

---

# The Python/C API

發[F] 3.11.11

Guido van Rossum and the Python development team

12 月 07, 2024

Python Software Foundation  
Email: [docs@python.org](mailto:docs@python.org)



<b>1</b>	<b>簡介</b>	<b>3</b>
1.1	編寫標準	3
1.2	引入檔案 (include files)	3
1.3	有用的巨集	4
1.4	物件、型別和參照計數	6
1.4.1	參照計數	7
1.4.2	型別	9
1.5	例外	10
1.6	嵌入式 Python	11
1.7	除錯建置	12
<b>2</b>	<b>C API 穩定性</b>	<b>13</b>
2.1	穩定的應用程式二進位介面	13
2.1.1	受限 C API	13
2.1.2	穩定 ABI	14
2.1.3	受限 API 範圍和性能	14
2.1.4	受限 API 注意事項	14
2.2	平台注意事項	15
2.3	受限 API 的內容	15
<b>3</b>	<b>极高层级 API</b>	<b>39</b>
<b>4</b>	<b>參照計數</b>	<b>43</b>
<b>5</b>	<b>例外處理</b>	<b>45</b>
5.1	打印和清理	45
5.2	拋出異常	46
5.3	發出警告	48
5.4	查詢錯誤指示器	49
5.5	信號處理	51
5.6	例外類型	52
5.7	例外物件	52
5.8	Unicode 異常對象	53
5.9	遞歸控制	54
5.10	標準異常	54
5.11	標準警告類別	56
<b>6</b>	<b>工具</b>	<b>57</b>
6.1	作業系統工具	57
6.2	系統函式	60
6.3	行程 (Process) 控制	62

6.4	引入模組	62
6.5	数据 marshal 操作支持	65
6.6	剖析引數與建置數值	66
6.6.1	解析参数	67
6.6.2	创建变量	72
6.7	字串轉碼與格式化	74
6.8	PyHash API	76
6.9	反射	77
6.10	编解码器注册与支持功能	77
6.10.1	Codec 查找 API	78
6.10.2	用于 Unicode 编码错误处理程序的注册表 API	78
<b>7</b>	<b>抽象物件層 (Abstract Objects Layer)</b>	<b>79</b>
7.1	对象协议	79
7.2	呼叫協定 (Call Protocol)	83
7.2.1	<i>tp_call</i> 協定	83
7.2.2	Vectorcall 協定	83
7.2.3	物件呼叫 API	85
7.2.4	呼叫支援 API	87
7.3	数字协议	87
7.4	序列协议	90
7.5	映射协议	92
7.6	代器協議	93
7.7	緩衝協定 (Buffer Protocol)	94
7.7.1	缓冲区结构	95
7.7.2	缓冲区请求的类型	96
7.7.3	复杂数组	98
7.7.4	缓冲区相关函数	99
7.8	舊式緩衝協定 (Buffer Protocol)	100
<b>8</b>	<b>具體物件層</b>	<b>103</b>
8.1	基礎物件	103
8.1.1	类型对象	103
8.1.2	None 物件	107
8.2	數值物件	107
8.2.1	整數物件	107
8.2.2	Boolean (布林) 物件	110
8.2.3	浮點數 (Floating Point) 物件	110
8.2.4	复数对象	112
8.3	序列物件	113
8.3.1	位元組物件 (Bytes Objects)	113
8.3.2	位元組串列物件 (Byte Array Objects)	115
8.3.3	Unicode 物件與編碼	116
8.3.4	元組 (Tuple) 物件	133
8.3.5	结构序列对象	134
8.3.6	List (串列) 物件	135
8.4	容器物件	136
8.4.1	字典物件	136
8.4.2	集合对象	139
8.5	函式物件	141
8.5.1	函式物件 (Function Objects)	141
8.5.2	實例方法物件 (Instance Method Objects)	142
8.5.3	方法物件 (Method Objects)	142
8.5.4	Cell 物件	143
8.5.5	程式碼物件	143
8.6	其他物件	145
8.6.1	檔案物件 (File Objects)	145
8.6.2	模組物件模組	146

