一定會執行成功。

int **PyDelta_CheckExact**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_DeltaType*，則回傳 true。*ob* 不得🄵 `NULL`。這個函式一定會執行成功。

int **PyTZInfo_Check**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_TZInfoType* 或 PyDateTime_TZInfoType 的子型🄵，則回傳 true。*ob* 不得🄵 NULL。這個函式一定會執行成功。

int **PyTZInfo_CheckExact**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_TZInfoType*，則回傳 true。*ob* 不得🄵 NULL。這個函式一定會執行成功。

建立物件的巨集：

PyObject ***PyDate_FromDate**(int year, int month, int day)

回傳值：新的參照。回傳一個有特定年、月、日的物件 datetime.date。

PyObject ***PyDateTime_FromDateAndTime**(int year, int month, int day, int hour, int minute, int second, int usecond)

回傳值：新的參照。回傳一個有特定年、月、日、時、分、秒、微秒的物件 datetime.datetime。

PyObject ***PyDateTime_FromDateAndTimeAndFold**(int year, int month, int day, int hour, int minute, int second, int usecond, int fold)

回傳值：新的參照。回傳一個有特定年、月、日、時、分、秒、微秒與 fold（時間折🄵）的物件 datetime.datetime。

在 3.6 版被加入.

PyObject ***PyTime_FromTime**(int hour, int minute, int second, int usecond)

回傳值：新的參照。回傳一個有特定時、分、秒、微秒的物件 datetime.date。

PyObject ***PyTime_FromTimeAndFold**(int hour, int minute, int second, int usecond, int fold)

回傳值：新的參照。回傳一個有特定時、分、秒、微秒與 fold（時間折🄵）的物件 datetime.time。

在 3.6 版被加入.

PyObject ***PyDelta_FromDSU**(int days, int seconds, int useconds)

回傳值：新的參照。回傳一個 datetime.timedelta 物件，表示給定的天數、秒數和微秒數。執行標準化 (normalization) 以便生成的微秒數和秒數位於 datetime.timedelta 物件記🄵的範圍🄵。

PyObject ***PyTimeZone_FromOffset**(*PyObject* *offset)

回傳值：新的參照。回傳一個 datetime.timezone 物件，其未命名的固定偏移量由 *offset* 引數表示。

在 3.7 版被加入.

PyObject ***PyTimeZone_FromOffsetAndName**(*PyObject* *offset, *PyObject* *name)

回傳值：新的參照。回傳一個 datetime.timezone 物件，其固定偏移量由 *offset* 引數表示，🄵帶有 tzname *name*。

在 3.7 版被加入.

從 date 物件中提取欄位的巨集。引數必須是個*PyDateTime_Date* 的實例，包括子類🄵（例如*PyDateTime_DateTime*）。引數不得🄵 NULL，🄵且不會檢查型🄵：

int **PyDateTime_GET_YEAR**(*PyDateTime_Date* *o)

回傳年份，🄵正整數。

int **PyDateTime_GET_MONTH**(*PyDateTime_Date* *o)

回傳月份，🄵正整數，從 1 到 12。

int **PyDateTime_GET_DAY**(*PyDateTime_Date* *o)

回傳日期，🄵正整數，從 1 到 31。

從 datetime 物件中提取欄位的巨集。引數必須是個*PyDateTime_DateTime* 的實例，包括子類🄵。引數不得🄵 NULL，🄵且不會檢查型🄵：

int **PyDateTime_DATE_GET_HOUR**(*PyDateTime_DateTime* *o)
回傳小時，🄵正整數，從 0 到 23。

int **PyDateTime_DATE_GET_MINUTE**(*PyDateTime_DateTime* *o)
回傳分鐘，🄵正整數，從 0 到 59。

int **PyDateTime_DATE_GET_SECOND**(*PyDateTime_DateTime* *o)
回傳秒，🄵正整數，從 0 到 59。

int **PyDateTime_DATE_GET_MICROSECOND**(*PyDateTime_DateTime* *o)
回傳微秒，🄵正整數，從 0 到 999999。

int **PyDateTime_DATE_GET_FOLD**(*PyDateTime_DateTime* *o)
回傳 fold，🄵 0 或 1 的正整數。

在 3.6 版被加入.

PyObject ***PyDateTime_DATE_GET_TZINFO**(*PyDateTime_DateTime* *o)
回傳 tzinfo（可能是 `None`）。

在 3.10 版被加入.

從 time 物件中提取欄位的巨集。引數必須是個*`PyDateTime_Time`* 的實例，包括子類🄵。引數不得🄵 `NULL`，🄵且不會檢查型🄵：

int **PyDateTime_TIME_GET_HOUR**(*PyDateTime_Time* *o)
回傳小時，🄵正整數，從 0 到 23。

int **PyDateTime_TIME_GET_MINUTE**(*PyDateTime_Time* *o)
回傳分鐘，🄵正整數，從 0 到 59。

int **PyDateTime_TIME_GET_SECOND**(*PyDateTime_Time* *o)
回傳秒，🄵正整數，從 0 到 59。

int **PyDateTime_TIME_GET_MICROSECOND**(*PyDateTime_Time* *o)
回傳微秒，🄵正整數，從 0 到 999999。

int **PyDateTime_TIME_GET_FOLD**(*PyDateTime_Time* *o)
回傳 fold，🄵 0 或 1 的正整數。

在 3.6 版被加入.

PyObject ***PyDateTime_TIME_GET_TZINFO**(*PyDateTime_Time* *o)
回傳 tzinfo（可能是 `None`）。

在 3.10 版被加入.

從 time delta 物件中提取欄位的巨集。引數必須是個*`PyDateTime_Delta`* 的實例，包括子類🄵。引數不能🄵 `NULL`，🄵且不會檢查型🄵：

int **PyDateTime_DELTA_GET_DAYS**(*PyDateTime_Delta* *o)
以 -999999999 到 999999999 之間的整數形式回傳天數。

在 3.3 版被加入.

int **PyDateTime_DELTA_GET_SECONDS**(*PyDateTime_Delta* *o)
以 0 到 86399 之間的整數形式回傳秒數。

在 3.3 版被加入.

int **PyDateTime_DELTA_GET_MICROSECONDS**(*PyDateTime_Delta* *o)
以 0 到 999999 之間的整數形式回傳微秒數。

在 3.3 版被加入.

🄵了方便模組實作 DB API 的巨集：

PyObject ***PyDateTime_FromTimestamp**(*PyObject* *args)

回傳值：新的參照。 給定一個適合傳遞給 `datetime.datetime.fromtimestamp()` 的引數元組，建立🄵回傳一個新的 `datetime.datetime` 物件。

PyObject ***PyDate_FromTimestamp**(*PyObject* *args)

回傳值：新的參照。 給定一個適合傳遞給 `datetime.date.fromtimestamp()` 的引數元組，建立🄵回傳一個新的 `datetime.date` 物件。

9.6.17 型🄵提示物件

提供了數個用於型🄵提示的🄵建型🄵。目前有兩種 -- GenericAlias 和 Union。只有 `GenericAlias` 有公開 (expose) 給 C。

PyObject ***Py_GenericAlias**(*PyObject* *origin, *PyObject* *args)

🄵穩定 ABI 的一部分 自 *3.9* 版本開始. 建立一個 GenericAlias 物件，等同於呼叫 Python 的 `types.GenericAlias` class。*origin* 和 *args* 引數分🄵設定了 `GenericAlias` 的 `__origin__` 與 `__args__` 屬性。*origin* 應該要是個 *PyTypeObject** 且 *args* 可以是個 *PyTupleObject** 或任意 `PyObject*`。如果傳入的 *args* 不是個 tuple（元組），則會自動建立一個長度🄵 1 的 tuple 且 `__args__` 會被設🄵 `(args,)`。只會進行最少的引數檢查，所以即便 *origin* 不是個型🄵，函式也會不會失敗。`GenericAlias` 的 `__parameters__` 屬性會自 `__args__` 惰性地建立 (constructed lazily)。當失敗時，會引發一個例外🄵回傳 `NULL`。

以下是個讓一個擴充型🄵泛用化 (generic) 的例子：

```
...
static PyMethodDef my_obj_methods[] = {
    // 其他方法。
    ...
    {"__class_getitem__", Py_GenericAlias, METH_O|METH_CLASS, "See PEP 585"}
    ...
}
```

也參考

資料模型方法 `__class_getitem__()`。

在 3.9 版被加入.

PyTypeObject **Py_GenericAliasType**

🄵穩定 ABI 的一部分 自 *3.9* 版本開始. *Py_GenericAlias()* 所回傳該物件的 C 型🄵。等價於 Python 中的 `types.GenericAlias`。

在 3.9 版被加入.

CHAPTER 10

初始化、終結化與執行緒

關於如何在初始化之前設定直譯器的細節，請參見*Python 初始化設定*。

10.1 Python 初始化之前

In an application embedding Python, the `Py_Initialize()` function must be called before using any other Python/C API functions; with the exception of a few functions and the *global configuration variables*.

The following functions can be safely called before Python is initialized:

- Functions that initialize the interpreter:
  - `Py_Initialize()`
  - `Py_InitializeEx()`
  - `Py_InitializeFromConfig()`
  - `Py_BytesMain()`
  - `Py_Main()`
  - the runtime pre-initialization functions covered in *Python 初始化設定*
- Configuration functions:
  - `PyImport_AppendInittab()`
  - `PyImport_ExtendInittab()`
  - `PyInitFrozenExtensions()`
  - `PyMem_SetAllocator()`
  - `PyMem_SetupDebugHooks()`
  - `PyObject_SetArenaAllocator()`
  - `Py_SetProgramName()`
  - `Py_SetPythonHome()`
  - `PySys_ResetWarnOptions()`
  - the configuration functions covered in *Python 初始化設定*
- Informative functions:

  - *Py_IsInitialized()*
  - *PyMem_GetAllocator()*
  - *PyObject_GetArenaAllocator()*
  - *Py_GetBuildInfo()*
  - *Py_GetCompiler()*
  - *Py_GetCopyright()*
  - *Py_GetPlatform()*
  - *Py_GetVersion()*
  - *Py_IsInitialized()*
- Utilities:
  - *Py_DecodeLocale()*
  - the status reporting and utility functions covered in *Python 初始化設定*
- Memory allocators:
  - *PyMem_RawMalloc()*
  - *PyMem_RawRealloc()*
  - *PyMem_RawCalloc()*
  - *PyMem_RawFree()*
- Synchronization:
  - *PyMutex_Lock()*
  - *PyMutex_Unlock()*

備🄵

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 Global configuration variables

Python has variables for the global configuration to control different features and options. By default, these flags are controlled by command line options.

When a flag is set by an option, the value of the flag is the number of times that the option was set. For example, `-b` sets *Py_BytesWarningFlag* to 1 and `-bb` sets *Py_BytesWarningFlag* to 2.

int **Py_BytesWarningFlag**

This API is kept for backward compatibility: setting *PyConfig.bytes_warning* should be used instead, see *Python Initialization Configuration*.

Issue a warning when comparing `bytes` or `bytearray` with `str` or `bytes` with `int`. Issue an error if greater or equal to `2`.

由 `-b` 選項設定。

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*.

Turn on parser debugging output (for expert only, depending on compilation options).

由 `-d` 選項與 `PYTHONDEBUG` 環境變數設定。

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*.

If set to non-zero, Python won't try to write `.pyc` files on the import of source modules.

由 `-B` 選項與 `PYTHONDONTWRITEBYTECODE` 環境變數設定。

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*.

Suppress error messages when calculating the module search path in *Py_GetPath()*.

Private flag used by `_freeze_module` and `frozenmain` programs.

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*.

如果環境變數 `PYTHONHASHSEED` 被設定🄴一個非空字串則設🄵 `1`。

If the flag is non-zero, read the `PYTHONHASHSEED` environment variable to initialize the secret hash seed.

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*.

忽略所有可能被設定的 `PYTHON*` 環境變數，例如 `PYTHONPATH` 與 `PYTHONHOME`。

由 `-E` 與 `-I` 選項設定。

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*.

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.

由 `-i` 選項與 `PYTHONINSPECT` 環境變數設定。

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*.

由 `-i` 選項設定。

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*.

Run Python in isolated mode. In isolated mode `sys.path` contains neither the script's directory nor the user's site-packages directory.

由 `-i` 選項設定。

在 3.4 版被加入.

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*.

If the flag is non-zero, use the `mbcs` encoding with `replace` error handler, instead of the UTF-8 encoding with `surrogatepass` error handler, for the *filesystem encoding and error handler*.

如果環境變數 `PYTHONLEGACYWINDOWSFSENCODING` 被設定🄵一個非空字串則設🄵 `1`。

更多詳情請見 **PEP 529**。

可用性: 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.

Set to `1` if the `PYTHONLEGACYWINDOWSSTDIO` environment variable is set to a non-empty string.

更多詳情請見 **PEP 528**。

可用性: 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*.

Disable the import of the module `site` and the site-dependent manipulations of `sys.path` that it entails. Also disable these manipulations if `site` is explicitly imported later (call `site.main()` if you want them to be triggered).

由 `-S` 選項設定。

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*.

Don't add the `user site-packages directory` to `sys.path`.

由 `-s` 選項、`-I` 選項與 `PYTHONNOUSERSITE` 環境變數設定。

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*.

由 `-O` 選項與 `PYTHONOPTIMIZE` 環境變數設定。

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*.

Don't display the copyright and version messages even in interactive mode.

由 `-q` 選項設定。

在 3.2 版被加入.

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*.

Force the stdout and stderr streams to be unbuffered.

由 `-u` 選項與 `PYTHONUNBUFFERED` 環境變數設定。

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*.

Print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. If greater 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.

由 `-v` 選項與 `PYTHONVERBOSE` 環境變數設定。

Deprecated since version 3.12, will be removed in version 3.15.

10.3 Initializing and finalizing the interpreter

void **Py_Initialize**()

🄵穩定 ABI 的一部分. Initialize the Python interpreter. In an application embedding Python, this should be called before using any other Python/C API functions; see *Before Python Initialization* for the few exceptions.

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*.

> **備🄵**
>
> On Windows, changes the console mode from `O_TEXT` to `O_BINARY`, which will also affect non-Python uses of the console using the C Runtime.

void **Py_InitializeEx**(int initsigs)

🄵穩定 ABI 的一部分. 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 *Initialization with PyConfig*.

See the *Python 初始化設定* section for details on pre-initializing the interpreter, populating the runtime configuration structure, and querying the returned status structure.

int **Py_IsInitialized**()

🄵穩定 ABI 的一部分. Return true (nonzero) when the Python interpreter has been initialized, false (zero) if not. After `Py_FinalizeEx()` is called, this returns false until `Py_Initialize()` is called again.

int **Py_IsFinalizing**()

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Return true (non-zero) if the main Python interpreter is *shutting down*. Return false (zero) otherwise.

在 3.13 版被加入.

int **Py_FinalizeEx**()

🄵穩定 ABI 的一部分 自 *3.6* 版本開始. 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.

This function is provided for a number of reasons. An embedding application might want to restart Python without having to restart the application itself. An application that has loaded the Python interpreter from a dynamically loadable library (or DLL) might want to free all memory allocated by Python before unloading the DLL. During a hunt for memory leaks in an application a developer might want to free all memory allocated by Python before exiting from the application.

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.

引發一個不附帶引數的稽核事件 `cpython._PySys_ClearAuditHooks`。

在 3.6 版被加入.

void **Py_Finalize**()

🄵穩定 ABI 的一部分. This is a backwards-compatible version of `Py_FinalizeEx()` that disregards the return value.

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

🄵穩定 ABI 的一部分 自 *3.8* 版本開始. 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.

在 3.8 版被加入.

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

🄵穩定 ABI 的一部分. 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)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

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*.

在 3.13 版被加入.

10.4 Process-wide parameters

void **Py_SetProgramName**(const wchar_t *name)

🄵穩定 ABI 的一部分. This API is kept for backward compatibility: setting *PyConfig.program_name* should be used instead, see *Python Initialization Configuration*.

This function should be called before *Py_Initialize()* is called for the first time, if it is called at all. It tells the interpreter the value of the `argv[0]` argument to the `main()` function of the program (converted to wide characters). This is used by *Py_GetPath()* and some other functions below to find the Python run-time libraries relative to the interpreter executable. The default value is `'python'`. The argument should point to a zero-terminated wide character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

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**()

🄵穩定 ABI 的一部分. 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.

此函式不應該在*Py_Initialize()* 之前呼叫，否則會回傳 `NULL`。

在 3.10 版的變更: 如果在*Py_Initialize()* 之前呼叫，現在會回傳 `NULL`。

Deprecated since version 3.13, will be removed in version 3.15: Use *PyConfig_Get("executable")* (`sys.executable`) instead.

wchar_t ***Py_GetPrefix**()

🄵穩定 ABI 的一部分. 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.

此函式不應該在*Py_Initialize()* 之前呼叫，否則會回傳 `NULL`。

在 3.10 版的變更: 如果在*Py_Initialize()* 之前呼叫，現在會回傳 `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**()

🄵穩定 ABI 的一部分. 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.

Background: The exec-prefix differs from the prefix when platform dependent files (such as executables and shared libraries) are installed in a different directory tree. In a typical installation, platform dependent files may be installed in the `/usr/local/plat` subtree while platform independent may be installed in `/usr/local`.

Generally speaking, a platform is a combination of hardware and software families, e.g. Sparc machines running the Solaris 2.x operating system are considered the same platform, but Intel machines running Solaris 2.x are another platform, and Intel machines running Linux are yet another platform. Different major revisions of the same operating system generally also form different platforms. Non-Unix operating systems are a different story; the installation strategies on those systems are so different that the prefix and exec-prefix are meaningless, and set to the empty string. Note that compiled Python bytecode files are platform independent (but not independent from the Python version by which they were compiled!).

System administrators will know how to configure the **mount** or **automount** programs to share `/usr/local` between platforms while having `/usr/local/plat` be a different filesystem for each platform.

此函式不應該在*Py_Initialize()* 之前呼叫，否則會回傳 `NULL`。

在 3.10 版的變更: 如果在*Py_Initialize()* 之前呼叫，現在會回傳 `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**()

🄵穩定 ABI 的一部分. 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`.

此函式不應該在*Py_Initialize()* 之前呼叫，否則會回傳 `NULL`。

在 3.10 版的變更: 如果在*Py_Initialize()* 之前呼叫，現在會回傳 `NULL`。

Deprecated since version 3.13, will be removed in version 3.15: Use *PyConfig_Get("executable")* (`sys.executable`) instead.

wchar_t ***Py_GetPath**()

🄵穩定 ABI 的一部分. 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.

此函式不應該在*Py_Initialize()* 之前呼叫，否則會回傳 `NULL`。

在 3.10 版的變更: 如果在*Py_Initialize()* 之前呼叫，現在會回傳 `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**()

🄵穩定 ABI 的一部分. Return the version of this Python interpreter. This is a string that looks something like

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

The first word (up to the first space character) is the current Python version; the first characters are the major and minor version separated by a period. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.version`.

另請參🄵*Py_Version* 常數。

const char ***Py_GetPlatform**()

🄵穩定 ABI 的一部分. Return the platform identifier for the current platform. On Unix, this is formed from the "official" name of the operating system, converted to lower case, followed by the major revision number; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is `'sunos5'`. On macOS, it is `'darwin'`. On Windows, it is `'win'`. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.platform`.

const char *__Py_GetCopyright__()

🄵穩定 ABI 的一部分. Return the official copyright string for the current Python version, for example

```
'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.copyright`.

const char *__Py_GetCompiler__()

🄵穩定 ABI 的一部分. Return an indication of the compiler used to build the current Python version, in square brackets, for example:

```
"[GCC 2.7.2.2]"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

const char *__Py_GetBuildInfo__()

🄵穩定 ABI 的一部分. Return information about the sequence number and build date and time of the current Python interpreter instance, for example

```
"#67, Aug  1 1997, 22:34:28"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

void __PySys_SetArgvEx__(int argc, wchar_t **argv, int updatepath)

🄵穩定 ABI 的一部分. This API is kept for backward compatibility: setting *PyConfig.argv*, *PyConfig.parse_argv* and *PyConfig.safe_path* should be used instead, see *Python Initialization Configuration*.

Set `sys.argv` based on *argc* and *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. If this function fails to initialize `sys.argv`, a fatal condition is signalled using *Py_FatalError()*.

If *updatepath* is zero, this is all the function does. If *updatepath* is non-zero, the function also modifies `sys.path` according to the following algorithm:

- If the name of an existing script is passed in `argv[0]`, the absolute path of the directory where the script is located is prepended to `sys.path`.
- Otherwise (that is, if *argc* is `0` or `argv[0]` doesn't point to an existing file name), an empty string is prepended to `sys.path`, which is the same as prepending the current working directory (`"."`).

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_t*` string.

See also *PyConfig.orig_argv* and *PyConfig.argv* members of the *Python Initialization Configuration*.

備🄵

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**.

On versions before 3.1.3, you can achieve the same effect by manually popping the first `sys.path` element after having called *PySys_SetArgv()*, for example using:

```
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```

在 3.1.3 版被加入.

Deprecated since version 3.11, will be removed in version 3.15.

void **PySys_SetArgv**(int argc, wchar_t **argv)

🄵穩定 ABI 的一部分. This API is kept for backward compatibility: setting *PyConfig.argv* and *PyConfig.parse_argv* should be used instead, see *Python Initialization Configuration*.

This function works like *PySys_SetArgvEx()* with *updatepath* set to `1` unless the **python** interpreter was started with the `-I`.

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_t*` string.

See also *PyConfig.orig_argv* and *PyConfig.argv* members of the *Python Initialization Configuration*.

在 3.4 版的變更: The *updatepath* value depends on `-I`.

Deprecated since version 3.11, will be removed in version 3.15.

void **Py_SetPythonHome**(const wchar_t *home)

🄵穩定 ABI 的一部分. This API is kept for backward compatibility: setting *PyConfig.home* should be used instead, see *Python Initialization Configuration*.

Set the default "home" directory, that is, the location of the standard Python libraries. See `PYTHONHOME` for the meaning of the argument string.

The argument should point to a zero-terminated character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

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**()

🄵穩定 ABI 的一部分. Return the default "home", that is, the value set by *PyConfig.home*, or the value of the `PYTHONHOME` environment variable if it is set.

此函式不應該在*Py_Initialize()* 之前呼叫，否則會回傳 `NULL`。

在 3.10 版的變更: 如果在*Py_Initialize()* 之前呼叫，現在會回傳 `NULL`。

Deprecated since version 3.13, will be removed in version 3.15: 改🄵使用*PyConfig_Get("home")* 或 `PYTHONHOME` 環境變數。

10.5 Thread State and the Global Interpreter Lock

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.

Therefore, the rule exists that only the thread that has acquired the *GIL* may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads (see `sys.setswitchinterval()`). The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.

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.
```

This is so common that a pair of macros exists to simplify it:

```
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

The *Py_BEGIN_ALLOW_THREADS* macro opens a new block and declares a hidden local variable; the *Py_END_ALLOW_THREADS* macro closes the block.

The block above expands to the following code:

```
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.

備🄵

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 Non-Python created threads

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.

The *PyGILState_Ensure()* and *PyGILState_Release()* functions do all of the above automatically. The typical idiom for calling into Python from a C thread is:

```
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 Cautions about fork()

Another important thing to note about threads is their behaviour in the face of the C `fork()` call. On most systems with `fork()`, after a process forks only the thread that issued the fork will exist. This has a concrete impact both on how locks must be handled and on all stored state in CPython's runtime.

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.

The fact that all other threads go away also means that CPython's runtime state there must be cleaned up properly, which `os.fork()` does. This means finalizing all other *`PyThreadState`* objects belonging to the current interpreter and all other *`PyInterpreterState`* objects. Due to this and the special nature of the *"main" interpreter*, `fork()` should only be called in that interpreter's "main" thread, where the CPython global runtime was originally initialized. The only exception is if `exec()` will be called immediately after.

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 高階 API

These are the most commonly used types and functions when writing C extension code, or when embedding the Python interpreter:

type **PyInterpreterState**

🄵受限 API 的一部分 （做🄵一個不透明結構 *(opaque struct)*）. This data structure represents the state shared by a number of cooperating threads. Threads belonging to the same interpreter share their module administration and a few other internal items. There are no public members in this structure.

Threads belonging to different interpreters initially share nothing, except process state like available memory, open file descriptors and such. The global interpreter lock is also shared by all threads, regardless of to which interpreter they belong.

在 3.12 版的變更: **PEP 684** introduced the possibility of a *per-interpreter GIL*. See `Py_NewInterpreterFromConfig()`.

type **PyThreadState**

🄵受限 API 的一部分 （做🄵一個不透明結構 *(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**()

🄵穩定 ABI 的一部分. Deprecated function which does nothing.

In Python 3.6 and older, this function created the GIL if it didn't exist.

在 3.9 版的變更: 此函式現在不會做任何事情。

在 3.7 版的變更: This function is now called by `Py_Initialize()`, so you don't have to call it yourself anymore.

在 3.2 版的變更: This function cannot be called before `Py_Initialize()` anymore.

在 3.9 版之後被🄵用.

PyThreadState ***PyEval_SaveThread**()

🄵穩定 ABI 的一部分. Detach the *attached thread state* and return it. The thread will have no *thread state* upon returning.

void **PyEval_RestoreThread**(*PyThreadState* *tstate)

🄵穩定 ABI 的一部分. Set the *attached thread state* to *tstate*. The passed *thread state* **should not** be *attached*, otherwise deadlock ensues. *tstate* will be attached upon returning.

> **備🄵**
>
> 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.

在 3.14 版的變更: Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

PyThreadState ***PyThreadState_Get**()

🄵穩定 ABI 的一部分. 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`).

也請見`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.

在 3.13 版被加入: In Python 3.5 to 3.12, the function was private and known as `_PyThreadState_UncheckedGet()`.

PyThreadState ***PyThreadState_Swap**(*PyThreadState* *tstate)

🄵穩定 ABI 的一部分. 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.

> **備🄵**
>
> Similar to `PyGILState_Ensure()`, this function will hang the thread if the runtime is finalizing.

The following functions use thread-local storage, and are not compatible with sub-interpreters:

PyGILState_STATE **PyGILState_Ensure**()

🄵穩定 ABI 的一部分. 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.

> **警告**
>
> 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.

在 3.14 版的變更: Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

void **PyGILState_Release**(PyGILState_STATE)

🄵穩定 ABI 的一部分. Release any resources previously acquired. After this call, Python's state will be the same as it was prior to the corresponding *PyGILState_Ensure()* call (but generally this state will be unknown to the caller, hence the use of the GILState API).

Every call to *PyGILState_Ensure()* must be matched by a call to *PyGILState_Release()* on the same thread.

PyThreadState ***PyGILState_GetThisThreadState**()

🄵穩定 ABI 的一部分. 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.

> **備🄵**
>
> 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.

> **備🄵**
>
> If the current Python process has ever created a subinterpreter, this function will *always* return `1`. Prefer *PyThreadState_GetUnchecked()* for most cases.

在 3.4 版被加入.

The following macros are normally used without a trailing semicolon; look for example usage in the Python source distribution.

Py_BEGIN_ALLOW_THREADS

🄵穩定 ABI 的一部分. This macro expands to `{ PyThreadState *_save; _save = PyEval_SaveThread();`. Note that it contains an opening brace; it must be matched with a following *Py_END_ALLOW_THREADS* macro. See above for further discussion of this macro.

Py_END_ALLOW_THREADS

🄵穩定 ABI 的一部分. This macro expands to `PyEval_RestoreThread(_save); }`. Note that it contains a closing brace; it must be matched with an earlier *Py_BEGIN_ALLOW_THREADS* macro. See above for further discussion of this macro.

Py_BLOCK_THREADS

🄵穩定 ABI 的一部分. This macro expands to `PyEval_RestoreThread(_save);`: it is equivalent to *Py_END_ALLOW_THREADS* without the closing brace.

Py_UNBLOCK_THREADS

🄵穩定 ABI 的一部分. This macro expands to `_save = PyEval_SaveThread();`: it is equivalent to *Py_BEGIN_ALLOW_THREADS* without the opening brace and variable declaration.

10.5.7 低階 API

All of the following functions must be called after *Py_Initialize()*.

在 3.7 版的變更: *Py_Initialize()* now initializes the *GIL* and sets an *attached thread state*.

PyInterpreterState ***PyInterpreterState_New**()

🄵穩定 ABI 的一部分. 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.

引發一個不附帶引數的稽核事件 `cpython.PyInterpreterState_New`。

void **PyInterpreterState_Clear**(*PyInterpreterState* *interp)

🄵穩定 ABI 的一部分. Reset all information in an interpreter state object. There must be an *attached thread state* for the interpreter.

引發一個不附帶引數的稽核事件 `cpython.PyInterpreterState_Clear`。

void **PyInterpreterState_Delete**(*PyInterpreterState* *interp)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. Create a new thread state object belonging to the given interpreter object. An *attached thread state* is not needed.

void **PyThreadState_Clear**(*PyThreadState* *tstate)

🄵穩定 ABI 的一部分. Reset all information in a *thread state* object. *tstate* must be *attached*

在 3.9 版的變更: This function now calls the `PyThreadState.on_delete` callback. Previously, that happened in *PyThreadState_Delete()*.

在 3.13 版的變更: `PyThreadState.on_delete` 回呼已被移除。

void **PyThreadState_Delete**(*PyThreadState* *tstate)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Get the current frame of the Python thread state *tstate*.

Return a *strong reference*. Return `NULL` if no frame is currently executing.

也請見*PyEval_GetFrame()*。

tstate must not be `NULL`, and must be *attached*.

在 3.9 版被加入.

uint64_t **PyThreadState_GetID**(*PyThreadState* *tstate)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Get the unique *thread state* identifier of the Python thread state *tstate*.

tstate must not be `NULL`, and must be *attached*.

在 3.9 版被加入.

PyInterpreterState ***PyThreadState_GetInterpreter**(*PyThreadState* *tstate)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Get the interpreter of the Python thread state *tstate*.

tstate must not be `NULL`, and must be *attached*.

在 3.9 版被加入.

void **PyThreadState_EnterTracing**(*PyThreadState* *tstate)

Suspend tracing and profiling in the Python thread state *tstate*.

Resume them using the `PyThreadState_LeaveTracing()` function.

在 3.11 版被加入.

void **PyThreadState_LeaveTracing**(*PyThreadState* *tstate)

Resume tracing and profiling in the Python thread state *tstate* suspended by the `PyThreadState_EnterTracing()` function.

See also `PyEval_SetTrace()` and `PyEval_SetProfile()` functions.

在 3.11 版被加入.

PyInterpreterState ***PyInterpreterState_Get**(void)

🄵穩定 ABI 的一部分 自 *3.9* 版本開始. Get the current interpreter.

Issue a fatal error if there no *attached thread state*. It cannot return NULL.

在 3.9 版被加入.

int64_t **PyInterpreterState_GetID**(*PyInterpreterState* *interp)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Return the interpreter's unique ID. If there was any error in doing so then `-1` is returned and an error is set.

呼叫者必須擁有一個*attached thread state*。

在 3.7 版被加入.

PyObject ***PyInterpreterState_GetDict**(*PyInterpreterState* *interp)

回傳值：借用參照。🄵穩定 ABI 的一部分 自 *3.8* 版本開始. Return a dictionary in which interpreter-specific data may be stored. If this function returns `NULL` then no exception has been raised and the caller should assume no interpreter-specific dict is available.

This is not a replacement for `PyModule_GetState()`, which extensions should use to store interpreter-specific state information.

The returned dictionary is borrowed from the interpreter and is valid until interpreter shutdown.

在 3.8 版被加入.

typedef *PyObject* *(***_PyFrameEvalFunction**)(*PyThreadState* *tstate, *_PyInterpreterFrame* *frame, int throwflag)

Type of a frame evaluation function.

The *throwflag* parameter is used by the `throw()` method of generators: if non-zero, handle the current exception.

在 3.9 版的變更: The function now takes a *tstate* parameter.

在 3.11 版的變更: The *frame* parameter changed from `PyFrameObject*` to `_PyInterpreterFrame*`.

_PyFrameEvalFunction **_PyInterpreterState_GetEvalFrameFunc**(*PyInterpreterState* *interp)

Get the frame evaluation function.

See the **PEP 523** "Adding a frame evaluation API to CPython".

在 3.9 版被加入.

void **_PyInterpreterState_SetEvalFrameFunc**(*PyInterpreterState* *interp, *_PyFrameEvalFunction* eval_frame)

Set the frame evaluation function.

See the **PEP 523** "Adding a frame evaluation API to CPython".

在 3.9 版被加入.

PyObject ***PyThreadState_GetDict**()

回傳值：借用參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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.

在 3.7 版的變更: The type of the *id* parameter changed from `long` to `unsigned long`.

void **PyEval_AcquireThread**(*PyThreadState* *tstate)

🄵穩定 ABI 的一部分. *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*.

> **備🄵**
>
> 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.

在 3.8 版的變更: Updated to be consistent with *PyEval_RestoreThread()*, *Py_END_ALLOW_THREADS()*, and *PyGILState_Ensure()*, and terminate the current thread if called while the interpreter is finalizing.

在 3.14 版的變更: Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

PyEval_RestoreThread() is a higher-level function which is always available (even when threads have not been initialized).

void **PyEval_ReleaseThread**(*PyThreadState* *tstate)

🄵穩定 ABI 的一部分. 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() is a higher-level function which is always available (even when threads have not been initialized).

10.6 Sub-interpreter support

While in most uses, you will only embed a single Python interpreter, there are cases where you need to create several independent interpreters in the same process and perhaps even in the same thread. Sub-interpreters allow you to do that.

The "main" interpreter is the first one created when the runtime initializes. It is usually the only Python interpreter in a process. Unlike sub-interpreters, the main interpreter has unique process-global responsibilities like signal handling. It is also responsible for execution during runtime initialization and is usually the active interpreter during runtime finalization. The *PyInterpreterState_Main()* function returns a pointer to its state.

You can switch between sub-interpreters using the *PyThreadState_Swap()* function. You can create and destroy them using the following functions:

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.

在 3.12 版被加入.

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)

Create a new sub-interpreter. This is an (almost) totally separate environment for the execution of Python code. In particular, the new interpreter has separate, independent versions of all imported modules, including the fundamental modules `builtins`, `__main__` and `sys`. The table of loaded modules (`sys.modules`) and the module search path (`sys.path`) are also separate. The new environment has no `sys.argv` variable. It

has new standard I/O stream file objects `sys.stdin`, `sys.stdout` and `sys.stderr` (however these refer to the same underlying file descriptors).

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.

在 3.12 版被加入.

Sub-interpreters are most effective when isolated from each other, with certain functionality restricted:

```
PyInterpreterConfig config = {
    .use_main_obmalloc = 0,
    .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*.

Extension modules are shared between (sub-)interpreters as follows:

- For modules using multi-phase initialization, e.g. *PyModule_FromDefAndSpec()*, a separate module object is created and initialized for each interpreter. Only C-level static and global variables are shared between these module objects.
- For modules using single-phase initialization, e.g. *PyModule_Create()*, the first time a particular extension is imported, it is initialized normally, and a (shallow) copy of its module's dictionary is squirreled away. When the same extension is imported by another (sub-)interpreter, a new module is initialized and filled with the contents of this copy; the extension's `init` function is not called. Objects in the module's dictionary thus end up shared across (sub-)interpreters, which might cause unwanted behavior (see *Bugs and caveats* below).

  Note that this is different from what happens when an extension is imported after the interpreter has been completely re-initialized by calling *Py_FinalizeEx()* and *Py_Initialize()*; in that case, the extension's `initmodule` function *is* called again. As with multi-phase initialization, this means that only C-level static and global variables are shared between these modules.

PyThreadState ***Py_NewInterpreter**(void)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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 直譯器各🄵持有的 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.

在 3.12 版被加入.

10.6.2 Bugs and caveats

Because sub-interpreters (and the main interpreter) are part of the same process, the insulation between them isn't perfect --- for example, using low-level file operations like `os.close()` they can (accidentally or maliciously) affect each other's open files. Because of the way extensions are shared between (sub-)interpreters, some extensions may not work properly; this is especially likely when using single-phase initialization or (static) global variables. It is possible to insert objects created in one sub-interpreter into a namespace of another (sub-)interpreter; this should be avoided if possible.

Special care should be taken to avoid sharing user-defined functions, methods, instances or classes between sub-interpreters, since import operations executed by such objects may affect the wrong (sub-)interpreter's dictionary of loaded modules. It is equally important to avoid sharing objects from which the above are reachable.

Also note that combining this functionality with `PyGILState_*` APIs is delicate, because these APIs assume a bijection between Python thread states and OS-level threads, an assumption broken by the presence of sub-interpreters. It is highly recommended that you don't switch sub-interpreters between a pair of matching *PyGILState_Ensure()* and *PyGILState_Release()* calls. Furthermore, extensions (such as `ctypes`) using these APIs to allow calling of Python code from non-Python created threads will probably be broken when using sub-interpreters.

10.7 Asynchronous Notifications

A mechanism is provided to make asynchronous notifications to the main interpreter thread. These notifications take the form of a function pointer and a void pointer argument.

int **Py_AddPendingCall**(int (*func)(void*), void *arg)

🄵穩定 ABI 的一部分. Schedule a function to be called from the main interpreter thread. On success, `0` is returned and *func* is queued for being called in the main thread. On failure, `-1` is returned without setting any exception.

When successfully queued, *func* will be *eventually* called from the main interpreter thread with the argument *arg*. It will be called asynchronously with respect to normally running Python code, but with both these conditions met:

- on a *bytecode* boundary;
- 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.

> **警告**
>
> This is a low-level function, only useful for very special cases. There is no guarantee that *func* will be called as quick as possible. If the main thread is busy executing a system call, *func* won't be called before the system call returns. This function is generally **not** suitable for calling Python code from arbitrary C threads. Instead, use the *PyGILState API*.

在 3.1 版被加入.

在 3.9 版的變更: If this function is called in a subinterpreter, the function *func* is now scheduled to be called from the subinterpreter, rather than being called from the main interpreter. Each subinterpreter now has its own list of scheduled calls.

10.8 Profiling and Tracing

The Python interpreter provides some low-level support for attaching profiling and execution tracing facilities. These are used for profiling, debugging, and coverage analysis tools.

This C interface allows the profiling or tracing code to avoid the overhead of calling through Python-level callable objects, making a direct C function call instead. The essential attributes of the facility have not changed; the interface allows trace functions to be installed per-thread, and the basic events reported to the trace function are the same as had been reported to the Python-level trace functions in previous versions.

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*:

what 的值	*arg* 的含義
`PyTrace_CALL`	Always `Py_None`.
`PyTrace_EXCEPTION`	Exception information as returned by `sys.exc_info()`.
`PyTrace_LINE`	Always `Py_None`.
`PyTrace_RETURN`	Value being returned to the caller, or `NULL` if caused by an exception.
`PyTrace_C_CALL`	被呼叫的函式物件。
`PyTrace_C_EXCEPTION`	被呼叫的函式物件。
`PyTrace_C_RETURN`	被呼叫的函式物件。
`PyTrace_OPCODE`	Always `Py_None`.

int **PyTrace_CALL**

The value of the *what* parameter to a *Py_tracefunc* function when a new call to a function or method is being reported, or a new entry into a generator. Note that the creation of the iterator for a generator function is not reported as there is no control transfer to the Python bytecode in the corresponding frame.

int **PyTrace_EXCEPTION**

The value of the *what* parameter to a *Py_tracefunc* function when an exception has been raised. The callback function is called with this value for *what* when after any bytecode is processed after which the exception becomes set within the frame being executed. The effect of this is that as exception propagation causes the Python stack to unwind, the callback is called upon return to each frame as the exception propagates. Only trace functions receives these events; they are not needed by the profiler.

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**

The value for the *what* parameter to *Py_tracefunc* functions when a call is about to return.

int **PyTrace_C_CALL**

The value for the *what* parameter to *Py_tracefunc* functions when a C function is about to be called.

int **PyTrace_C_EXCEPTION**

The value for the *what* parameter to *Py_tracefunc* functions when a C function has raised an exception.

int **PyTrace_C_RETURN**

The value for the *what* parameter to *Py_tracefunc* functions when a C function has returned.

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*.

另請參🄵 `sys.setprofile()` 函式。

呼叫者必須擁有一個*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.

呼叫者必須擁有一個*attached thread state*。

As *PyEval_SetProfile()*, this function ignores any exceptions raised while setting the profile functions in all threads.

在 3.12 版被加入.

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.

也請見 `sys.settrace()` 函式。

呼叫者必須擁有一個*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.

呼叫者必須擁有一個*attached thread state*。

As `PyEval_SetTrace()`, this function ignores any exceptions raised while setting the trace functions in all threads.

在 3.12 版被加入.

10.9 Reference tracing

在 3.13 版被加入.

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.

在 3.13 版被加入.

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.

在 3.13 版被加入.

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.

在 3.13 版被加入.

10.10 Advanced Debugger Support

These functions are only intended to be used by advanced debugging tools.

PyInterpreterState ***PyInterpreterState_Head**()

Return the interpreter state object at the head of the list of all such objects.

PyInterpreterState ***PyInterpreterState_Main**()

Return the main interpreter state object.

PyInterpreterState ***PyInterpreterState_Next**(*PyInterpreterState* *interp)

Return the next interpreter state object after *interp* from the list of all such objects.

PyThreadState ***PyInterpreterState_ThreadHead**(*PyInterpreterState* *interp)

Return the pointer to the first `PyThreadState` object in the list of threads associated with the interpreter *interp*.

PyThreadState ***PyThreadState_Next**(*PyThreadState* *tstate)

Return the next thread state object after *tstate* from the list of all such objects belonging to the same `PyInterpreterState` object.

10.11 Thread Local Storage Support

The Python interpreter provides low-level support for thread-local storage (TLS) which wraps the underlying native TLS implementation to support the Python-level thread local storage API (`threading.local`). The CPython C level APIs are similar to those offered by pthreads and Windows: use a thread key and functions to associate a `void*` value per thread.

A *thread state* does *not* need to be *attached* when calling these functions; they suppl their own locking.

Note that `Python.h` does not include the declaration of the TLS APIs, you need to include `pythread.h` to use thread-local storage.

備🄵

None of these API functions handle memory management on behalf of the `void*` values. You need to allocate and deallocate them yourself. If the `void*` values happen to be `PyObject*`, these functions don't do refcount operations on them either.

10.11.1 Thread Specific Storage (TSS) API

TSS API is introduced to supersede the use of the existing TLS API within the CPython interpreter. This API uses a new type `Py_tss_t` instead of `int` to represent thread keys.

在 3.7 版被加入.

也參考

"A New C-API for Thread-Local Storage in CPython" (**PEP 539**)

type **Py_tss_t**

This data structure represents the state of a thread key, the definition of which may depend on the underlying TLS implementation, and it has an internal field representing the key's initialization state. There are no public members in this structure.

When *Py_LIMITED_API* is not defined, static allocation of this type by `Py_tss_NEEDS_INIT` is allowed.

Py_tss_NEEDS_INIT

This macro expands to the initializer for `Py_tss_t` variables. Note that this macro won't be defined with *Py_LIMITED_API*.

Dynamic Allocation

Dynamic allocation of the `Py_tss_t`, required in extension modules built with *Py_LIMITED_API*, where static allocation of this type is not possible due to its implementation being opaque at build time.

Py_tss_t ***PyThread_tss_alloc**()

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Return a value which is the same state as a value initialized with `Py_tss_NEEDS_INIT`, or `NULL` in the case of dynamic allocation failure.

void **PyThread_tss_free**(*Py_tss_t* *key)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Free the given *key* allocated by `PyThread_tss_alloc()`, after first calling `PyThread_tss_delete()` to ensure any associated thread locals have been unassigned. This is a no-op if the *key* argument is `NULL`.

> **備🄵**
>
> A freed key becomes a dangling pointer. You should reset the key to `NULL`.

方法

The parameter *key* of these functions must not be `NULL`. Moreover, the behaviors of `PyThread_tss_set()` and `PyThread_tss_get()` are undefined if the given `Py_tss_t` has not been initialized by `PyThread_tss_create()`.

int **PyThread_tss_is_created**(*Py_tss_t* *key)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Return a non-zero value if the given `Py_tss_t` has been initialized by `PyThread_tss_create()`.

int **PyThread_tss_create**(*Py_tss_t* *key)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Return a zero value on successful initialization of a TSS key. The behavior is undefined if the value pointed to by the *key* argument is not initialized by `Py_tss_NEEDS_INIT`. This function can be called repeatedly on the same key -- calling it on an already initialized key is a no-op and immediately returns success.

void **PyThread_tss_delete**(*Py_tss_t* *key)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Destroy a TSS key to forget the values associated with the key across all threads, and change the key's initialization state to uninitialized. A destroyed key is able to be initialized again by `PyThread_tss_create()`. This function can be called repeatedly on the same key -- calling it on an already destroyed key is a no-op.

int **PyThread_tss_set**(*Py_tss_t* *key, void *value)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Return a zero value to indicate successfully associating a `void*` value with a TSS key in the current thread. Each thread has a distinct mapping of the key to a `void*` value.

void ***PyThread_tss_get**(*Py_tss_t* *key)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Return the `void*` value associated with a TSS key in the current thread. This returns `NULL` if no value is associated with the key in the current thread.

10.11.2 執行緒局部儲存 (Thread Local Storage, TLS) API:

在 3.7 版之後被🄵用: This API is superseded by *Thread Specific Storage (TSS) API*.

> **備🄵**
>
> This version of the API does not support platforms where the native TLS key is defined in a way that cannot be safely cast to `int`. On such platforms, `PyThread_create_key()` will return immediately with a failure status, and the other TLS functions will all be no-ops on such platforms.

Due to the compatibility problem noted above, this version of the API should not be used in new code.

int **PyThread_create_key**()

🄵穩定 ABI 的一部分.

void **PyThread_delete_key**(int key)

🄵穩定 ABI 的一部分.

int **PyThread_set_key_value**(int key, void *value)

🄵穩定 ABI 的一部分.

void ***PyThread_get_key_value**(int key)

🄵穩定 ABI 的一部分.

void **PyThread_delete_key_value**(int key)

🄵穩定 ABI 的一部分.

void **PyThread_ReInitTLS**()

🄵穩定 ABI 的一部分.

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.

備🄵

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.

在 3.13 版被加入.

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.

在 3.13 版被加入.

void **PyMutex_Unlock**(*PyMutex* *m)

Unlock mutex *m*. The mutex must be locked --- otherwise, the function will issue a fatal error.

在 3.13 版被加入.

int **PyMutex_IsLocked**(*PyMutex* *m)

Returns non-zero if the mutex *m* is currently locked, zero otherwise.

備🄵

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.

在 3.14 版被加入.

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.

備🄵

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.

範例用法：

```
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 `{`.

在 3.13 版被加入.

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 `{`.

在 3.14 版被加入.

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 `}`.

在 3.13 版被加入.

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 `{`.

在 3.13 版被加入.

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 `{`.

在 3.14 版被加入.

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 `}`.

在 3.13 版被加入.

CHAPTER 11

Python 初始化設定

11.1 PyInitConfig C API

在 3.14 版被加入.

Python 可以使用*Py_InitializeFromInitConfig()* 來初始化。

The *Py_RunMain()* function can be used to write a customized Python program.

See also *Initialization, Finalization, and Threads*.

也參考

PEP 741 ”Python Configuration C API”.

11.1.1 范例

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);
    return 0;
```

(繼續下一頁)

(繼續上一頁)

```
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 建立設定

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 錯誤處理

int **PyInitConfig_GetError**(*PyInitConfig* *config, const char **err_msg)

取得 *config* 錯誤訊息。

- 如果設定了錯誤，則設置 **err_msg* 🄵回傳 `1`。
- 如果🄵有設定錯誤，則設定 **err_msg* 🄵 `NULL` 🄵回傳 `0`。

錯誤訊息是 UTF-8 編碼的字串。

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.

- 成功時回傳 `0`。
- 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.

- 成功時回傳 `0`。
- 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.

- 成功時回傳 `0`。
- Set an error in *config* and return `-1` on error.

11.1.6 模組

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.

- 成功時回傳 `0`。
- 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.

和*PyImport_AppendInittab()* 函式類似。

11.1.7 初始化 Python

int **Py_InitializeFromInitConfig**(*PyInitConfig* *config)

Initialize Python from the initialization configuration.

- 成功時回傳 `0`。
- 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

選項	PyConfig/PyPreConfig 成員	Type	Visibility
"allocator"	*allocator*	int	唯讀
"argv"	*argv*	list[str]	公開
"base_exec_prefix"	*base_exec_prefix*	str	公開
"base_executable"	*base_executable*	str	公開
"base_prefix"	*base_prefix*	str	公開
"buffered_stdio"	*buffered_stdio*	bool	唯讀
"bytes_warning"	*bytes_warning*	int	公開
"check_hash_pycs_mode"	*check_hash_pycs_mode*	str	唯讀
"code_debug_ranges"	*code_debug_ranges*	bool	唯讀
"coerce_c_locale"	*coerce_c_locale*	bool	唯讀
"coerce_c_locale_warn"	*coerce_c_locale_warn*	bool	唯讀
"configure_c_stdio"	*configure_c_stdio*	bool	唯讀
"configure_locale"	*configure_locale*	bool	唯讀
"cpu_count"	*cpu_count*	int	公開
"dev_mode"	*dev_mode*	bool	唯讀
"dump_refs"	*dump_refs*	bool	唯讀
"dump_refs_file"	*dump_refs_file*	str	唯讀
"exec_prefix"	*exec_prefix*	str	公開
"executable"	*executable*	str	公開
"faulthandler"	*faulthandler*	bool	唯讀
"filesystem_encoding"	*filesystem_encoding*	str	唯讀
"filesystem_errors"	*filesystem_errors*	str	唯讀
"hash_seed"	*hash_seed*	int	唯讀
"home"	*home*	str	唯讀
"import_time"	*import_time*	int	唯讀
"inspect"	*inspect*	bool	公開

繼續下一頁

表格 1 – 繼續上一頁

選項	PyConfig/PyPreConfig 成員	Type	Visibility
"install_signal_handlers"	*install_signal_handlers*	bool	唯讀
"int_max_str_digits"	*int_max_str_digits*	int	公開
"interactive"	*interactive*	bool	公開
"isolated"	*isolated*	bool	唯讀
"legacy_windows_fs_encoding"	*legacy_windows_fs_encoding*	bool	唯讀
"legacy_windows_stdio"	*legacy_windows_stdio*	bool	唯讀
"malloc_stats"	*malloc_stats*	bool	唯讀
"module_search_paths"	*module_search_paths*	list[str]	公開
"optimization_level"	*optimization_level*	int	公開
"orig_argv"	*orig_argv*	list[str]	唯讀
"parse_argv"	*parse_argv*	bool	唯讀
"parser_debug"	*parser_debug*	bool	公開
"pathconfig_warnings"	*pathconfig_warnings*	bool	唯讀
"perf_profiling"	*perf_profiling*	bool	唯讀
"platlibdir"	*platlibdir*	str	公開
"prefix"	*prefix*	str	公開
"surrogateescape"	*program_name*	str	唯讀
"pycache_prefix"	*pycache_prefix*	str	公開
"quiet"	*quiet*	bool	公開
"run_command"	*run_command*	str	唯讀
"run_filename"	*run_filename*	str	唯讀
"run_module"	*run_module*	str	唯讀
"run_presite"	*run_presite*	str	唯讀
"safe_path"	*safe_path*	bool	唯讀
"show_ref_count"	*show_ref_count*	bool	唯讀
"site_import"	*site_import*	bool	唯讀
"skip_source_first_line"	*skip_source_first_line*	bool	唯讀
"stdio_encoding"	*stdio_encoding*	str	唯讀
"stdio_errors"	*stdio_errors*	str	唯讀
"stdlib_dir"	*stdlib_dir*	str	公開
"tracemalloc"	*tracemalloc*	int	唯讀
"use_environment"	*use_environment*	bool	公開
"use_frozen_modules"	*use_frozen_modules*	bool	唯讀
"use_hash_seed"	*use_hash_seed*	bool	唯讀
"use_system_logger"	*use_system_logger*	bool	唯讀
"user_site_directory"	*user_site_directory*	bool	唯讀
"utf8_mode"	*utf8_mode*	bool	唯讀
"verbose"	*verbose*	int	公開
"warn_default_encoding"	*warn_default_encoding*	bool	唯讀
"warnoptions"	*warnoptions*	list[str]	公開
"write_bytecode"	*write_bytecode*	bool	公開
"xoptions"	*xoptions*	dict[str, str]	公開
"_pystats"	*_pystats*	bool	唯讀

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.

在 3.14 版被加入.

int **PyConfig_GetInt**(const char *name, int *value)

Similar to *`PyConfig_Get()`*, but get the value as a C int.

- 成功時回傳 `0`。
- Set an exception and return `-1` on error.

在 3.14 版被加入.

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.

在 3.14 版被加入.

int **PyConfig_Set**(const char *name, *PyObject* *value)

Set the current runtime value of a configuration option.

- 如果🄵有選項 *name*，則引發 `ValueError`。
- 如果 *value* 是無效值，則引發 `ValueError`。
- 如果選項是唯讀的（無法設定），則引發 `ValueError`。
- 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`.

在 3.14 版被加入.

11.4 PyConfig C API

在 3.8 版被加入.

Python can be initialized with `Py_InitializeFromConfig()` and the `PyConfig` structure. It can be preinitialized with `Py_PreInitialize()` and the `PyPreConfig` structure.

There are two kinds of configuration:

- 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.

The `Py_RunMain()` function can be used to write a customized Python program.

See also *Initialization, Finalization, and Threads*.

也參考

PEP 587 "Python Initialization Configuration".

11.4.1 范例

Example of customized Python always running in isolated mode:

```
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**

`wchar_t*` 字串串列。

If *length* is non-zero, *items* must be non-`NULL` and all strings must be non-`NULL`.

方法：

PyStatus **PyWideStringList_Append**(*PyWideStringList* *list, const wchar_t *item)

Append *item* to *list*.

Python must be preinitialized to call this function.

PyStatus **PyWideStringList_Insert**(*PyWideStringList* *list, *Py_ssize_t* index, const wchar_t *item)

Insert *item* into *list* at *index*.

If *index* is greater than or equal to *list* length, append *item* to *list*.

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

Python must be preinitialized to call this function.

Structure fields:

Py_ssize_t **length**

串列長度。

wchar_t ****items**

List items.

11.4.3 PyStatus

type **PyStatus**

Structure to store an initialization function status: success, error or exit.

For an error, it can store the C function name which created the error.

Structure fields:

int **exitcode**

Exit code. Argument passed to `exit()`.

const char ***err_msg**

錯誤訊息。

const char ***func**

Name of the function which created an error, can be `NULL`.

Functions to create a status:

PyStatus **PyStatus_Ok**(void)

成功。

PyStatus **PyStatus_Error**(const char *err_msg)

附帶訊息的初始化錯誤。

err_msg 不可🄵 `NULL`。

PyStatus **PyStatus_NoMemory**(void)

Memory allocation failure (out of memory).

PyStatus **PyStatus_Exit**(int exitcode)

Exit Python with the specified exit code.

Functions to handle a status:

int **PyStatus_Exception**(*PyStatus* status)

Is the status an error or an exit? If true, the exception must be handled; by calling *Py_ExitStatusException()* for example.

int **PyStatus_IsError**(*PyStatus* status)

Is the result an error?

int **PyStatus_IsExit**(*PyStatus* status)

Is the result an exit?

void **Py_ExitStatusException**(*PyStatus* status)

Call `exit(exitcode)` if *status* is an exit. Print the error message and exit with a non-zero exit code if *status* is an error. Must only be called if `PyStatus_Exception(status)` is non-zero.

備🄵

Internally, Python uses macros which set `PyStatus.func`, whereas functions to create a status set `func` to `NULL`.

範例:

```
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**

Structure used to preinitialize Python.

Function to initialize a preconfiguration:

void **PyPreConfig_InitPythonConfig**(*PyPreConfig* *preconfig)

Initialize the preconfiguration with *Python Configuration*.

void **PyPreConfig_InitIsolatedConfig**(*PyPreConfig* *preconfig)

Initialize the preconfiguration with *Isolated Configuration*.

Structure fields:

int **allocator**

Name of the Python memory allocators:

- `PYMEM_ALLOCATOR_NOT_SET` (`0`): don't change memory allocators (use defaults).
- `PYMEM_ALLOCATOR_DEFAULT` (`1`): *default memory allocators*.

- `PYMEM_ALLOCATOR_DEBUG` (`2`): *default memory allocators* with *debug hooks*.
- `PYMEM_ALLOCATOR_MALLOC` (`3`): use `malloc()` of the C library.
- `PYMEM_ALLOCATOR_MALLOC_DEBUG` (`4`): force usage of `malloc()` with *debug hooks*.
- `PYMEM_ALLOCATOR_PYMALLOC` (`5`): *Python pymalloc memory allocator*.
- `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` (`6`): *Python pymalloc memory allocator* with *debug hooks*.
- `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` and `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` are not supported if Python is `configured using --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.

請見記憶體管理。

預設：`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`.

請見*locale encoding*。

Default: `1` in Python config, `0` in isolated config.

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.

請見*locale encoding*。

Default: `-1` in Python config, `0` in isolated config.

int **coerce_c_locale_warn**

If non-zero, emit a warning if the C locale is coerced.

Default: `-1` in Python config, `0` in isolated config.

int **dev_mode**

Python Development Mode: see *PyConfig.dev_mode*.

Default: `-1` in Python mode, `0` in isolated mode.

int **isolated**

Isolated mode: see *PyConfig.isolated*.

Default: `0` in Python mode, `1` in isolated mode.

int **legacy_windows_fs_encoding**

如果不🄵 0：

- 將*PyPreConfig.utf8_mode* 設🄵 `0`、
- 將*PyConfig.filesystem_encoding* 設🄵 `"mbcs"`、
- 將*PyConfig.filesystem_errors* 設🄵 `"replace"`。

Initialized from the `PYTHONLEGACYWINDOWSFSENCODING` environment variable value.

Only available on Windows. `#ifdef MS_WINDOWS` macro can be used for Windows specific code.

預設：`0`。

int **parse_argv**

If non-zero, *`Py_PreInitializeFromArgs()`* and *`Py_PreInitializeFromBytesArgs()`* parse their `argv` argument the same way the regular Python parses command line arguments: see Command Line Arguments.

Default: `1` in Python config, `0` in isolated config.

int **use_environment**

Use environment variables? See *`PyConfig.use_environment`*.

Default: `1` in Python config and `0` in isolated config.

int **utf8_mode**

If non-zero, enable the Python UTF-8 Mode.

Set to `0` or `1` by the `-X utf8` command line option and the `PYTHONUTF8` environment variable.

Also set to `1` if the `LC_CTYPE` locale is `C` or `POSIX`.

Default: `-1` in Python config and `0` in isolated config.

11.4.5 Preinitialize Python with PyPreConfig

The preinitialization of Python:

- Set the Python memory allocators (*`PyPreConfig.allocator`*)
- Configure the LC_CTYPE locale (*locale encoding*)
- Set the Python UTF-8 Mode (*`PyPreConfig.utf8_mode`*)

The current preconfiguration (`PyPreConfig` type) is stored in `_PyRuntime.preconfig`.

Functions to preinitialize Python:

PyStatus **Py_PreInitialize**(const *PyPreConfig* *preconfig)

Preinitialize Python from *preconfig* preconfiguration.

preconfig 不可🄵 `NULL`。

PyStatus **Py_PreInitializeFromBytesArgs**(const *PyPreConfig* *preconfig, int argc, char *const *argv)

Preinitialize Python from *preconfig* preconfiguration.

Parse *argv* command line arguments (bytes strings) if *`parse_argv`* of *preconfig* is non-zero.

preconfig 不可🄵 `NULL`。

PyStatus **Py_PreInitializeFromArgs**(const *PyPreConfig* *preconfig, int argc, wchar_t *const *argv)

Preinitialize Python from *preconfig* preconfiguration.

Parse *argv* command line arguments (wide strings) if *`parse_argv`* of *preconfig* is non-zero.

preconfig 不可🄵 `NULL`。

The caller is responsible to handle exceptions (error or exit) using *`PyStatus_Exception()`* and *`Py_ExitStatusException()`*.

For *Python Configuration* (*`PyPreConfig_InitPythonConfig()`*), if Python is initialized with command line arguments, the command line arguments must also be passed to preinitialize Python, since they have an effect on the pre-configuration like encodings. For example, the `-X utf8` command line option enables the Python UTF-8 Mode.

`PyMem_SetAllocator()` can be called after *`Py_PreInitialize()`* and before *`Py_InitializeFromConfig()`* to install a custom memory allocator. It can be called before *`Py_PreInitialize()`* if *`PyPreConfig.allocator`* is set to `PYMEM_ALLOCATOR_NOT_SET`.

Python memory allocation functions like *PyMem_RawMalloc()* must not be used before the Python preinitialization, whereas calling directly `malloc()` and `free()` is always safe. *Py_DecodeLocale()* must not be called before the Python preinitialization.

Example using the preinitialization to enable the 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**

Structure containing most parameters to configure Python.

When done, the *PyConfig_Clear()* function must be used to release the configuration memory.

Structure methods:

void **PyConfig_InitPythonConfig**(*PyConfig* *config)

Initialize configuration with the *Python Configuration*.

void **PyConfig_InitIsolatedConfig**(*PyConfig* *config)

Initialize configuration with the *Isolated Configuration*.

PyStatus **PyConfig_SetString**(*PyConfig* *config, wchar_t *const *config_str, const wchar_t *str)

Copy the wide character string *str* into `*config_str`.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesString**(*PyConfig* *config, wchar_t *const *config_str, const char *str)

Decode *str* using *Py_DecodeLocale()* and set the result into `*config_str`.

Preinitialize Python if needed.

PyStatus **PyConfig_SetArgv**(*PyConfig* *config, int argc, wchar_t *const *argv)

Set command line arguments (*argv* member of *config*) from the *argv* list of wide character strings.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesArgv**(*PyConfig* *config, int argc, char *const *argv)

Set command line arguments (*argv* member of *config*) from the *argv* list of bytes strings. Decode bytes using *Py_DecodeLocale()*.

Preinitialize Python if needed.

PyStatus **PyConfig_SetWideStringList**(*PyConfig* *config, *PyWideStringList* *list, *Py_ssize_t* length, wchar_t **items)

Set the list of wide strings *list* to *length* and *items*.

Preinitialize Python if needed.

PyStatus **PyConfig_Read**(*PyConfig* *config)

Read all Python configuration.

Fields which are already initialized are left unchanged.

Fields for *path configuration* are no longer calculated or modified when calling this function, as of 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.

Preinitialize Python if needed.

在 3.10 版的變更: The *PyConfig.argv* arguments are now only parsed once, *PyConfig.parse_argv* is set to `2` after arguments are parsed, and arguments are only parsed if *PyConfig.parse_argv* equals `1`.

在 3.11 版的變更: *PyConfig_Read()* no longer calculates all paths, and so fields listed under *Python Path Configuration* may no longer be updated until *Py_InitializeFromConfig()* is called.

void **PyConfig_Clear**(*PyConfig* *config)

Release configuration memory.

Most `PyConfig` methods *preinitialize Python* if needed. In that case, the Python preinitialization configuration (*PyPreConfig*) in based on the *PyConfig*. If configuration fields which are in common with *PyPreConfig* are tuned, they must be set before calling a *PyConfig* method:

- *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).

The caller of these methods is responsible to handle exceptions (error or exit) using `PyStatus_Exception()` and `Py_ExitStatusException()`.

Structure fields:

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.

Set *parse_argv* to `1` to parse *argv* the same way the regular Python parses Python command line arguments and then to strip Python arguments from *argv*.

If *argv* is empty, an empty string is added to ensure that `sys.argv` always exists and is never empty.

預設值：`NULL`。

另請參🄵*orig_argv* 成員。

int **safe_path**

If equals to zero, `Py_RunMain()` prepends a potentially unsafe path to `sys.path` at startup:

- If *argv[0]* is equal to `L"-m"` (`python -m module`), prepend the current working directory.
- If running a script (`python script.py`), prepend the script's directory. If it's a symbolic link, resolve symbolic links.

- Otherwise (`python -c code` and `python`), prepend an empty string, which means the current working directory.

Set to `1` by the `-P` command line option and the `PYTHONSAFEPATH` environment variable.

Default: `0` in Python config, `1` in isolated config.

在 3.11 版被加入.

wchar_t ***base_exec_prefix**

`sys.base_exec_prefix`。

預設值：`NULL`。

Part of the *Python Path Configuration* output.

也請見*PyConfig.exec_prefix*

wchar_t ***base_executable**

Python base executable: `sys._base_executable`.

Set by the `__PYVENV_LAUNCHER__` environment variable.

Set from *PyConfig.executable* if `NULL`.

預設值：`NULL`。

Part of the *Python Path Configuration* output.

也請見*PyConfig.executable*

wchar_t ***base_prefix**

`sys.base_prefix`。

預設值：`NULL`。

Part of the *Python Path Configuration* output.

也請見*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 is always opened in buffered mode.

預設值：`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.

Incremented by the `-b` command line option.

預設：`0`。

int **warn_default_encoding**

If non-zero, emit a `EncodingWarning` warning when `io.TextIOWrapper` uses its default encoding. See io-encoding-warning for details.

預設：`0`。

在 3.10 版被加入.

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.

預設值：`1`。

在 3.11 版被加入.

wchar_t ***check_hash_pycs_mode**

Control the validation behavior of hash-based `.pyc` files: value of the `--check-hash-based-pycs` command line option.

Valid values:

- `L"always"`: Hash the source file for invalidation regardless of value of the 'check_source' flag.
- `L"never"`: Assume that hash-based pycs always are valid.
- `L"default"`: The 'check_source' flag in hash-based pycs determines invalidation.

預設：`L"default"`。

See also **PEP 552** "Deterministic pycs".

int **configure_c_stdio**

If non-zero, configure C standard streams:

- On Windows, set the binary mode (`O_BINARY`) on stdin, stdout and stderr.
- If *`buffered_stdio`* equals zero, disable buffering of stdin, stdout and stderr streams.
- If *`interactive`* is non-zero, enable stream buffering on stdin and stdout (only stdout on Windows).

Default: `1` in Python config, `0` in isolated config.

int **dev_mode**

If non-zero, enable the Python Development Mode.

Set to `1` by the `-X dev` option and the `PYTHONDEVMODE` environment variable.

Default: `-1` in Python mode, `0` in isolated mode.

int **dump_refs**

Dump Python references?

If non-zero, dump all objects which are still alive at exit.

Set to `1` by the `PYTHONDUMPREFS` environment variable.

Needs a special build of Python with the `Py_TRACE_REFS` macro defined: see the `configure --with-trace-refs option`.

預設：`0`。

wchar_t ***dump_refs_file**

Filename where to dump Python references.

由 `PYTHONDUMPREFSFILE` 環境變數設定。

預設值：`NULL`。

在 3.11 版被加入.

wchar_t ***exec_prefix**

The site-specific directory prefix where the platform-dependent Python files are installed: `sys.exec_prefix`.

預設值：`NULL`。

Part of the *Python Path Configuration* output.

也請見*`PyConfig.base_exec_prefix`*

wchar_t ***executable**

The absolute path of the executable binary for the Python interpreter: `sys.executable`.

預設值：`NULL`。

Part of the *Python Path Configuration* output.

也請見*`PyConfig.base_executable`*

int **faulthandler**

Enable faulthandler?

If non-zero, call `faulthandler.enable()` at startup.

Set to `1` by `-X faulthandler` and the `PYTHONFAULTHANDLER` environment variable.

Default: `-1` in Python mode, `0` in isolated mode.

wchar_t ***filesystem_encoding**

Filesystem encoding: `sys.getfilesystemencoding()`.

On macOS, Android and VxWorks: use `"utf-8"` by default.

On Windows: use `"utf-8"` by default, or `"mbcs"` if *`legacy_windows_fs_encoding`* of *`PyPreConfig`* is non-zero.

Default encoding on other platforms:

- `"utf-8"` if *`PyPreConfig.utf8_mode`* is non-zero.
- `"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"` if `nl_langinfo(CODESET)` returns an empty string.
- Otherwise, use the *locale encoding*: `nl_langinfo(CODESET)` result.

At Python startup, the encoding name is normalized to the Python codec name. For example, `"ANSI_X3.4-1968"` is replaced with `"ascii"`.

另請參🄵*`filesystem_errors`* 成員。

wchar_t ***filesystem_errors**

Filesystem error handler: `sys.getfilesystemencodeerrors()`.

On Windows: use `"surrogatepass"` by default, or `"replace"` if *`legacy_windows_fs_encoding`* of *`PyPreConfig`* is non-zero.

On other platforms: use `"surrogateescape"` by default.

Supported error handlers:

- `"strict"`
- `"surrogateescape"`
- `"surrogatepass"`（僅支援 UTF-8 編碼）

另請參🄵*`filesystem_encoding`* 成員。

int **use_frozen_modules**

If non-zero, use frozen modules.

由 PYTHON_FROZEN_MODULES 環境變數設定。

Default: 1 in a release build, or 0 in a debug build.

unsigned long **hash_seed**

int **use_hash_seed**

Randomized hash function seed.

If *use_hash_seed* is zero, a seed is chosen randomly at Python startup, and *hash_seed* is ignored.

由 PYTHONHASHSEED 環境變數設定。

Default *use_hash_seed* value: -1 in Python mode, 0 in isolated mode.

wchar_t ***home**

Set the default Python "home" directory, that is, the location of the standard Python libraries (see PYTHONHOME).

由 PYTHONHOME 環境變數設定。

預設值：NULL。

Part of the *Python Path Configuration* input.

int **import_time**

If 1, profile import time. If 2, include additional output that indicates when an imported module has already been loaded.

由 -X importtime 選項與 PYTHONPROFILEIMPORTTIME 環境變數設定。

預設：0。

在 3.14 版的變更: Added support for import_time = 2

int **inspect**

Enter interactive mode after executing a script or a command.

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.

Incremented by the -i command line option. Set to 1 if the PYTHONINSPECT environment variable is non-empty.

預設：0。

int **install_signal_handlers**

Install Python signal handlers?

Default: 1 in Python mode, 0 in isolated mode.

int **interactive**

If greater than 0, enable the interactive mode (REPL).

Incremented by the -i command line option.

預設：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.

在 3.12 版被加入.

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.

預設值：`1`。

在 3.13 版被加入.

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.
- 將*use_environment* 設定🄵 `0`：忽略 `PYTHON` 環境變數。
- Set *user_site_directory* to `0`: don't add the user site directory to `sys.path`.
- Python REPL doesn't import `readline` nor enable default readline configuration on interactive prompts.

Set to `1` by the `-I` command line option.

Default: `0` in Python mode, `1` in isolated mode.

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`.

Set to `1` if the `PYTHONLEGACYWINDOWSSTDIO` environment variable is set to a non-empty string.

Only available on Windows. `#ifdef MS_WINDOWS` macro can be used for Windows specific code.

預設：`0`。

See also the **PEP 528** (Change Windows console encoding to UTF-8).

int **malloc_stats**

If non-zero, dump statistics on *Python pymalloc memory allocator* at exit.

Set to `1` by the `PYTHONMALLOCSTATS` environment variable.

The option is ignored if Python is `configured using the --without-pymalloc option`.

預設：`0`。

wchar_t ***platlibdir**

Platform library directory name: `sys.platlibdir`.

Set by the `PYTHONPLATLIBDIR` environment variable.

Default: value of the `PLATLIBDIR` macro which is set by the `configure --with-platlibdir option` (default: `"lib"`, or `"DLLs"` on Windows).

Part of the *Python Path Configuration* input.

在 3.9 版被加入.

在 3.11 版的變更: This macro is now used on Windows to locate the standard library extension modules, typically under `DLLs`. However, for compatibility, note that this value is ignored for any non-standard layouts, including in-tree builds and virtual environments.

wchar_t ***pythonpath_env**

Module search paths (`sys.path`) as a string separated by `DELIM` (`os.pathsep`).

Set by the `PYTHONPATH` environment variable.

預設值：`NULL`。

Part of the *Python Path Configuration* input.

PyWideStringList **module_search_paths**

int **module_search_paths_set**

Module search paths: `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`.

Default: empty list (`module_search_paths`) and `0` (`module_search_paths_set`).

Part of the *Python Path Configuration* output.

int **optimization_level**

Compilation optimization level:

- `0`: Peephole optimizer, set `__debug__` to `True`.
- `1`: Level 0, remove assertions, set `__debug__` to `False`.
- `2`: Level 1, strip docstrings.

Incremented by the `-O` command line option. Set to the `PYTHONOPTIMIZE` environment variable value.

預設：`0`。

PyWideStringList **orig_argv**

The list of the original command line arguments passed to the Python executable: `sys.orig_argv`.

If *orig_argv* list is empty and *argv* is not a list only containing an empty string, *PyConfig_Read()* copies *argv* into *orig_argv* before modifying *argv* (if *parse_argv* is non-zero).

See also the *argv* member and the *Py_GetArgcArgv()* function.

Default: empty list.

在 3.10 版被加入.

int **parse_argv**

Parse command line arguments?

If equals to `1`, parse *argv* the same way the regular Python parses command line arguments, and strip Python arguments from *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.

Default: `1` in Python mode, `0` in isolated mode.

在 3.10 版的變更: The *PyConfig.argv* arguments are now only parsed if *PyConfig.parse_argv* equals to `1`.

int **parser_debug**

Parser debug mode. If greater than `0`, turn on parser debugging output (for expert only, depending on compilation options).

Incremented by the `-d` command line option. Set to the `PYTHONDEBUG` environment variable value.

Needs a debug build of Python (the `Py_DEBUG` macro must be defined).

預設：`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.

Default: `1` in Python mode, `0` in isolated mode.

Part of the *Python Path Configuration* input.

在 3.11 版的變更: Now also applies on Windows.

wchar_t ***prefix**

The site-specific directory prefix where the platform independent Python files are installed: `sys.prefix`.

預設值：`NULL`。

Part of the *Python Path Configuration* output.

也請見*PyConfig.base_prefix*

wchar_t ***program_name**

Program name used to initialize *executable* and in early error messages during Python initialization.

- On macOS, use `PYTHONEXECUTABLE` environment variable if set.
- If the `WITH_NEXT_FRAMEWORK` macro is defined, use `__PYVENV_LAUNCHER__` environment variable if set.
- Use `argv[0]` of *argv* if available and non-empty.
- Otherwise, use `L"python"` on Windows, or `L"python3"` on other platforms.

預設值：`NULL`。

Part of the *Python Path Configuration* input.

wchar_t ***pycache_prefix**

Directory where cached `.pyc` files are written: `sys.pycache_prefix`.

Set by the `-X pycache_prefix=PATH` command line option and the `PYTHONPYCACHEPREFIX` environment variable. The command-line option takes precedence.

If `NULL`, `sys.pycache_prefix` is set to `None`.

預設值：`NULL`。

int **quiet**

Quiet mode. If greater than `0`, don't display the copyright and version at Python startup in interactive mode.

Incremented by the `-q` command line option.

預設：`0`。

wchar_t ***run_command**

Value of the `-c` command line option.

Used by *Py_RunMain()*.

預設值：`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.

也請見*PyConfig.skip_source_first_line* 選項。

預設值：NULL。

wchar_t ***run_module**

Value of the `-m` command line option.

Used by *Py_RunMain()*.

預設值：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).

預設值：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).

預設：0。

int **site_import**

Import the `site` module at startup?

If equal to zero, disable the import of the module site and the site-dependent manipulations of `sys.path` that it entails.

Also disable these manipulations if the `site` module is explicitly imported later (call `site.main()` if you want them to be triggered).

Set to `0` by the `-S` command line option.