8.6.3	迭代器 (Iterator) 物件	153
8.6.4	Descriptor (描述器) 物件	153
8.6.5	切片物件	154
8.6.6	MemoryView 物件	155
8.6.7	弱参照物件	156
8.6.8	Capsule 对象	157
8.6.9	帧对象	158
8.6.10	生成器 (Generator) 物件	159
8.6.11	Coroutine (协程) 物件	160
8.6.12	上下文变量对象	160
8.6.13	DateTime 物件	161
8.6.14	类型提示物件	165
<b>9</b>	<b>初始化, 最终化和线程</b>	<b>167</b>
9.1	在 Python 初始化之前	167
9.2	全局配置变量	168
9.3	初始化和最终化解释器	170
9.4	进程级参数	171
9.5	线程状态和全局解释器锁	174
9.5.1	从扩展扩展代码中释放 GIL	175
9.5.2	非 Python 创建的线程	175
9.5.3	有关 fork() 的注意事项	176
9.5.4	高阶 API	176
9.5.5	低阶 API	178
9.6	子解释器支持	181
9.6.1	错误和警告	182
9.7	异步通知	182
9.8	分析和跟踪	183
9.9	高级调试器支持	184
9.10	线程本地存储支持	184
9.10.1	线程专属存储 (TSS) API	185
9.10.2	线程本地存储 (TLS) API	186
<b>10</b>	<b>Python 初始化配置</b>	<b>187</b>
10.1	范例	187
10.2	PyWideStringList	188
10.3	PyStatus	189
10.4	PyPreConfig	190
10.5	使用 PyPreConfig 预初始化 Python	192
10.6	PyConfig	193
10.7	使用 PyConfig 初始化	203
10.8	隔离配置	204
10.9	Python 配置	205
10.10	Python 路径配置	205
10.11	Py_RunMain()	206
10.12	Py_GetArgcArgv()	206
10.13	多阶段初始化私有暂定 API	207
<b>11</b>	<b>記憶體管理</b>	<b>209</b>
11.1	總覽	209
11.2	分配器域	210
11.3	原始内存接口	210
11.4	内存接口	211
11.5	对象分配器	212
11.6	默认内存分配器	213
11.7	自定义内存分配器	213
11.8	Python 内存分配器的调试钩子	215
11.9	pymalloc 分配器	216
11.9.1	自定义 pymalloc Arena 分配器	216

11.10	tracemalloc C API	217
11.11	範例	217
<b>12</b>	<b>对象实现支持</b>	<b>219</b>
12.1	在 heap 上分配物件	219
12.2	通用物件結構	220
12.2.1	基本的对象类型和宏	220
12.2.2	实现函数和方法	222
12.2.3	访问扩展类型的属性	224
12.3	型物件	226
12.3.1	快速参考	226
12.3.2	PyObject 定义	231
12.3.3	PyObject 槽位	232
12.3.4	PyVarObject 槽位	233
12.3.5	PyObject 槽	233
12.3.6	静态类型	250
12.3.7	堆类型	250
12.4	数字对象结构体	250
12.5	映射对象结构体	252
12.6	序列对象结构体	253
12.7	缓冲区对象结构体	253
12.8	异步对象结构体	254
12.9	槽位类型 typedef	255
12.10	範例	257
12.11	使对象类型支持循环垃圾回收	259
12.11.1	控制垃圾回收器状态	261
<b>13</b>	<b>API 和 ABI 版本管理</b>	<b>263</b>
<b>A</b>	<b>術語表</b>	<b>265</b>
<b>B</b>	<b>關於這些文件</b>	<b>281</b>
B.1	Python 文件的貢獻者們	281
<b>C</b>	<b>沿革與授權</b>	<b>283</b>
C.1	軟體沿革	283
C.2	關於存取或以其他方式使用 Python 的合約條款	284
C.2.1	用於 PYTHON 3.11.11 的 PSF 授權合約	284
C.2.2	用於 PYTHON 2.0 的 BEOPEN.COM 授權合約	285
C.2.3	用於 PYTHON 1.6.1 的 CNRI 授權合約	286
C.2.4	用於 PYTHON 0.9.0 至 1.2 的 CWI 授權合約	287
C.2.5	用於 PYTHON 3.11.11 文件程式碼的 ZERO-CLAUSE BSD 授權	287
C.3	被收軟體的授權與致謝	287
C.3.1	Mersenne Twister	288
C.3.2	Sockets	288
C.3.3	非同步 socket 服務	289
C.3.4	Cookie 管理	289
C.3.5	執行追	290
C.3.6	UUencode 與 UUdecode 函式	290
C.3.7	XML 遠端程序呼叫	291
C.3.8	test_epoll	291
C.3.9	Select kqueue	292
C.3.10	SipHash24	292
C.3.11	strtod 與 dtoa	293
C.3.12	OpenSSL	293
C.3.13	expat	296
C.3.14	libffi	297
C.3.15	zlib	297
C.3.16	cfuhash	298

C.3.17	libmpdec	298
C.3.18	W3C C14N 測試套件	299
C.3.19	audioop	300
C.3.20	asyncio	300
<b>D</b>	<b>版權宣告</b>	<b>301</b>
	<b>索引</b>	<b>303</b>





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，這份手冊記述了可使用的 API（應用程式介面）。在 `extending-index` 中也有相關的內容，它描述了編寫擴充的一般原則，但沒有詳細說明 API 函式。



---

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++，但為簡潔起見，通常將其稱作 Python/C API。使用 Python/C API 有兩個不同的原因，第一個是為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 作為零件使用；這種技術通常在應用程式中稱作 *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，儘管人們從早期就將 Python 嵌入到其他應用程式中，但嵌入 Python 的過程並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 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。

## 1.2 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準標頭：<stdio.h>、<string.h>、<errno.h>、<limits.h>、<assert.h> 和 <stdlib.h>（如果可用）。

---

**備註：**由於 Python 可能會定義一些會影響某些系統上標準標頭檔的預處理器 (pre-processor)，因此你必須在引入任何標準標頭檔之前引入 `Python.h`。

建議在引入 `Python.h` 之前都要定義 `PY_SSIZE_T_CLEAN`。有關此巨集的說明，請參閱[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（另外透過標準標頭檔引入的除外）都具有 `Py` 或 `_Py` 前綴。以 `_Py` 開頭的名稱供 Python 實作內部使用，擴充編寫者不應使用。結構成員名稱有保留前綴。

**備註：** 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，且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 上它們位於目錄 `prefix/include/pythonversion/` 和 `exec_prefix/include/pythonversion/`，其中 `prefix` 和 `exec_prefix` 由 Python 的 `configure` 腳本的相應參數定義，`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) 用的安裝目錄。

要引入標頭，請將兩個（如果不同）目錄放在編譯器的引入搜索路徑 (search path) 中。不要將父目錄放在搜索路徑上，然後使用 `#include <pythonX.Y/Python.h>`；這會在多平台建置上壞掉，因為 `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，儘管 API 完全使用 C 來定義，但標頭檔適當地將入口點聲明為 `extern "C"`。因此，無需執行任何特殊操作即可使用 C++ 中的 API。

## 1.3 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`），其他是更通用的工具程式。以下不一定是完整的列表。

### PyMODINIT\_FUNC

聲明擴展模块 `PyInit` 初始化函数。函数返回类型为 `PyObject*`。该宏声明了平台所要求的任何特殊链接声明，并针对 C++ 将函数声明为 `extern "C"`。

初始化函数必须命名为 `PyInit_name`，其中 `name` 是模块名称，并且应在模块文件中定义的唯一非 `static` 项。例如：

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

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

### Py\_ABS(x)

回傳 `x` 的絕對值。

在 3.3 版新加入。

### Py\_ALWAYS\_INLINE

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

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 是在調試模式下構建的 (即定義了 `Py_DEBUG` 宏), 則 `Py_ALWAYS_INLINE` 宏將不做任何事情。

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 版新加入。

#### **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` (即如果設定了 `Py_IgnoreEnvironmentFlag`) 則回傳 `NULL`。

#### **Py\_MAX** (x, y)

回傳 `x` 和 `y` 之間的最大值。

在 3.3 版新加入。

#### **Py\_MEMBER\_SIZE** (type, member)

以位元組單位回傳結構 (type) member 的大小。

在 3.6 版新加入。

#### **Py\_MIN** (x, y)

回傳 `x` 和 `y` 之間的最小值。

在 3.3 版新加入。

#### **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 版新加入。

#### **Py\_STRINGIFY** (x)

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

在 3.4 版新加入。

#### **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況下可以到達，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。在這種情況下，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 版新加入。