`sys.flags.no_site` is set to the inverted value of *site_import*.

預設值：1。

int **skip_source_first_line**

If non-zero, skip the first line of the *PyConfig.run_filename* source.

It allows the usage of non-Unix forms of `#!cmd`. This is intended for a DOS specific hack only.

Set to `1` by the `-x` command line option.

預設：0。

wchar_t ***stdio_encoding**

wchar_t ***stdio_errors**

Encoding and encoding errors of `sys.stdin`, `sys.stdout` and `sys.stderr` (but `sys.stderr` always uses `"backslashreplace"` error handler).

Use the `PYTHONIOENCODING` environment variable if it is non-empty.

Default encoding:

- `"UTF-8"` if *PyPreConfig.utf8_mode* is non-zero.
- Otherwise, use the *locale encoding*.

Default error handler:

- On Windows: use `"surrogateescape"`.

- `"surrogateescape"` if *PyPreConfig.utf8_mode* is non-zero, or if the LC_CTYPE locale is "C" or "POSIX".
- `"strict"` otherwise.

也請見*PyConfig.legacy_windows_stdio*。

int **tracemalloc**

Enable tracemalloc?

If non-zero, call `tracemalloc.start()` at startup.

Set by `-X tracemalloc=N` command line option and by the `PYTHONTRACEMALLOC` environment variable.

Default: `-1` in Python mode, `0` in isolated mode.

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.

預設值：`1`。

也參考

See perf_profiling for more information.

在 3.12 版被加入.

wchar_t ***stdlib_dir**

Directory of the Python standard library.

預設值：`NULL`。

在 3.11 版被加入.

int **use_environment**

Use environment variables?

If equals to zero, ignore the environment variables.

Set to `0` by the `-E` environment variable.

Default: `1` in Python config and `0` in isolated config.

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).

在 3.14 版被加入.

int **user_site_directory**

If non-zero, add the user site directory to `sys.path`.

Set to `0` by the `-s` and `-I` command line options.

Set to `0` by the `PYTHONNOUSERSITE` environment variable.

Default: `1` in Python mode, `0` in isolated mode.

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.

Incremented by the `-v` command line option.

Set by the `PYTHONVERBOSE` environment variable value.

預設：`0`。

PyWideStringList **warnoptions**

Options of the `warnings` module to build warnings filters, lowest to highest priority: `sys.warnoptions`.

The `warnings` module adds `sys.warnoptions` in the reverse order: the last *PyConfig.warnoptions* item becomes the first item of `warnings.filters` which is checked first (highest priority).

The `-W` command line options adds its value to *warnoptions*, it can be used multiple times.

The `PYTHONWARNINGS` environment variable can also be used to add warning options. Multiple options can be specified, separated by commas (`,`).

Default: empty list.

int **write_bytecode**

If equal to `0`, Python won't try to write `.pyc` files on the import of source modules.

Set to `0` by the `-B` command line option and the `PYTHONDONTWRITEBYTECODE` environment variable.

`sys.dont_write_bytecode` is initialized to the inverted value of *write_bytecode*.

預設值：`1`。

PyWideStringList **xoptions**

Values of the `-X` command line options: `sys._xoptions`.

Default: empty list.

int **_pystats**

If non-zero, write performance statistics at Python exit.

Need a special build with the `Py_STATS` macro: see `--enable-pystats`.

預設：`0`。

If *parse_argv* is non-zero, *argv* arguments are parsed the same way the regular Python parses command line arguments, and Python arguments are stripped from *argv*.

The *xoptions* options are parsed to set other options: see the `-X` command line option.

在 3.9 版的變更: The `show_alloc_count` field has been removed.

11.4.7 Initialization with PyConfig

Initializing the interpreter from a populated configuration struct is handled by calling *Py_InitializeFromConfig()*.

The caller is responsible to handle exceptions (error or exit) using *PyStatus_Exception()* and *Py_ExitStatusException()*.

If *PyImport_FrozenModules()*, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* are used, they must be set or called after Python preinitialization and before the Python initialization. If Python is initialized multiple times, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* must be called before each Python initialization.

The current configuration (`PyConfig` type) is stored in `PyInterpreterState.config`.

Example setting the program name:

```
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 */
```

(繼續下一頁)

(繼續上一頁)

```
    /* 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 Isolated Configuration

PyPreConfig_InitIsolatedConfig() and *PyConfig_InitIsolatedConfig()* functions create a configuration to isolate Python from the system. For example, to embed Python into an application.

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.

Configuration files are still used with this configuration to determine paths that are unspecified. Ensure *PyConfig.home* is specified to avoid computing the default path configuration.

11.4.9 Python Configuration

PyPreConfig_InitPythonConfig() and *PyConfig_InitPythonConfig()* functions create a configuration to build a customized Python which behaves as the regular Python.

Environments variables and command line arguments are used to configure Python, whereas global configuration variables are ignored.

This function enables C locale coercion (**PEP 538**) and Python UTF-8 Mode (**PEP 540**) depending on the LC_CTYPE locale, `PYTHONUTF8` and `PYTHONCOERCECLOCALE` environment variables.

11.4.10 Python Path Configuration

PyConfig contains multiple fields for the path configuration:

- Path configuration inputs:
  - *PyConfig.home*
  - *PyConfig.platlibdir*
  - *PyConfig.pathconfig_warnings*

  - *PyConfig.program_name*
  - *PyConfig.pythonpath_env*
  - current working directory: to get absolute paths
  - `PATH` environment variable to get the program full path (from *PyConfig.program_name*)
  - `__PYVENV_LAUNCHER__` 環境變數
  - (Windows only) Application paths in the registry under "SoftwarePythonPythonCoreX.YPythonPath" of HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE (where X.Y is the Python version).
- Path configuration output fields:
  - *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*

If at least one "output field" is not set, Python calculates the path configuration to fill unset fields. If *module_search_paths_set* is equal to `0`, *module_search_paths* is overridden and *module_search_paths_set* is set to `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).

If *base_prefix* or *base_exec_prefix* fields are not set, they inherit their value from *prefix* and *exec_prefix* respectively.

Py_RunMain() and *Py_Main()* modify `sys.path`:

- If *run_filename* is set and is a directory which contains a `__main__.py` script, prepend *run_filename* to `sys.path`.
- 如果*isolated* 🄵零:
  - If *run_module* is set, prepend the current directory to `sys.path`. Do nothing if the current directory cannot be read.
  - If *run_filename* is set, prepend the directory of the filename to `sys.path`.
  - Otherwise, prepend an empty string to `sys.path`.

If *site_import* is non-zero, `sys.path` can be modified by the `site` module. If *user_site_directory* is non-zero and the user's site-package directory exists, the `site` module appends the user's site-package directory to `sys.path`.

The following configuration files are used by the path configuration:

- `pyvenv.cfg`
- `._pth` file (ex: `python._pth`)
- `pybuilddir.txt`（僅限 Unix）

If a `._pth` file is present:

- 將*isolated* 設定🄵 `1`。

- 將*use_environment* 設定🄵 0。
- 將*site_import* 設定🄵 0。
- 將*safe_path* 設定🄵 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*.

在 3.14 版的變更: *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)

Get the original command line arguments, before Python modified them.

也請參與*PyConfig.orig_argv* 成員。

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.

CHAPTER 12

記憶體管理

12.1 總覽

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the *Python memory manager*. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

To avoid memory corruption, extension writers should never try to operate on Python objects with the functions exported by the C library: `malloc()`, `calloc()`, `realloc()` and `free()`. This will result in mixed calls between the C allocator and the Python memory manager with fatal consequences, because they implement different algorithms and operate on different heaps. However, one may safely allocate and release memory blocks with the C library allocator for individual purposes, as shown in the following example:

```
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);
free(buf); /* malloc'ed */
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the bytes object returned as a result.

In most situations, however, it is recommended to allocate memory from the Python heap specifically because the latter is under control of the Python memory manager. For example, this is required when the interpreter is extended with new object types written in C. Another reason for using the Python heap is the desire to *inform* the Python memory manager about the memory needs of the extension module. Even when the requested memory is used exclusively for internal, highly specific purposes, delegating all memory requests to the Python memory manager causes the interpreter to have a more accurate image of its memory footprint as a whole. Consequently, under certain circumstances, the Python memory manager may or may not trigger appropriate actions, like garbage collection, memory compaction or other preventive procedures. Note that by using the C library allocator as shown in the previous example, the allocated memory for the I/O buffer escapes completely the Python memory manager.

也參考

The `PYTHONMALLOC` environment variable can be used to configure the memory allocators used by Python.

The `PYTHONMALLOCSTATS` environment variable can be used to print statistics of the *pymalloc memory allocator* every time a new pymalloc object arena is created, and on shutdown.

12.2 Allocator Domains

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()`.

The three allocation domains are:

- 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*.

備🄵

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 Raw Memory Interface

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.

在 3.4 版被加入.

void *__PyMem_RawMalloc__(size_t n)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawMalloc(1)` had been called instead. The memory will not have been initialized in any way.

void *__PyMem_RawCalloc__(size_t nelem, size_t elsize)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawCalloc(1, 1)` had been called instead.

在 3.5 版被加入.

void *__PyMem_RawRealloc__(void *p, size_t n)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyMem_RawMalloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to *PyMem_RawMalloc()*, *PyMem_RawRealloc()* or *PyMem_RawCalloc()*.

If the request fails, *PyMem_RawRealloc()* returns `NULL` and *p* remains a valid pointer to the previous memory area.

void __PyMem_RawFree__(void *p)

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. Frees the memory block pointed to by *p*, which must have been returned by a previous call to *PyMem_RawMalloc()*, *PyMem_RawRealloc()* or *PyMem_RawCalloc()*. Otherwise, or if `PyMem_RawFree(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

12.4 記憶體介面

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

The *default memory allocator* uses the *pymalloc memory allocator*.

> **警告**
>
> There must be an *attached thread state* when using these functions.

在 3.6 版的變更: The default allocator is now pymalloc instead of system `malloc()`.

void *__PyMem_Malloc__(size_t n)

🄵穩定 ABI 的一部分. Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void *__PyMem_Calloc__(size_t nelem, size_t elsize)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-NULL pointer if possible, as if PyMem_Calloc(1, 1) had been called instead.

在 3.5 版被加入.

void ***PyMem_Realloc**(void *p, size_t n)

🄵穩定 ABI 的一部分. Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is NULL, the call is equivalent to PyMem_Malloc(n); else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-NULL.

Unless *p* is NULL, it must have been returned by a previous call to *PyMem_Malloc()*, *PyMem_Realloc()* or *PyMem_Calloc()*.

If the request fails, *PyMem_Realloc()* returns NULL and *p* remains a valid pointer to the previous memory area.

void **PyMem_Free**(void *p)

🄵穩定 ABI 的一部分. Frees the memory block pointed to by *p*, which must have been returned by a previous call to *PyMem_Malloc()*, *PyMem_Realloc()* or *PyMem_Calloc()*. Otherwise, or if PyMem_Free(p) has been called before, undefined behavior occurs.

If *p* is NULL, no operation is performed.

The following type-oriented macros are provided for convenience. Note that *TYPE* refers to any C type.

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.

This is a C preprocessor macro; *p* is always reassigned. Save the original value of *p* to avoid losing memory when handling errors.

void **PyMem_Del**(void *p)

和*PyMem_Free()* 相同。

In addition, the following macro sets are provided for calling the Python memory allocator directly, without involving the C API functions listed above. However, note that their use does not preserve binary compatibility across Python versions and is therefore deprecated in extension modules.

- PyMem_MALLOC(size)
- PyMem_NEW(type, size)
- PyMem_REALLOC(ptr, size)
- PyMem_RESIZE(ptr, type, size)
- PyMem_FREE(ptr)
- PyMem_DEL(ptr)

12.5 Object allocators

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

備🄵

There is no guarantee that the memory returned by these allocators can be successfully cast to a Python object when intercepting the allocating functions in this domain by the methods described in the *Customize Memory Allocators* section.

The *default object allocator* uses the *pymalloc memory allocator*.

警告

There must be an *attached thread state* when using these functions.

void ***PyObject_Malloc**(size_t n)

🄵穩定 ABI 的一部分. Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void ***PyObject_Calloc**(size_t nelem, size_t elsize)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Calloc(1, 1)` had been called instead.

在 3.5 版被加入.

void ***PyObject_Realloc**(void *p, size_t n)

🄵穩定 ABI 的一部分. Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyObject_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to *`PyObject_Malloc()`*, *`PyObject_Realloc()`* or *`PyObject_Calloc()`*.

If the request fails, *`PyObject_Realloc()`* returns `NULL` and *p* remains a valid pointer to the previous memory area.

void **PyObject_Free**(void *p)

🄵穩定 ABI 的一部分. Frees the memory block pointed to by *p*, which must have been returned by a previous call to *`PyObject_Malloc()`*, *`PyObject_Realloc()`* or *`PyObject_Calloc()`*. Otherwise, or if `PyObject_Free(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

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.

也參考

- *`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 Default Memory Allocators

Default memory allocators:

配置	名徵	PyMem_RawMallo	PyMem_Malloc	PyOb-ject_Malloc
Release build	`"pymalloc"`	`malloc`	`pymalloc`	`pymalloc`
Debug build	`"pymalloc_debug`	`malloc` + debug	`pymalloc` + de-bug	`pymalloc` + de-bug
Release build, without py-malloc	`"malloc"`	`malloc`	`malloc`	`malloc`
Debug build, without py-malloc	`"malloc_debug"`	`malloc` + debug	`malloc` + debug	`malloc` + debug

Legend:

- Name: value for `PYTHONMALLOC` environment variable.
- `malloc`: system allocators from the standard C library, C functions: `malloc()`, `calloc()`, `realloc()` and `free()`.
- `pymalloc`: *pymalloc memory allocator*.
- `mimalloc`: *mimalloc memory allocator*. The pymalloc allocator will be used if mimalloc support isn't available.
- "+ debug": with *debug hooks on the Python memory allocators*.
- "Debug build": Python build in debug mode.

12.7 Customize Memory Allocators

在 3.4 版被加入.

type **PyMemAllocatorEx**

Structure used to describe a memory block allocator. The structure has the following fields:

欄位	意義
`void *ctx`	user context passed as first argument
`void* malloc(void *ctx, size_t size)`	allocate a memory block
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	allocate a memory block initialized with zeros
`void* realloc(void *ctx, void *ptr, size_t new_size)`	allocate or resize a memory block
`void free(void *ctx, void *ptr)`	free a memory block

在 3.5 版的變更: The `PyMemAllocator` structure was renamed to *`PyMemAllocatorEx`* and a new `calloc` field was added.

type **`PyMemAllocatorDomain`**

Enum used to identify an allocator domain. Domains:

`PYMEM_DOMAIN_RAW`

函式:

- *`PyMem_RawMalloc()`*
- *`PyMem_RawRealloc()`*
- *`PyMem_RawCalloc()`*
- *`PyMem_RawFree()`*

`PYMEM_DOMAIN_MEM`

函式:

- *`PyMem_Malloc()`*,
- *`PyMem_Realloc()`*
- *`PyMem_Calloc()`*
- *`PyMem_Free()`*

`PYMEM_DOMAIN_OBJ`

函式:

- *`PyObject_Malloc()`*
- *`PyObject_Realloc()`*
- *`PyObject_Calloc()`*
- *`PyObject_Free()`*

void **`PyMem_GetAllocator`**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

Get the memory block allocator of the specified domain.

void **`PyMem_SetAllocator`**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

Set the memory block allocator of the specified domain.

The new allocator must return a distinct non-`NULL` pointer when requesting zero 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*.

If the new allocator is not a hook (does not call the previous allocator), the *`PyMem_SetupDebugHooks()`* function must be called to reinstall the debug hooks on top on the new allocator.

See also *`PyPreConfig.allocator`* and *Preinitialize Python with PyPreConfig*.

警告

`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**.

在 3.12 版的變更: All allocators must be thread-safe.

void **PyMem_SetupDebugHooks**(void)

Setup *debug hooks in the Python memory allocators* to detect memory errors.

12.8 Debug hooks on the Python memory allocators

When Python is built in debug mode, the *PyMem_SetupDebugHooks()* function is called at the *Python preinitialization* to setup debug hooks on Python memory allocators to detect memory errors.

The PYTHONMALLOC environment variable can be used to install debug hooks on a Python compiled in release mode (ex: PYTHONMALLOC=debug).

The *PyMem_SetupDebugHooks()* function can be used to set debug hooks after calling *PyMem_SetAllocator()*.

These debug hooks fill dynamically allocated memory blocks with special, recognizable bit patterns. Newly allocated memory is filled with the byte 0xCD (PYMEM_CLEANBYTE), freed memory is filled with the byte 0xDD (PYMEM_DEADBYTE). Memory blocks are surrounded by "forbidden bytes" filled with the byte 0xFD (PYMEM_FORBIDDENBYTE). Strings of these bytes are unlikely to be valid addresses, floats, or ASCII strings.

Runtime 檢查:

- Detect API violations. For example, detect if *PyObject_Free()* is called on a memory block allocated by *PyMem_Malloc()*.
- Detect write before the start of the buffer (buffer underflow).
- Detect write after the end of the buffer (buffer overflow).
- 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.

On error, the debug hooks use the tracemalloc module to get the traceback where a memory block was allocated. The traceback is only displayed if tracemalloc is tracing Python memory allocations and the memory block was traced.

Let *S* = sizeof(size_t). 2*S bytes are added at each end of each block of *N* bytes requested. The memory layout is like so, where p represents the address returned by a malloc-like or realloc-like function (p[i:j] means the slice of bytes from *(p+i) inclusive up to *(p+j) exclusive; note that the treatment of negative indices differs from a Python slice):

p[-2*S:-S]
Number of bytes originally asked for. This is a size_t, big-endian (easier to read in a memory dump).

p[-S]
API identifier (ASCII character):

- 'r' for *PYMEM_DOMAIN_RAW*.
- 'm' for *PYMEM_DOMAIN_MEM*.
- 'o' for *PYMEM_DOMAIN_OBJ*.

p[-S+1:0]
Copies of PYMEM_FORBIDDENBYTE. Used to catch under- writes and reads.

p[0:N]
The requested memory, filled with copies of PYMEM_CLEANBYTE, used to catch reference to uninitialized memory. When a realloc-like function is called requesting a larger memory block, the new excess bytes are also filled with PYMEM_CLEANBYTE. When a free-like function is called, these are overwritten with

PYMEM_DEADBYTE, to catch reference to freed memory. When a realloc- like function is called requesting a smaller memory block, the excess old bytes are also filled with PYMEM_DEADBYTE.

p[N:N+S]
Copies of PYMEM_FORBIDDENBYTE. Used to catch over- writes and reads.

p[N+S:N+2*S]
Only used if the `PYMEM_DEBUG_SERIALNO` macro is defined (not defined by default).

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.

A realloc-like or free-like function first checks that the PYMEM_FORBIDDENBYTE bytes at each end are intact. If they've been altered, diagnostic output is written to stderr, and the program is aborted via Py_FatalError(). The other main failure mode is provoking a memory error when a program reads up one of the special bit patterns and tries to use it as an address. If you get in a debugger then and look at the object, you're likely to see that it's entirely filled with PYMEM_DEADBYTE (meaning freed memory is getting used) or PYMEM_CLEANBYTE (meaning uninitialized memory is getting used).

在 3.6 版的變更: 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.

在 3.8 版的變更: Byte patterns `0xCB` (`PYMEM_CLEANBYTE`), `0xDB` (`PYMEM_DEADBYTE`) and `0xFB` (`PYMEM_FORBIDDENBYTE`) have been replaced with `0xCD`, `0xDD` and `0xFD` to use the same values than Windows CRT debug `malloc()` and `free()`.

12.9 The pymalloc allocator

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.

The arena allocator uses the following functions:

- `VirtualAlloc()` and `VirtualFree()` on Windows,
- `mmap()` and `munmap()` if available,
- `malloc()` and `free()` otherwise.

This allocator is disabled if Python is configured with the `--without-pymalloc` option. It can also be disabled at runtime using the `PYTHONMALLOC` environment variable (ex: `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 Customize pymalloc Arena Allocator

在 3.4 版被加入.

type **PyObjectArenaAllocator**
Structure used to describe an arena allocator. The structure has three fields:

欄位	意義
`void *ctx`	user context passed as first argument
`void* alloc(void *ctx, size_t size)`	allocate an arena of size bytes
`void free(void *ctx, void *ptr, size_t size)`	free an arena

void **PyObject_GetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

Get the arena allocator.

void **PyObject_SetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

Set the arena allocator.

12.10 The mimalloc allocator

在 3.13 版被加入.

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

在 3.7 版被加入.

int **PyTraceMalloc_Track**(unsigned int domain, uintptr_t ptr, size_t size)

Track an allocated memory block in the `tracemalloc` module.

Return `0` on success, return `-1` on error (failed to allocate memory to store the trace). Return `-2` if tracemalloc is disabled.

If memory block is already tracked, update the existing trace.

int **PyTraceMalloc_Untrack**(unsigned int domain, uintptr_t ptr)

Untrack an allocated memory block in the `tracemalloc` module. Do nothing if the block was not tracked.

Return `-2` if tracemalloc is disabled, otherwise return `0`.

12.12 范例

Here is the example from section 總覽, rewritten so that the I/O buffer is allocated from the Python heap by using the first function set:

```
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;
```

The same code using the type-oriented function set:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
```

(繼續下一頁)

(繼續上一頁)

```
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_New */
return res;
```

Note that in the two examples above, the buffer is always manipulated via functions belonging to the same set. Indeed, it is required to use the same memory API family for a given memory block, so that the risk of mixing different allocators is reduced to a minimum. The following code sequence contains two errors, one of which is labeled as *fatal* because it mixes two different allocators operating on different heaps.

```
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()*.

These will be explained in the next chapter on defining and implementing new object types in C.

CHAPTER 13

物件實作支援

本章節🄵明在定義新的物件型🄵時所使用的函式、型🄵和巨集。

13.1 在 heap 上分配物件

PyObject ***_PyObject_New**(*PyTypeObject* *type)

回傳值：新的參照。

PyVarObject ***_PyObject_NewVar**(*PyTypeObject* *type, *Py_ssize_t* size)

回傳值：新的參照。

PyObject ***PyObject_Init**(*PyObject* *op, *PyTypeObject* *type)

回傳值：借用參照。🄵穩定 ABI 的一部分. 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()`*.

> **備🄵**
>
> 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)

回傳值：借用參照。🄵穩定 ABI 的一部分. 它會做到*`PyObject_Init()`* 的所有功能，🄵且會初始化一個大小可變物件的長度資訊。

> **備🄵**
>
> This function only initializes some of the object's memory. It does not zero the rest.

PyObject_New(TYPE, typeobj)

使用 C 結構型🄴 *TYPE* 和 Python 型🄴物件 *typeobj* (`PyTypeObject*`) 分配一個新的 Python 物件。它會呼叫*PyObject_Malloc()* 來分配記憶體，🄴且會像*PyObject_Init()* 一樣初始化它。呼叫者會擁有該物件的唯一參照（也就是它的參照計數會是 1）。

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.

這個巨集不會呼叫*tp_alloc*、*tp_new* (`__new__()`)、或*tp_init* (`__init__()`)。

這不能用於有在*tp_flags* 中設定*Py_TPFLAGS_HAVE_GC* 的物件；請改用*PyObject_GC_New*。

Memory allocated by this macro must be freed with *PyObject_Free()* (usually called via the object's *tp_free* slot).

備🄵

The returned memory is not guaranteed to have been completely zeroed before it was initialized.

備🄵

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);
```

也參考

- *PyObject_Free()*
- *PyObject_GC_New*
- *PyType_GenericAlloc()*
- *tp_alloc*

PyObject_NewVar(TYPE, typeobj, size)

和*PyObject_New* 類似，但有以下差🄵:

- 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*.
- 記憶體會像*PyObject_InitVar()* 一樣被初始化。

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.

這不能用於有在*tp_flags* 中設定*Py_TPFLAGS_HAVE_GC* 的物件；請改用*PyObject_GC_NewVar*。

Memory allocated by this function must be freed with *PyObject_Free()* (usually called via the object's *tp_free* slot).

備🄵

The returned memory is not guaranteed to have been completely zeroed before it was initialized.

備🄵

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);
```

也參考

- *PyObject_Free()*
- *PyObject_GC_NewVar*
- *PyType_GenericAlloc()*
- *tp_alloc*

void **PyObject_Del**(void *op)

和*PyObject_Free()* 相同。

PyObject **_Py_NoneStruct**

這個物件像是 Python 中的 `None`。它只應該透過*Py_None* 巨集來存取，該巨集的拿到指向該物件的指標。

也參考

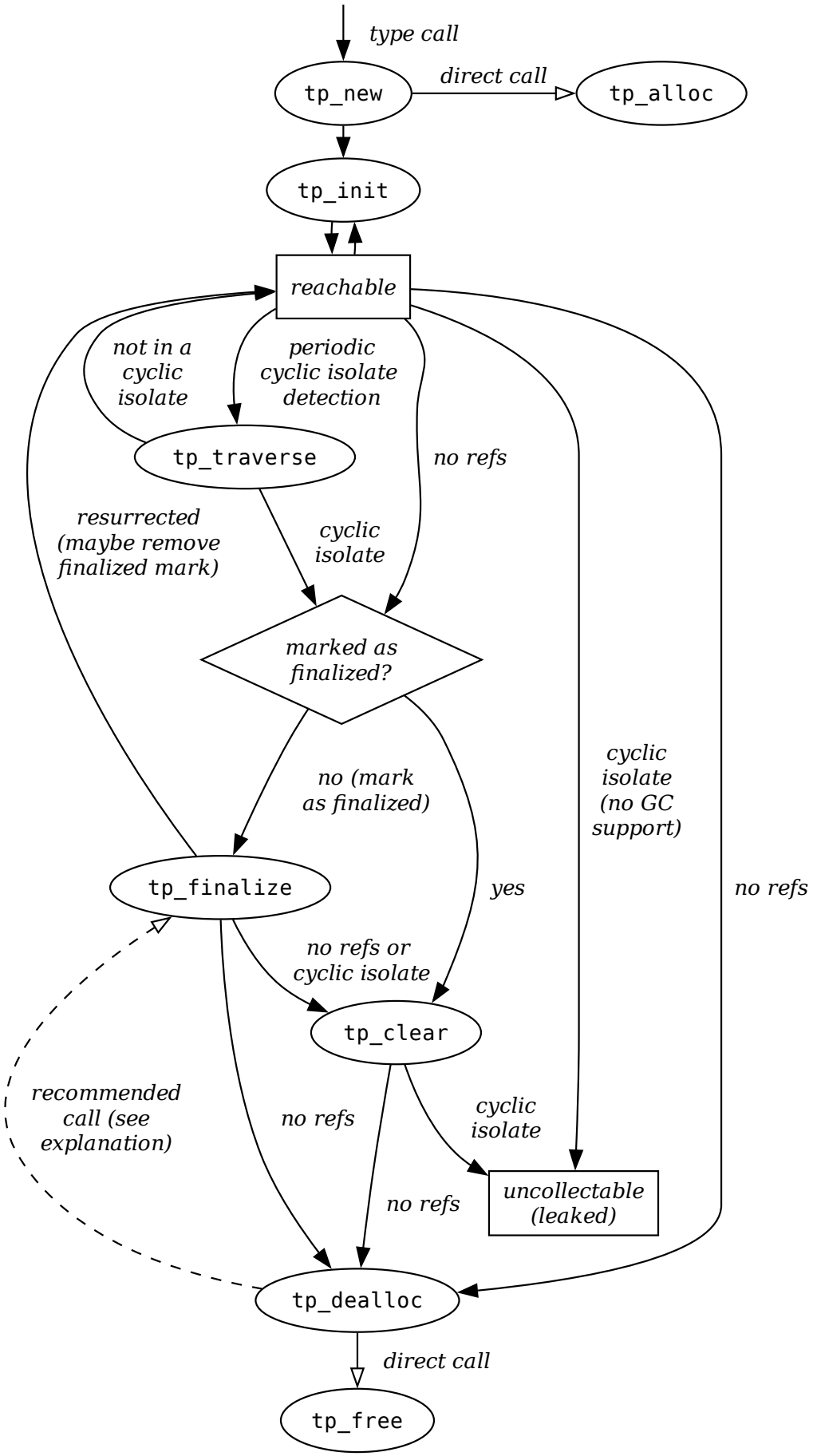
模組物件
分配記憶體和建立擴充模組。

13.2 物件生命🄵期

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 型🄵物件結構 for details about a particular slot.

13.2.1 生命事件

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

🄵明：

- 當一個新物件透過其型🄵來建構時：
  1. *tp_new* 會被呼叫來建立一個新物件。
  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__()`.
- 在 `tp_init` 完成後，該物件即可使用。
- 在最後一個對物件的參照被移除一段時間後：
  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 函式

要分配和釋放記憶體，請參見*在 heap 上分配物件*。

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.

也參考

`tp_dealloc` for example code.

13.3 通用物件結構

There are a large number of structures which are used in the definition of object types for Python. This section describes these structures and how they are used.

13.3.1 Base object types and 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**

🄵受限 API 的一部分. （只有部分成員是穩定 *ABI* 的一部分。） 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**

🄵穩定 ABI 的一部分. 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**

🄵穩定 ABI 的一部分. The object's type. Do not use this field directly; use `Py_TYPE` and `Py_SET_TYPE()` instead.

type **PyVarObject**

🄵受限 API 的一部分. （只有部分成員是穩定 *ABI* 的一部分。） 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**

🄵穩定 ABI 的一部分. 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

This is a macro used when declaring new types which represent objects without a varying length. The PyObject_HEAD macro expands to:

```
PyObject ob_base;
```

See documentation of *PyObject* above.

PyObject_VAR_HEAD

This is a macro used when declaring new types which represent objects with a length that varies from instance to instance. The PyObject_VAR_HEAD macro expands to:

```
PyVarObject ob_base;
```

請見上面*PyVarObject* 的文件。

PyTypeObject **PyBaseObject_Type**

🄵穩定 ABI 的一部分. The base class of all other objects, the same as `object` in Python.

int **Py_Is**(*PyObject* *x, *PyObject* *y)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Test if the *x* object is the *y* object, the same as `x is y` in Python.

在 3.10 版被加入.

int **Py_IsNone**(*PyObject* *x)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Test if an object is the `None` singleton, the same as `x is None` in Python.

在 3.10 版被加入.

int **Py_IsTrue**(*PyObject* *x)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Test if an object is the `True` singleton, the same as `x is True` in Python.

在 3.10 版被加入.

int **Py_IsFalse**(*PyObject* *x)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Test if an object is the `False` singleton, the same as `x is False` in Python.

在 3.10 版被加入.

PyTypeObject ***Py_TYPE**(*PyObject* *o)

回傳值：借用參照。🄵穩定 ABI 的一部分 自 *3.14* 版本開始. Get the type of the Python object *o*.

The returned reference is *borrowed* from *o*. Do not release it with *Py_DECREF()* or similar.

在 3.11 版的變更: *Py_TYPE()* is changed to an inline static function. The parameter type is no longer `const` *PyObject*`*`.

int **Py_IS_TYPE**(*PyObject* *o, *PyTypeObject* *type)

Return non-zero if the object *o* type is *type*. Return zero otherwise. Equivalent to: `Py_TYPE(o) == type`.

在 3.9 版被加入.

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.

在 3.9 版被加入.

Py_ssize_t **Py_SIZE**(*PyVarObject* *o)

Get the `ob_size` field of *o*.

在 3.11 版的變更: `Py_SIZE()` is changed to an inline static function. The parameter type is no longer `const PyVarObject*`.

void **Py_SET_SIZE**(*PyVarObject* *o, *Py_ssize_t* size)

Set the `ob_size` field of *o* to *size*.

在 3.9 版被加入.

PyObject_HEAD_INIT(type)

This is a macro which expands to initialization values for a new `PyObject` type. This macro expands to:

```
_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 實作函式與方法

type **PyCFunction**

🄵穩定 ABI 的一部分. 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.

The function signature is:

```
PyObject *PyCFunction(PyObject *self,
                      PyObject *args);
```

type **PyCFunctionWithKeywords**

🄵穩定 ABI 的一部分. 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**

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 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**

🄵穩定 ABI 的一部分 自 *3.13* 版本開始. 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)
```

在 3.9 版被加入.

type **PyMethodDef**

🄵穩定 ABI 的一部分 （包含所有成員）. Structure used to describe a method of an extension type. This structure has four fields:

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.

There are these calling conventions:

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
Methods with these flags must be of type *PyCFunctionWithKeywords*. The function expects three parameters: *self*, *args*, *kwargs* where *kwargs* is a dictionary of all the keyword arguments or possibly `NULL` if there are no keyword arguments. The parameters are typically processed using *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).

在 3.7 版被加入.

在 3.10 版的變更: `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.

在 3.7 版被加入.

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)`.

The method needs to be of type *PyCMethod*, the same as for `METH_FASTCALL | METH_KEYWORDS` with `defining_class` argument added after `self`.

在 3.9 版被加入.

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`.

The function must have 2 parameters. Since the second parameter is unused, *Py_UNUSED* can be used to prevent a compiler warning.

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.

These two constants are not used to indicate the calling convention but the binding when use with methods of classes. These may not be used for functions defined for modules. At most one of these flags may be set for any given method.

METH_CLASS
The method will be passed the type object as the first parameter rather than an instance of the type. This is used to create *class methods*, similar to what is created when using the `classmethod()` built-in function.

METH_STATIC
The method will be passed `NULL` as the first parameter rather than an instance of the type. This is used to create *static methods*, similar to what is created when using the `staticmethod()` built-in function.

One other constant controls whether a method is loaded in place of another definition with the same method name.

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)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.9* 版本開始. 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`.

也參考

`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`.

在 3.9 版被加入.

PyObject ***PyCFunction_NewEx**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module)

回傳值：新的參照。🄵穩定 ABI 的一部分. 等價於 `PyCMethod_New(ml, self, module, NULL)`。

PyObject ***PyCFunction_New**(*PyMethodDef* *ml, *PyObject* *self)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.4* 版本開始. 等價於 `PyCMethod_New(ml, self, NULL, NULL)`。

13.3.3 Accessing attributes of extension types

type **PyMemberDef**

🄵穩定 ABI 的一部分 （包含所有成員）. 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.

在 3.12 版的變更: `PyMemberDef` is always available. Previously, it required including `"structmember.h"`.

在 3.14 版的變更: *Py_RELATIVE_OFFSET* is now allowed for `"__vectorcalloffset__"`, `"__dictoffset__"` and `"__weaklistoffset__"`.

PyObject ***PyMember_GetOne**(const char *obj_addr, struct *PyMemberDef* *m)

🄵穩定 ABI 的一部分. Get an attribute belonging to the object at address *obj_addr*. The attribute is described by `PyMemberDef` *m*. Returns `NULL` on error.

在 3.12 版的變更: `PyMember_GetOne` is always available. Previously, it required including `"structmember.h"`.

int **PyMember_SetOne**(char *obj_addr, struct *PyMemberDef* *m, *PyObject* *o)

🄵穩定 ABI 的一部分. Set an attribute belonging to the object at address *obj_addr* to object *o*. The attribute to set is described by `PyMemberDef` *m*. Returns `0` if successful and a negative value on failure.

在 3.12 版的變更: `PyMember_SetOne` is always available. Previously, it required including `"structmember.h"`.

Member flags

The following flags can be used with *PyMemberDef.flags*:

Py_READONLY

不可寫入。

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.

在 3.10 版的變更: 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.

在 3.12 版的變更: 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()`.

巨集名懲	C type	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` （寫成 0 或 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.

在 3.12 版被加入: 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**

🄵穩定 ABI 的一部分 （包含所有成員）. Structure to define property-like access for a type. See also description of the *PyTypeObject.tp_getset* slot.

const char ***name**

屬性名稱

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**

可選的文件字串

void ***closure**

Optional user data pointer, providing additional data for getter and setter.

typedef *PyObject* *(***getter**)(*PyObject**, void*)

🄵穩定 ABI 的一部分. The get function takes one *PyObject** parameter (the instance) and a user data pointer (the associated closure):

It should return a new reference on success or NULL with a set exception on failure.

typedef int (***setter**)(*PyObject**, *PyObject**, void*)

🄵穩定 ABI 的一部分. set functions take two *PyObject** parameters (the instance and the value to be set) and a user data pointer (the associated closure):

In case the attribute should be deleted the second parameter is NULL. Should return 0 on success or -1 with a set exception on failure.

13.4 型🄵物件結構

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.

Type objects are fairly large compared to most of the standard types. The reason for the size is that each type object stores a large number of values, mostly C function pointers, each of which implements a small part of the type's functionality. The fields of the type object are examined in detail in this section. The fields will be described in the order in which they occur in the structure.

In addition to the following quick reference, the 範例 section provides at-a-glance insight into the meaning and use of *PyTypeObject*.

13.4.1 Quick Reference

"tp slots"

PyTypeObject Slot[Page 312, 1]	*Type*	special methods/attrs	Info[Page 312, 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-slots*				%
tp_repr	*reprfunc*	__repr__	X	X		X
tp_as_number	*PyNumberMethods* *	*sub-slots*				%
tp_as_sequence	*PySequenceMethods* *	*sub-slots*				%
tp_as_mapping	*PyMappingMethods* *	*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-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__				
[*tp_weaklist*]	*PyObject* *					
(*tp_del*)	*destructor*					
[*tp_version_tag*]	unsigned int					

繼續下一頁

表格 1 – 繼續上一頁

PyTypeObject Slot[1]	*Type*	special methods/attrs	Info[2] C T D I
tp_finalize	*destructor*	__del__	X
tp_vectorcall	*vectorcallfunc*		
[*tp_watched*]	unsigned char		

sub-slots

Slot	*Type*	special methods
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__
nb_invert	*unaryfunc*	__invert__
nb_lshift	*binaryfunc*	__lshift__ __rlshift__
nb_inplace_lshift	*binaryfunc*	__ilshift__
nb_rshift	*binaryfunc*	__rshift__ __rrshift__

繼續下一頁

[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] Columns:
"O": set on *PyBaseObject_Type*
"T": set on *PyType_Type*
"D": default (if slot is set to `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": inheritance

```
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
```

Note that some slots are effectively inherited through the normal attribute lookup chain.