#### **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 版新加入。

#### **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況下建置，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況下建置 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 sqlite_row_methods[] = {
    {"keys", (PyCFunction)sqlite_row_keys, METH_NOARGS,
     PyDoc_STR("Returns the keys of the row.")},
    {NULL, NULL}
};
```

## 1.4 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。由於在大多數情況下，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麼種類的物件 (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。

### 1.4.1 參照計數

引用計數之所以重要是因為現有計算機的內存大小是有限的（並且往往限制得很嚴格）；它會計算有多少不同的地方對一個對象進行了 *strong reference*。這些地方可以是另一個對象，也可以是全局（或靜態）C 變量，或是某個 C 函數中的局部變量。當某個對象的最后一个 *strong reference* 被釋放時（即其引用計數變為零），該對象就會被取消分配。如果該對象包含對其他對象的引用，則會釋放這些引用。如果不再有其他對象的引用，這些對象也會同樣地被取消分配，依此類推。（在這裡對象之間的相互引用顯然是個問題；目前的解決辦法，就是“不要這樣做”。）

對於引用計數總是會顯式地執行操作。通常的做法是使用 `Py_INCREF()` 宏來獲取對象的新引用（即讓引用計數加一），並使用 `Py_DECREF()` 宏來釋放引用（即讓引用計數減一）。`Py_DECREF()` 宏比 `incr` 宏複雜得多，因為它必須檢查引用計數是否為零然後再調用對象的釋放器。釋放器是一個函數指針，它包含在對象的類型結構體中。如果對象是複合對象類型，如列表，則特定於類型的釋放器會負責釋放對象中包含的其他對象的引用，並執行所需的其他終結化操作。引用計數不會發生溢出；用於保存引用計數的位數至少會與虛擬內存中不同內存位置的位數相同（假設 `sizeof(Py_ssize_t) >= sizeof(void*)`）。因此，引用計數的遞增是一個簡單的操作。

沒有必要為每個包含指向對象指針的局部變量持有 *strong reference*（即增加引用計數）。理論上說，當變量指向對象時對象的引用計數就會加一，而當變量離開其作用域時引用計數就會減一。不過，這兩種情況會相互抵消，所以最后引用計數並沒有改變。使用引用計數的唯一真正原因在於只要我們的變量指向對象就可以防止對象被釋放。只要我們知道至少還有一個指向某對象的引用與我們的變量同時存在，就沒有必要臨時獲取一個新的 *strong reference*（即增加引用計數）。出現引用計數增加的一種重要情況是對象作為參數被傳遞給擴展模塊中的 C 函數而這些函數又在 Python 中被調用；調用機制會保證在調用期間對每個參數持有一個引用。

然而，一個常見的陷阱是從列表中提取對象並在不獲取新引用的情況下將其保留一段時間。某個其他操作可能在無意中從列表中移除該對象，釋放這個引用，並可能撤銷分配其資源。真正的危險在於看似無害的操作可能會發起調用任意的 Python 代碼來做這件事；有一條代碼路徑允許控制權從 `Py_DECREF()` 流回到用戶，因此幾乎任何操作都有潛在的危險。

安全的做法是始終使用泛型操作（名稱以 `PyObject_`、`PyNumber_`、`PySequence_` 或 `PyMapping_` 開頭的函數）。這些操作總是為其返回的對象創建一個新的 *strong reference*（即增加引用計數）。這使得調用者有責任在獲得結果之後調用 `Py_DECREF()`；這種做法很快就能習慣成自然。

#### 參照計數詳細資訊

Python/C API 中函數的引用計數最好是使用引用所有權來解釋。所有權是關聯到引用，而不是對象（對象不能被擁有：它們總是會被共享）。“擁有一個引用”意味著當不再需要該引用時必須在其上調用 `Py_DECREF`。所有權也可以被轉移，這意味著接受該引用所有權的代碼在不再需要它時必須通過調用 `Py_DECREF()` 或 `Py_XDECREF()` 來最終釋放它 --- 或是繼續轉移這個責任（通常是轉給其調用方）。當一個函數將引用所有權轉給其調用方時，則稱調用方收到一個新的引用。當未轉移所有權時，則稱調用方是借入這個引用。對於 *borrowed reference* 來說不需要任何額外操作。

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()` 來獲取另一個參照。



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");
```

在对条目使用 `PyObject_SetItem()` 等操作时更常见的做法是只借入引用，比如将参数传递给你正在编写的函数。在这种情况下，它们在引用方面的行为更为清晰，因为你不必为了把引用转走而获取一个新的引用（“让它被偷取”）。例如，这个函数将列表（实际上是任何可变序列）中的所有条目都设为给定的条目：

```
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：在很多情況下，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 中獲取相同的項目（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)
```

(繼續下一頁)



(繼續上一頁)

```

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

```

```

long
sum_sequence(PyObject *sequence)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;
    n = PySequence_Length(sequence);
    if (n < 0)
        return -1; /* Has no length */
    for (i = 0; i < n; i++) {
        item = PySequence_GetItem(sequence, i);
        if (item == NULL)
            return -1; /* Not a sequence, or other failure */
        if (PyLong_Check(item)) {
            value = PyLong_AsLong(item);
            Py_DECREF(item);
            if (value == -1 && PyErr_Occurred())
                /* Integer too big to fit in a C long, bail out */
                return -1;
            total += value;
        }
        else {
            Py_DECREF(item); /* Discard reference ownership */
        }
    }
    return total;
}