表格 2 – 繼續上一頁

Slot	*Type*	special methods
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__

slot typedefs

typedef	Parameter Types	Return Type
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* *

更多細節請見下方的*Slot Type typedefs*。

13.4.2 PyTypeObject Definition

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;

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

(繼續下一頁)

(繼續上一頁)

```
    /* 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 PyObject Slots

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.

Inheritance:

This field is not inherited by subtypes.

PyObject.ob_type

This is the type's type, in other words its metatype. It is initialized by the argument to the `PyObject_HEAD_INIT` macro, and its value should normally be `&PyType_Type`. However, for dynamically loadable extension modules that must be usable on Windows (at least), the compiler complains that this is not a valid initializer. Therefore, the convention is to pass `NULL` to the

`PyObject_HEAD_INIT` macro and to initialize this field explicitly at the start of the module's initialization function, before doing anything else. This is typically done like this:

```
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.

Inheritance:

This field is inherited by subtypes.

13.4.4 PyVarObject Slots

`PyVarObject.ob_size`

For *statically allocated type objects*, this should be initialized to zero. For *dynamically allocated type objects*, this field has a special internal meaning.

This field should be accessed using the *`Py_SIZE()`* macro.

Inheritance:

This field is not inherited by subtypes.

13.4.5 PyTypeObject Slots

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"`.

For *dynamically allocated type objects*, this should just be the type name, and the module name explicitly stored in the type dict as the value for key `'__module__'`.

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.

This field must not be `NULL`. It is the only required field in *`PyTypeObject()`* (other than potentially *`tp_itemsize`*).

Inheritance:

This field is not inherited by subtypes.

Py_ssize_t *`PyTypeObject`*.**tp_basicsize**

Py_ssize_t *`PyTypeObject`*.**tp_itemsize**

These fields allow calculating the size in bytes of instances of the type.

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()*.

> **備🄵**
>
> 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)`.

Inheritance:

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()*).

範例：

```
static void
foo_dealloc(PyObject *self)
{
    PyObject *exc = PyErr_GetRaisedException();

    if (PyObject_CallFinalizerFromDealloc(self) < 0) {
        // self was resurrected.
        goto done;
    }

    PyTypeObject *tp = Py_TYPE(self);

    if (tp->tp_flags & Py_TPFLAGS_HAVE_GC) {
        PyObject_GC_UnTrack(self);
```

(繼續下一頁)

(繼續上一頁)

```
    }

    // 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.

Inheritance:

This field is inherited by subtypes.

也參考

物件生命🄵期 for details about how this slot relates to other slots.

Py_ssize_t *PyTypeObject*.**tp_vectorcall_offset**

An optional offset to a per-instance function that implements calling the object using the *vectorcall protocol*, a more efficient alternative of the simpler *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*.

Any class that sets `Py_TPFLAGS_HAVE_VECTORCALL` must also set *tp_call* and make sure its behaviour is consistent with the *vectorcallfunc* function. This can be done by setting *tp_call* to *PyVectorcall_Call()*.

在 3.8 版的變更: Before version 3.8, this slot was named `tp_print`. In Python 2.x, it was used for printing to a file. In Python 3.0 to 3.7, it was unused.

在 3.12 版的變更: 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.

Inheritance:

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**

An optional pointer to the get-attribute-string function.

This field is deprecated. When it is defined, it should point to a function that acts the same as the *tp_getattro* function, but taking a C string instead of a Python string object to give the attribute name.

Inheritance:

群組：*tp_getattr*、*tp_getattro*

This field is inherited by subtypes together with *tp_getattro*: a subtype inherits both *tp_getattr* and *tp_getattro* from its base type when the subtype's *tp_getattr* and *tp_getattro* are both `NULL`.

setattrfunc *PyTypeObject*.**tp_setattr**

An optional pointer to the function for setting and deleting attributes.

This field is deprecated. When it is defined, it should point to a function that acts the same as the *tp_setattro* function, but taking a C string instead of a Python string object to give the attribute name.

Inheritance:

群組：*tp_setattr*、*tp_setattro*

This field is inherited by subtypes together with *tp_setattro*: a subtype inherits both *tp_setattr* and *tp_setattro* from its base type when the subtype's *tp_setattr* and *tp_setattro* are both `NULL`.

PyAsyncMethods **PyTypeObject*.**tp_as_async**

Pointer to an additional structure that contains fields relevant only to objects which implement *awaitable* and *asynchronous iterator* protocols at the C-level. See *Async Object Structures* for details.

在 3.5 版被加入: Formerly known as `tp_compare` and `tp_reserved`.

Inheritance:

The *tp_as_async* field is not inherited, but the contained fields are inherited individually.

reprfunc *PyTypeObject*.**tp_repr**

An optional pointer to a function that implements the built-in function `repr()`.

The signature is the same as for *PyObject_Repr()*:

```
PyObject *tp_repr(PyObject *self);
```

The function must return a string or a Unicode object. Ideally, this function should return a string that, when passed to `eval()`, given a suitable environment, returns an object with the same value. If this is not feasible, it should return a string starting with `'<'` and ending with `'>'` from which both the type and the value of the object can be deduced.

Inheritance:

This field is inherited by subtypes.

預設：

When this field is not set, a string of the form `<%s object at %p>` is returned, where `%s` is replaced by the type name, and `%p` by the object's memory address.

PyNumberMethods **PyTypeObject*.**tp_as_number**

Pointer to an additional structure that contains fields relevant only to objects which implement the number protocol. These fields are documented in *Number Object Structures*.

Inheritance:

The *tp_as_number* field is not inherited, but the contained fields are inherited individually.

PySequenceMethods **PyTypeObject.tp_as_sequence*

Pointer to an additional structure that contains fields relevant only to objects which implement the sequence protocol. These fields are documented in *Sequence Object Structures*.

Inheritance:

The *tp_as_sequence* field is not inherited, but the contained fields are inherited individually.

PyMappingMethods **PyTypeObject.tp_as_mapping**

Pointer to an additional structure that contains fields relevant only to objects which implement the mapping protocol. These fields are documented in *Mapping Object Structures*.

Inheritance:

The *tp_as_mapping* field is not inherited, but the contained fields are inherited individually.

hashfunc *PyTypeObject*.**tp_hash**

An optional pointer to a function that implements the built-in function `hash()`.

The signature is the same as for *PyObject_Hash()*:

```
Py_hash_t tp_hash(PyObject *);
```

The value `-1` should not be returned as a normal return value; when an error occurs during the computation of the hash value, the function should set an exception and return `-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()*.

This field can be set explicitly to *PyObject_HashNotImplemented()* to block inheritance of the hash method from a parent type. This is interpreted as the equivalent of `__hash__ = None` at the Python level, causing `isinstance(o, collections.Hashable)` to correctly return `False`. Note that the converse is also true - setting `__hash__ = None` on a class at the Python level will result in the `tp_hash` slot being set to *PyObject_HashNotImplemented()*.

Inheritance:

群組：*tp_hash*、*tp_richcompare*

This field is inherited by subtypes together with *tp_richcompare*: a subtype inherits both of *tp_richcompare* and *tp_hash*, when the subtype's *tp_richcompare* and *tp_hash* are both `NULL`.

預設：

PyBaseObject_Type uses *PyObject_GenericHash()*.

ternaryfunc *PyTypeObject*.**tp_call**

An optional pointer to a function that implements calling the object. This should be `NULL` if the object is not callable. The signature is the same as for *PyObject_Call()*:

```
PyObject *tp_call(PyObject *self, PyObject *args, PyObject *kwargs);
```

Inheritance:

This field is inherited by subtypes.

reprfunc *PyTypeObject*.**tp_str**

An optional pointer to a function that implements the built-in operation `str()`. (Note that `str` is a type now, and `str()` calls the constructor for that type. This constructor calls *PyObject_Str()* to do the actual work, and *PyObject_Str()* will call this handler.)

The signature is the same as for *PyObject_Str()*:

```
PyObject *tp_str(PyObject *self);
```

The function must return a string or a Unicode object. It should be a "friendly" string representation of the object, as this is the representation that will be used, among other things, by the `print()` function.

Inheritance:

This field is inherited by subtypes.

預設：

When this field is not set, *PyObject_Repr()* is called to return a string representation.

getattrofunc *PyTypeObject*.**tp_getattro**

An optional pointer to the get-attribute function.

The signature is the same as for *PyObject_GetAttr()*:

```
PyObject *tp_getattro(PyObject *self, PyObject *attr);
```

It is usually convenient to set this field to *PyObject_GenericGetAttr()*, which implements the normal way of looking for object attributes.

Inheritance:

群組：*tp_getattr*、*tp_getattro*

This field is inherited by subtypes together with *tp_getattr*: a subtype inherits both *tp_getattr* and *tp_getattro* from its base type when the subtype's *tp_getattr* and *tp_getattro* are both `NULL`.

預設：

PyBaseObject_Type uses *PyObject_GenericGetAttr()*.

setattrofunc *PyTypeObject*.**tp_setattro**

An optional pointer to the function for setting and deleting attributes.

The signature is the same as for *PyObject_SetAttr()*:

```
int tp_setattro(PyObject *self, PyObject *attr, PyObject *value);
```

In addition, setting *value* to `NULL` to delete an attribute must be supported. It is usually convenient to set this field to *PyObject_GenericSetAttr()*, which implements the normal way of setting object attributes.

Inheritance:

群組：*tp_setattr*、*tp_setattro*

This field is inherited by subtypes together with *tp_setattr*: a subtype inherits both *tp_setattr* and *tp_setattro* from its base type when the subtype's *tp_setattr* and *tp_setattro* are both `NULL`.

預設：

PyBaseObject_Type uses *PyObject_GenericSetAttr()*.

PyBufferProcs **PyTypeObject*.**tp_as_buffer**

Pointer to an additional structure that contains fields relevant only to objects which implement the buffer interface. These fields are documented in *Buffer Object Structures*.

Inheritance:

The *tp_as_buffer* field is not inherited, but the contained fields are inherited individually.

unsigned long *PyTypeObject*.**tp_flags**

This field is a bit mask of various flags. Some flags indicate variant semantics for certain situations; others are used to indicate that certain fields in the type object (or in the extension structures referenced via *tp_as_number*, *tp_as_sequence*, *tp_as_mapping*, and *tp_as_buffer*) that were historically not always present are valid; if such a flag bit is clear, the type fields it guards must not be accessed and must be considered to have a zero or `NULL` value instead.

Inheritance:

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.

預設：

PyBaseObject_Type 使用 `Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE`。

Bit Masks:

The following bit masks are currently defined; these can be ORed together using the `|` operator to form the value of the *tp_flags* field. The macro *PyType_HasFeature()* takes a type and a flags value, *tp* and *f*, and checks whether `tp->tp_flags & f` is non-zero.

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.

Inheritance:

???

Py_TPFLAGS_BASETYPE

This bit is set when the type can be used as the base type of another type. If this bit is clear, the type cannot be subtyped (similar to a "final" class in Java).

Inheritance:

???

Py_TPFLAGS_READY

This bit is set when the type object has been fully initialized by *PyType_Ready()*.

Inheritance:

???

Py_TPFLAGS_READYING

This bit is set while *PyType_Ready()* is in the process of initializing the type object.

Inheritance:

???

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 循環垃圾回收的支援.

Inheritance:

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`.

Inheritance:

???

Py_TPFLAGS_METHOD_DESCRIPTOR

This bit indicates that objects behave like unbound methods.

If this flag is set for `type(meth)`, then:

- `meth.__get__(obj, cls)(*args, **kwds)` (with `obj` not None) must be equivalent to `meth(obj, *args, **kwds)`.
- `meth.__get__(None, cls)(*args, **kwds)` must be equivalent to `meth(*args, **kwds)`.

This flag enables an optimization for typical method calls like `obj.meth()`: it avoids creating a temporary "bound method" object for `obj.meth`.

在 3.8 版被加入.

Inheritance:

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()*.

在 3.12 版被加入.

Inheritance:

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.

在 3.12 版被加入.

Inheritance:

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.

在 3.12 版被加入.

Inheritance:

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

These flags are used by functions such as *PyLong_Check()* to quickly determine if a type is a subclass of a built-in type; such specific checks are faster than a generic check, like *PyObject_IsInstance()*. Custom types that inherit from built-ins should have their *tp_flags* set appropriately, or the code that interacts with such types will behave differently depending on what kind of check is used.

Py_TPFLAGS_HAVE_FINALIZE

This bit is set when the *tp_finalize* slot is present in the type structure.

在 3.4 版被加入.

在 3.8 版之後被🄵用: This flag isn't necessary anymore, as the interpreter assumes the *tp_finalize* slot is always present in the type structure.

Py_TPFLAGS_HAVE_VECTORCALL

This bit is set when the class implements the *vectorcall protocol*. See *tp_vectorcall_offset* for details.

Inheritance:

This bit is inherited if *tp_call* is also inherited.

在 3.9 版被加入.

在 3.12 版的變更: 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

This bit is set for type objects that are immutable: type attributes cannot be set nor deleted.

PyType_Ready() automatically applies this flag to *static types*.

Inheritance:

This flag is not inherited.

在 3.10 版被加入.

Py_TPFLAGS_DISALLOW_INSTANTIATION

Disallow creating instances of the type: set *tp_new* to NULL and don't create the `__new__` key in the type dictionary.

The flag must be set before creating the type, not after. For example, it must be set before *PyType_Ready()* is called on the type.

The flag is set automatically on *static types* if *tp_base* is NULL or `&PyBaseObject_Type` and *tp_new* is NULL.

Inheritance:

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).

> **備🄵**
>
> 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.

在 3.10 版被加入.

Py_TPFLAGS_MAPPING

This bit indicates that instances of the class may match mapping patterns when used as the subject of a `match` block. It is automatically set when registering or subclassing `collections.abc.Mapping`, and unset when registering `collections.abc.Sequence`.

> **備🄵**
>
> *Py_TPFLAGS_MAPPING* and *Py_TPFLAGS_SEQUENCE* are mutually exclusive; it is an error to enable both flags simultaneously.

Inheritance:

This flag is inherited by types that do not already set *Py_TPFLAGS_SEQUENCE*.

> **也參考**
>
> **PEP 634** -- Structural Pattern Matching: Specification

在 3.10 版被加入.

Py_TPFLAGS_SEQUENCE

This bit indicates that instances of the class may match sequence patterns when used as the subject of a `match` block. It is automatically set when registering or subclassing `collections.abc.Sequence`, and unset when registering `collections.abc.Mapping`.

> **備🄵**
>
> *Py_TPFLAGS_MAPPING* and *Py_TPFLAGS_SEQUENCE* are mutually exclusive; it is an error to enable both flags simultaneously.

Inheritance:

This flag is inherited by types that do not already set *Py_TPFLAGS_MAPPING*.

> **也參考**
>
> **PEP 634** -- Structural Pattern Matching: Specification

在 3.10 版被加入.

Py_TPFLAGS_VALID_VERSION_TAG

Internal. Do not set or unset this flag. To indicate that a class has changed call *PyType_Modified()*

> **警告**
>
> 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.

Inheritance:

This field is *not* inherited by subtypes.

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);
```

More information about Python's garbage collection scheme can be found in section 循環垃圾回收的支援.

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;
}
```

Note that *Py_VISIT()* is called only on those members that can participate in reference cycles. Although there is also a `self->key` member, it can only be `NULL` or a Python string and therefore cannot be part of a reference cycle.

On the other hand, even if you know a member can never be part of a cycle, as a debugging aid you may want to visit it anyway just so the `gc` module's `get_referents()` function will include it.

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);
```

> **警告**
>
> When implementing *tp_traverse*, only the members that the instance *owns* (by having *strong references* to them) must be visited. For instance, if an object supports weak references via the *tp_weaklist* slot, the pointer supporting the linked list (what *tp_weaklist* points to) must **not** be visited as the instance does not directly own the weak references to itself (the weakreference list is there to support the weak reference machinery, but the instance has no strong reference to the elements inside it, as they are allowed to be removed even if the instance is still alive).

Note that *Py_VISIT()* requires the *visit* and *arg* parameters to `local_traverse()` to have these specific names; don't name them just anything.

Instances of *heap-allocated types* hold a reference to their type. Their traversal function must therefore either visit *Py_TYPE(self)*, or delegate this responsibility by calling `tp_traverse` of another heap-allocated type (such as a heap-allocated superclass). If they do not, the type object may not be garbage-collected.

> **備🄵**
>
> *tp_traverse* 函式可以從任何執行緒呼叫。

在 3.9 版的變更: Heap-allocated types are expected to visit `Py_TYPE(self)` in `tp_traverse`. In earlier versions of Python, due to bug 40217, doing this may lead to crashes in subclasses.

Inheritance:

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`.

> **備🄵**
>
> If `tp_clear` fails to break a reference cycle then the objects in the *cyclic isolate* may remain indefinitely uncollectable ("leak"). See `gc.garbage`.

> **備🄵**
>
> Referents (direct and indirect) might have already been cleared; they are not guaranteed to be in a consistent state.

> **備🄵**
>
> *tp_clear* 函式可以從任何執行緒呼叫。

> **備🄵**
>
> 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*.

Implementations of *tp_clear* should drop the instance's references to those of its members that may be Python objects, and set its pointers to those members to `NULL`, as in the following example:

```
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);
```

More information about Python's garbage collection scheme can be found in section 循環垃圾回收的支援.

Inheritance:

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.

也參考

物件生命🄵期 for details about how this slot relates to other slots.

richcmpfunc *PyTypeObject*.**tp_richcompare**

An optional pointer to the rich comparison function, whose signature is:

```
PyObject *tp_richcompare(PyObject *self, PyObject *other, int op);
```

The first parameter is guaranteed to be an instance of the type that is defined by *PyTypeObject*.

The function should return the result of the comparison (usually `Py_True` or `Py_False`). If the comparison is undefined, it must return `Py_NotImplemented`, if another error occurred it must return `NULL` and set an exception condition.

The following constants are defined to be used as the third argument for *tp_richcompare* and for *PyObject_RichCompare()*:

常數	Comparison
Py_LT	<
Py_LE	<=
Py_EQ	==
Py_NE	!=
Py_GT	>
Py_GE	>=

The following macro is defined to ease writing rich comparison functions:

Py_RETURN_RICHCOMPARE(VAL_A, VAL_B, op)

Return `Py_True` or `Py_False` from the function, depending on the result of a comparison. VAL_A and VAL_B must be orderable by C comparison operators (for example, they may be C ints or floats). The third argument specifies the requested operation, as for *PyObject_RichCompare()*.

The returned value is a new *strong reference*.

On error, sets an exception and returns NULL from the function.

在 3.7 版被加入.

Inheritance:

群組：*tp_hash*、*tp_richcompare*

This field is inherited by subtypes together with *tp_hash*: a subtype inherits *tp_richcompare* and *tp_hash* when the subtype's *tp_richcompare* and *tp_hash* are both NULL.

預設：

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.

Do not confuse this field with *tp_weaklist*; that is the list head for weak references to the type object itself.

It is an error to set both the *Py_TPFLAGS_MANAGED_WEAKREF* bit and *tp_weaklistoffset*.

Inheritance:

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype uses a different weak reference list head than the base type. Since the list head is always found via *tp_weaklistoffset*, this should not be a problem.

預設：

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).

This function has the same signature as *PyObject_GetIter()*:

```
PyObject *tp_iter(PyObject *self);
```

Inheritance:

This field is inherited by subtypes.

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);
```

When the iterator is exhausted, it must return NULL; a StopIteration exception may or may not be set. When another error occurs, it must return NULL too. Its presence signals that the instances of this type are iterators.

Iterator types should also define the *tp_iter* function, and that function should return the iterator instance itself (not a new iterator instance).

This function has the same signature as *PyIter_Next()*.

Inheritance:

This field is inherited by subtypes.

struct *PyMethodDef* **PyTypeObject.tp_methods*

An optional pointer to a static NULL-terminated array of *PyMethodDef* structures, declaring regular methods of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a method descriptor.

Inheritance:

This field is not inherited by subtypes (methods are inherited through a different mechanism).

struct *PyMemberDef* **PyTypeObject.tp_members*

An optional pointer to a static NULL-terminated array of *PyMemberDef* structures, declaring regular data members (fields or slots) of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a member descriptor.

Inheritance:

This field is not inherited by subtypes (members are inherited through a different mechanism).

struct *PyGetSetDef* **PyTypeObject.tp_getset*

An optional pointer to a static NULL-terminated array of *PyGetSetDef* structures, declaring computed attributes of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a getset descriptor.

Inheritance:

This field is not inherited by subtypes (computed attributes are inherited through a different mechanism).

PyTypeObject **PyTypeObject.tp_base*

An optional pointer to a base type from which type properties are inherited. At this level, only single inheritance is supported; multiple inheritance require dynamically creating a type object by calling the metatype.

> **備🄵**
>
> Slot initialization is subject to the rules of initializing globals. C99 requires the initializers to be "address constants". Function designators like *PyType_GenericNew()*, with implicit conversion to a pointer, are valid C99 address constants.
>
> 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.
>
> Consequently, *tp_base* should be set in the extension module's init function.

Inheritance:

This field is not inherited by subtypes (obviously).

預設：

This field defaults to `&PyBaseObject_Type` (which to Python programmers is known as the type `object`).

PyObject **PyTypeObject.tp_dict*

The type's dictionary is stored here by *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.

在 3.12 版的變更: 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.

Inheritance:

This field is not inherited by subtypes (though the attributes defined in here are inherited through a different mechanism).

預設:

If this field is NULL, *PyType_Ready()* will assign a new dictionary to it.

> **警告**
>
> It is not safe to use *PyDict_SetItem()* on or otherwise modify *tp_dict* with the dictionary C-API.

descrgetfunc *PyTypeObject*.**tp_descr_get**

An optional pointer to a "descriptor get" function.

The function signature is:

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *type);
```

Inheritance:

This field is inherited by subtypes.

descrsetfunc *PyTypeObject*.**tp_descr_set**

An optional pointer to a function for setting and deleting a descriptor's value.

The function signature is:

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

The *value* argument is set to NULL to delete the value.

Inheritance:

This field is inherited by subtypes.

Py_ssize_t *PyTypeObject*.**tp_dictoffset**

While this field is still supported, *Py_TPFLAGS_MANAGED_DICT* should be used instead, if at all possible.

If the instances of this type have a dictionary containing instance variables, this field is non-zero and contains the offset in the instances of the type of the instance variable dictionary; this offset is used by *PyObject_GenericGetAttr()*.

Do not confuse this field with *tp_dict*; that is the dictionary for attributes of the type object itself.

The value specifies the offset of the dictionary from the start of the instance structure.

The *tp_dictoffset* should be regarded as write-only. To get the pointer to the dictionary call *PyObject_GenericGetDict()*. Calling *PyObject_GenericGetDict()* may need to allocate memory for the dictionary, so it is may be more efficient to call *PyObject_GetAttr()* when accessing an attribute on the object.

It is an error to set both the *Py_TPFLAGS_MANAGED_DICT* bit and *tp_dictoffset*.

Inheritance:

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*.

預設：

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**

An optional pointer to an instance initialization function.

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.

The function signature is:

```
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__()`.

The *tp_init* function, if not `NULL`, is called when an instance is created normally by calling its type, after the type's *tp_new* function has returned an instance of the type. If the *tp_new* function returns an instance of some other type that is not a subtype of the original type, no *tp_init* function is called; if *tp_new* returns an instance of a subtype of the original type, the subtype's *tp_init* is called.

Returns `0` on success, `-1` and sets an exception on error.

Inheritance:

This field is inherited by subtypes.

預設：

For *static types* this field does not have a default.

allocfunc *PyTypeObject*.**tp_alloc**

An optional pointer to an instance allocation function.

The function signature is:

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems);
```

Inheritance:

Static subtypes inherit this slot, which will be *PyType_GenericAlloc()* if inherited from `object`.

Heap subtypes do not inherit this slot.

預設：

For heap subtypes, this field is always set to *PyType_GenericAlloc()*.

For static subtypes, this slot is inherited (see above).

newfunc *PyTypeObject*.**tp_new**

An optional pointer to an instance creation function.

The function signature is:

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds);
```

The *subtype* argument is the type of the object being created; the *args* and *kwds* arguments represent positional and keyword arguments of the call to the type. Note that *subtype* doesn't have to equal the type whose *tp_new* function is called; it may be a subtype of that type (but not an unrelated type).

The *tp_new* function should call `subtype->tp_alloc(subtype, nitems)` to allocate space for the object, and then do only as much further initialization as is absolutely necessary. Initialization that can safely be ignored or repeated should be placed in the *tp_init* handler. A good rule of thumb is that for immutable

types, all initialization should take place in *tp_new*, while for mutable types, most initialization should be deferred to *tp_init*.

Set the *Py_TPFLAGS_DISALLOW_INSTANTIATION* flag to disallow creating instances of the type in Python.

Inheritance:

This field is inherited by subtypes, except it is not inherited by *static types* whose *tp_base* is `NULL` or `&PyBaseObject_Type`.

預設：

For *static types* this field has no default. This means if the slot is defined as `NULL`, the type cannot be called to create new instances; presumably there is some other way to create instances, like a factory function.

freefunc *PyTypeObject*.**tp_free**

An optional pointer to an instance deallocation function. Its signature is:

```
void tp_free(void *self);
```

This function must free the memory allocated by *tp_alloc*.

Inheritance:

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.

預設：

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**

An optional pointer to a function called by the garbage collector.

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);
```

(The only example of this are types themselves. The metatype, *PyType_Type*, defines this function to distinguish between statically and *dynamically allocated types*.)

Inheritance:

This field is inherited by subtypes.

預設：

This slot has no default. If this field is `NULL`, *Py_TPFLAGS_HAVE_GC* is used as the functional equivalent.

PyObject ***PyTypeObject**.**tp_bases**

Tuple of base types.

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.

> **警告**
>
> 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.

Inheritance:

This field is not inherited.

PyObject **PyTypeObject*.**tp_mro**

Tuple containing the expanded set of base types, starting with the type itself and ending with `object`, in Method Resolution Order.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is *initialized*.

Inheritance:

This field is not inherited; it is calculated fresh by *PyType_Ready()*.

PyObject **PyTypeObject*.**tp_cache**

Unused. Internal use only.

Inheritance:

This field is not inherited.

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__()`.

在 3.12 版的變更: For some types, this field does not hold a valid *PyObject**. The type was changed to `void*` to indicate this.

Inheritance:

This field is not inherited.

PyObject **PyTypeObject*.**tp_weaklist**

Weak reference list head, for weak references to this type object. Not inherited. Internal use only.

在 3.12 版的變更: 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.

Inheritance:

This field is not inherited.

destructor *PyTypeObject*.**tp_del**

This field is deprecated. Use *tp_finalize* instead.

unsigned int *PyTypeObject*.**tp_version_tag**

Used to index into the method cache. Internal use only.

Inheritance:

This field is not inherited.

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).

備🄵

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.

備🄵

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.

備🄵

The *tp_finalize* function can be called from any thread, although the *GIL* will be held.

備🄵

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 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);
}
```

Inheritance:

This field is inherited by subtypes.

在 3.4 版被加入.

在 3.8 版的變更: 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.

也參考

- **PEP 442**: ”Safe object finalization”
- 物件生命🄵期 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.

備🄵

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).

Inheritance:

This field is never inherited.

在 3.9 版被加入: (the field exists since 3.8 but it's only used since 3.9)

unsigned char *PyTypeObject*.**tp_watched**

Internal. Do not use.

在 3.12 版被加入.

13.4.6 Static Types

Traditionally, types defined in C code are *static*, that is, a static *PyTypeObject* structure is defined directly in code and initialized using *PyType_Ready()*.

This results in types that are limited relative to types defined in Python:

- Static types are limited to one base, i.e. they cannot use multiple inheritance.
- Static type objects (but not necessarily their instances) are immutable. It is not possible to add or modify the type object's attributes from Python.
- Static type objects are shared across *sub-interpreters*, so they should not include any subinterpreter-specific state.

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 Heap Types

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 Number Object Structures

type **PyNumberMethods**

This structure holds pointers to the functions which an object uses to implement the number protocol. Each function is used by the function of similar name documented in the 數字協定 section.

Here is the structure definition:

```
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;
```

> **備🄵**
>
> Binary and ternary functions must check the type of all their operands, and implement the necessary con-

versions (at least one of the operands is an instance of the defined type). If the operation is not defined for the given operands, binary and ternary functions must return `Py_NotImplemented`, if another error occurred they must return `NULL` and set an exception.

備🄵

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 Mapping Object Structures

type **PyMappingMethods**

This structure holds pointers to the functions which an object uses to implement the mapping protocol. It has three members:

lenfunc *PyMappingMethods*.**mp_length**

This function is used by *PyMapping_Size()* and *PyObject_Size()*, and has the same signature. This slot may be set to `NULL` if the object has no defined length.

binaryfunc *PyMappingMethods*.**mp_subscript**

This function is used by *PyObject_GetItem()* and *PySequence_GetSlice()*, and has the same signature as `PyObject_GetItem()`. This slot must be filled for the *PyMapping_Check()* function to return `1`, it can be `NULL` otherwise.

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 Sequence Object Structures

type **PySequenceMethods**

This structure holds pointers to the functions which an object uses to implement the sequence protocol.

lenfunc *PySequenceMethods*.**sq_length**

This function is used by *PySequence_Size()* and *PyObject_Size()*, and has the same signature. It is also used for handling negative indices via the *sq_item* and the *sq_ass_item* slots.

binaryfunc *PySequenceMethods*.**sq_concat**

This function is used by *PySequence_Concat()* and has the same signature. It is also used by the `+` operator, after trying the numeric addition via the *nb_add* slot.

ssizeargfunc *PySequenceMethods*.**sq_repeat**

This function is used by *PySequence_Repeat()* and has the same signature. It is also used by the `*` operator, after trying numeric multiplication via the *nb_multiply* slot.

ssizeargfunc *PySequenceMethods*.**sq_item**

This function is used by *PySequence_GetItem()* and has the same signature. It is also used by *PyObject_GetItem()*, after trying the subscription via the *mp_subscript* slot. This slot must be filled for the *PySequence_Check()* function to return `1`, it can be `NULL` otherwise.

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**

This function is used by *PySequence_SetItem()* and has the same signature. It is also used by *PyObject_SetItem()* and *PyObject_DelItem()*, after trying the item assignment and deletion via the *mp_ass_subscript* slot. This slot may be left to `NULL` if the object does not support item assignment and deletion.

objobjproc *PySequenceMethods*.**sq_contains**

This function may be used by *PySequence_Contains()* and has the same signature. This slot may be left to `NULL`, in this case `PySequence_Contains()` simply traverses the sequence until it finds a match.

binaryfunc *PySequenceMethods*.**sq_inplace_concat**

This function is used by *PySequence_InPlaceConcat()* and has the same signature. It should modify its first operand, and return it. This slot may be left to `NULL`, in this case `PySequence_InPlaceConcat()` will fall back to *PySequence_Concat()*. It is also used by the augmented assignment `+=`, after trying numeric in-place addition via the *nb_inplace_add* slot.

ssizeargfunc *PySequenceMethods*.**sq_inplace_repeat**

This function is used by *PySequence_InPlaceRepeat()* and has the same signature. It should modify its first operand, and return it. This slot may be left to `NULL`, in this case `PySequence_InPlaceRepeat()` will fall back to *PySequence_Repeat()*. It is also used by the augmented assignment `*=`, after trying numeric in-place multiplication via the *nb_inplace_multiply* slot.

13.4.11 Buffer Object Structures

type **PyBufferProcs**

This structure holds pointers to the functions required by the *Buffer protocol*. The protocol defines how an exporter object can expose its internal data to consumer objects.

getbufferproc *PyBufferProcs*.**bf_getbuffer**

The signature of this function is:

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

Handle a request to *exporter* to fill in *view* as specified by *flags*. Except for point (3), an implementation of this function MUST take these steps:

(1) Check if the request can be met. If not, raise `BufferError`, set `view->obj` to `NULL` and return `-1`.

(2) Fill in the requested fields.

(3) Increment an internal counter for the number of exports.

(4) Set `view->obj` to *exporter* and increment `view->obj`.

(5) 回傳 `0`。

If *exporter* is part of a chain or tree of buffer providers, two main schemes can be used:

- 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.

The individual fields of *view* are described in section *Buffer structure*, the rules how an exporter must react to specific requests are in section *Buffer request types*.

All memory pointed to in the `Py_buffer` structure belongs to the exporter and must remain valid until there are no consumers left. `format`, `shape`, `strides`, `suboffsets` and `internal` are read-only for the consumer.

`PyBuffer_FillInfo()` provides an easy way of exposing a simple bytes buffer while dealing correctly with all request types.

`PyObject_GetBuffer()` is the interface for the consumer that wraps this function.

releasebufferproc `PyBufferProcs`**.bf_releasebuffer**

The signature of this function is:

```
void (PyObject *exporter, Py_buffer *view);
```

Handle a request to release the resources of the buffer. If no resources need to be released, `PyBufferProcs.bf_releasebuffer` may be `NULL`. Otherwise, a standard implementation of this function will take these optional steps:

(1) Decrement an internal counter for the number of exports.

(2) If the counter is `0`, free all memory associated with *view*.

The exporter MUST use the `internal` field to keep track of buffer-specific resources. This field is guaranteed to remain constant, while a consumer MAY pass a copy of the original buffer as the *view* argument.

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()` is the interface for the consumer that wraps this function.

13.4.12 Async Object Structures

在 3.5 版被加入.

type **PyAsyncMethods**

This structure holds pointers to the functions required to implement *awaitable* and *asynchronous iterator* objects.

Here is the structure definition:

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
    sendfunc am_send;
} PyAsyncMethods;
```

unaryfunc `PyAsyncMethods`**.am_await**

The signature of this function is:

```
PyObject *am_await(PyObject *self);
```

The returned object must be an *iterator*, i.e. `PyIter_Check()` must return `1` for it.

This slot may be set to `NULL` if an object is not an *awaitable*.

unaryfunc `PyAsyncMethods`**.am_aiter**

The signature of this function is:

```
PyObject *am_aiter(PyObject *self);
```

Must return an *asynchronous iterator* object. See `__anext__()` for details.