```

## 1.4.2 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`

屬於穩定 ABI。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。

## 1.5 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，在那兒它們透過堆回溯 (stack trace) 回報給使用者。

然而，對於 C 開發者來說，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。如果有另外文件記錄，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錄於文件。

例外的狀態會在個執行緒的存儲空間 (per-thread storage) 中維護（這相當於在非執行緒應用程式中使用全域存儲空間）。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（儘管不是最通用的）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；它還替回溯中被堆棧 (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。它應該回傳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()` 示例是一个检测异常并将其传递出去的简单例子。碰巧的是这个示例在检测到错误时不需要清理所拥有的任何引用。下面的示例函数展示了一些错误清理操作。首先，为了提醒你 Python 的受欢迎程度，我们展示了等价的 Python 代码：

```
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();
```

(繼續下一頁)

(繼續上一頁)

```

    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為 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為 NULL 以使其能順利作用；同樣地，回傳值被初始化為 -1（失敗），且僅在最後一次呼叫成功後才設定為成功。

## 1.6 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（特別是在 Unix 和 Windows 上，儘管細節略有不同），`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，根據其對標準 Python 直譯器可執行檔案位置的最佳猜測來計算模組搜索路徑。或者更詳細地說，它會在 shell 命令搜索路徑（環境變數 `PATH`）中找到名為 `python` 的可執行檔案，在其父目錄中查找一個名為 `lib/pythonX.Y` 的目錄的相對位置。

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/pythonX.Y` 中。（事實上這個特定的路徑也是「後備 (fallback)」位置，當在 `PATH` 中找不到名為 `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` 在標準路徑前面插入額外的目錄。

嵌入的應用程式可以透過在呼叫 `Py_Initialize()` 之前呼叫 `Py_SetProgramName(file)` 來引導搜索。請注意 `PYTHONHOME` 仍然覆蓋它且 `PYTHONPATH` 仍然插入在標準路徑的前面。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 當前處於初始化狀態，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。

## 1.7 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)，因此預設情況下不用它們。

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，支援追蹤計數、記憶體分配器除錯或對主直譯器圈進行低階分析的建置。本節的其餘部分將僅描述最常用的建置。

### Py\_DEBUG

在定義了 `Py_DEBUG` 宏的情況下編譯解釋器將產生通常所稱的 Python 調試構建版。`Py_DEBUG` 在 Unix 編譯版中是通过添加 `--with-pydebug` 到 `./configure` 命令來啟用的。它也可以通过提供非 Python 專屬的 `_DEBUG` 宏來啟用。當 `Py_DEBUG` 在 Unix 編譯版中啟用時，編譯器優化將被禁用。

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` 來用參照追蹤（參見調用 `--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)。全體分配也有被追蹤。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`。

---

C API 穩定性

---

Python 的 C API 被包含在向後相容性策略 [PEP 387](#) 中。雖然 C API 會隨著每個次要版本（例如從 3.9 到 3.10）而變化，但大多數變化都是與原始碼相容的，通常只是加入新的 API。更改現有 API 或刪除 API 僅在長期後或修復嚴重問題時進行。

CPython 的应用程序二进制接口（ABI）可以跨微版本向上和向下兼容（在以相同方式编译的情况下，参见下文[平台注意事項](#)一节）。因此，针对 Python 3.10.0 编译的代码将适用于 3.10.8，反之亦然，但对于 3.9.x 和 3.11.x 则需要单独编译。

帶有底線前綴的名稱是私有 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。

## 2.1 穩定的應用程式二進位介面

简单起见，本文档只讨论了 扩展，但受限 API 和稳定 ABI 对于 API 的所有用法都能发挥相同的作用—例如嵌入版的 Python 等。

### 2.1.1 受限 C API

Python 3.2 引入了 受限 API，它是 Python 的 C API 的一个子集。只使用受限 API 扩展可以一次编译即适用于多个 Python 版本。受限 API 的内容[如下所示](#)。

#### **Py\_LIMITED\_API**

在包含 `Python.h` 之前定義此巨集以選擇只使用受限 API，[選擇](#)受限 API 版本。

將 `Py_LIMITED_API` 定義為與你的擴展所支持的最低 Python 版本的 `PY_VERSION_HEX` 的值。擴展將無需重編譯即可適用於從該指定版本開始的所有 Python 3 發布版，並可使用到該版本為止所引入的受限 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` 定義為 3，這與 `0x03020000`（Python 3.2，引入了受限 API 的版本）相同。

## 2.1.2 穩定 ABI

为启用此特性，Python 提供了一个 穩定 ABI: 将能跨 Python 3.x 版本保持兼容的一组符号。

穩定 ABI 包含在受限 API 中对外公开的符号，但还包含其他符号—例如，为支持旧版本受限 API 所需的函数。

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 將查找並加載以 `abi3` 標識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 中的所有函式都作為函式存在於 Python 的共享函式庫中，而不僅是作為巨集。這使得它們可被用於不使用 C 預處理器 (preprocessor) 的語言。

## 2.1.3 受限 API 範圍和性能

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，因為它可以依賴 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換巨集。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。

通過省略 `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` 編譯將生成一個擴充，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)。

## 2.1.4 受限 API 注意事項

请注意使用 `Py_LIMITED_API` 进行编译 无法完全保证代码能够兼容受限 API 或穩定 ABI。`Py_LIMITED_API` 仅仅涵盖定义部分，但一个 API 还包括其他因素，如预期的语义等。

`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。特別是如果穩定 ABI 保持穩定，部分受限 API 可能會被用和除。



## 2.2 平台注意事項

ABI 的穩定性不僅取決於 Python，取決於所使用的編譯器、低層級庫和編譯器選項等。對於穩定 ABI 的目標來說，這些細節定義了一個“平台”。它們通常會依賴於 OS 類型和處理器架構等。

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<sup>②</sup>。

## 2.3 受限 API 的<sup>③</sup>容

目前受限 API 包括下面這些項：

- `PyAlter_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*
- *PyCFunctionWithKeywords*
- *PyCFunction\_Call()*
- *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_GetItemString()`
- `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_ExceptionMatches()`
- `PyErr_Fetch()`
- `PyErr_Format()`
- `PyErr_FormatV()`
- `PyErr_GetExcInfo()`
- `PyErr_GetHandledException()`
- `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_SetString()`
- `PyErr_SyntaxLocation()`
- `PyErr_SyntaxLocationEx()`
- `PyErr_WarnEx()`
- `PyErr_WarnExplicit()`
- `PyErr_WarnFormat()`
- `PyErr_WriteUnraisable()`
- `PyEval_AcquireLock()`
- `PyEval_AcquireThread()`
- `PyEval_CallFunction()`
- `PyEval_CallMethod()`
- `PyEval_CallObjectWithKeywords()`
- `PyEval_EvalCode()`
- `PyEval_EvalCodeEx()`
- `PyEval_EvalFrame()`
- `PyEval_EvalFrameEx()`
- `PyEval_GetBuiltins()`

- `PyEval_GetFrame()`
- `PyEval_GetFuncDesc()`
- `PyEval_GetFuncName()`
- `PyEval_GetGlobals()`
- `PyEval_GetLocals()`
- `PyEval_InitThreads()`
- `PyEval_ReleaseLock()`
- `PyEval_ReleaseThread()`
- `PyEval_RestoreThread()`
- `PyEval_SaveThread()`
- `PyEval_ThreadsInitialized()`
- `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_GetCause()`
- `PyException_GetContext()`
- `PyException_GetTraceback()`
- `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_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_Send()`
- `PyListIter_Type`
- `PyListRevIter_Type`
- `PyList_Append()`
- `PyList_AsTuple()`
- `PyList_GetItem()`
- `PyList_GetSlice()`
- `PyList_Insert()`
- `PyList_New()`
- `PyList_Reverse()`
- `PyList_SetItem()`
- `PyList_SetSlice()`

- `PyList_Size()`
- `PyList_Sort()`
- `PyList_Type`
- `PyLongObject`
- `PyLongRangeIter_Type`
- `PyLong_AsDouble()`
- `PyLong_AsLong()`
- `PyLong_AsLongAndOverflow()`
- `PyLong_AsLongLong()`
- `PyLong_AsLongLongAndOverflow()`
- `PyLong_AsSize_t()`
- `PyLong_AsSsize_t()`
- `PyLong_AsUnsignedLong()`