This slot may be set to `NULL` if an object does not implement asynchronous iteration protocol.

unaryfunc *PyAsyncMethods*.**am_anext**

The signature of this function is:

```
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**

The signature of this function is:

```
PySendResult am_send(PyObject *self, PyObject *arg, PyObject **result);
```

See *PyIter_Send()* for details. This slot may be set to `NULL`.

在 3.10 版被加入.

13.4.13 Slot Type typedefs

typedef *PyObject* *(***allocfunc**)(*PyTypeObject* *cls, *Py_ssize_t* nitems)

🄵穩定 ABI 的一部分. 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*.

This function should not do any other instance initialization, not even to allocate additional memory; that should be done by *tp_new*.

typedef void (***destructor**)(*PyObject**)

🄵穩定 ABI 的一部分.

typedef void (***freefunc**)(void*)

請見*tp_free*。

typedef *PyObject* *(***newfunc**)(*PyTypeObject**, *PyObject**, *PyObject**)

🄵穩定 ABI 的一部分. 請見*tp_new*。

typedef int (***initproc**)(*PyObject**, *PyObject**, *PyObject**)

🄵穩定 ABI 的一部分. 請見*tp_init*。

typedef *PyObject* *(***reprfunc**)(*PyObject**)

🄵穩定 ABI 的一部分. 請見*tp_repr*。

typedef *PyObject* *(***getattrfunc**)(*PyObject* *self, char *attr)

🄵穩定 ABI 的一部分. Return the value of the named attribute for the object.

typedef int (***setattrfunc**)(*PyObject* *self, char *attr, *PyObject* *value)

🄵穩定 ABI 的一部分. Set the value of the named attribute for the object. The value argument is set to `NULL` to delete the attribute.

typedef *PyObject* *(***getattrofunc**)(*PyObject* *self, *PyObject* *attr)

🄵穩定 ABI 的一部分. Return the value of the named attribute for the object.

請見*tp_getattro*。

typedef int (***setattrofunc**)(*PyObject* *self, *PyObject* *attr, *PyObject* *value)

🄵穩定 ABI 的一部分. Set the value of the named attribute for the object. The value argument is set to `NULL` to delete the attribute.

請見*tp_setattro*。

typedef *PyObject* *(***descrgetfunc**)(*PyObject**, *PyObject**, *PyObject**)

🄵穩定 ABI 的一部分. 請見*tp_descr_get*。

typedef int (***descrsetfunc**)(*PyObject**, *PyObject**, *PyObject**)

🄵穩定 ABI 的一部分. 請見*tp_descr_set*。

typedef *Py_hash_t* (***hashfunc**)(*PyObject**)

🄵穩定 ABI 的一部分. 請見*tp_hash*。

typedef *PyObject* *(***richcmpfunc**)(*PyObject**, *PyObject**, int)

🄵穩定 ABI 的一部分. 請見*tp_richcompare*。

typedef *PyObject* *(***getiterfunc**)(*PyObject**)

🄵穩定 ABI 的一部分. 請見*tp_iter*。

typedef *PyObject* *(***iternextfunc**)(*PyObject**)

🄵穩定 ABI 的一部分. 請見*tp_iternext*。

typedef *Py_ssize_t* (***lenfunc**)(*PyObject**)

🄵穩定 ABI 的一部分.

typedef int (***getbufferproc**)(*PyObject**, *Py_buffer**, int)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始.

typedef void (***releasebufferproc**)(*PyObject**, *Py_buffer**)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始.

typedef *PyObject* *(***unaryfunc**)(*PyObject**)

🄵穩定 ABI 的一部分.

typedef *PyObject* *(***binaryfunc**)(*PyObject**, *PyObject**)

🄵穩定 ABI 的一部分.

typedef *PySendResult* (***sendfunc**)(*PyObject**, *PyObject**, *PyObject***)

請見*am_send*。

typedef *PyObject* *(***ternaryfunc**)(*PyObject**, *PyObject**, *PyObject**)

🄵穩定 ABI 的一部分.

typedef *PyObject* *(***ssizeargfunc**)(*PyObject**, *Py_ssize_t*)

🄵穩定 ABI 的一部分.

typedef int (***ssizeobjargproc**)(*PyObject**, *Py_ssize_t*, *PyObject**)

🄵穩定 ABI 的一部分.

typedef int (***objobjproc**)(*PyObject**, *PyObject**)

🄵穩定 ABI 的一部分.

typedef int (***objobjargproc**)(*PyObject**, *PyObject**, *PyObject**)

🄵穩定 ABI 的一部分.

13.4.14 范例

The following are simple examples of Python type definitions. They include common usage you may encounter. Some demonstrate tricky corner cases. For more examples, practical info, and a tutorial, see defining-new-types and new-types-topics.

A basic *static type*:

```
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,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
};
```

You may also find older code (especially in the CPython code base) with a more verbose initializer:

```
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 */
};
```

A type that supports weakrefs, instance dicts, and hashing:

```
typedef struct {
    PyObject_HEAD
```

(繼續下一頁)

(繼續上一頁)

```
    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_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,
};
```

The simplest *static type* with fixed-length instances:

```
typedef struct {
    PyObject_HEAD
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
};
```

The simplest *static type* with variable-length instances:

```
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 *),
```

(繼續下一頁)

(繼續上一頁)

```
    .tp_itemsize = sizeof(char *),
};
```

13.5 循環垃圾回收的支援

Python's support for detecting and collecting garbage which involves circular references requires support from object types which are "containers" for other objects which may also be containers. Types which do not store references to other objects, or which only store references to atomic types (such as numbers or strings), do not need to provide any explicit support for garbage collection.

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
: Objects with a type with this flag set must conform with the rules documented here. For convenience these objects will be referred to as container objects.

Constructors for container types must conform to two rules:

1. The memory for the object must be allocated using *PyObject_GC_New* or *PyObject_GC_NewVar*.
2. Once all the fields which may contain references to other containers are initialized, it must call *PyObject_GC_Track()*.

Similarly, the deallocator for the object must conform to a similar pair of rules:

1. Before fields which refer to other containers are invalidated, *PyObject_GC_UnTrack()* must be called.
2. The object's memory must be deallocated using *PyObject_GC_Del()*.

> **警告**
>
> If a type adds the Py_TPFLAGS_HAVE_GC, then it *must* implement at least a *tp_traverse* handler or explicitly use one from its subclass or subclasses.
>
> When calling *PyType_Ready()* or some of the APIs that indirectly call it like *PyType_FromSpecWithBases()* or *PyType_FromSpec()* the interpreter will automatically populate the *tp_flags*, *tp_traverse* and *tp_clear* fields if the type inherits from a class that implements the garbage collector protocol and the child class does *not* include the *Py_TPFLAGS_HAVE_GC* flag.

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).

> **也參考**
>
> - *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).

也參考

- *PyObject_GC_Del()*
- *PyObject_NewVar*
- *PyType_GenericAlloc()*
- *tp_alloc*

PyObject ***PyUnstable_Object_GC_NewWithExtraData**(*PyTypeObject* *type, size_t extra_size)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

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).

警告

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.

在 3.12 版被加入.

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)

🄵穩定 ABI 的一部分. Adds the object *op* to the set of container objects tracked by the collector. The collector can run at unexpected times so objects must be valid while being tracked. This should be called once all the fields followed by the *tp_traverse* handler become valid, usually near the end of the constructor.

int **PyObject_IS_GC**(*PyObject* *obj)

Returns non-zero if the object implements the garbage collector protocol, otherwise returns 0.

The object cannot be tracked by the garbage collector if this function returns 0.

int **PyObject_GC_IsTracked**(*PyObject* *op)

🄵穩定 ABI 的一部分 自 *3.9* 版本開始. Returns 1 if the object type of *op* implements the GC protocol and *op* is being currently tracked by the garbage collector and 0 otherwise.

This is analogous to the Python function `gc.is_tracked()`.

在 3.9 版被加入.

int **PyObject_GC_IsFinalized**(*PyObject* *op)

🄵穩定 ABI 的一部分 自 *3.9* 版本開始. Returns 1 if the object type of *op* implements the GC protocol and *op* has been already finalized by the garbage collector and 0 otherwise.

This is analogous to the Python function `gc.is_finalized()`.

在 3.9 版被加入.

void **PyObject_GC_Del**(void *op)

🄵穩定 ABI 的一部分. 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.

也參考

- *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)

🄵穩定 ABI 的一部分. Remove the object *op* from the set of container objects tracked by the collector. Note that *PyObject_GC_Track()* can be called again on this object to add it back to the set of tracked objects. The deallocator (*tp_dealloc* handler) should call this for the object before any of the fields used by the *tp_traverse* handler become invalid.

在 3.8 版的變更: The `_PyObject_GC_TRACK()` and `_PyObject_GC_UNTRACK()` macros have been removed from the public C API.

The *tp_traverse* handler accepts a function parameter of this type:

typedef int (***visitproc**)(*PyObject* *object, void *arg)

🄵穩定 ABI 的一部分. Type of the visitor function passed to the *tp_traverse* handler. The function should be called with an object to traverse as *object* and the third parameter to the *tp_traverse* handler as *arg*. The Python core uses several visitor functions to implement cyclic garbage detection; it's not expected that users will need to write their own visitor functions.

The *tp_traverse* handler must have the following type:

typedef int (***traverseproc**)(*PyObject* *self, *visitproc* visit, void *arg)

🄵穩定 ABI 的一部分. Traversal function for a container object. Implementations must call the *visit* function for each object directly contained by *self*, with the parameters to *visit* being the contained object and the *arg* value passed to the handler. The *visit* function must not be called with a `NULL` object argument. If *visit* returns a non-zero value that value should be returned immediately.

To simplify writing *tp_traverse* handlers, a *Py_VISIT()* macro is provided. In order to use this macro, the *tp_traverse* implementation must name its arguments exactly *visit* and *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;
}
```

The *tp_clear* handler must be of the *inquiry* type, or NULL if the object is immutable.

typedef int (***inquiry**)(*PyObject* *self)

🄵穩定 ABI 的一部分. Drop references that may have created reference cycles. Immutable objects do not have to define this method since they can never directly create reference cycles. Note that the object must still be valid after calling this method (don't just call *Py_DECREF()* on a reference). The collector will call this method if it detects that this object is involved in a reference cycle.

13.5.1 Controlling the Garbage Collector State

The C-API provides the following functions for controlling garbage collection runs.

Py_ssize_t **PyGC_Collect**(void)

🄵穩定 ABI 的一部分. Perform a full garbage collection, if the garbage collector is enabled. (Note that `gc.collect()` runs it unconditionally.)

Returns the number of collected + unreachable objects which cannot be collected. If the garbage collector is disabled or already collecting, returns `0` immediately. Errors during garbage collection are passed to `sys.unraisablehook`. This function does not raise exceptions.

int **PyGC_Enable**(void)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Enable the garbage collector: similar to `gc.enable()`. Returns the previous state, 0 for disabled and 1 for enabled.

在 3.10 版被加入.

int **PyGC_Disable**(void)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Disable the garbage collector: similar to `gc.disable()`. Returns the previous state, 0 for disabled and 1 for enabled.

在 3.10 版被加入.

int **PyGC_IsEnabled**(void)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. Query the state of the garbage collector: similar to `gc.isenabled()`. Returns the current state, 0 for disabled and 1 for enabled.

在 3.10 版被加入.

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)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

Run supplied *callback* on all live GC-capable objects. *arg* is passed through to all invocations of *callback*.

> **警告**
>
> 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.

在 3.12 版被加入.

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.

在 3.12 版被加入.

CHAPTER 14

API 和 ABI 版本管理

14.1 Build-time version constants

CPython 透過以下巨集 (macro) 公開其版本號。請注意，對應到的是**建置 (built)** 所用到的版本。關於 **run time** 所使用的版本，請見*Py_Version*。

關於跨版本 API 和 ABI 穩定性的討論，請見*C API 穩定性*。

PY_MAJOR_VERSION

在 3.4.1a2 中的 3。

PY_MINOR_VERSION

在 3.4.1a2 中的 4。

PY_MICRO_VERSION

在 3.4.1a2 中的 1。

PY_RELEASE_LEVEL

在 3.4.1a2 中的 a。0xA 代表 alpha 版本、0xB 代表 beta 版本、0xC 🄵發布候選版本、0xF 則🄵最終版。

PY_RELEASE_SERIAL

在 3.4.1a2 中的 2。零則🄵最終發布版本。

PY_VERSION_HEX

被編碼🄵單一整數的 Python 版本號。詳見*Py_PACK_FULL_VERSION()* 以了解編碼詳細資訊。

使用它進行數值比較，例如 `#if PY_VERSION_HEX >= ...`。

14.2 Run-time version

const unsigned long **Py_Version**

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. 編碼🄵單個常數整數的 Python runtime 版本號。詳見*Py_PACK_FULL_VERSION()* 以了解編碼詳細資訊。這包含在 runtime 使用的 Python 版本。

使用它進行數值比較，例如 `if (Py_Version >= ...)`。

在 3.11 版被加入.

14.3 Bit-packing macros

uint32_t **Py_PACK_FULL_VERSION**(int major, int minor, int micro, int release_level, int release_serial)

🄵穩定 ABI 的一部分 自 *3.14* 版本開始. Return the given version, encoded as a single 32-bit integer with the following structure:

引數	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`

例如:

版本	`Py_PACK_FULL_VERSION` 引數	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.

在 3.14 版被加入.

uint32_t **Py_PACK_VERSION**(int major, int minor)

🄵穩定 ABI 的一部分 自 *3.14* 版本開始. 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.

在 3.14 版被加入.

CHAPTER 15

監控 C API

於 3.13 版本中新增。

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`.

CHAPTER 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.

關於事件的𝖥述請見 `sys.monitoring`。

int **PyMonitoring_FirePyStartEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

發出一個 `PY_START` 事件。

int **PyMonitoring_FirePyResumeEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

發出一個 `PY_RESUME` 事件。

int **PyMonitoring_FirePyReturnEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

發出一個 `PY_RETURN` 事件。

int **PyMonitoring_FirePyYieldEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

發出一個 `PY_YIELD` 事件。

int **PyMonitoring_FireCallEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *callable, *PyObject* *arg0)

發出一個 `CALL` 事件。

int **PyMonitoring_FireLineEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, int lineno)

發出一個 `LINE` 事件。

int **PyMonitoring_FireJumpEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

發出一個 JUMP 事件。

int **PyMonitoring_FireBranchLeftEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

發出一個 BRANCH_LEFT 事件。

int **PyMonitoring_FireBranchRightEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

發出一個 BRANCH_RIGHT 事件。

int **PyMonitoring_FireCReturnEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

發出一個 C_RETURN 事件。

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.

event_types 的巨集有：

巨集	事件
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.

在 3.13 版被加入.

在 3.14 版之後被🄴用: This function is *soft deprecated*.

APPENDIX A

術語表

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

...
可以表示：

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

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

annotate function（🄵釋函式）
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*.

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

在 runtime 的區域變數🄵釋無法被存取，但全域變數、類🄵屬性和函式的🄵釋，分🄵能🄵以對模組、類🄵和函式呼叫 `annotationlib.get_annotations()` 來取得。

請參🄵*variable annotation*、*function annotation*、**PEP 484**、**PEP 526** 和 **PEP 649**，這些章節皆有此功能的🄵明。關於🄵釋的最佳實踐方法也請參🄵 annotations-howto。

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

位元組碼的指令列表可以在 dis 模組的🄵明文件中找到。

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

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

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

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

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

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

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

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

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

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

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

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

- The temporary state or environment established by a *context manager* via a `with` statement.
- The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
- 一個 `contextvars.Context` 物件。另請參🄵*current context*。

context management protocol（情境管理協定）
由 `with` 陳述式所呼叫的 `__enter__()` 和 `__exit__()` 方法。另請參🄵 **PEP 343**。

context manager（情境管理器）
An object which implements the *context management protocol* and controls the environment seen in a `with` statement. See **PEP 343**.

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

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

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

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

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

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

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.

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

裝飾器語法只是語法糖。以下兩個函式定義在語義上是等效的：

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

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

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

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

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

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

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

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

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

duck-typing（鴨子型🄵）
一種程式設計風格，它不是藉由檢查一個物件的型🄵來確定它是否具有正確的介面；取而代之的是，method 或屬性會單純地被呼叫或使用。（「如果它看起來像一🄵鴨子而且叫起來像一🄵鴨子，那🄵它一定是一🄵鴨子。」）因🄵🄵調介面而非特定型🄵，精心設計的程式碼能讓多形替代 (polymorphic

substitution) 來增進它的靈活性。鴨子型🄵要避免使用 `type()` 或 `isinstance()` 進行測試。(但是請注意，鴨子型🄵可以用抽象基底類🄵 *(abstract base class)* 來補充。) 然而，它通常會🄵用 `hasattr()` 測試，或是*EAFP* 程式設計風格。

dunder（雙底🄵）
一個非正式的縮寫，代表「雙底🄵 (double underscore)」，當談論到特殊方法時使用。例如，`__init__` 通常被叫做"dunder init"。

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

evaluate function（求值函式）
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.

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

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

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

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

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

file-like object（類檔案物件）
file object（檔案物件）的同義字。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）
Python 所使用的一種編碼和錯誤處理函式，用來解碼來自作業系統的位元組，以及將 Unicode 編碼到作業系統。

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

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

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

另請參🄵*locale encoding*（區域編碼）。

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

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

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

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

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

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

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

function annotation（函式🄵釋）
函式參數或回傳值的一個*annotation*（🄵釋）。

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

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

函式🄵釋的語法在 function 章節有詳細解釋。

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

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

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

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

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

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

generator iterator（🄵生器🄵代器）
一個由*generator*（🄵生器）函式所建立的物件。

每個 `yield` 會暫停處理程序，🄵記住執行狀態（包括區域變數及擱置中的 try 陳述式）。當🄵生器🄵代器回復時，它會從停止的地方繼續執行（與那些每次呼叫時都要重新開始的函式有所不同）。

generator expression（🄵生器運算式）
一個會回傳🄵代器的運算式。它看起來像一個正常的運算式，後面接著一個 `for` 子句，該子句定義了🄵圈變數、範圍以及一個選擇性的 `if` 子句。該組合運算式會🄴外層函式🄵生多個值：

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

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

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

generic type（泛型型🄵）
一個能🄵被參數化 (parameterized) 的*type*（型🄵）；通常是一個 容器型🄵，像是 `list` 和 `dict`。它被用於型🄵提示和🄵釋。

詳情請參🄵泛型🄵名型🄵、**PEP 483**、**PEP 484**、**PEP 585** 和 `typing` 模組。

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

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

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

從 Python 3.13 開始可以使用 `--disable-gil` 建置設定來停用 GIL。使用此選項建立 Python 後，必須使用 `-X gil=0` 來執行程式碼，或者設定 `PYTHON_GIL=0` 環境變數後再執行程式碼。此功能可以提高多執行緒應用程式的效能，🄵使多核心 CPU 的高效使用變得更加容易。有關更多詳細資訊，請參🄵 **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（雜🄵架構的 pyc）
一個位元組碼 (bytecode) 暫存檔，它使用雜🄵值而不是對應原始檔案的最後修改時間，來確定其有效性。請參🄵 pyc-invalidation。

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

可雜🄵性 (hashability) 使一個物件可用作 dictionary（字典）的鍵和 set（集合）的成員，因🄵這些資料結構都在其🄴部使用了雜🄵值。

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

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

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

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

Immortal objects can be identified via `sys._is_immortal()`, or via *`PyUnstable_IsImmortal()`* in the C API.

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

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

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

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

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

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

interpreter shutdown（直譯器關閉）
當 Python 直譯器被要求關閉時，它會進入一個特殊階段，在此它逐漸釋放所有被配置的資源，例如模組和各種關鍵F部結構。它也會多次呼叫*垃圾回收器 (garbage collector)*。這能F觸發使用者自定的解構函式 (destructor) 或弱引用的回呼 (weakref callback)，F執行其中的程式碼。在關閉階段被執行的程式碼會遇到各種例外，因F它所依賴的資源可能不再有作用了（常見的例子是函式庫模組或是警告機制）。

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

iterable（可F代物件）
一種能F一次回傳一個其中成員的物件。可F代物件的例子包括所有的序列型F（像是 `list`、`str` 和 `tuple`）和某些非序列型F，像是 `dict`、*檔案物件*，以及你所定義的任何 class 物件，只要那些 class 有實作*sequence*（序列）語意的 `__iter__()` 或是 `__getitem__()` method，該物件就是可F代物件。

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

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

在 typeiter 文中可以找到更多資訊。

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

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

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

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

keyword argument（關鍵字引數）
請參🄵*argument*（引數）。

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

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

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

lexical analyzer（詞法分析器）
tokenizer 的正式名稱；請參🄵*token*。

list（串列）
一個 Python 🄵建的*sequence*（序列）。🄵管它的名字是 list，它其實更類似其他語言中的一個陣列 (array) 而較不像一個鏈結串列 (linked list)，因🄵存取元素的時間🄵雜度是 *O*(1)。

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

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

- finders-and-loaders
- `importlib.abc.Loader`
- **PEP 302**

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

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

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

`locale.getencoding()` 可以用來取得區域編碼。

也請參考*filesystem encoding and error handler*。

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

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

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

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

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

更多資訊可以在 metaclasses 章節中找到。

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

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

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

另請參🄵*package*（套件）。

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

另請參🄵 module-specs。

MRO
請參🄵*method resolution order*（方法解析順序）。

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

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

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

```
>>> sys.float_info[1]                   # 以索引存取
1024
>>> sys.float_info.max_exp              # 以欄位名稱存取
1024
>>> isinstance(sys.float_info, tuple)   # 屬於 tuple 型🄴
True
```

有些 named tuple 是🄴建型🄵（如上例）。或者，一個 named tuple 也可以從一個正規的 class 定義來建立，只要該 class 是繼承自 `tuple`，且定義了附名欄位 (named field) 即可。這類的 class 可以手工編

寫、可以繼承自 `typing.NamedTuple` 來建立，也可以使用工廠函式 (factory function) `collections.namedtuple()` 來建立。後者技術也增加了一些額外的 method，這些 method 可能是在手寫或🄵建的 named tuple 中，無法找到的。

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

namespace package（命名空間套件）
一個*package*（套件），它只能作🄵子套件 (subpackage) 的一個容器。命名空間套件可能🄵有實體的表示法，而且具體來🄵它們不像是一個*regular package*（正規套件），因🄵它們🄵🄵有 `__init__.py` 這個檔案。

命名空間套件允許數個可獨立安裝的套件擁有一個共同的父套件。除此之外，建議使用*regular package*。

更多資訊，請參🄵 **PEP 420** 和 reference-namespace-package。

另請參🄵*module*（模組）。

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

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

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

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

package（套件）
一個 Python 的*module*（模組），它可以包含子模組 (submodule) 或是遞🄵的子套件 (subpackage)。技術上而言，套件就是具有 `__path__` 屬性的一個 Python 模組。

另請參🄵*regular package*（正規套件）和*namespace package*（命名空間套件）。

parameter（參數）
在*function*（函式）或 method 定義中的一個命名實體 (named entity)，它指明該函式能🄵接受的一個*argument*（引數），或在某些情🄵下指示多個引數。共有有五種不同的參數類型：

- *positional-or-keyword*（位置或關鍵字）：指明一個可以按照位置或是作🄵關鍵字引數被傳遞的引數。這是參數的預設類型，例如以下的 *foo* 和 *bar*：

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

- *positional-only*（僅限位置）：指明一個只能按照位置被提供的引數。在函式定義的參數列表中包含一個 / 字元，就可以在該字元前面定義僅限位置參數，例如以下的 *posonly1* 和 *posonly2*：

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

- *keyword-only*（僅限關鍵字）：指明一個只能以關鍵字被提供的引數。在函式定義的參數列表中，包含一個任意數量位置參數 (var-positional parameter) 或是單純的 `*` 字元，就可以在其後方定義僅限關鍵字參數，例如以下的 *kw_only1* 和 *kw_only2*：

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

- *var-positional*（任意數量位置）：指明一串能以任意序列被提供的位置引數（在已被其他參數接受的任何位置引數之外）。這類參數是透過在其參數名稱字首加上 `*` 來定義的，例如以下的 *args*：

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

- *var-keyword*（任意數量關鍵字）：指明可被提供的任意數量關鍵字引數（在已被其他參數接受的任何關鍵字引數之外）。這類參數是透過在其參數名稱字首加上 `**` 來定義的，例如上面範例中的 *kwargs*。

參數可以指明引數是選擇性的或必需的，也可以🄵一些選擇性的引數指定預設值。

另請參🄵術語表的*argument*（引數）條目、常見問題中的引數和參數之間的差🄵、`inspect.Parameter` class、function 章節，以及 **PEP 362**。

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

path entry finder（路徑項目尋檢器）
被 `sys.path_hooks` 中的一個可呼叫物件 (callable)（意即一個*path entry hook*）所回傳的一種*finder*，它知道如何以一個*path entry*定位模組。

關於路徑項目尋檢器實作的 method，請參🄵 `importlib.abc.PathEntryFinder`。

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

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

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

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

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

請參🄵 **PEP 1**。

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

positional argument（位置引數）
請參🄵*argument*（引數）。

provisional API（暫行 API）
暫行 API 是指，從標準函式庫的向後相容性 (backwards compatibility) 保證中，故意被排除的 API。雖然此類介面，只要它們被標示🄵暫行的，理論上🄵不會有重大的變更，但如果核心開發人員認

🄵有必要，也可能會出現向後不相容的變更（甚至包括移除該介面）。這種變更🄵不會無端地🄵生——只有 API 被納入之前未察覺的嚴重基本缺陷被揭露時，它們才會發生。

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

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

provisional package（暫行套件）
請參🄵*provisional API*（暫行 API）。

Python 3000
Python 3.x 系列版本的🄵稱（很久以前創造的，當時第 3 版的發布是在🄵遠的未來。）也可以縮寫🄵「Py3k」。

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

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

相較之下，以下方法更簡潔、更具有 Python 風格：

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

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

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

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

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

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

在*CPython* 中，參照計數不被視🄵穩定或明確定義的值；對物件的參照數量，以及該數量如何受到 Python 程式碼的影響，在不同版本之間可能會有所不同。

regular package（正規套件）
一個傳統的*package*（套件），例如一個包含 `__init__.py` 檔案的目🄵。

另請參🄵*namespace package*（命名空間套件）。

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

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

sequence（序列）
一個*iterable*（可🄵代物件），它透過 `__getitem__()` special method（特殊方法），使用整數索引來支援高效率的元素存取，🄵定義了一個 `__len__()` method 來回傳該序列的長度。一些🄵建序列型🄵包括 `list`、`str`、`tuple` 和 `bytes`。請注意，雖然 `dict` 也支援 `__getitem__()` 和 `__len__()`，但它被視🄵對映 (mapping) 而不是序列，因🄵其查找方式是使用任意的*hashable* 鍵，而不是整數。

抽象基底類🄵 (abstract base class) `collections.abc.Sequence` 定義了一個更加豐富的介面，🄵不僅止於 `__getitem__()` 和 `__len__()`，還增加了 `count()`、`index()`、`__contains__()` 和 `__reversed__()`。實作此擴充介面的型🄵，可以使用 `register()` 被明確地🄵🄵。更多關於序列方法的文件，請見常見序列操作。

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

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

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

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

與正常🄵用不同，軟性🄵用🄵有🄵除 API 的規劃，也不會發出警告。

請參🄵 PEP 387：軟性🄵用。

special method（特殊方法）
一種會被 Python 自動呼叫的 method，用於對某種型🄵執行某種運算，例如加法。這種 method 的名稱會在開頭和結尾有兩個下底🄵。Special method 在 specialnames 中有詳細🄵明。

標准函式庫
包含套件、模組和擴充模組的集合，它們是作🄵官方 Python 直譯器套件的一部分來發行。該集合的成員可能會因平台、可用的系統函式庫或其他條件而有所不同。相關文件可以在 library-index 中找到。

請參🄵 `sys.stdlib_module_names` 以取得所有可能的標准函式庫模組名稱的列表。

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

static type checker（🄵態型🄵檢查器）
會讀取 Python 程式碼🄵分析的外部工具，能🄵找出錯誤，像是使用了不正確的型🄵。另請參🄵型🄵提示 *(type hints)* 以及 `typing` 模組。

stdlib（標准函式庫）
standard library 的縮寫。

strong reference（🄵參照）
在 Python 的 C API 中，🄵參照是對物件的參照，該物件🄵持有該參照的程式碼所擁有。建立參照時透過呼叫*Py_INCREF()* 來獲得🄵參照、🄵除參照時透過*Py_DECREF()* 釋放🄵參照。

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

另請參🄵*borrowed reference*（借用參照）。

t-string（t 字串）
t-strings（t 字串）
以 `t` 或 `T` 🄵前綴的字串文本通常被稱🄵「t 字串」，它是模板化的字串文本的縮寫。

text encoding（文字編碼）
Python 中的字串是一個 Unicode 編碼位置 (code point) 的序列（範圍在 `U+0000` -- `U+10FFFF` 之間）。若要儲存或傳送一個字串，它必須被序列化🄵一個位元組序列。

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

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

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

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

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
原始碼的小單位，由 詞法分析器（也稱🄵 *tokenizer*）🄵生。名稱、數字、字串、運算子、🄵行符號等都以 token 表示。

`tokenize` 模組公開了 Python 的詞法分析器。`token` 模組包含各種 token 類型的資訊。

triple-quoted string（三引號🄵字串）
由三個雙引號 (") 或單引號 (') 的作🄵邊界的一個字串。雖然它們🄵🄵有提供🄵於單引號字串的任何額外功能，但基於許多原因，它們仍是很有用的。它們讓你可以在字串中包含未跳🄵 (unescaped) 的單引號和雙引號，而且它們不需使用連續字元 (continuation character) 就可以跨越多行，這使得它們在編寫🄵明字串時特🄵有用。

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

type alias（型🄵🄵名）
一個型🄵的同義詞，透過將型🄵指定給一個識🄵符 (identifier) 來建立。

型🄵🄵名對於簡化*型🄵提示 (type hint)* 很有用。例如：

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

可以寫成這樣，更具有可讀性：

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

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

請參🄵 `typing` 和 **PEP 484**，有此功能的描述。

type hint（型🄵提示）
一種*annotation*（🄵釋），它指定一個變數、一個 class 屬性或一個函式的參數或回傳值的預期型🄵。

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

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

請參🄵 `typing` 和 **PEP 484**，有此功能的描述。

universal newlines（通用🄵行字元）
一種解譯文字流 (text stream) 的方式，會將以下所有的情🄵識🄵🄵一行的結束：Unix 行尾慣例 `'\n'`、Windows 慣例 `'\r\n'` 和舊的 Macintosh 慣例 `'\r'`。請參🄵 **PEP 278** 和 **PEP 3116**，以及用於 `bytes.splitlines()` 的附加用途。

variable annotation（變數🄵釋）
一個變數或 class 屬性的*annotation*（🄵釋）。

🄵釋變數或 class 屬性時，賦值是選擇性的：

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

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

```
count: int = 0
```

變數🄵釋的語法在 annassign 章節有詳細的解釋。

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

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

另請參🄵 `venv`。

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

walrus operator（海象運算子）
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 of Python（Python 之🄵）
Python 設計原則與哲學的列表，其🄵容有助於理解和使用此語言。此列表可以透過在互動式提式字元後輸入「`import this`」來找到它。

APPENDIX B

關於這份🄵明文件

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

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

致謝：

- Fred L. Drake, Jr.，原始 Python 文件工具集的創造者以及一大部份🄵容的作者；
- 創造 reStructuredText 和 Docutils 工具組的 Docutils 專案；
- Fredrik Lundh 先生，Sphinx 從他的 Alternative Python Reference 計劃中獲得許多的好主意。

B.1 Python 文件的貢獻者們

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

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

APPENDIX C

沿革與授權

C.1 軟體沿革

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

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

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

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

發🄵版本	源自	年份	擁有者	GPL 相容性？(1)
0.9.0 至 1.2	不適用	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	是 (2)
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 以上	2.1.1	2001 至今	PSF	是

備🄵

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

在 GPL 下發🄵的軟體一起使用；但其它的授權則不行。

(2) 根據 Richard Stallman 的🄵法，1.6.1 不是 GPL 相容的，因🄵其授權有一個法律選擇條款。然而根據 CNRI 的🄵法，Stallman 的律師告訴 CNRI 的律師，1.6.1 與 GPL「不相容」。

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

C.2 關於存取或以其他方式使用 Python 的合約條款

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

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

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

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001 Python Software Foundation; All Rights
   Reserved" are retained in Python alone or in any derivative version
   prepared by Licensee.

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

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

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

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

7. Nothing in this License Agreement shall be deemed to create any relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This License
   Agreement does not grant permission to use PSF trademarks or trade name in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.

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

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

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

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

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

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

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

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

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

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

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

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

(繼續下一頁)

(繼續上一頁)

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

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

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

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

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

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

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

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

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

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

(繼續下一頁)

(繼續上一頁)

```
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 被收🄵軟體的授權與致謝

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

C.3.1 Mersenne Twister

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

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

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

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

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

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

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
```

(繼續下一頁)

(繼續上一頁)

```
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

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

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 非同步 socket 服務

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

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

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

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

(繼續下一頁)

(繼續上一頁)

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

C.3.4 Cookie 管理

`http.cookies` 模組包含以下聲明：

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

                All Rights Reserved

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

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 執行追🄵

`trace` 模組包含以下聲明：

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
```

(繼續下一頁)

(繼續上一頁)

```
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode 與 UUdecode 函式

uu 編解碼器包含以下聲明：

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 遠端程序呼叫

xmlrpc.client 模組包含以下聲明：

```
    The XML-RPC client interface is

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

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

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

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

(繼續下一頁)

(繼續上一頁)

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

C.3.8 test_epoll

`test.test_epoll` 模組包含以下聲明：

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

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

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

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

C.3.9 Select kqueue

`select` 模組對於 kqueue 介面包含以下聲明：

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

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

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

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

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

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

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

C.3.11 strtod 與 dtoa

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

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

如果 OpenSSL 函式庫可被作業系統使用，則 `hashlib`、`posix`、`ssl` 模組會使用它來提升效能。此外，因🄵 Windows 和 macOS 的 Python 安裝程式可能包含 OpenSSL 函式庫的副本，所以我們也在此收🄵 OpenSSL 授權的副本。對於 OpenSSL 3.0 版本以及由此衍生的更新版本則適用 Apache 許可證 v2：

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

(繼續下一頁)

(繼續上一頁)

```
TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

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

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

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

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

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

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

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

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

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

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

(繼續下一頁)

(繼續上一頁)

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

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

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

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

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

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

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

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
```

(繼續下一頁)

(繼續上一頁)

```
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

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

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

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

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

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

END OF TERMS AND CONDITIONS
```

C.3.13 expat

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

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper
```

(繼續下一頁)

(繼續上一頁)

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

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

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

C.3.14 libffi

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

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

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

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

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

C.3.15 zlib

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

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
```

(繼續下一頁)

(繼續上一頁)

```
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

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

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

This code is released under the BSD license:

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

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

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

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

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.
```

(繼續下一頁)

(繼續上一頁)

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

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

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N 測試套件

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

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

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

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT 授權：

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

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

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

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

C.3.20 asyncio

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

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

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

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

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

C.3.21 Global Unbounded Sequences (GUS)

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

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

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
```

(繼續下一頁)

(繼續上一頁)

```
   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
   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.
```

APPENDIX D

版權宣告

Python 和這份說明文件的版權:

Copyright © 2001 Python 軟體基金會。保留所有權利。

Copyright © 2000 BeOpen.com 保留所有權利。

Copyright © 1995-2000 Corporation for National Research Initiatives 保留所有權利。

Copyright © 1991-1995 Stichting Mathematisch Centrum 保留所有權利。

完整的授權條款資訊請參見沿革與授權。

Bibliography

[win] `PyExc_WindowsError` is only defined on Windows; protect code that uses this by testing that the preprocessor macro `MS_WINDOWS` is defined.

索引

非依字母順序

..., **365**
    🄵節號字面值, 365
>>>, **365**
__all__（套件變數）, 80
__dict__（模組屬性）, 200
__doc__（模組屬性）, 200
__file__（模組屬性）, 200
__future__, **371**
__import__
    built-in function（🄵建函式）, 81
__loader__（模組屬性）, 200
__main__
    module（模組）, 11
    模組, 229, 244, 245
__name__（模組屬性）, 200
__package__（模組屬性）, 200
__PYVENV_LAUNCHER__, 268, 274
__slots__, **379**
_frozen (*C struct*), 83
_inittab (*C struct*), 84
_inittab.initfunc (*C member*), 84
_inittab.name (*C member*), 84
_Py_c_diff (*C function*), 153
_Py_c_neg (*C function*), 153
_Py_c_pow (*C function*), 153
_Py_c_prod (*C function*), 153
_Py_c_quot (*C function*), 153
_Py_c_sum (*C function*), 152
_Py_NoneStruct (*C var*), 297
_PyBytes_Resize (*C function*), 156
_PyCode_GetExtra（C 函式）, 198
_PyCode_SetExtra（C 函式）, 198
_PyEval_RequestCodeExtraIndex（C 函式）, 198
_PyFrameEvalFunction (*C type*), 242
_PyInterpreterFrame (*C struct*), 216
_PyInterpreterState_GetEvalFrameFunc (*C function*), 242
_PyInterpreterState_SetEvalFrameFunc (*C function*), 242
_PyObject_GetDictPtr (*C function*), 107
_PyObject_New (*C function*), 295
_PyObject_NewVar (*C function*), 295
_PyTuple_Resize (*C function*), 180
_thread
    模組, 238
環境變數
    __PYVENV_LAUNCHER__, 268, 274
    PATH, 11
    PYTHON_CPU_COUNT, 272
    PYTHON_FROZEN_MODULES, 271
    PYTHON_GIL, 372
    PYTHON_PERF_JIT_SUPPORT, 276
    PYTHON_PRESITE, 275
    PYTHONCOERCECLOCALE, 279
    PYTHONDEBUG, 227, 273
    PYTHONDEVMODE, 269
    PYTHONDONTWRITEBYTECODE, 227, 277
    PYTHONDUMPREFS, 269
    PYTHONDUMPREFSFILE, 269
    PYTHONEXECUTABLE, 274
    PYTHONFAULTHANDLER, 270
    PYTHONHASHSEED, 227, 271
    PYTHONHOME, 11, 227, 235, 271
    PYTHONINSPECT, 227, 271
    PYTHONINTMAXSTRDIGITS, 272
    PYTHONIOENCODING, 275
    PYTHONLEGACYWINDOWSFSENCODING, 228, 265
    PYTHONLEGACYWINDOWSSTDIO, 228, 272
    PYTHONMALLOC, 284, 288, 290, 291
    PYTHONMALLOCSTATS, 272, 284
    PYTHONNODEBUGRANGES, 269
    PYTHONNOUSERSITE, 228, 276
    PYTHONOPTIMIZE, 228, 273
    PYTHONPATH, 11, 227, 273
    PYTHONPERFSUPPORT, 276
    PYTHONPLATLIBDIR, 272
    PYTHONPROFILEIMPORTTIME, 271
    PYTHONPYCACHEPREFIX, 274
    PYTHONSAFEPATH, 268
    PYTHONTRACEMALLOC, 276
    PYTHONUNBUFFERED, 229, 268
    PYTHONUTF8, 265, 279
    PYTHONVERBOSE, 229, 277
    PYTHONWARNINGS, 277

A

abort（C 函式）, 80
abs
    built-in function（🄵建函式）, 118
abstract base class（抽象基底類🄵）, **365**
allocfunc (*C type*), 346
annotate function（🄵釋函式）, **365**
annotation（🄵釋）, **365**
argument（引數）, **365**
argv (*in module sys*), 267
argv（sys 模組中）, 234
ascii
    bulit-in function（🄵建函式）, 107
asynchronous context manager（非同步情境管理器）, **366**
asynchronous generator iterator（非同步🄵生器🄵代器）, **366**
asynchronous generator（非同步🄵生器）, **366**
asynchronous iterable（非同步可🄵代物件）, **366**
asynchronous iterator（非同步🄵代器）, **366**
attached thread state, **366**
attribute（屬性）, **367**
awaitable（可等待物件）, **367**

B

BDFL, **367**
binary file（二進位檔案）, **367**
binaryfunc (*C type*), 347
borrowed reference（借用參照）, **367**
buffer interface（緩衝介面）
    （請見緩衝協定）, 124
buffer object（緩衝物件）
    （請見緩衝協定）, 124
buffer protocol（緩衝協定）, 124
built-in function（🄵建函式）
    __import__, 81
    abs, 118
    classmethod, 305
    compile（編譯）, 82
    divmod, 118
    float, 120
    hash（雜🄵）, 322
    int, 120
    len, 120, 122, 182, 186, 189
    pow, 118, 119
    repr, 321
    staticmethod, 305
    tuple（元組）, 122, 183
builtins（🄵建）
    module（模組）, 11
    模組, 229, 244, 245
bulit-in function（🄵建函式）
    ascii, 107
    bytes（位元組）, 108
    hash（雜🄵）, 108
    len, 109
    repr, 107
    type（型🄵）, 108
bytearray（位元組陣列）
    object（物件）, 156
bytecode（位元組碼）, **367**
bytes-like object（類位元組串物件）, **367**
bytes（位元組）
    bulit-in function（🄵建函式）, 108
    object（物件）, 154

C

callable（可呼叫物件）, **367**
callback（回呼）, **367**
calloc（C 函式）, 283
Capsule
    object（物件）, 212
C-contiguous（C 連續的）, 128, 368
class variable（類🄵變數）, **368**
classmethod
    built-in function（🄵建函式）, 305
class（類🄵）, **367**
cleanup functions（清理函式）, 80
close（os 模組中）, 245
closure variable（閉包變數）, **368**
CO_ASYNC_GENERATOR (*C macro*), 197
CO_COROUTINE (*C macro*), 197
CO_FUTURE_ABSOLUTE_IMPORT (*C macro*), 197
CO_FUTURE_ANNOTATIONS (*C macro*), 197
CO_FUTURE_DIVISION (*C macro*), 197
CO_FUTURE_GENERATOR_STOP (*C macro*), 197
CO_FUTURE_PRINT_FUNCTION (*C macro*), 197
CO_FUTURE_UNICODE_LITERALS (*C macro*), 197
CO_FUTURE_WITH_STATEMENT (*C macro*), 197
CO_GENERATOR (*C macro*), 197
CO_HAS_DOCSTRING (*C macro*), 197
CO_ITERABLE_COROUTINE (*C macro*), 197
CO_METHOD (*C macro*), 197
CO_NESTED (*C macro*), 197
CO_NEWLOCALS (*C macro*), 197
CO_OPTIMIZED (*C macro*), 197
CO_VARARGS (*C macro*), 197
CO_VARKEYWORDS (*C macro*), 197
code object（程式碼物件）, 193
Common Vulnerabilities and Exposures
    CVE 2008-5983, 234
compile（編譯）
    built-in function（🄵建函式）, 82
complex number（🄵數）, **368**
    object（物件）, 152
context management protocol（情境管理協定）, **368**
context manager（情境管理器）, **368**
context variable（情境變數）, **368**
context（情境）, **368**
contiguous（連續的）, 128, **368**
copyright（sys 模組中）, 234
coroutine function（協程函式）, **368**
coroutine（協程）, **368**
CPython, **369**

current context, **369**
cyclic isolate, **369**

D

decorator（裝飾器）, **369**
descrgetfunc (*C type*), 346
descriptor（描述器）, **369**
descrsetfunc (*C type*), 347
destructor (*C type*), 346
dictionary comprehension（字典綜合運算）, **369**
dictionary view（字典檢視）, **369**
dictionary（字典）, **369**
    object（物件）, 183
divmod
    built-in function（🄵建函式）, 118
docstring（🄵明字串）, **369**
duck-typing（鴨子型🄵）, **369**
dunder（雙底🄵）, **370**

E

EAFP, **370**
EOFError（🄵建例外）, 199
evaluate function（求值函式）, **370**
exc_info （sys 模組中）, 9
executable（sys 模組中）, 233
exit（C 函式）, 80
expression（運算式）, **370**
extension module（擴充模組）, **370**

F

f-strings（f 字串）, **370**
f-string（f 字串）, **370**
file object（檔案物件）, **370**
file-like object（類檔案物件）, **370**
filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）, **370**
file（檔案）
    object（物件）, 199
finder（尋檢器）, **370**
float
    built-in function（🄵建函式）, 120
floating-point（浮點）
    object（物件）, 150
floor division（向下取整除法）, **371**
Fortran contiguous（Fortran 連續的）, 128, 368
free threading（自由執行緒）, **371**
free variable（自由變數）, **371**
freefunc (*C type*), 346
free（C 函式）, 283
freeze utility（凍結工具）, 83
frozenset（凍結集合）
    object（物件）, 188
function annotation（函式🄵釋）, **371**
function（函式）, **371**
    object（物件）, 190

G

garbage collection（垃圾回收）, **371**
gcvisitobjects_t (*C type*), 354
generator expression（🄵生器運算式）, 371, **372**
generator iterator（🄵生器🄵代器）, **371**
generator（🄵生器）, **371**
generic function（泛型函式）, **372**
generic type（泛型型🄵）, **372**
getattrfunc (*C type*), 346
getattrofunc (*C type*), 346
getbufferproc (*C type*), 347
getiterfunc (*C type*), 347
getter (*C type*), 310
GIL, **372**
global interpreter lock（全域直譯器鎖）, 235, **372**

H

hash-based pyc（雜🄵架構的 pyc）, **372**
hashable（可雜🄵的）, **372**
hashfunc (*C type*), 347
hash（雜🄵）
    built-in function（🄵建函式）, 322
    bulit-in function（🄵建函式）, 108

I

IDLE, **372**
immortal（不滅）, **372**
immutable（不可變物件）, **373**
import path（引入路徑）, **373**
importer（引入器）, **373**
importing（引入）, **373**
incr_item(), 10, 11
initproc (*C type*), 346
inquiry (*C type*), 353
instancemethod
    object（物件）, 192
int
    built-in function（🄵建函式）, 120
integer（整數）
    object（物件）, 140
interactive（互動的）, **373**
interpreted（直譯的）, **373**
interpreter lock（直譯器鎖）, 235
interpreter shutdown（直譯器關閉）, **373**
iterable（可🄵代物件）, **373**
iterator（🄵代器）, **373**
iternextfunc (*C type*), 347

K

key function（鍵函式）, **374**
KeyboardInterrupt（🄵建例外）, 59
keyword argument（關鍵字引數）, **374**

L

lambda, **374**
LBYL, **374**
len
    built-in function（🄵建函式）, 120, 122, 182, 186, 189

    bulit-in function（🄵建函式）, 109
lenfunc (*C type*), 347
lexical analyzer（詞法分析器）, **374**
list comprehension（串列綜合運算）, **374**
list（串列）, **374**
    object（物件）, 181
loader（載入器）, **374**
locale encoding（區域編碼）, **374**
lock, interpreter（鎖、直譯器）, 235
long integer（長整數）
    object（物件）, 140
LONG_MAX（C 巨集）, 142

M

magic
    method（方法）, 375
magic method（魔術方法）, **375**
main(), 232, 234, 267
malloc（C 函式）, 283
mapping（對映）, **375**
    object（物件）, 183
memoryview（記憶體視圖）
    object（物件）, 210
meta path finder（元路徑尋檢器）, **375**
metaclass（元類🄵）, **375**
METH_CLASS (*C macro*), 305
METH_COEXIST (*C macro*), 305
METH_FASTCALL (*C macro*), 305
METH_KEYWORDS (*C macro*), 304
METH_METHOD (*C macro*), 305
METH_NOARGS (*C macro*), 305
METH_O (*C macro*), 305
METH_STATIC (*C macro*), 305
METH_VARARGS (*C macro*), 304
method resolution order（方法解析順序）, **375**
MethodType（types 模組中）, 190, 192, 200
method（方法）, **375**
    magic, 375
    object（物件）, 192
    special, 379
module spec（模組規格）, **375**
modules（sys 模組中）, 80, 229
module（模組）, **375**
    __main__, 11
    builtins（🄵建）, 11
    object（模組）, 200
    search（搜尋） path（路徑）, 11
    signal（訊號）, 59
    sys, 11
MRO, **375**
mutable（可變物件）, **375**

N

named tuple（附名元組）, **375**
namespace package（命名空間套件）, **376**
namespace（命名空間）, **376**
nested scope（巢狀作用域）, **376**
new-style class（新式類🄵）, **376**
newfunc (*C type*), 346
None
    object（物件）, 140
numeric（數值）
    object（物件）, 140

O

object（模組）
    module（模組）, 200
object（物件）, **376**
    bytearray（位元組陣列）, 156
    bytes（位元組）, 154
    Capsule, 212
    code（程式碼）, 193
    complex number（🄵數）, 152
    dictionary（字典）, 183
    file（檔案）, 199
    floating-point（浮點）, 150
    frozenset（凍結集合）, 188
    function（函式）, 190
    instancemethod, 192
    integer（整數）, 140
    list（串列）, 181
    long integer（長整數）, 140
    mapping（對映）, 183
    memoryview（記憶體視圖）, 210
    method（方法）, 192
    None, 140
    numeric（數值）, 140
    sequence（序列）, 154
    set（集合）, 188
    tuple（元組）, 179
    type（型🄵）, 6, 133
objobjargproc (*C type*), 347
objobjproc (*C type*), 347
optimized scope（最佳化作用域）, **376**
OverflowError（🄵建例外）, 142, 143

P

package variable（套件變數）
    __all__, 80
package（套件）, **376**
parameter（參數）, **376**
PATH, 11
path based finder（基於路徑的尋檢器）, **377**
path entry finder（路徑項目尋檢器）, **377**
path entry hook（路徑項目🄵）, **377**
path entry（路徑項目）, **377**
path-like object（類路徑物件）, **377**
path（sys 模組中）, 11, 229, 233
path（路徑）
    module（模組） search（搜尋）, 11
    模組 search（搜尋）, 229, 233
PEP, **377**
platform（sys 模組中）, 233
portion（部分）, **377**
positional argument（位置引數）, **377**
pow

built-in function（🄵建函式）, 118, 119
provisional API（暫行 API）, **377**
provisional package（暫行套件）, **378**
Py_ABS (*C macro*), 4
Py_AddPendingCall (*C function*), 246
Py_ALWAYS_INLINE (*C macro*), 4
Py_ASNATIVEBYTES_ALLOW_INDEX (*C macro*), 146
Py_ASNATIVEBYTES_BIG_ENDIAN (*C macro*), 146
Py_ASNATIVEBYTES_DEFAULTS (*C macro*), 146
Py_ASNATIVEBYTES_LITTLE_ENDIAN (*C macro*), 146
Py_ASNATIVEBYTES_NATIVE_ENDIAN (*C macro*), 146
Py_ASNATIVEBYTES_REJECT_NEGATIVE (*C macro*), 146
Py_ASNATIVEBYTES_UNSIGNED_BUFFER (*C macro*), 146
Py_AtExit (*C function*), 80
Py_AUDIT_READ (*C macro*), 307
Py_AuditHookFunction (*C type*), 80
Py_BEGIN_ALLOW_THREADS (*C macro*), 240
Py_BEGIN_ALLOW_THREADS（C 巨集）, 236
Py_BEGIN_CRITICAL_SECTION (*C macro*), 253
Py_BEGIN_CRITICAL_SECTION2 (*C macro*), 254
Py_BEGIN_CRITICAL_SECTION2_MUTEX (*C macro*), 254
Py_BEGIN_CRITICAL_SECTION_MUTEX (*C macro*), 253
Py_BLOCK_THREADS (*C macro*), 240
Py_buffer (*C type*), 125
Py_buffer.buf (*C member*), 125
Py_buffer.format (*C member*), 126
Py_buffer.internal (*C member*), 127
Py_buffer.itemsize (*C member*), 126
Py_buffer.len (*C member*), 126
Py_buffer.ndim (*C member*), 126
Py_buffer.obj (*C member*), 125
Py_buffer.readonly (*C member*), 126
Py_buffer.shape (*C member*), 126
Py_buffer.strides (*C member*), 126
Py_buffer.suboffsets (*C member*), 126
Py_BuildValue (*C function*), 92
Py_BytesMain (*C function*), 230
Py_BytesWarningFlag (*C var*), 226
Py_CHARMASK (*C macro*), 4
Py_CLEANUP_SUPPORTED (*C macro*), 89
Py_CLEAR (*C function*), 49
Py_CompileString (*C function*), 43
Py_CompileStringExFlags (*C function*), 43
Py_CompileStringFlags (*C function*), 43
Py_CompileStringObject (*C function*), 43
Py_CompileString（C 函式）, 44
Py_complex (*C type*), 152
Py_complex.imag (*C member*), 152
Py_complex.real (*C member*), 152
Py_CONSTANT_ELLIPSIS (*C macro*), 104
Py_CONSTANT_EMPTY_BYTES (*C macro*), 104
Py_CONSTANT_EMPTY_STR (*C macro*), 104
Py_CONSTANT_EMPTY_TUPLE (*C macro*), 104
Py_CONSTANT_FALSE (*C macro*), 104
Py_CONSTANT_NONE (*C macro*), 104
Py_CONSTANT_NOT_IMPLEMENTED (*C macro*), 104
Py_CONSTANT_ONE (*C macro*), 104
Py_CONSTANT_TRUE (*C macro*), 104
Py_CONSTANT_ZERO (*C macro*), 104
PY_CXX_CONST (*C macro*), 91
Py_DEBUG (*C macro*), 11
Py_DebugFlag (*C var*), 226
Py_DecodeLocale (*C function*), 76
Py_DECREF (*C function*), 48
Py_DecRef (*C function*), 49
Py_DECREF（C 函式）, 6
Py_DEPRECATED (*C macro*), 4
Py_DontWriteBytecodeFlag (*C var*), 227
Py_Ellipsis (*C var*), 210
Py_EncodeLocale (*C function*), 77
Py_END_ALLOW_THREADS (*C macro*), 240
Py_END_ALLOW_THREADS（C 巨集）, 236
Py_END_CRITICAL_SECTION (*C macro*), 254
Py_END_CRITICAL_SECTION2 (*C macro*), 254
Py_EndInterpreter (*C function*), 245
Py_EnterRecursiveCall (*C function*), 62
Py_EQ (*C macro*), 331
Py_eval_input (*C var*), 44
Py_Exit (*C function*), 80
Py_ExitStatusException (*C function*), 263
Py_False (*C var*), 150
Py_FatalError (*C function*), 80
Py_FatalError(), 234
Py_fclose (*C function*), 78
Py_FdIsInteractive (*C function*), 75
Py_file_input (*C var*), 44
Py_Finalize (*C function*), 230
Py_FinalizeEx (*C function*), 230
Py_FinalizeEx（C 函式）, 80, 229, 245, 246
Py_fopen (*C function*), 78
Py_FrozenFlag (*C var*), 227
Py_GE (*C macro*), 331
Py_GenericAlias (*C function*), 223
Py_GenericAliasType (*C var*), 223
Py_GetArgcArgv (*C function*), 281
Py_GetBuildInfo (*C function*), 234
Py_GetCompiler (*C function*), 234
Py_GetConstant (*C function*), 103
Py_GetConstantBorrowed (*C function*), 104
Py_GetCopyright (*C function*), 233
Py_GETENV (*C macro*), 5
Py_GetExecPrefix (*C function*), 232
Py_GetExecPrefix（C 函式）, 11
Py_GetPath (*C function*), 233
Py_GetPath(), 232
Py_GetPath（C 函式）, 11
Py_GetPlatform (*C function*), 233
Py_GetPrefix (*C function*), 232
Py_GetPrefix（C 函式）, 11
Py_GetProgramFullPath (*C function*), 233

Py_GetProgramFullPath（C 函式），11
Py_GetProgramName (*C function*), 232
Py_GetPythonHome (*C function*), 235
Py_GetVersion (*C function*), 233
Py_GT (*C macro*), 331
Py_hash_t (*C type*), 96
Py_HashBuffer (*C function*), 97
Py_HashPointer (*C function*), 96
Py_HashRandomizationFlag (*C var*), 227
Py_IgnoreEnvironmentFlag (*C var*), 227
Py_INCREF (*C function*), 47
Py_IncRef (*C function*), 49
Py_INCREF（C 函式），6
Py_Initialize (*C function*), 229
Py_Initialize(), 232
Py_InitializeEx (*C function*), 229
Py_InitializeFromConfig (*C function*), 229
Py_InitializeFromInitConfig (*C function*), 258
Py_Initialize（C 函式），11, 245
Py_InspectFlag (*C var*), 227
Py_InteractiveFlag (*C var*), 227
Py_Is (*C function*), 302
Py_IS_TYPE (*C function*), 302
Py_IsFalse (*C function*), 302
Py_IsFinalizing (*C function*), 230
Py_IsInitialized (*C function*), 230
Py_IsInitialized（C 函式），11
Py_IsNone (*C function*), 302
Py_IsolatedFlag (*C var*), 227
Py_IsTrue (*C function*), 302
Py_LE (*C macro*), 331
Py_LeaveRecursiveCall (*C function*), 62
Py_LegacyWindowsFSEncodingFlag (*C var*), 228
Py_LegacyWindowsStdioFlag (*C var*), 228
Py_LIMITED_API (*C macro*), 14
Py_LT (*C macro*), 331
Py_Main (*C function*), 230
PY_MAJOR_VERSION (*C macro*), 355
Py_MARSHAL_VERSION (*C macro*), 84
Py_MAX (*C macro*), 5
Py_MEMBER_SIZE (*C macro*), 5
PY_MICRO_VERSION (*C macro*), 355
Py_MIN (*C macro*), 5
PY_MINOR_VERSION (*C macro*), 355
Py_mod_create (*C macro*), 202
Py_mod_exec (*C macro*), 203
Py_mod_gil (*C macro*), 203
Py_MOD_GIL_NOT_USED (*C macro*), 203
Py_MOD_GIL_USED (*C macro*), 203
Py_mod_multiple_interpreters (*C macro*), 203
Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED (*C macro*), 203
Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED (*C macro*), 203
Py_MOD_PER_INTERPRETER_GIL_SUPPORTED (*C macro*), 203
PY_MONITORING_EVENT_BRANCH_LEFT (*C macro*), 362
PY_MONITORING_EVENT_BRANCH_RIGHT (*C macro*), 362
PY_MONITORING_EVENT_C_RAISE (*C macro*), 362
PY_MONITORING_EVENT_C_RETURN (*C macro*), 362
PY_MONITORING_EVENT_CALL (*C macro*), 362
PY_MONITORING_EVENT_EXCEPTION_HANDLED (*C macro*), 362
PY_MONITORING_EVENT_INSTRUCTION (*C macro*), 362
PY_MONITORING_EVENT_JUMP (*C macro*), 362
PY_MONITORING_EVENT_LINE (*C macro*), 362
PY_MONITORING_EVENT_PY_RESUME (*C macro*), 362
PY_MONITORING_EVENT_PY_RETURN (*C macro*), 362
PY_MONITORING_EVENT_PY_START (*C macro*), 362
PY_MONITORING_EVENT_PY_THROW (*C macro*), 362
PY_MONITORING_EVENT_PY_UNWIND (*C macro*), 362
PY_MONITORING_EVENT_PY_YIELD (*C macro*), 362
PY_MONITORING_EVENT_RAISE (*C macro*), 362
PY_MONITORING_EVENT_RERAISE (*C macro*), 362
PY_MONITORING_EVENT_STOP_ITERATION (*C macro*), 362
PY_MONITORING_IS_INSTRUMENTED_EVENT (*C function*), 363
Py_NE (*C macro*), 331
Py_NewInterpreter (*C function*), 245
Py_NewInterpreterFromConfig (*C function*), 244
Py_NewRef (*C function*), 48
Py_NO_INLINE (*C macro*), 5
Py_None (*C var*), 140
Py_NoSiteFlag (*C var*), 228
Py_NotImplemented (*C var*), 104
Py_NoUserSiteDirectory (*C var*), 228
Py_OpenCodeHookFunction (*C type*), 199
Py_OptimizeFlag (*C var*), 228
Py_PACK_FULL_VERSION (*C function*), 356
Py_PACK_VERSION (*C function*), 356
Py_PreInitialize (*C function*), 265
Py_PreInitializeFromArgs (*C function*), 265
Py_PreInitializeFromBytesArgs (*C function*), 265
Py_PRINT_RAW (*C macro*), 104
Py_PRINT_RAW（C 巨集），199
Py_QuietFlag (*C var*), 229
Py_READONLY (*C macro*), 307
Py_REFCNT (*C function*), 47
Py_RELATIVE_OFFSET (*C macro*), 307
PY_RELEASE_LEVEL (*C macro*), 355
PY_RELEASE_SERIAL (*C macro*), 355
Py_ReprEnter (*C function*), 63
Py_ReprLeave (*C function*), 63
Py_RETURN_FALSE (*C macro*), 150
Py_RETURN_NONE (*C macro*), 140
Py_RETURN_NOTIMPLEMENTED (*C macro*), 104
Py_RETURN_RICHCOMPARE (*C macro*), 331
Py_RETURN_TRUE (*C macro*), 150
Py_RunMain (*C function*), 231
Py_SET_REFCNT (*C function*), 47
Py_SET_SIZE (*C function*), 303

Py_SET_TYPE (*C function*), 302
Py_SetProgramName (*C function*), 232
Py_SetPythonHome (*C function*), 235
Py_SETREF (*C macro*), 49
Py_single_input (*C var*), 44
Py_SIZE (*C function*), 303
Py_ssize_t (*C type*), 9
PY_SSIZE_T_MAX（C 巨集）, 143
Py_STRINGIFY (*C macro*), 5
Py_T_BOOL (*C macro*), 309
Py_T_BYTE (*C macro*), 309
Py_T_CHAR (*C macro*), 309
Py_T_DOUBLE (*C macro*), 309
Py_T_FLOAT (*C macro*), 309
Py_T_INT (*C macro*), 309
Py_T_LONG (*C macro*), 309
Py_T_LONGLONG (*C macro*), 309
Py_T_OBJECT_EX (*C macro*), 309
Py_T_PYSSIZET (*C macro*), 309
Py_T_SHORT (*C macro*), 309
Py_T_STRING (*C macro*), 309
Py_T_STRING_INPLACE (*C macro*), 309
Py_T_UBYTE (*C macro*), 309
Py_T_UINT (*C macro*), 309
Py_T_ULONG (*C macro*), 309
Py_T_ULONGLONG (*C macro*), 309
Py_T_USHORT (*C macro*), 309
Py_tp_token (*C macro*), 140
Py_TP_USE_SPEC (*C macro*), 140
Py_TPFLAGS_BASE_EXC_SUBCLASS (*C macro*), 326
Py_TPFLAGS_BASETYPE (*C macro*), 324
Py_TPFLAGS_BYTES_SUBCLASS (*C macro*), 326
Py_TPFLAGS_DEFAULT (*C macro*), 324
Py_TPFLAGS_DICT_SUBCLASS (*C macro*), 326
Py_TPFLAGS_DISALLOW_INSTANTIATION (*C macro*), 326
Py_TPFLAGS_HAVE_FINALIZE (*C macro*), 326
Py_TPFLAGS_HAVE_GC (*C macro*), 324
Py_TPFLAGS_HAVE_VECTORCALL (*C macro*), 326
Py_TPFLAGS_HEAPTYPE (*C macro*), 324
Py_TPFLAGS_IMMUTABLETYPE (*C macro*), 326
Py_TPFLAGS_ITEMS_AT_END (*C macro*), 325
Py_TPFLAGS_LIST_SUBCLASS (*C macro*), 326
Py_TPFLAGS_LONG_SUBCLASS (*C macro*), 325
Py_TPFLAGS_MANAGED_DICT (*C macro*), 325
Py_TPFLAGS_MANAGED_WEAKREF (*C macro*), 325
Py_TPFLAGS_MAPPING (*C macro*), 327
Py_TPFLAGS_METHOD_DESCRIPTOR (*C macro*), 325
Py_TPFLAGS_READY (*C macro*), 324
Py_TPFLAGS_READYING (*C macro*), 324
Py_TPFLAGS_SEQUENCE (*C macro*), 327
Py_TPFLAGS_TUPLE_SUBCLASS (*C macro*), 326
Py_TPFLAGS_TYPE_SUBCLASS (*C macro*), 326
Py_TPFLAGS_UNICODE_SUBCLASS (*C macro*), 326
Py_TPFLAGS_VALID_VERSION_TAG (*C macro*), 327
Py_tracefunc (*C type*), 247
Py_True (*C var*), 150
Py_tss_NEEDS_INIT (*C macro*), 250
Py_tss_t (*C type*), 250
Py_TYPE (*C function*), 302
Py_UCS1 (*C type*), 157
Py_UCS2 (*C type*), 157
Py_UCS4 (*C type*), 157
Py_uhash_t (*C type*), 96
Py_UNBLOCK_THREADS (*C macro*), 240
Py_UnbufferedStdioFlag (*C var*), 229
Py_UNICODE (*C type*), 178
Py_UNICODE_IS_HIGH_SURROGATE (*C function*), 160
Py_UNICODE_IS_LOW_SURROGATE (*C function*), 160
Py_UNICODE_IS_SURROGATE (*C function*), 160
Py_UNICODE_ISALNUM (*C function*), 160
Py_UNICODE_ISALPHA (*C function*), 160
Py_UNICODE_ISDECIMAL (*C function*), 159
Py_UNICODE_ISDIGIT (*C function*), 159
Py_UNICODE_ISLINEBREAK (*C function*), 159
Py_UNICODE_ISLOWER (*C function*), 159
Py_UNICODE_ISNUMERIC (*C function*), 160
Py_UNICODE_ISPRINTABLE (*C function*), 160
Py_UNICODE_ISSPACE (*C function*), 159
Py_UNICODE_ISTITLE (*C function*), 159
Py_UNICODE_ISUPPER (*C function*), 159
Py_UNICODE_JOIN_SURROGATES (*C function*), 160
Py_UNICODE_REPLACEMENT_CHARACTER (*C macro*), 169
Py_UNICODE_TODECIMAL (*C function*), 160
Py_UNICODE_TODIGIT (*C function*), 160
Py_UNICODE_TOLOWER (*C function*), 160
Py_UNICODE_TONUMERIC (*C function*), 160
Py_UNICODE_TOTITLE (*C function*), 160
Py_UNICODE_TOUPPER (*C function*), 160
Py_UNREACHABLE (*C macro*), 5
Py_UNUSED (*C macro*), 5
Py_VaBuildValue (*C function*), 94
PY_VECTORCALL_ARGUMENTS_OFFSET (*C macro*), 114
Py_VerboseFlag (*C var*), 229
Py_Version (*C var*), 355
PY_VERSION_HEX (*C macro*), 355
Py_VISIT (*C macro*), 352
Py_XDECREF (*C function*), 48
Py_XDECREF（C 函式）, 11
Py_XINCREF (*C function*), 48
Py_XNewRef (*C function*), 48
Py_XSETREF (*C macro*), 49
PyAIter_Check (*C function*), 124
PyAnySet_Check (*C function*), 188
PyAnySet_CheckExact (*C function*), 189
PyArg_Parse (*C function*), 90
PyArg_ParseTuple (*C function*), 90
PyArg_ParseTupleAndKeywords (*C function*), 90
PyArg_UnpackTuple (*C function*), 91
PyArg_ValidateKeywordArguments (*C function*), 90
PyArg_VaParse (*C function*), 90
PyArg_VaParseTupleAndKeywords (*C function*), 90
PyASCIIObject (*C type*), 158

PyAsyncMethods (*C type*), 345
PyAsyncMethods.am_aiter (*C member*), 345
PyAsyncMethods.am_anext (*C member*), 345
PyAsyncMethods.am_await (*C member*), 345
PyAsyncMethods.am_send (*C member*), 346
PyBaseObject_Type (*C var*), 302
PyBool_Check (*C function*), 150
PyBool_FromLong (*C function*), 150
PyBool_Type (*C var*), 150
PyBUF_ANY_CONTIGUOUS (*C macro*), 128
PyBUF_C_CONTIGUOUS (*C macro*), 128
PyBUF_CONTIG (*C macro*), 129
PyBUF_CONTIG_RO (*C macro*), 129
PyBUF_F_CONTIGUOUS (*C macro*), 128
PyBUF_FORMAT (*C macro*), 127
PyBUF_FULL (*C macro*), 129
PyBUF_FULL_RO (*C macro*), 129
PyBUF_INDIRECT (*C macro*), 128
PyBUF_MAX_NDIM (*C macro*), 127
PyBUF_ND (*C macro*), 128
PyBUF_READ (*C macro*), 210
PyBUF_RECORDS (*C macro*), 129
PyBUF_RECORDS_RO (*C macro*), 129
PyBUF_SIMPLE (*C macro*), 128
PyBUF_STRIDED (*C macro*), 129
PyBUF_STRIDED_RO (*C macro*), 129
PyBUF_STRIDES (*C macro*), 128
PyBUF_WRITABLE (*C macro*), 127
PyBUF_WRITE (*C macro*), 210
PyBuffer_FillContiguousStrides (*C function*), 131
PyBuffer_FillInfo (*C function*), 131
PyBuffer_FromContiguous (*C function*), 131
PyBuffer_GetPointer (*C function*), 131
PyBuffer_IsContiguous (*C function*), 131
PyBuffer_Release (*C function*), 130
PyBuffer_SizeFromFormat (*C function*), 130
PyBuffer_ToContiguous (*C function*), 131
PyBufferProcs (*C type*), 344
PyBufferProcs.bf_getbuffer (*C member*), 344
PyBufferProcs.bf_releasebuffer (*C member*), 345
PyBufferProcs (C 型🄵) , 125
PyByteArray_AS_STRING (*C function*), 157
PyByteArray_AsString (*C function*), 157
PyByteArray_Check (*C function*), 156
PyByteArray_CheckExact (*C function*), 156
PyByteArray_Concat (*C function*), 156
PyByteArray_FromObject (*C function*), 156
PyByteArray_FromStringAndSize (*C function*), 156
PyByteArray_GET_SIZE (*C function*), 157
PyByteArray_Resize (*C function*), 157
PyByteArray_Size (*C function*), 157
PyByteArray_Type (*C var*), 156
PyByteArrayObject (*C type*), 156
PyBytes_AS_STRING (*C function*), 155
PyBytes_AsString (*C function*), 155
PyBytes_AsStringAndSize (*C function*), 155
PyBytes_Check (*C function*), 154
PyBytes_CheckExact (*C function*), 154
PyBytes_Concat (*C function*), 155
PyBytes_ConcatAndDel (*C function*), 156
PyBytes_FromFormat (*C function*), 154
PyBytes_FromFormatV (*C function*), 155
PyBytes_FromObject (*C function*), 155
PyBytes_FromString (*C function*), 154
PyBytes_FromStringAndSize (*C function*), 154
PyBytes_GET_SIZE (*C function*), 155
PyBytes_Join (*C function*), 156
PyBytes_Size (*C function*), 155
PyBytes_Type (*C var*), 154
PyBytesObject (*C type*), 154
PyCallable_Check (*C function*), 117
PyCallIter_Check (*C function*), 208
PyCallIter_New (*C function*), 208
PyCallIter_Type (*C var*), 208
PyCapsule (*C type*), 212
PyCapsule_CheckExact (*C function*), 212
PyCapsule_Destructor (*C type*), 212
PyCapsule_GetContext (*C function*), 213
PyCapsule_GetDestructor (*C function*), 213
PyCapsule_GetName (*C function*), 213
PyCapsule_GetPointer (*C function*), 213
PyCapsule_Import (*C function*), 213
PyCapsule_IsValid (*C function*), 213
PyCapsule_New (*C function*), 213
PyCapsule_SetContext (*C function*), 214
PyCapsule_SetDestructor (*C function*), 214
PyCapsule_SetName (*C function*), 214
PyCapsule_SetPointer (*C function*), 214
PyCell_Check (*C function*), 193
PyCell_GET (*C function*), 193
PyCell_Get (*C function*), 193
PyCell_New (*C function*), 193
PyCell_SET (*C function*), 193
PyCell_Set (*C function*), 193
PyCell_Type (*C var*), 193
PyCellObject (*C type*), 193
PyCF_ALLOW_TOP_LEVEL_AWAIT (*C macro*), 44
PyCF_ONLY_AST (*C macro*), 44
PyCF_OPTIMIZED_AST (*C macro*), 44
PyCF_TYPE_COMMENTS (*C macro*), 44
PyCFunction (*C type*), 303
PyCFunction_New (*C function*), 306
PyCFunction_NewEx (*C function*), 306
PyCFunctionFast (*C type*), 303
PyCFunctionFastWithKeywords (*C type*), 304
PyCFunctionWithKeywords (*C type*), 303
PyCMethod (*C type*), 304
PyCMethod_New (*C function*), 306
PyCode_Addr2Line (*C function*), 195
PyCode_Addr2Location (*C function*), 195
PyCode_AddWatcher (*C function*), 195
PyCode_Check (*C function*), 194
PyCode_ClearWatcher (*C function*), 195

PyCode_GetCellvars (*C function*), 195
PyCode_GetCode (*C function*), 195
PyCode_GetFreevars (*C function*), 195
PyCode_GetNumFree (*C function*), 194
PyCode_GetVarnames (*C function*), 195
PyCode_NewEmpty (*C function*), 195
PyCode_NewWithPosOnlyArgs（C 函式）, 195
PyCode_New（C 函式）, 194
PyCode_Type (*C var*), 194
PyCode_WatchCallback (*C type*), 196
PyCodec_BackslashReplaceErrors (*C function*), 99
PyCodec_Decode (*C function*), 98
PyCodec_Decoder (*C function*), 99
PyCodec_Encode (*C function*), 98
PyCodec_Encoder (*C function*), 99
PyCodec_IgnoreErrors (*C function*), 99
PyCodec_IncrementalDecoder (*C function*), 99
PyCodec_IncrementalEncoder (*C function*), 99
PyCodec_KnownEncoding (*C function*), 98
PyCodec_LookupError (*C function*), 99
PyCodec_NameReplaceErrors (*C function*), 99
PyCodec_Register (*C function*), 98
PyCodec_RegisterError (*C function*), 99
PyCodec_ReplaceErrors (*C function*), 99
PyCodec_StreamReader (*C function*), 99
PyCodec_StreamWriter (*C function*), 99
PyCodec_StrictErrors (*C function*), 99
PyCodec_Unregister (*C function*), 98
PyCodec_XMLCharRefReplaceErrors (*C function*), 99
PyCodeEvent (*C type*), 196
PyCodeObject (*C type*), 193
PyCompactUnicodeObject (*C type*), 158
PyCompilerFlags (*C struct*), 44
PyCompilerFlags.cf_feature_version (*C member*), 44
PyCompilerFlags.cf_flags (*C member*), 44
PyComplex_AsCComplex (*C function*), 154
PyComplex_Check (*C function*), 153
PyComplex_CheckExact (*C function*), 153
PyComplex_FromCComplex (*C function*), 153
PyComplex_FromDoubles (*C function*), 153
PyComplex_ImagAsDouble (*C function*), 153
PyComplex_RealAsDouble (*C function*), 153
PyComplex_Type (*C var*), 153
PyComplexObject (*C type*), 153
PyConfig (*C type*), 266
PyConfig_Clear (*C function*), 267
PyConfig_Get (*C function*), 260
PyConfig_GetInt (*C function*), 260
PyConfig_InitIsolatedConfig (*C function*), 266
PyConfig_InitPythonConfig (*C function*), 266
PyConfig_Names (*C function*), 260
PyConfig._pystats (*C member*), 277
PyConfig_Read (*C function*), 266
PyConfig_Set (*C function*), 260
PyConfig_SetArgv (*C function*), 266
PyConfig_SetBytesArgv (*C function*), 266
PyConfig_SetBytesString (*C function*), 266
PyConfig_SetString (*C function*), 266
PyConfig_SetWideStringList (*C function*), 266
PyConfig.argv (*C member*), 267
PyConfig.base_exec_prefix (*C member*), 268
PyConfig.base_executable (*C member*), 268
PyConfig.base_prefix (*C member*), 268
PyConfig.buffered_stdio (*C member*), 268
PyConfig.bytes_warning (*C member*), 268
PyConfig.check_hash_pycs_mode (*C member*), 269
PyConfig.code_debug_ranges (*C member*), 268
PyConfig.configure_c_stdio (*C member*), 269
PyConfig.cpu_count (*C member*), 272
PyConfig.dev_mode (*C member*), 269
PyConfig.dump_refs (*C member*), 269
PyConfig.dump_refs_file (*C member*), 269
PyConfig.exec_prefix (*C member*), 269
PyConfig.executable (*C member*), 270
PyConfig.faulthandler (*C member*), 270
PyConfig.filesystem_encoding (*C member*), 270
PyConfig.filesystem_errors (*C member*), 270
PyConfig.hash_seed (*C member*), 271
PyConfig.home (*C member*), 271
PyConfig.import_time (*C member*), 271
PyConfig.inspect (*C member*), 271
PyConfig.install_signal_handlers (*C member*), 271
PyConfig.int_max_str_digits (*C member*), 271
PyConfig.interactive (*C member*), 271
PyConfig.isolated (*C member*), 272
PyConfig.legacy_windows_stdio (*C member*), 272
PyConfig.malloc_stats (*C member*), 272
PyConfig.module_search_paths (*C member*), 273
PyConfig.module_search_paths_set (*C member*), 273
PyConfig.optimization_level (*C member*), 273
PyConfig.orig_argv (*C member*), 273
PyConfig.parse_argv (*C member*), 273
PyConfig.parser_debug (*C member*), 273
PyConfig.pathconfig_warnings (*C member*), 274