- `PyLong_AsUnsignedLongLong()`
- `PyLong_AsUnsignedLongLongMask()`
- `PyLong_AsUnsignedLongMask()`
- `PyLong_AsVoidPtr()`
- `PyLong_FromDouble()`
- `PyLong_FromLong()`
- `PyLong_FromLongLong()`
- `PyLong_FromSize_t()`
- `PyLong_FromSsize_t()`
- `PyLong_FromString()`
- `PyLong_FromUnsignedLong()`
- `PyLong_FromUnsignedLongLong()`
- `PyLong_FromVoidPtr()`
- `PyLong_GetInfo()`
- `PyLong_Type`
- `PyMap_Type`
- `PyMapping_Check()`
- `PyMapping_GetItemString()`
- `PyMapping_HasKey()`
- `PyMapping_HasKeyString()`
- `PyMapping_Items()`
- `PyMapping_Keys()`
- `PyMapping_Length()`
- `PyMapping_SetItemString()`
- `PyMapping_Size()`
- `PyMapping_Values()`



- `PyMem_Calloc()`
- `PyMem_Free()`
- `PyMem_Malloc()`
- `PyMem_Realloc()`
- `PyMemberDef`
- `PyMemberDescr_Type`
- `PyMemoryView_FromBuffer()`
- `PyMemoryView_FromMemory()`
- `PyMemoryView_FromObject()`
- `PyMemoryView_GetContiguous()`
- `PyMemoryView_Type`
- `PyMethodDef`
- `PyMethodDescr_Type`
- `PyModuleDef`
- `PyModuleDef_Base`
- `PyModuleDef_Init()`
- `PyModuleDef_Type`
- `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_AsCharBuffer()`
- `PyObject_AsFileDescriptor()`
- `PyObject_AsReadBuffer()`
- `PyObject_AsWriteBuffer()`
- `PyObject_Bytes()`
- `PyObject_Call()`
- `PyObject_CallFunction()`
- `PyObject_CallFunctionObjArgs()`
- `PyObject_CallMethod()`
- `PyObject_CallMethodObjArgs()`
- `PyObject_CallNoArgs()`
- `PyObject_CallObject()`
- `PyObject_Calloc()`
- `PyObject_CheckBuffer()`
- `PyObject_CheckReadBuffer()`
- `PyObject_ClearWeakRefs()`
- `PyObject_CopyData()`
- `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_HasAttr()`
- `PyObject_HasAttrString()`
- `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()`
- `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_AddWarnOption()`
- `PySys_AddWarnOptionUnicode()`
- `PySys_AddXOption()`
- `PySys_FormatStderr()`
- `PySys_FormatStdout()`
- `PySys_GetObject()`
- `PySys_GetXOptions()`
- `PySys_HasWarnOptions()`
- `PySys_ResetWarnOptions()`
- `PySys_SetArgv()`
- `PySys_SetArgvEx()`
- `PySys_SetObject()`
- `PySys_SetPath()`
- `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_FromModuleAndSpec()`
- `PyType_FromSpec()`
- `PyType_FromSpecWithBases()`
- `PyType_GenericAlloc()`
- `PyType_GenericNew()`
- `PyType_GetFlags()`
- `PyType_GetModule()`
- `PyType_GetModuleState()`
- `PyType_GetName()`
- `PyType_GetQualName()`
- `PyType_GetSlot()`
- `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_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_GetSize()`
- `PyUnicode_InternFromString()`
- `PyUnicode_InternImmortal()`
- `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`
- `PyWeakReference`
- `PyWeakref_GetObject()`
- `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_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_IsInitialized()`
- `Py_IsNone()`
- `Py_IsTrue()`
- `Py_LeaveRecursiveCall()`
- `Py_Main()`
- `Py_MakePendingCalls()`
- `Py_NewInterpreter()`
- `Py_NewRef()`
- `Py_ReprEnter()`
- `Py_ReprLeave()`
- `Py_SetPath()`
- `Py_SetProgramName()`
- `Py_SetPythonHome()`
- `Py_SetRecursionLimit()`
- `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`