PyConfig.perf_profiling (*C member*), 276
PyConfig.platlibdir (*C member*), 272
PyConfig.prefix (*C member*), 274
PyConfig.program_name (*C member*), 274
PyConfig.pycache_prefix (*C member*), 274
PyConfig.pythonpath_env (*C member*), 273
PyConfig.quiet (*C member*), 274
PyConfig.run_command (*C member*), 274
PyConfig.run_filename (*C member*), 274
PyConfig.run_module (*C member*), 275
PyConfig.run_presite (*C member*), 275
PyConfig.safe_path (*C member*), 267
PyConfig.show_ref_count (*C member*), 275
PyConfig.site_import (*C member*), 275

PyConfig.skip_source_first_line (*C member*), 275
PyConfig.stdio_encoding (*C member*), 275
PyConfig.stdio_errors (*C member*), 275
PyConfig.stdlib_dir (*C member*), 276
PyConfig.tracemalloc (*C member*), 276
PyConfig.use_environment (*C member*), 276
PyConfig.use_frozen_modules (*C member*), 270
PyConfig.use_hash_seed (*C member*), 271
PyConfig.use_system_logger (*C member*), 276
PyConfig.user_site_directory (*C member*), 276
PyConfig.verbose (*C member*), 277
PyConfig.warn_default_encoding (*C member*), 268
PyConfig.warnoptions (*C member*), 277
PyConfig.write_bytecode (*C member*), 277
PyConfig.xoptions (*C member*), 277
PyContext (*C type*), 218
PyContext_AddWatcher (*C function*), 218
PyContext_CheckExact (*C function*), 218
PyContext_ClearWatcher (*C function*), 218
PyContext_Copy (*C function*), 218
PyContext_CopyCurrent (*C function*), 218
PyContext_Enter (*C function*), 218
PyContext_Exit (*C function*), 218
PyContext_New (*C function*), 218
PyContext_Type (*C var*), 218
PyContext_WatchCallback (*C type*), 219
PyContextEvent (*C type*), 219
PyContextToken (*C type*), 218
PyContextToken_CheckExact (*C function*), 218
PyContextToken_Type (*C var*), 218
PyContextVar (*C type*), 218
PyContextVar_CheckExact (*C function*), 218
PyContextVar_Get (*C function*), 219
PyContextVar_New (*C function*), 219
PyContextVar_Reset (*C function*), 219
PyContextVar_Set (*C function*), 219
PyContextVar_Type (*C var*), 218
PyCoro_CheckExact (*C function*), 217
PyCoro_New (*C function*), 217
PyCoro_Type (*C var*), 217
PyCoroObject (*C type*), 217
PyDate_Check (*C function*), 220
PyDate_CheckExact (*C function*), 220
PyDate_FromDate (*C function*), 221
PyDate_FromTimestamp (*C function*), 223
PyDateTime_Check (*C function*), 220
PyDateTime_CheckExact (*C function*), 220
PyDateTime_Date (*C type*), 219
PyDateTime_DATE_GET_FOLD (*C function*), 222
PyDateTime_DATE_GET_HOUR (*C function*), 221
PyDateTime_DATE_GET_MICROSECOND (*C function*), 222
PyDateTime_DATE_GET_MINUTE (*C function*), 222
PyDateTime_DATE_GET_SECOND (*C function*), 222
PyDateTime_DATE_GET_TZINFO (*C function*), 222
PyDateTime_DateTime (*C type*), 219
PyDateTime_DateTimeType (*C var*), 220
PyDateTime_DateType (*C var*), 220
PyDateTime_Delta (*C type*), 219
PyDateTime_DELTA_GET_DAYS (*C function*), 222
PyDateTime_DELTA_GET_MICROSECONDS (*C function*), 222
PyDateTime_DELTA_GET_SECONDS (*C function*), 222
PyDateTime_DeltaType (*C var*), 220
PyDateTime_FromDateAndTime (*C function*), 221
PyDateTime_FromDateAndTimeAndFold (*C function*), 221
PyDateTime_FromTimestamp (*C function*), 222
PyDateTime_GET_DAY (*C function*), 221
PyDateTime_GET_MONTH (*C function*), 221
PyDateTime_GET_YEAR (*C function*), 221
PyDateTime_Time (*C type*), 219
PyDateTime_TIME_GET_FOLD (*C function*), 222
PyDateTime_TIME_GET_HOUR (*C function*), 222
PyDateTime_TIME_GET_MICROSECOND (*C function*), 222
PyDateTime_TIME_GET_MINUTE (*C function*), 222
PyDateTime_TIME_GET_SECOND (*C function*), 222
PyDateTime_TIME_GET_TZINFO (*C function*), 222
PyDateTime_TimeType (*C var*), 220
PyDateTime_TimeZone_UTC (*C var*), 220
PyDateTime_TZInfoType (*C var*), 220
PyDelta_Check (*C function*), 220
PyDelta_CheckExact (*C function*), 220
PyDelta_FromDSU (*C function*), 221
PyDescr_IsData (*C function*), 208
PyDescr_NewClassMethod (*C function*), 208
PyDescr_NewGetSet (*C function*), 208
PyDescr_NewMember (*C function*), 208
PyDescr_NewMethod (*C function*), 208
PyDescr_NewWrapper (*C function*), 208
PyDict_AddWatcher (*C function*), 187
PyDict_Check (*C function*), 183
PyDict_CheckExact (*C function*), 183
PyDict_Clear (*C function*), 184
PyDict_ClearWatcher (*C function*), 187
PyDict_Contains (*C function*), 184
PyDict_ContainsString (*C function*), 184
PyDict_Copy (*C function*), 184
PyDict_DelItem (*C function*), 184
PyDict_DelItemString (*C function*), 184
PyDict_GetItem (*C function*), 184
PyDict_GetItemRef (*C function*), 184
PyDict_GetItemString (*C function*), 185
PyDict_GetItemStringRef (*C function*), 185
PyDict_GetItemWithError (*C function*), 185
PyDict_Items (*C function*), 186
PyDict_Keys (*C function*), 186
PyDict_Merge (*C function*), 187
PyDict_MergeFromSeq2 (*C function*), 187
PyDict_New (*C function*), 183
PyDict_Next (*C function*), 186
PyDict_Pop (*C function*), 185
PyDict_PopString (*C function*), 185

PyDict_SetDefault (*C function*), 185
PyDict_SetDefaultRef (*C function*), 185
PyDict_SetItem (*C function*), 184
PyDict_SetItemString (*C function*), 184
PyDict_Size (*C function*), 186
PyDict_Type (*C var*), 183
PyDict_Unwatch (*C function*), 187
PyDict_Update (*C function*), 187
PyDict_Values (*C function*), 186
PyDict_Watch (*C function*), 187
PyDict_WatchCallback (*C type*), 187
PyDict_WatchEvent (*C type*), 187
PyDictObject (*C type*), 183
PyDictProxy_New (*C function*), 183
PyDoc_STR (*C macro*), 6
PyDoc_STRVAR (*C macro*), 5
PyEllipsis_Type (*C var*), 210
PyErr_BadArgument (*C function*), 53
PyErr_BadInternalCall (*C function*), 54
PyErr_CheckSignals (*C function*), 59
PyErr_Clear (*C function*), 51
PyErr_Clear（C 函式）, 9, 11
PyErr_DisplayException (*C function*), 52
PyErr_ExceptionMatches (*C function*), 56
PyErr_ExceptionMatches（C 函式）, 11
PyErr_Fetch (*C function*), 56
PyErr_Format (*C function*), 52
PyErr_FormatUnraisable (*C function*), 52
PyErr_FormatV (*C function*), 52
PyErr_GetExcInfo (*C function*), 58
PyErr_GetHandledException (*C function*), 57
PyErr_GetRaisedException (*C function*), 56
PyErr_GivenExceptionMatches (*C function*), 56
PyErr_NewException (*C function*), 60
PyErr_NewExceptionWithDoc (*C function*), 60
PyErr_NoMemory (*C function*), 53
PyErr_NormalizeException (*C function*), 57
PyErr_Occurred (*C function*), 55
PyErr_Occurred（C 函式）, 9
PyErr_Print (*C function*), 52
PyErr_PrintEx (*C function*), 51
PyErr_ResourceWarning (*C function*), 55
PyErr_Restore (*C function*), 57
PyErr_SetExcFromWindowsErr (*C function*), 53
PyErr_SetExcFromWindowsErrWithFilename (*C function*), 54
PyErr_SetExcFromWindowsErrWithFilenameObject (*C function*), 54
PyErr_SetExcFromWindowsErrWithFilenameObjects (*C function*), 54
PyErr_SetExcInfo (*C function*), 58
PyErr_SetFromErrno (*C function*), 53
PyErr_SetFromErrnoWithFilename (*C function*), 53
PyErr_SetFromErrnoWithFilenameObject (*C function*), 53
PyErr_SetFromErrnoWithFilenameObjects (*C function*), 53
PyErr_SetFromWindowsErr (*C function*), 53
PyErr_SetFromWindowsErrWithFilename (*C function*), 53
PyErr_SetHandledException (*C function*), 58
PyErr_SetImportError (*C function*), 54
PyErr_SetImportErrorSubclass (*C function*), 54
PyErr_SetInterrupt (*C function*), 59
PyErr_SetInterruptEx (*C function*), 59
PyErr_SetNone (*C function*), 53
PyErr_SetObject (*C function*), 52
PyErr_SetRaisedException (*C function*), 56
PyErr_SetString (*C function*), 52
PyErr_SetString（C 函式）, 9
PyErr_SyntaxLocation (*C function*), 54
PyErr_SyntaxLocationEx (*C function*), 54
PyErr_SyntaxLocationObject (*C function*), 54
PyErr_WarnEx (*C function*), 55
PyErr_WarnExplicit (*C function*), 55
PyErr_WarnExplicitObject (*C function*), 55
PyErr_WarnFormat (*C function*), 55
PyErr_WriteUnraisable (*C function*), 52
PyEval_AcquireThread (*C function*), 243
PyEval_AcquireThread(), 238
PyEval_EvalCode (*C function*), 43
PyEval_EvalCodeEx (*C function*), 44
PyEval_EvalFrame (*C function*), 44
PyEval_EvalFrameEx (*C function*), 44
PyEval_GetBuiltins (*C function*), 97
PyEval_GetFrame (*C function*), 97
PyEval_GetFrameBuiltins (*C function*), 98
PyEval_GetFrameGlobals (*C function*), 98
PyEval_GetFrameLocals (*C function*), 98
PyEval_GetFuncDesc (*C function*), 98
PyEval_GetFuncName (*C function*), 98
PyEval_GetGlobals (*C function*), 97
PyEval_GetLocals (*C function*), 97
PyEval_InitThreads (*C function*), 238
PyEval_InitThreads(), 229
PyEval_MergeCompilerFlags (*C function*), 44
PyEval_ReleaseThread (*C function*), 243
PyEval_ReleaseThread(), 238
PyEval_RestoreThread (*C function*), 238
PyEval_RestoreThread(), 238
PyEval_RestoreThread（C 函式）, 236
PyEval_SaveThread (*C function*), 238
PyEval_SaveThread(), 238
PyEval_SaveThread（C 函式）, 236
PyEval_SetProfile (*C function*), 248
PyEval_SetProfileAllThreads (*C function*), 248
PyEval_SetTrace (*C function*), 248
PyEval_SetTraceAllThreads (*C function*), 248
PyExc_ArithmeticError (*C var*), 63
PyExc_AssertionError (*C var*), 63
PyExc_AttributeError (*C var*), 64
PyExc_BaseException (*C var*), 63
PyExc_BaseExceptionGroup (*C var*), 63
PyExc_BlockingIOError (*C var*), 64
PyExc_BrokenPipeError (*C var*), 64

PyExc_BufferError (*C var*), 64
PyExc_BytesWarning (*C var*), 69
PyExc_ChildProcessError (*C var*), 64
PyExc_ConnectionAbortedError (*C var*), 64
PyExc_ConnectionError (*C var*), 64
PyExc_ConnectionRefusedError (*C var*), 64
PyExc_ConnectionResetError (*C var*), 64
PyExc_DeprecationWarning (*C var*), 69
PyExc_EncodingWarning (*C var*), 69
PyExc_EnvironmentError (*C var*), 68
PyExc_EOFError (*C var*), 64
PyExc_Exception (*C var*), 63
PyExc_FileExistsError (*C var*), 64
PyExc_FileNotFoundError (*C var*), 64
PyExc_FloatingPointError (*C var*), 64
PyExc_FutureWarning (*C var*), 69
PyExc_GeneratorExit (*C var*), 65
PyExc_ImportError (*C var*), 65
PyExc_ImportWarning (*C var*), 69
PyExc_IndentationError (*C var*), 65
PyExc_IndexError (*C var*), 65
PyExc_InterruptedError (*C var*), 65
PyExc_IOError (*C var*), 68
PyExc_IsADirectoryError (*C var*), 65
PyExc_KeyboardInterrupt (*C var*), 65
PyExc_KeyError (*C var*), 65
PyExc_LookupError (*C var*), 65
PyExc_MemoryError (*C var*), 65
PyExc_ModuleNotFoundError (*C var*), 65
PyExc_NameError (*C var*), 65
PyExc_NotADirectoryError (*C var*), 65
PyExc_NotImplementedError (*C var*), 66
PyExc_OSError (*C var*), 66
PyExc_OverflowError (*C var*), 66
PyExc_PendingDeprecationWarning (*C var*), 69
PyExc_PermissionError (*C var*), 66
PyExc_ProcessLookupError (*C var*), 66
PyExc_PythonFinalizationError (*C var*), 66
PyExc_RecursionError (*C var*), 66
PyExc_ReferenceError (*C var*), 66
PyExc_ResourceWarning (*C var*), 69
PyExc_RuntimeError (*C var*), 66
PyExc_RuntimeWarning (*C var*), 69
PyExc_StopAsyncIteration (*C var*), 66
PyExc_StopIteration (*C var*), 66
PyExc_SyntaxError (*C var*), 66
PyExc_SyntaxWarning (*C var*), 69
PyExc_SystemError (*C var*), 66
PyExc_SystemExit (*C var*), 66
PyExc_TabError (*C var*), 67
PyExc_TimeoutError (*C var*), 67
PyExc_TypeError (*C var*), 67
PyExc_UnboundLocalError (*C var*), 67
PyExc_UnicodeDecodeError (*C var*), 67
PyExc_UnicodeEncodeError (*C var*), 67
PyExc_UnicodeError (*C var*), 67
PyExc_UnicodeTranslateError (*C var*), 67
PyExc_UnicodeWarning (*C var*), 69
PyExc_UserWarning (*C var*), 69
PyExc_ValueError (*C var*), 67
PyExc_Warning (*C var*), 69
PyExc_WindowsError (*C var*), 68
PyExc_ZeroDivisionError (*C var*), 67
PyException_GetArgs (*C function*), 61
PyException_GetCause (*C function*), 60
PyException_GetContext (*C function*), 60
PyException_GetTraceback (*C function*), 60
PyException_SetArgs (*C function*), 61
PyException_SetCause (*C function*), 60
PyException_SetContext (*C function*), 60
PyException_SetTraceback (*C function*), 60
PyExceptionClass_Check (*C function*), 60
PyExceptionClass_Name (*C function*), 60
PyFile_FromFd (*C function*), 199
PyFile_GetLine (*C function*), 199
PyFile_SetOpenCodeHook (*C function*), 199
PyFile_WriteObject (*C function*), 199
PyFile_WriteString (*C function*), 199
PyFloat_AS_DOUBLE (*C function*), 151
PyFloat_AsDouble (*C function*), 151
PyFloat_Check (*C function*), 150
PyFloat_CheckExact (*C function*), 151
PyFloat_FromDouble (*C function*), 151
PyFloat_FromString (*C function*), 151
PyFloat_GetInfo (*C function*), 151
PyFloat_GetMax (*C function*), 151
PyFloat_GetMin (*C function*), 151
PyFloat_Pack2 (*C function*), 152
PyFloat_Pack4 (*C function*), 152
PyFloat_Pack8 (*C function*), 152
PyFloat_Type (*C var*), 150
PyFloat_Unpack2 (*C function*), 152
PyFloat_Unpack4 (*C function*), 152
PyFloat_Unpack8 (*C function*), 152
PyFloatObject (*C type*), 150
PyFrame_Check (*C function*), 214
PyFrame_GetBack (*C function*), 214
PyFrame_GetBuiltins (*C function*), 214
PyFrame_GetCode (*C function*), 215
PyFrame_GetGenerator (*C function*), 215
PyFrame_GetGlobals (*C function*), 215
PyFrame_GetLasti (*C function*), 215
PyFrame_GetLineNumber (*C function*), 215
PyFrame_GetLocals (*C function*), 215
PyFrame_GetVar (*C function*), 215
PyFrame_GetVarString (*C function*), 215
PyFrame_Type (*C var*), 214
PyFrameLocalsProxy_Check (*C function*), 216
PyFrameLocalsProxy_Type (*C var*), 216
PyFrameObject (*C type*), 214
PyFrozenSet_Check (*C function*), 188
PyFrozenSet_CheckExact (*C function*), 189
PyFrozenSet_New (*C function*), 189
PyFrozenSet_Type (*C var*), 188
PyFunction_AddWatcher (*C function*), 191
PyFunction_Check (*C function*), 190

PyFunction_ClearWatcher (*C function*), 191
PyFunction_GET_ANNOTATIONS (*C function*), 191
PyFunction_GET_CLOSURE (*C function*), 191
PyFunction_GET_CODE (*C function*), 191
PyFunction_GET_DEFAULTS (*C function*), 191
PyFunction_GET_GLOBALS (*C function*), 191
PyFunction_GET_KW_DEFAULTS (*C function*), 191
PyFunction_GET_MODULE (*C function*), 191
PyFunction_GetAnnotations (*C function*), 191
PyFunction_GetClosure (*C function*), 191
PyFunction_GetCode (*C function*), 190
PyFunction_GetDefaults (*C function*), 190
PyFunction_GetGlobals (*C function*), 190
PyFunction_GetKwDefaults (*C function*), 190
PyFunction_GetModule (*C function*), 190
PyFunction_New (*C function*), 190
PyFunction_NewWithQualName (*C function*), 190
PyFunction_SetAnnotations (*C function*), 191
PyFunction_SetClosure (*C function*), 191
PyFunction_SetDefaults (*C function*), 190
PyFunction_SetVectorcall (*C function*), 190
PyFunction_Type (*C var*), 190
PyFunction_WatchCallback (*C type*), 191
PyFunction_WatchEvent (*C type*), 191
PyFunctionObject (*C type*), 190
PyGC_Collect (*C function*), 353
PyGC_Disable (*C function*), 353
PyGC_Enable (*C function*), 353
PyGC_IsEnabled (*C function*), 353
PyGen_Check (*C function*), 217
PyGen_CheckExact (*C function*), 217
PyGen_New (*C function*), 217
PyGen_NewWithQualName (*C function*), 217
PyGen_Type (*C var*), 217
PyGenObject (*C type*), 217
PyGetSetDef (*C type*), 310
PyGetSetDef.closure (*C member*), 310
PyGetSetDef.doc (*C member*), 310
PyGetSetDef.get (*C member*), 310
PyGetSetDef.name (*C member*), 310
PyGetSetDef.set (*C member*), 310
PyGILState_Check (*C function*), 240
PyGILState_Ensure (*C function*), 239
PyGILState_GetThisThreadState (*C function*), 240
PyGILState_Release (*C function*), 240
PyHASH_BITS (*C macro*), 96
PyHash_FuncDef (*C type*), 96
PyHash_FuncDef.hash (*C member*), 96
PyHash_FuncDef.hash_bits (*C member*), 96
PyHash_FuncDef.name (*C member*), 96
PyHash_FuncDef.seed_bits (*C member*), 96
PyHash_GetFuncDef (*C function*), 96
PyHASH_IMAG (*C macro*), 96
PyHASH_INF (*C macro*), 96
PyHASH_MODULUS (*C macro*), 96
PyHASH_MULTIPLIER (*C macro*), 96
PyImport_AddModule (*C function*), 81
PyImport_AddModuleObject (*C function*), 81
PyImport_AddModuleRef (*C function*), 81
PyImport_AppendInittab (*C function*), 83
PyImport_ExecCodeModule (*C function*), 82
PyImport_ExecCodeModuleEx (*C function*), 82
PyImport_ExecCodeModuleObject (*C function*), 82
PyImport_ExecCodeModuleWithPathnames (*C function*), 82
PyImport_ExtendInittab (*C function*), 84
PyImport_FrozenModules (*C var*), 83
PyImport_GetImporter (*C function*), 83
PyImport_GetMagicNumber (*C function*), 82
PyImport_GetMagicTag (*C function*), 83
PyImport_GetModule (*C function*), 83
PyImport_GetModuleDict (*C function*), 83
PyImport_Import (*C function*), 81
PyImport_ImportFrozenModule (*C function*), 83
PyImport_ImportFrozenModuleObject (*C function*), 83
PyImport_ImportModule (*C function*), 80
PyImport_ImportModuleAttr (*C function*), 84
PyImport_ImportModuleAttrString (*C function*), 84
PyImport_ImportModuleEx (*C function*), 80
PyImport_ImportModuleLevel (*C function*), 81
PyImport_ImportModuleLevelObject (*C function*), 81
PyImport_ImportModuleNoBlock (*C function*), 80
PyImport_ReloadModule (*C function*), 81
PyIndex_Check (*C function*), 120
PyInit_modulename (*C function*), 72
PyInitConfig (*C struct*), 256
PyInitConfig_AddModule (*C function*), 258
PyInitConfig_Create (*C function*), 256
PyInitConfig_Free (*C function*), 256
PyInitConfig_FreeStrList (*C function*), 257
PyInitConfig_GetError (*C function*), 256
PyInitConfig_GetExitCode (*C function*), 256
PyInitConfig_GetInt (*C function*), 257
PyInitConfig_GetStr (*C function*), 257
PyInitConfig_GetStrList (*C function*), 257
PyInitConfig_HasOption (*C function*), 257
PyInitConfig_SetInt (*C function*), 257
PyInitConfig_SetStr (*C function*), 257
PyInitConfig_SetStrList (*C function*), 257
PyInstanceMethod_Check (*C function*), 192
PyInstanceMethod_Function (*C function*), 192
PyInstanceMethod_GET_FUNCTION (*C function*), 192
PyInstanceMethod_New (*C function*), 192
PyInstanceMethod_Type (*C var*), 192
PyInterpreterConfig (*C type*), 244
PyInterpreterConfig_DEFAULT_GIL (*C macro*), 244
PyInterpreterConfig_OWN_GIL (*C macro*), 244
PyInterpreterConfig_SHARED_GIL (*C macro*), 244

PyInterpreterConfig.allow_daemon_threads (*C member*), 244
PyInterpreterConfig.allow_exec (*C member*), 244
PyInterpreterConfig.allow_fork (*C member*), 244
PyInterpreterConfig.allow_threads (*C member*), 244
PyInterpreterConfig.check_multi_interp_extensions (*C member*), 244
PyInterpreterConfig.gil (*C member*), 244
PyInterpreterConfig.use_main_obmalloc (*C member*), 244
PyInterpreterState (*C type*), 238
PyInterpreterState_Clear (*C function*), 241
PyInterpreterState_Delete (*C function*), 241
PyInterpreterState_Get (*C function*), 242
PyInterpreterState_GetDict (*C function*), 242
PyInterpreterState_GetID (*C function*), 242
PyInterpreterState_Head (*C function*), 249
PyInterpreterState_Main (*C function*), 249
PyInterpreterState_New (*C function*), 241
PyInterpreterState_Next (*C function*), 249
PyInterpreterState_ThreadHead (*C function*), 250
PyIter_Check (*C function*), 124
PyIter_Next (*C function*), 124
PyIter_NextItem (*C function*), 124
PyIter_Send (*C function*), 124
PyList_Append (*C function*), 182
PyList_AsTuple (*C function*), 183
PyList_Check (*C function*), 181
PyList_CheckExact (*C function*), 182
PyList_Clear (*C function*), 183
PyList_Extend (*C function*), 183
PyList_GET_ITEM (*C function*), 182
PyList_GET_SIZE (*C function*), 182
PyList_GetItem (*C function*), 182
PyList_GetItemRef (*C function*), 182
PyList_GetItem（C 函式）, 8
PyList_GetSlice (*C function*), 183
PyList_Insert (*C function*), 182
PyList_New (*C function*), 182
PyList_Reverse (*C function*), 183
PyList_SET_ITEM (*C function*), 182
PyList_SetItem (*C function*), 182
PyList_SetItem（C 函式）, 7
PyList_SetSlice (*C function*), 183
PyList_Size (*C function*), 182
PyList_Sort (*C function*), 183
PyList_Type (*C var*), 181
PyListObject (*C type*), 181
PyLong_AS_LONG (*C function*), 142
PyLong_AsDouble (*C function*), 144
PyLong_AsInt (*C function*), 142
PyLong_AsInt32 (*C function*), 144
PyLong_AsInt64 (*C function*), 144
PyLong_AsLong (*C function*), 142
PyLong_AsLongAndOverflow (*C function*), 142
PyLong_AsLongLong (*C function*), 143
PyLong_AsLongLongAndOverflow (*C function*), 143
PyLong_AsNativeBytes (*C function*), 145
PyLong_AsSize_t (*C function*), 143
PyLong_AsSsize_t (*C function*), 143
PyLong_AsUInt32 (*C function*), 144
PyLong_AsUInt64 (*C function*), 144
PyLong_AsUnsignedLong (*C function*), 143
PyLong_AsUnsignedLongLong (*C function*), 143
PyLong_AsUnsignedLongLongMask (*C function*), 144
PyLong_AsUnsignedLongMask (*C function*), 143
PyLong_AsVoidPtr (*C function*), 144
PyLong_Check (*C function*), 140
PyLong_CheckExact (*C function*), 140
PyLong_Export (*C function*), 149
PyLong_FreeExport (*C function*), 149
PyLong_FromDouble (*C function*), 141
PyLong_FromInt32 (*C function*), 141
PyLong_FromInt64 (*C function*), 141
PyLong_FromLong (*C function*), 141
PyLong_FromLongLong (*C function*), 141
PyLong_FromNativeBytes (*C function*), 142
PyLong_FromSize_t (*C function*), 141
PyLong_FromSsize_t (*C function*), 141
PyLong_FromString (*C function*), 141
PyLong_FromUInt32 (*C function*), 141
PyLong_FromUInt64 (*C function*), 141
PyLong_FromUnicodeObject (*C function*), 142
PyLong_FromUnsignedLong (*C function*), 141
PyLong_FromUnsignedLongLong (*C function*), 141
PyLong_FromUnsignedNativeBytes (*C function*), 142
PyLong_FromVoidPtr (*C function*), 142
PyLong_GetInfo (*C function*), 147
PyLong_GetNativeLayout (*C function*), 148
PyLong_GetSign (*C function*), 147
PyLong_IsNegative (*C function*), 147
PyLong_IsPositive (*C function*), 147
PyLong_IsZero (*C function*), 147
PyLong_Type (*C var*), 140
PyLongExport (*C struct*), 149
PyLongExport.digits (*C member*), 149
PyLongExport.ndigits (*C member*), 149
PyLongExport.negative (*C member*), 149
PyLongExport.value (*C member*), 149
PyLongLayout (*C struct*), 148
PyLongLayout.bits_per_digit (*C member*), 148
PyLongLayout.digit_endianness (*C member*), 148
PyLongLayout.digit_size (*C member*), 148
PyLongLayout.digits_order (*C member*), 148
PyLongObject (*C type*), 140
PyLongWriter (*C struct*), 149
PyLongWriter_Create (*C function*), 149
PyLongWriter_Discard (*C function*), 150
PyLongWriter_Finish (*C function*), 150

PyMapping_Check (*C function*), 122
PyMapping_DelItem (*C function*), 123
PyMapping_DelItemString (*C function*), 123
PyMapping_GetItemString (*C function*), 122
PyMapping_GetOptionalItem (*C function*), 122
PyMapping_GetOptionalItemString (*C function*), 123
PyMapping_HasKey (*C function*), 123
PyMapping_HasKeyString (*C function*), 123
PyMapping_HasKeyStringWithError (*C function*), 123
PyMapping_HasKeyWithError (*C function*), 123
PyMapping_Items (*C function*), 124
PyMapping_Keys (*C function*), 123
PyMapping_Length (*C function*), 122
PyMapping_SetItemString (*C function*), 123
PyMapping_Size (*C function*), 122
PyMapping_Values (*C function*), 124
PyMappingMethods (*C type*), 343
PyMappingMethods.mp_ass_subscript (*C member*), 343
PyMappingMethods.mp_length (*C member*), 343
PyMappingMethods.mp_subscript (*C member*), 343
PyMarshal_ReadLastObjectFromFile (*C function*), 85
PyMarshal_ReadLongFromFile (*C function*), 84
PyMarshal_ReadObjectFromFile (*C function*), 85
PyMarshal_ReadObjectFromString (*C function*), 85
PyMarshal_ReadShortFromFile (*C function*), 85
PyMarshal_WriteLongToFile (*C function*), 84
PyMarshal_WriteObjectToFile (*C function*), 84
PyMarshal_WriteObjectToString (*C function*), 84
PyMem_Calloc (*C function*), 285
PyMem_Del (*C function*), 286
PYMEM_DOMAIN_MEM (*C macro*), 289
PYMEM_DOMAIN_OBJ (*C macro*), 289
PYMEM_DOMAIN_RAW (*C macro*), 289
PyMem_Free (*C function*), 286
PyMem_GetAllocator (*C function*), 289
PyMem_Malloc (*C function*), 285
PyMem_New (*C macro*), 286
PyMem_RawCalloc (*C function*), 285
PyMem_RawFree (*C function*), 285
PyMem_RawMalloc (*C function*), 284
PyMem_RawRealloc (*C function*), 285
PyMem_Realloc (*C function*), 286
PyMem_Resize (*C macro*), 286
PyMem_SetAllocator (*C function*), 289
PyMem_SetupDebugHooks (*C function*), 290
PyMemAllocatorDomain (*C type*), 289
PyMemAllocatorEx (*C type*), 288
PyMember_GetOne (*C function*), 307
PyMember_SetOne (*C function*), 307
PyMemberDef (*C type*), 306
PyMemberDef.doc (*C member*), 306
PyMemberDef.flags (*C member*), 306
PyMemberDef.name (*C member*), 306
PyMemberDef.offset (*C member*), 306
PyMemberDef.type (*C member*), 306
PyMemoryView_Check (*C function*), 210
PyMemoryView_FromBuffer (*C function*), 210
PyMemoryView_FromMemory (*C function*), 210
PyMemoryView_FromObject (*C function*), 210
PyMemoryView_GET_BASE (*C function*), 211
PyMemoryView_GET_BUFFER (*C function*), 211
PyMemoryView_GetContiguous (*C function*), 210
PyMethod_Check (*C function*), 192
PyMethod_Function (*C function*), 193
PyMethod_GET_FUNCTION (*C function*), 193
PyMethod_GET_SELF (*C function*), 193
PyMethod_New (*C function*), 193
PyMethod_Self (*C function*), 193
PyMethod_Type (*C var*), 192
PyMethodDef (*C type*), 304
PyMethodDef.ml_doc (*C member*), 304
PyMethodDef.ml_flags (*C member*), 304
PyMethodDef.ml_meth (*C member*), 304
PyMethodDef.ml_name (*C member*), 304
PyMODINIT_FUNC (*C macro*), 72
PyModule_Add (*C function*), 205
PyModule_AddFunctions (*C function*), 206
PyModule_AddIntConstant (*C function*), 206
PyModule_AddIntMacro (*C macro*), 206
PyModule_AddObject (*C function*), 205
PyModule_AddObjectRef (*C function*), 204
PyModule_AddStringConstant (*C function*), 206
PyModule_AddStringMacro (*C macro*), 206
PyModule_AddType (*C function*), 206
PyModule_Check (*C function*), 200
PyModule_CheckExact (*C function*), 200
PyModule_Create (*C function*), 203
PyModule_Create2 (*C function*), 203
PyModule_ExecDef (*C function*), 204
PyModule_FromDefAndSpec (*C function*), 204
PyModule_FromDefAndSpec2 (*C function*), 204
PyModule_GetDef (*C function*), 200
PyModule_GetDict (*C function*), 200
PyModule_GetFilename (*C function*), 200
PyModule_GetFilenameObject (*C function*), 200
PyModule_GetName (*C function*), 200
PyModule_GetNameObject (*C function*), 200
PyModule_GetState (*C function*), 200
PyModule_New (*C function*), 200
PyModule_NewObject (*C function*), 200
PyModule_SetDocString (*C function*), 207
PyModule_Type (*C var*), 200
PyModuleDef (*C type*), 201
PyModuleDef_Init (*C function*), 73
PyModuleDef_Slot (*C type*), 202
PyModuleDef_Slot.slot (*C member*), 202
PyModuleDef_Slot.value (*C member*), 202
PyModuleDef.m_base (*C member*), 201
PyModuleDef.m_clear (*C member*), 202
PyModuleDef.m_doc (*C member*), 201

PyModuleDef.m_free (*C member*), 202
PyModuleDef.m_methods (*C member*), 201
PyModuleDef.m_name (*C member*), 201
PyModuleDef.m_size (*C member*), 201
PyModuleDef.m_slots (*C member*), 201
PyModuleDef.m_slots.m_reload (*C member*), 201
PyModuleDef.m_traverse (*C member*), 201
PyMonitoring_EnterScope (*C function*), 360
PyMonitoring_ExitScope (*C function*), 362
PyMonitoring_FireBranchLeftEvent (*C function*), 360
PyMonitoring_FireBranchRightEvent (*C function*), 360
PyMonitoring_FireCallEvent (*C function*), 359
PyMonitoring_FireCRaiseEvent (*C function*), 360
PyMonitoring_FireCReturnEvent (*C function*), 360
PyMonitoring_FireExceptionHandledEvent (*C function*), 360
PyMonitoring_FireJumpEvent (*C function*), 359
PyMonitoring_FireLineEvent (*C function*), 359
PyMonitoring_FirePyResumeEvent (*C function*), 359
PyMonitoring_FirePyReturnEvent (*C function*), 359
PyMonitoring_FirePyStartEvent (*C function*), 359
PyMonitoring_FirePyThrowEvent (*C function*), 360
PyMonitoring_FirePyUnwindEvent (*C function*), 360
PyMonitoring_FirePyYieldEvent (*C function*), 359
PyMonitoring_FireRaiseEvent (*C function*), 360
PyMonitoring_FireReraiseEvent (*C function*), 360
PyMonitoring_FireStopIterationEvent (*C function*), 360
PyMonitoringState (*C type*), 359
PyMutex (*C type*), 252
PyMutex_IsLocked (*C function*), 252
PyMutex_Lock (*C function*), 252
PyMutex_Unlock (*C function*), 252
PyNumber_Absolute (*C function*), 118
PyNumber_Add (*C function*), 117
PyNumber_And (*C function*), 118
PyNumber_AsSsize_t (*C function*), 120
PyNumber_Check (*C function*), 117
PyNumber_Divmod (*C function*), 118
PyNumber_Float (*C function*), 120
PyNumber_FloorDivide (*C function*), 118
PyNumber_Index (*C function*), 120
PyNumber_InPlaceAdd (*C function*), 119
PyNumber_InPlaceAnd (*C function*), 119
PyNumber_InPlaceFloorDivide (*C function*), 119
PyNumber_InPlaceLshift (*C function*), 119
PyNumber_InPlaceMatrixMultiply (*C function*), 119
PyNumber_InPlaceMultiply (*C function*), 119
PyNumber_InPlaceOr (*C function*), 120
PyNumber_InPlacePower (*C function*), 119
PyNumber_InPlaceRemainder (*C function*), 119
PyNumber_InPlaceRshift (*C function*), 119
PyNumber_InPlaceSubtract (*C function*), 119
PyNumber_InPlaceTrueDivide (*C function*), 119
PyNumber_InPlaceXor (*C function*), 120
PyNumber_Invert (*C function*), 118
PyNumber_Long (*C function*), 120
PyNumber_Lshift (*C function*), 118
PyNumber_MatrixMultiply (*C function*), 118
PyNumber_Multiply (*C function*), 118
PyNumber_Negative (*C function*), 118
PyNumber_Or (*C function*), 119
PyNumber_Positive (*C function*), 118
PyNumber_Power (*C function*), 118
PyNumber_Remainder (*C function*), 118
PyNumber_Rshift (*C function*), 118
PyNumber_Subtract (*C function*), 117
PyNumber_ToBase (*C function*), 120
PyNumber_TrueDivide (*C function*), 118
PyNumber_Xor (*C function*), 119
PyNumberMethods (*C type*), 341
PyNumberMethods.nb_absolute (*C member*), 342
PyNumberMethods.nb_add (*C member*), 342
PyNumberMethods.nb_and (*C member*), 342
PyNumberMethods.nb_bool (*C member*), 342
PyNumberMethods.nb_divmod (*C member*), 342
PyNumberMethods.nb_float (*C member*), 342
PyNumberMethods.nb_floor_divide (*C member*), 343
PyNumberMethods.nb_index (*C member*), 343
PyNumberMethods.nb_inplace_add (*C member*), 342
PyNumberMethods.nb_inplace_and (*C member*), 342
PyNumberMethods.nb_inplace_floor_divide (*C member*), 343
PyNumberMethods.nb_inplace_lshift (*C member*), 342
PyNumberMethods.nb_inplace_matrix_multiply (*C member*), 343
PyNumberMethods.nb_inplace_multiply (*C member*), 342
PyNumberMethods.nb_inplace_or (*C member*), 343
PyNumberMethods.nb_inplace_power (*C member*), 342
PyNumberMethods.nb_inplace_remainder (*C member*), 342
PyNumberMethods.nb_inplace_rshift (*C member*), 342
PyNumberMethods.nb_inplace_subtract (*C member*), 342
PyNumberMethods.nb_inplace_true_divide (*C member*), 343

PyNumberMethods.nb_inplace_xor (*C member*), 343
PyNumberMethods.nb_int (*C member*), 342
PyNumberMethods.nb_invert (*C member*), 342
PyNumberMethods.nb_lshift (*C member*), 342
PyNumberMethods.nb_matrix_multiply (*C member*), 343
PyNumberMethods.nb_multiply (*C member*), 342
PyNumberMethods.nb_negative (*C member*), 342
PyNumberMethods.nb_or (*C member*), 342
PyNumberMethods.nb_positive (*C member*), 342
PyNumberMethods.nb_power (*C member*), 342
PyNumberMethods.nb_remainder (*C member*), 342
PyNumberMethods.nb_reserved (*C member*), 342
PyNumberMethods.nb_rshift (*C member*), 342
PyNumberMethods.nb_subtract (*C member*), 342
PyNumberMethods.nb_true_divide (*C member*), 343
PyNumberMethods.nb_xor (*C member*), 342
PyObject (*C type*), 301
PyObject_ASCII (*C function*), 107
PyObject_AsFileDescriptor (*C function*), 199
PyObject_Bytes (*C function*), 107
PyObject_Call (*C function*), 115
PyObject_CallFinalizer (*C function*), 301
PyObject_CallFinalizerFromDealloc (*C function*), 301
PyObject_CallFunction (*C function*), 116
PyObject_CallFunctionObjArgs (*C function*), 116
PyObject_CallMethod (*C function*), 116
PyObject_CallMethodNoArgs (*C function*), 116
PyObject_CallMethodObjArgs (*C function*), 116
PyObject_CallMethodOneArg (*C function*), 116
PyObject_CallNoArgs (*C function*), 115
PyObject_CallObject (*C function*), 116
PyObject_Calloc (*C function*), 287
PyObject_CallOneArg (*C function*), 115
PyObject_CheckBuffer (*C function*), 130
PyObject_ClearManagedDict (*C function*), 110
PyObject_ClearWeakRefs (*C function*), 212
PyObject_CopyData (*C function*), 131
PyObject_Del (*C function*), 297
PyObject_DelAttr (*C function*), 106
PyObject_DelAttrString (*C function*), 106
PyObject_DelItem (*C function*), 109
PyObject_DelItemString (*C function*), 109
PyObject_Dir (*C function*), 109
PyObject_Format (*C function*), 107
PyObject_Free (*C function*), 287
PyObject_GC_Del (*C function*), 352
PyObject_GC_IsFinalized (*C function*), 352
PyObject_GC_IsTracked (*C function*), 351
PyObject_GC_New (*C macro*), 350
PyObject_GC_NewVar (*C macro*), 351
PyObject_GC_Resize (*C macro*), 351
PyObject_GC_Track (*C function*), 351
PyObject_GC_UnTrack (*C function*), 352
PyObject_GenericGetAttr (*C function*), 106
PyObject_GenericGetDict (*C function*), 106
PyObject_GenericHash (*C function*), 97
PyObject_GenericSetAttr (*C function*), 106
PyObject_GenericSetDict (*C function*), 107
PyObject_GetAIter (*C function*), 109
PyObject_GetArenaAllocator (*C function*), 292
PyObject_GetAttr (*C function*), 105
PyObject_GetAttrString (*C function*), 105
PyObject_GetBuffer (*C function*), 130
PyObject_GetItem (*C function*), 109
PyObject_GetItemData (*C function*), 110
PyObject_GetIter (*C function*), 109
PyObject_GetOptionalAttr (*C function*), 105
PyObject_GetOptionalAttrString (*C function*), 106
PyObject_GetTypeData (*C function*), 109
PyObject_HasAttr (*C function*), 105
PyObject_HasAttrString (*C function*), 105
PyObject_HasAttrStringWithError (*C function*), 105
PyObject_HasAttrWithError (*C function*), 105
PyObject_Hash (*C function*), 108
PyObject_HashNotImplemented (*C function*), 108
PyObject_HEAD (*C macro*), 302
PyObject_HEAD_INIT (*C macro*), 303
PyObject_Init (*C function*), 295
PyObject_InitVar (*C function*), 295
PyObject_IS_GC (*C function*), 351
PyObject_IsInstance (*C function*), 108
PyObject_IsSubclass (*C function*), 108
PyObject_IsTrue (*C function*), 108
PyObject_Length (*C function*), 108
PyObject_LengthHint (*C function*), 109
PyObject_Malloc (*C function*), 287
PyObject_New (*C macro*), 295
PyObject_NewVar (*C macro*), 296
PyObject_Not (*C function*), 108
PyObject_Print (*C function*), 104
PyObject_Realloc (*C function*), 287
PyObject_Repr (*C function*), 107
PyObject_RichCompare (*C function*), 107
PyObject_RichCompareBool (*C function*), 107
PyObject_SelfIter (*C function*), 109
PyObject_SetArenaAllocator (*C function*), 292
PyObject_SetAttr (*C function*), 106
PyObject_SetAttrString (*C function*), 106
PyObject_SetItem (*C function*), 109
PyObject_Size (*C function*), 108
PyObject_Str (*C function*), 107
PyObject_Type (*C function*), 108
PyObject_TypeCheck (*C function*), 108
PyObject_VAR_HEAD (*C macro*), 302
PyObject_Vectorcall (*C function*), 117
PyObject_VectorcallDict (*C function*), 117
PyObject_VectorcallMethod (*C function*), 117
PyObject_VisitManagedDict (*C function*), 110
PyObjectArenaAllocator (*C type*), 291
PyObject.ob_refcnt (*C member*), 301

PyObject.ob_type (*C member*), 301
PyOS_AfterFork (*C function*), 76
PyOS_AfterFork_Child (*C function*), 76
PyOS_AfterFork_Parent (*C function*), 75
PyOS_BeforeFork (*C function*), 75
PyOS_CheckStack (*C function*), 76
PyOS_double_to_string (*C function*), 95
PyOS_FSPath (*C function*), 75
PyOS_getsig (*C function*), 76
PyOS_InputHook (*C var*), 42
PyOS_ReadlineFunctionPointer (*C var*), 42
PyOS_setsig (*C function*), 76
PyOS_sighandler_t (*C type*), 76
PyOS_snprintf (*C function*), 94
PyOS_stricmp (*C function*), 95
PyOS_string_to_double (*C function*), 95
PyOS_strnicmp (*C function*), 95
PyOS_strtol (*C function*), 94
PyOS_strtoul (*C function*), 94
PyOS_vsnprintf (*C function*), 94
PyPreConfig (*C type*), 263
PyPreConfig_InitIsolatedConfig (*C function*), 263
PyPreConfig_InitPythonConfig (*C function*), 263
PyPreConfig.allocator (*C member*), 263
PyPreConfig.coerce_c_locale (*C member*), 264
PyPreConfig.coerce_c_locale_warn (*C member*), 264
PyPreConfig.configure_locale (*C member*), 264
PyPreConfig.dev_mode (*C member*), 264
PyPreConfig.isolated (*C member*), 264
PyPreConfig.legacy_windows_fs_encoding (*C member*), 264
PyPreConfig.parse_argv (*C member*), 265
PyPreConfig.use_environment (*C member*), 265
PyPreConfig.utf8_mode (*C member*), 265
PyProperty_Type (*C var*), 208
PyRefTracer (*C type*), 249
PyRefTracer_CREATE (*C var*), 249
PyRefTracer_DESTROY (*C var*), 249
PyRefTracer_GetTracer (*C function*), 249
PyRefTracer_SetTracer (*C function*), 249
PyRun_AnyFile (*C function*), 41
PyRun_AnyFileEx (*C function*), 41
PyRun_AnyFileExFlags (*C function*), 41
PyRun_AnyFileFlags (*C function*), 41
PyRun_File (*C function*), 43
PyRun_FileEx (*C function*), 43
PyRun_FileExFlags (*C function*), 43
PyRun_FileFlags (*C function*), 43
PyRun_InteractiveLoop (*C function*), 42
PyRun_InteractiveLoopFlags (*C function*), 42
PyRun_InteractiveOne (*C function*), 42
PyRun_InteractiveOneFlags (*C function*), 42
PyRun_SimpleFile (*C function*), 42
PyRun_SimpleFileEx (*C function*), 42
PyRun_SimpleFileExFlags (*C function*), 42
PyRun_SimpleString (*C function*), 41
PyRun_SimpleStringFlags (*C function*), 41
PyRun_String (*C function*), 43
PyRun_StringFlags (*C function*), 43
PySendResult (*C type*), 124
PySeqIter_Check (*C function*), 208
PySeqIter_New (*C function*), 208
PySeqIter_Type (*C var*), 208
PySequence_Check (*C function*), 120
PySequence_Concat (*C function*), 120
PySequence_Contains (*C function*), 121
PySequence_Count (*C function*), 121
PySequence_DelItem (*C function*), 121
PySequence_DelSlice (*C function*), 121
PySequence_Fast (*C function*), 122
PySequence_Fast_GET_ITEM (*C function*), 122
PySequence_Fast_GET_SIZE (*C function*), 122
PySequence_Fast_ITEMS (*C function*), 122
PySequence_GetItem (*C function*), 121
PySequence_GetItem（C 函式）, 8
PySequence_GetSlice (*C function*), 121
PySequence_In (*C function*), 121
PySequence_Index (*C function*), 121
PySequence_InPlaceConcat (*C function*), 121
PySequence_InPlaceRepeat (*C function*), 121
PySequence_ITEM (*C function*), 122
PySequence_Length (*C function*), 120
PySequence_List (*C function*), 121
PySequence_Repeat (*C function*), 121
PySequence_SetItem (*C function*), 121
PySequence_SetSlice (*C function*), 121
PySequence_Size (*C function*), 120
PySequence_Tuple (*C function*), 122
PySequenceMethods (*C type*), 343
PySequenceMethods.sq_ass_item (*C member*), 344
PySequenceMethods.sq_concat (*C member*), 343
PySequenceMethods.sq_contains (*C member*), 344
PySequenceMethods.sq_inplace_concat (*C member*), 344
PySequenceMethods.sq_inplace_repeat (*C member*), 344
PySequenceMethods.sq_item (*C member*), 343
PySequenceMethods.sq_length (*C member*), 343
PySequenceMethods.sq_repeat (*C member*), 343
PySet_Add (*C function*), 189
PySet_Check (*C function*), 188
PySet_CheckExact (*C function*), 189
PySet_Clear (*C function*), 189
PySet_Contains (*C function*), 189
PySet_Discard (*C function*), 189
PySet_GET_SIZE (*C function*), 189
PySet_New (*C function*), 189
PySet_Pop (*C function*), 189
PySet_Size (*C function*), 189
PySet_Type (*C var*), 188
PySetObject (*C type*), 188
PySignal_SetWakeupFd (*C function*), 59

PySlice_AdjustIndices (*C function*), 210
PySlice_Check (*C function*), 209
PySlice_GetIndices (*C function*), 209
PySlice_GetIndicesEx (*C function*), 209
PySlice_New (*C function*), 209
PySlice_Type (*C var*), 209
PySlice_Unpack (*C function*), 209
PyState_AddModule (*C function*), 207
PyState_FindModule (*C function*), 207
PyState_RemoveModule (*C function*), 207
PyStatus (*C type*), 262
PyStatus_Error (*C function*), 262
PyStatus_Exception (*C function*), 262
PyStatus_Exit (*C function*), 262
PyStatus_IsError (*C function*), 263
PyStatus_IsExit (*C function*), 263
PyStatus_NoMemory (*C function*), 262
PyStatus_Ok (*C function*), 262
PyStatus.err_msg (*C member*), 262
PyStatus.exitcode (*C member*), 262
PyStatus.func (*C member*), 262
PyStructSequence_Desc (*C type*), 180
PyStructSequence_Desc.doc (*C member*), 180
PyStructSequence_Desc.fields (*C member*), 180
PyStructSequence_Desc.n_in_sequence (*C member*), 180
PyStructSequence_Desc.name (*C member*), 180
PyStructSequence_Field (*C type*), 181
PyStructSequence_Field.doc (*C member*), 181
PyStructSequence_Field.name (*C member*), 181
PyStructSequence_GET_ITEM (*C function*), 181
PyStructSequence_GetItem (*C function*), 181
PyStructSequence_InitType (*C function*), 180
PyStructSequence_InitType2 (*C function*), 180
PyStructSequence_New (*C function*), 181
PyStructSequence_NewType (*C function*), 180
PyStructSequence_SET_ITEM (*C function*), 181
PyStructSequence_SetItem (*C function*), 181
PyStructSequence_UnnamedField (*C var*), 181
PySys_AddAuditHook (*C function*), 79
PySys_Audit (*C function*), 79
PySys_AuditTuple (*C function*), 79
PySys_FormatStderr (*C function*), 79
PySys_FormatStdout (*C function*), 79
PySys_GetObject (*C function*), 78
PySys_GetXOptions (*C function*), 79
PySys_ResetWarnOptions (*C function*), 78
PySys_SetArgv (*C function*), 234
PySys_SetArgvEx (*C function*), 234
PySys_SetObject (*C function*), 78
PySys_WriteStderr (*C function*), 79
PySys_WriteStdout (*C function*), 78
Python 3000, **378**
Python Enhancement Proposals
    PEP 1, 377
    PEP 7, 3, 6
    PEP 238, 371
    PEP 278, 381
    PEP 302, 374
    PEP 343, 368
    PEP 353, 9
    PEP 362, 366, 377
    PEP 383, 166, 167
    PEP 387, 13, 14
    PEP 393, 157
    PEP 411, 378
    PEP 420, 376, 377
    PEP 442, 300, 339
    PEP 443, 372
    PEP 446, 78
    PEP 451, 202
    PEP 456, 96
    PEP 483, 372
    PEP 484, 365, 371, 372, 381
    PEP 489, 71, 72, 244
    PEP 492, 366368
    PEP 498, 370
    PEP 519, 377
    PEP 523, 216, 242, 243
    PEP 525, 366
    PEP 526, 365, 381
    PEP 528, 228, 272
    PEP 529, 167, 228
    PEP 538, 279
    PEP 539, 250
    PEP 540, 279
    PEP 552, 269
    PEP 554, 246
    PEP 578, 79
    PEP 585, 372
    PEP 587, 261
    PEP 590, 113
    PEP 623, 157
    PEP 0626#out-of-process-debuggers-and-profilers, 195
    PEP 634, 327
    PEP 649, 365
    PEP 667, 97, 215, 216
    PEP 0683, 48, 372
    PEP 684, 238, 246
    PEP 703, 371, 372
    PEP 741, 255
    PEP 3116, 381
    PEP 3119, 108
    PEP 3121, 201
    PEP 3147, 83
    PEP 3151, 67
    PEP 3155, 378
PYTHON_ABI_VERSION (*C macro*), 204
PYTHON_API_VERSION (*C macro*), 204
PYTHON_CPU_COUNT, 272
PYTHON_FROZEN_MODULES, 271
PYTHON_GIL, 372
PYTHON_PERF_JIT_SUPPORT, 276
PYTHON_PRESITE, 275
PYTHONCOERCECLOCALE, 279

PYTHONDEBUG, 227, 273
PYTHONDEVMODE, 269
PYTHONDONTWRITEBYTECODE, 227, 277
PYTHONDUMPREFS, 269
PYTHONDUMPREFSFILE, 269
PYTHONEXECUTABLE, 274
PYTHONFAULTHANDLER, 270
PYTHONHASHSEED, 227, 271
PYTHONHOME, 11, 227, 235, 271
Pythonic（Python 風格的）, **378**
PYTHONINSPECT, 227, 271
PYTHONINTMAXSTRDIGITS, 272
PYTHONIOENCODING, 275
PYTHONLEGACYWINDOWSFSENCODING, 228, 265
PYTHONLEGACYWINDOWSSTDIO, 228, 272
PYTHONMALLOC, 284, 288, 290, 291
PYTHONMALLOCSTATS, 272, 284
PYTHONNODEBUGRANGES, 269
PYTHONNOUSERSITE, 228, 276
PYTHONOPTIMIZE, 228, 273
PYTHONPATH, 11, 227, 273
PYTHONPERFSUPPORT, 276
PYTHONPLATLIBDIR, 272
PYTHONPROFILEIMPORTTIME, 271
PYTHONPYCACHEPREFIX, 274
PYTHONSAFEPATH, 268
PYTHONTRACEMALLOC, 276
PYTHONUNBUFFERED, 229, 268
PYTHONUTF8, 265, 279
PYTHONVERBOSE, 229, 277
PYTHONWARNINGS, 277
PyThread_create_key (*C function*), 251
PyThread_delete_key (*C function*), 251
PyThread_delete_key_value (*C function*), 252
PyThread_get_key_value (*C function*), 252
PyThread_ReInitTLS (*C function*), 252
PyThread_set_key_value (*C function*), 252
PyThread_tss_alloc (*C function*), 250
PyThread_tss_create (*C function*), 251
PyThread_tss_delete (*C function*), 251
PyThread_tss_free (*C function*), 251
PyThread_tss_get (*C function*), 251
PyThread_tss_is_created (*C function*), 251
PyThread_tss_set (*C function*), 251
PyThreadState (*C type*), 238
PyThreadState_Clear (*C function*), 241
PyThreadState_Delete (*C function*), 241
PyThreadState_DeleteCurrent (*C function*), 241
PyThreadState_EnterTracing (*C function*), 242
PyThreadState_Get (*C function*), 239
PyThreadState_GetDict (*C function*), 243
PyThreadState_GetFrame (*C function*), 241
PyThreadState_GetID (*C function*), 241
PyThreadState_GetInterpreter (*C function*), 241
PyThreadState_GetUnchecked (*C function*), 239
PyThreadState_LeaveTracing (*C function*), 242
PyThreadState_New (*C function*), 241
PyThreadState_Next (*C function*), 250
PyThreadState_SetAsyncExc (*C function*), 243
PyThreadState_Swap (*C function*), 239
PyThreadState.interp (*C member*), 238
PyThreadState（C 型🄵）, 235
PyTime_AsSecondsDouble (*C function*), 101
PyTime_Check (*C function*), 220
PyTime_CheckExact (*C function*), 220
PyTime_FromTime (*C function*), 221
PyTime_FromTimeAndFold (*C function*), 221
PyTime_MAX (*C var*), 100
PyTime_MIN (*C var*), 100
PyTime_Monotonic (*C function*), 100
PyTime_MonotonicRaw (*C function*), 100
PyTime_PerfCounter (*C function*), 100
PyTime_PerfCounterRaw (*C function*), 100
PyTime_t (*C type*), 100
PyTime_Time (*C function*), 100
PyTime_TimeRaw (*C function*), 101
PyTimeZone_FromOffset (*C function*), 221
PyTimeZone_FromOffsetAndName (*C function*), 221
PyTrace_C_CALL (*C var*), 248
PyTrace_C_EXCEPTION (*C var*), 248
PyTrace_C_RETURN (*C var*), 248
PyTrace_CALL (*C var*), 247
PyTrace_EXCEPTION (*C var*), 248
PyTrace_LINE (*C var*), 248
PyTrace_OPCODE (*C var*), 248
PyTrace_RETURN (*C var*), 248
PyTraceMalloc_Track (*C function*), 292
PyTraceMalloc_Untrack (*C function*), 292
PyTuple_Check (*C function*), 179
PyTuple_CheckExact (*C function*), 179
PyTuple_GET_ITEM (*C function*), 179
PyTuple_GET_SIZE (*C function*), 179
PyTuple_GetItem (*C function*), 179
PyTuple_GetSlice (*C function*), 179
PyTuple_New (*C function*), 179
PyTuple_Pack (*C function*), 179
PyTuple_SET_ITEM (*C function*), 179
PyTuple_SetItem (*C function*), 179
PyTuple_SetItem（C 函式）, 7
PyTuple_Size (*C function*), 179
PyTuple_Type (*C var*), 179
PyTupleObject (*C type*), 179
PyType_AddWatcher (*C function*), 134
PyType_Check (*C function*), 133
PyType_CheckExact (*C function*), 133
PyType_ClearCache (*C function*), 133
PyType_ClearWatcher (*C function*), 134
PyType_Freeze (*C function*), 138
PyType_FromMetaclass (*C function*), 137
PyType_FromModuleAndSpec (*C function*), 137
PyType_FromSpec (*C function*), 138
PyType_FromSpecWithBases (*C function*), 138
PyType_GenericAlloc (*C function*), 135
PyType_GenericNew (*C function*), 135
PyType_GetBaseByToken (*C function*), 136
PyType_GetDict (*C function*), 134

PyType_GetFlags (*C function*), 133
PyType_GetFullyQualifiedName (*C function*), 135
PyType_GetModule (*C function*), 136
PyType_GetModuleByDef (*C function*), 136
PyType_GetModuleName (*C function*), 135
PyType_GetModuleState (*C function*), 136
PyType_GetName (*C function*), 135
PyType_GetQualName (*C function*), 135
PyType_GetSlot (*C function*), 136
PyType_GetTypeDataSize (*C function*), 109
PyType_HasFeature (*C function*), 134
PyType_IS_GC (*C function*), 134
PyType_IsSubtype (*C function*), 135
PyType_Modified (*C function*), 134
PyType_Ready (*C function*), 135
PyType_Slot (*C type*), 139
PyType_Slot.pfunc (*C member*), 139
PyType_Slot.slot (*C member*), 139
PyType_Spec (*C type*), 138
PyType_Spec.basicsize (*C member*), 138
PyType_Spec.flags (*C member*), 139
PyType_Spec.itemsize (*C member*), 138
PyType_Spec.name (*C member*), 138
PyType_Spec.slots (*C member*), 139
PyType_Type (*C var*), 133
PyType_Watch (*C function*), 134
PyType_WatchCallback (*C type*), 134
PyTypeObject (*C type*), 133
PyTypeObject.tp_alloc (*C member*), 335
PyTypeObject.tp_as_async (*C member*), 321
PyTypeObject.tp_as_buffer (*C member*), 323
PyTypeObject.tp_as_mapping (*C member*), 322
PyTypeObject.tp_as_number (*C member*), 321
PyTypeObject.tp_as_sequence (*C member*), 321
PyTypeObject.tp_base (*C member*), 333
PyTypeObject.tp_bases (*C member*), 336
PyTypeObject.tp_basicsize (*C member*), 317
PyTypeObject.tp_cache (*C member*), 337
PyTypeObject.tp_call (*C member*), 322
PyTypeObject.tp_clear (*C member*), 329
PyTypeObject.tp_dealloc (*C member*), 318
PyTypeObject.tp_del (*C member*), 337
PyTypeObject.tp_descr_get (*C member*), 334
PyTypeObject.tp_descr_set (*C member*), 334
PyTypeObject.tp_dict (*C member*), 333
PyTypeObject.tp_dictoffset (*C member*), 334
PyTypeObject.tp_doc (*C member*), 327
PyTypeObject.tp_finalize (*C member*), 337
PyTypeObject.tp_flags (*C member*), 323
PyTypeObject.tp_free (*C member*), 336
PyTypeObject.tp_getattr (*C member*), 321
PyTypeObject.tp_getattro (*C member*), 323
PyTypeObject.tp_getset (*C member*), 333
PyTypeObject.tp_hash (*C member*), 322
PyTypeObject.tp_init (*C member*), 335
PyTypeObject.tp_is_gc (*C member*), 336
PyTypeObject.tp_itemsize (*C member*), 317
PyTypeObject.tp_iter (*C member*), 332
PyTypeObject.tp_iternext (*C member*), 332
PyTypeObject.tp_members (*C member*), 333
PyTypeObject.tp_methods (*C member*), 332
PyTypeObject.tp_mro (*C member*), 337
PyTypeObject.tp_name (*C member*), 317
PyTypeObject.tp_new (*C member*), 335
PyTypeObject.tp_repr (*C member*), 321
PyTypeObject.tp_richcompare (*C member*), 331
PyTypeObject.tp_setattr (*C member*), 321
PyTypeObject.tp_setattro (*C member*), 323
PyTypeObject.tp_str (*C member*), 322
PyTypeObject.tp_subclasses (*C member*), 337
PyTypeObject.tp_traverse (*C member*), 328
PyTypeObject.tp_vectorcall (*C member*), 340
PyTypeObject.tp_vectorcall_offset (*C member*), 320
PyTypeObject.tp_version_tag (*C member*), 337
PyTypeObject.tp_watched (*C member*), 340
PyTypeObject.tp_weaklist (*C member*), 337
PyTypeObject.tp_weaklistoffset (*C member*), 332
PyTZInfo_Check (*C function*), 220
PyTZInfo_CheckExact (*C function*), 221
PyUnicode_1BYTE_DATA (*C function*), 158
PyUnicode_1BYTE_KIND (*C macro*), 158
PyUnicode_2BYTE_DATA (*C function*), 158
PyUnicode_2BYTE_KIND (*C macro*), 158
PyUnicode_4BYTE_DATA (*C function*), 158
PyUnicode_4BYTE_KIND (*C macro*), 158
PyUnicode_Append (*C function*), 164
PyUnicode_AppendAndDel (*C function*), 164
PyUnicode_AsASCIIString (*C function*), 172
PyUnicode_AsCharmapString (*C function*), 172
PyUnicode_AsEncodedString (*C function*), 169
PyUnicode_AsLatin1String (*C function*), 172
PyUnicode_AsMBCSString (*C function*), 173
PyUnicode_AsRawUnicodeEscapeString (*C function*), 172
PyUnicode_AsUCS4 (*C function*), 165
PyUnicode_AsUCS4Copy (*C function*), 166
PyUnicode_AsUnicodeEscapeString (*C function*), 171
PyUnicode_AsUTF8 (*C function*), 170
PyUnicode_AsUTF8AndSize (*C function*), 169
PyUnicode_AsUTF8String (*C function*), 169
PyUnicode_AsUTF16String (*C function*), 171
PyUnicode_AsUTF32String (*C function*), 170
PyUnicode_AsWideChar (*C function*), 168
PyUnicode_AsWideCharString (*C function*), 168
PyUnicode_BuildEncodingMap (*C function*), 164
PyUnicode_Check (*C function*), 158
PyUnicode_CHECK_INTERNED (*C function*), 176
PyUnicode_CheckExact (*C function*), 158
PyUnicode_Compare (*C function*), 175
PyUnicode_CompareWithASCIIString (*C function*), 175
PyUnicode_Concat (*C function*), 173
PyUnicode_Contains (*C function*), 176

PyUnicode_CopyCharacters (*C function*), 165
PyUnicode_Count (*C function*), 175
PyUnicode_DATA (*C function*), 158
PyUnicode_Decode (*C function*), 169
PyUnicode_DecodeASCII (*C function*), 172
PyUnicode_DecodeCharmap (*C function*), 172
PyUnicode_DecodeCodePageStateful (*C function*), 173
PyUnicode_DecodeFSDefault (*C function*), 167
PyUnicode_DecodeFSDefaultAndSize (*C function*), 167
PyUnicode_DecodeLatin1 (*C function*), 172
PyUnicode_DecodeLocale (*C function*), 166
PyUnicode_DecodeLocaleAndSize (*C function*), 166
PyUnicode_DecodeMBCS (*C function*), 173
PyUnicode_DecodeMBCSStateful (*C function*), 173
PyUnicode_DecodeRawUnicodeEscape (*C function*), 172
PyUnicode_DecodeUnicodeEscape (*C function*), 171
PyUnicode_DecodeUTF7 (*C function*), 171
PyUnicode_DecodeUTF7Stateful (*C function*), 171
PyUnicode_DecodeUTF8 (*C function*), 169
PyUnicode_DecodeUTF8Stateful (*C function*), 169
PyUnicode_DecodeUTF16 (*C function*), 170
PyUnicode_DecodeUTF16Stateful (*C function*), 171
PyUnicode_DecodeUTF32 (*C function*), 170
PyUnicode_DecodeUTF32Stateful (*C function*), 170
PyUnicode_EncodeCodePage (*C function*), 173
PyUnicode_EncodeFSDefault (*C function*), 167
PyUnicode_EncodeLocale (*C function*), 166
PyUnicode_Equal (*C function*), 175
PyUnicode_EqualToUTF8 (*C function*), 175
PyUnicode_EqualToUTF8AndSize (*C function*), 175
PyUnicode_Fill (*C function*), 165
PyUnicode_Find (*C function*), 174
PyUnicode_FindChar (*C function*), 174
PyUnicode_Format (*C function*), 176
PyUnicode_FromEncodedObject (*C function*), 164
PyUnicode_FromFormat (*C function*), 161
PyUnicode_FromFormatV (*C function*), 164
PyUnicode_FromKindAndData (*C function*), 161
PyUnicode_FromObject (*C function*), 164
PyUnicode_FromOrdinal (*C function*), 164
PyUnicode_FromString (*C function*), 161
PyUnicode_FromStringAndSize (*C function*), 161
PyUnicode_FromWideChar (*C function*), 168
PyUnicode_FSConverter (*C function*), 167
PyUnicode_FSDecoder (*C function*), 167
PyUnicode_GET_LENGTH (*C function*), 158
PyUnicode_GetDefaultEncoding (*C function*), 164
PyUnicode_GetLength (*C function*), 164
PyUnicode_InternFromString (*C function*), 176
PyUnicode_InternInPlace (*C function*), 176
PyUnicode_IS_ASCII (*C function*), 159
PyUnicode_IS_READY (*C function*), 178
PyUnicode_IsIdentifier (*C function*), 159
PyUnicode_Join (*C function*), 174
PyUnicode_KIND (*C function*), 158
PyUnicode_MAX_CHAR_VALUE (*C function*), 159
PyUnicode_New (*C function*), 160
PyUnicode_Partition (*C function*), 174
PyUnicode_READ (*C function*), 159
PyUnicode_READ_CHAR (*C function*), 159
PyUnicode_ReadChar (*C function*), 165
PyUnicode_READY (*C function*), 178
PyUnicode_Replace (*C function*), 175
PyUnicode_Resize (*C function*), 165
PyUnicode_RichCompare (*C function*), 175
PyUnicode_RPartition (*C function*), 174
PyUnicode_RSplit (*C function*), 174
PyUnicode_Split (*C function*), 173
PyUnicode_Splitlines (*C function*), 174
PyUnicode_Substring (*C function*), 165
PyUnicode_Tailmatch (*C function*), 174
PyUnicode_Translate (*C function*), 173
PyUnicode_Type (*C var*), 157
PyUnicode_WRITE (*C function*), 158
PyUnicode_WriteChar (*C function*), 165
PyUnicodeDecodeError_Create (*C function*), 61
PyUnicodeDecodeError_GetEncoding (*C function*), 61
PyUnicodeDecodeError_GetEnd (*C function*), 62
PyUnicodeDecodeError_GetObject (*C function*), 61
PyUnicodeDecodeError_GetReason (*C function*), 62
PyUnicodeDecodeError_GetStart (*C function*), 61
PyUnicodeDecodeError_SetEnd (*C function*), 62
PyUnicodeDecodeError_SetReason (*C function*), 62
PyUnicodeDecodeError_SetStart (*C function*), 61
PyUnicodeEncodeError_GetEncoding (*C function*), 61
PyUnicodeEncodeError_GetEnd (*C function*), 62
PyUnicodeEncodeError_GetObject (*C function*), 61
PyUnicodeEncodeError_GetReason (*C function*), 62
PyUnicodeEncodeError_GetStart (*C function*), 61
PyUnicodeEncodeError_SetEnd (*C function*), 62
PyUnicodeEncodeError_SetReason (*C function*), 62
PyUnicodeEncodeError_SetStart (*C function*), 61
PyUnicodeIter_Type (*C var*), 157
PyUnicodeObject (*C type*), 158
PyUnicodeTranslateError_GetEnd (*C function*), 62
PyUnicodeTranslateError_GetObject (*C function*), 61
PyUnicodeTranslateError_GetReason (*C function*), 62

PyUnicodeTranslateError_GetStart (*C function*), 61
PyUnicodeTranslateError_SetEnd (*C function*), 62
PyUnicodeTranslateError_SetReason (*C function*), 62
PyUnicodeTranslateError_SetStart (*C function*), 61
PyUnicodeWriter (*C type*), 176
PyUnicodeWriter_Create (*C function*), 177
PyUnicodeWriter_DecodeUTF8Stateful (*C function*), 178
PyUnicodeWriter_Discard (*C function*), 177
PyUnicodeWriter_Finish (*C function*), 177
PyUnicodeWriter_Format (*C function*), 178
PyUnicodeWriter_WriteASCII (*C function*), 177
PyUnicodeWriter_WriteChar (*C function*), 177
PyUnicodeWriter_WriteRepr (*C function*), 178
PyUnicodeWriter_WriteStr (*C function*), 177
PyUnicodeWriter_WriteSubstring (*C function*), 178
PyUnicodeWriter_WriteUCS4 (*C function*), 177
PyUnicodeWriter_WriteUTF8 (*C function*), 177
PyUnicodeWriter_WriteWideChar (*C function*), 177
PyUnstable, 13
PyUnstable_AtExit (*C function*), 231
PyUnstable_Code_GetExtra (*C function*), 198
PyUnstable_Code_GetFirstFree (*C function*), 194
PyUnstable_Code_New (*C function*), 194
PyUnstable_Code_NewWithPosOnlyArgs (*C function*), 194
PyUnstable_Code_SetExtra (*C function*), 198
PyUnstable_EnableTryIncRef (*C function*), 112
PyUnstable_Eval_RequestCodeExtraIndex (*C function*), 198
PyUnstable_Exc_PrepReraiseStar (*C function*), 61
PyUnstable_GC_VisitObjects (*C function*), 353
PyUnstable_InterpreterFrame_GetCode (*C function*), 216
PyUnstable_InterpreterFrame_GetLasti (*C function*), 216
PyUnstable_InterpreterFrame_GetLine (*C function*), 216
PyUnstable_IsImmortal (*C function*), 111
PyUnstable_Long_CompactValue (*C function*), 148
PyUnstable_Long_IsCompact (*C function*), 147
PyUnstable_Module_SetGIL (*C function*), 207
PyUnstable_Object_ClearWeakRefsNoCallbacks (*C function*), 212
PyUnstable_Object_EnableDeferredRefcount (*C function*), 110
PyUnstable_Object_GC_NewWithExtraData (*C function*), 351
PyUnstable_Object_IsUniquelyReferenced (*C function*), 113
PyUnstable_Object_IsUniqueReferencedTemporary (*C function*), 110
PyUnstable_PerfMapState_Fini (*C function*), 101
PyUnstable_PerfMapState_Init (*C function*), 101
PyUnstable_TryIncRef (*C function*), 111
PyUnstable_Type_AssignVersionTag (*C function*), 136
PyUnstable_WritePerfMapEntry (*C function*), 101
PyVarObject (*C type*), 301
PyVarObject_HEAD_INIT (*C macro*), 303
PyVarObject.ob_size (*C member*), 301
PyVectorcall_Call (*C function*), 115
PyVectorcall_Function (*C function*), 114
PyVectorcall_NARGS (*C function*), 114
PyWeakref_Check (*C function*), 211
PyWeakref_CheckProxy (*C function*), 211
PyWeakref_CheckRef (*C function*), 211
PyWeakref_GET_OBJECT (*C function*), 212
PyWeakref_GetObject (*C function*), 211
PyWeakref_GetRef (*C function*), 211
PyWeakref_IsDead (*C function*), 212
PyWeakref_NewProxy (*C function*), 211
PyWeakref_NewRef (*C function*), 211
PyWideStringList (*C type*), 262
PyWideStringList_Append (*C function*), 262
PyWideStringList_Insert (*C function*), 262
PyWideStringList.items (*C member*), 262
PyWideStringList.length (*C member*), 262
PyWrapper_New (*C function*), 208

Q

qualified name（限定名稱）, **378**

R

READ_RESTRICTED（C 巨集）, 307
READONLY（C 巨集）, 307
realloc（C 函式）, 283
reference count（參照計數）, **378**
regular package（正規套件）, **378**
releasebufferproc (*C type*), 347
REPL, **379**
repr
    built-in function（🄵建函式）, 321
    bulit-in function（🄵建函式）, 107
reprfunc (*C type*), 346
RESTRICTED（C 巨集）, 307
richcmpfunc (*C type*), 347

S

search（搜尋）
    path（路徑）, module（模組）, 11
    path（路徑）, 模組, 229, 233
sendfunc (*C type*), 347
sequence（序列）, **379**
    object（物件）, 154
set comprehension（集合綜合運算）, **379**
set_all(), 8
setattrfunc (*C type*), 346
setattrofunc (*C type*), 346

setswitchinterval （sys 模組中）, 235
setter (*C type*), 310
set（集合）
    object（物件）, 188
SIGINT（C 巨集）, 59
signal（訊號）
    module（模組）, 59
single dispatch（單一調度）, **379**
SIZE_MAX（C 巨集）, 143
slice（切片）, **379**
soft deprecated（軟性🄵用）, **379**
special
    method（方法）, 379
special method（特殊方法）, **379**
ssizeargfunc (*C type*), 347
ssizeobjargproc (*C type*), 347
statement（陳述式）, **379**
static type checker（🄵態型🄵檢查器）, **379**
staticmethod
    built-in function（🄵建函式）, 305
stderr（sys 模組中）, 244, 245
stdin（sys 模組中）, 244, 245
stdlib（標准函式庫）, **379**
stdout（sys 模組中）, 244, 245
strerror（C 函式）, 53
string（字串）
    PyObject_Str（C 函式）, 107
strong reference（🄵參照）, **379**
structmember.h, 310
sum_list(), 8
sum_sequence(), 9, 10
sys
    module（模組）, 11
    模組, 229, 244, 245
SystemError（🄵建例外）, 200

T

t-strings（t 字串）, **380**
t-string（t 字串）, **380**
T_BOOL（C 巨集）, 310
T_BYTE（C 巨集）, 310
T_CHAR（C 巨集）, 310
T_DOUBLE（C 巨集）, 310
T_FLOAT（C 巨集）, 310
T_INT（C 巨集）, 310
T_LONGLONG（C 巨集）, 310
T_LONG（C 巨集）, 310
T_NONE (*C macro*), 310
T_OBJECT (*C macro*), 310
T_OBJECT_EX（C 巨集）, 310
T_PYSSIZET（C 巨集）, 310
T_SHORT（C 巨集）, 310
T_STRING_INPLACE（C 巨集）, 310
T_STRING（C 巨集）, 310
T_UBYTE（C 巨集）, 310
T_UINT（C 巨集）, 310
T_ULONGULONG（C 巨集）, 310
T_ULONG（C 巨集）, 310
T_USHORT（C 巨集）, 310
ternaryfunc (*C type*), 347
text encoding（文字編碼）, **380**
text file（文字檔案）, **380**
thread state, **380**
token, **380**
traverseproc (*C type*), 352
triple-quoted string（三引號🄵字串）, **380**
tuple（元組）
    built-in function（🄵建函式）, 122, 183
    object（物件）, 179
type alias（型🄵🄵名）, **380**
type hint（型🄵提示）, **381**
type（型🄵）, **380**
    bulit-in function（🄵建函式）, 108
    object（物件）, 6, 133

U

ULONG_MAX（C 巨集）, 143
unaryfunc (*C type*), 347
universal newlines（通用🄵行字元）, **381**
USE_STACKCHECK（C 巨集）, 76

V

variable annotation（變數🄵釋）, **381**
vectorcallfunc (*C type*), 114
version（sys 模組中）, 233, 234
virtual environment（🄵擬環境）, **381**
virtual machine（🄵擬機器）, **381**
visitproc (*C type*), 352

W

walrus operator（海象運算子）, **381**
標准函式庫, **379**
模組
    __main__, 229, 244, 245
    _thread, 238
    builtins（🄵建）, 229, 244, 245
    search（搜尋） path（路徑）, 229, 233
    sys, 229, 244, 245
WRITE_RESTRICTED（C 巨集）, 307

Z

Zen of Python（Python 之🄵）, **381**