- `getiterfunc`
- `getter`
- `hashfunc`
- `initproc`
- `inquiry`
- `iternextfunc`
- `lenfunc`
- `newfunc`
- `objobjargproc`
- `objobjproc`
- `reprfunc`
- `richcmpfunc`
- `setattrfunc`
- `setattrofunc`
- `setter`
- `ssizeargfunc`
- `ssizeobjargproc`
- `ssizessizeargfunc`
- `ssizessizeobjargproc`
- `symtable`
- `ternaryfunc`
- `traverseproc`
- `unaryfunc`
- `visitproc`



本章节的函数将允许你执行在文件或缓冲区中提供的 Python 源代码，但它们将不允许你在更细节化的方式与解释器进行交互。

这些函数中有几个可以接受特定的语法前缀符号作为形参。可用的前缀符号有 `Py_eval_input`, `Py_file_input` 和 `Py_single_input`。这些符号会在接受它们作为形参的函数文档中加以说明。

还要注意这些函数中有几个可以接受 `FILE*` 形参。有一个需要小心处理的特别问题是针对不同 C 库的 `FILE` 结构体可能是不相同且不兼容的。(至少是)在 Windows 中，动态链接的扩展实际上有可能会使用不同的库，所以应当特别注意只有在确定这些函数是由 Python 运行时所使用的相同的库创建的情况下才将 `FILE*` 形参传给它们。

**int Py\_Main** (int argc, wchar\_t \*\*argv)

属于稳定 ABI。针对标准解释器的主程序。嵌入了 Python 的程序将可使用此程序。所提供的 `argc` 和 `argv` 形参应当与传给 C 程序的 `main()` 函数的形参相同（将根据用户的语言区域转换为）。一个重要的注意事项是参数列表可能会被修改（但参数列表中字符串所指向的内容不会被修改）。如果解释器正常退出（即未引发异常）则返回值将为 0，如果解释器因引发异常而退出则返回 1，或者如果形参列表不能表示有效的 Python 命令行则返回 2。

请注意如果引发了一个在其他场合下未处理的 `SystemExit`，此函数将不会返回 1，而是退出进程，只要 `Py_InspectFlag` 还未被设置。

**int Py\_BytesMain** (int argc, char \*\*argv)

属于稳定 ABI 自 3.8 版起。类似于 `Py_Main()` 但 `argv` 是一个包含字节串的数组。

在 3.8 版新加入。

**int PyRun\_AnyFile** (FILE \*fp, const char \*filename)

这是针对下面 `PyRun_AnyFileExFlags()` 的简化版接口，将 `closeit` 设为 0 而将 `flags` 设为 NULL。

**int PyRun\_AnyFileFlags** (FILE \*fp, const char \*filename, *PyCompilerFlags* \*flags)

这是针对下面 `PyRun_AnyFileExFlags()` 的简化版接口，将 `closeit` 参数设为 0。

**int PyRun\_AnyFileEx** (FILE \*fp, const char \*filename, int closeit)

这是针对下面 `PyRun_AnyFileExFlags()` 的简化版接口，将 `flags` 参数设为 NULL。

**int PyRun\_AnyFileExFlags** (FILE \*fp, const char \*filename, int closeit, *PyCompilerFlags* \*flags)

如果 `fp` 指向一个关联到交互设备（控制台或终端输入或 Unix 伪终端）的文件，则返回 `PyRun_InteractiveLoop()` 的值，否则返回 `PyRun_SimpleFile()` 的结果。`filename` 会使用文件系统的编码格式 (`sys.getfilesystemencoding()`) 来解码。如果 `filename` 为 NULL，此

函数会使用 "???" 作为文件名。如果 *closeit* 为真值，文件会在 `PyRun_SimpleFileExFlags()` 返回之前被关闭。

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

这是针对下面 `PyRun_SimpleStringFlags()` 的简化版接口，将 `PyCompilerFlags*` 参数设为 `NULL`。

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

根据 *flags* 参数，在 `__main__` 模块中执行 Python 源代码。如果 `__main__` 尚不存在，它将被创建。成功时返回 0，如果引发异常则返回 -1。如果发生错误，则将无法获得异常信息。对于 *flags* 的含义，请参阅下文。

请注意如果引发了一个在其他场合下未处理的 `SystemExit`，此函数将不会返回 -1，而是退出进程，只要 `Py_InspectFlag` 还未被设置。

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

这是针对下面 `PyRun_SimpleFileExFlags()` 的简化版接口，将 *closeit* 设为 0 而将 *flags* 设为 `NULL`。

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

这是针对下面 `PyRun_SimpleFileExFlags()` 的简化版接口，将 *flags* 设为 `NULL`。

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

类似于 `PyRun_SimpleStringFlags()`，但 Python 源代码是从 *fp* 读取而不是一个内存中的字符串。*filename* 应为文件名，它将使用 *filesystem encoding and error handler* 来解码。如果 *closeit* 为真值，则文件将在 `PyRun_SimpleFileExFlags()` 返回之前被关闭。

---

**備註：** 在 Windows 上，*fp* 应当以二进制模式打开（即 `fopen(filename, "rb")`）。否则，Python 可能无法正确地处理使用 LF 行结束符的脚本文件。

---

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

这是针对下面 `PyRun_InteractiveOneFlags()` 的简化版接口，将 *flags* 设为 `NULL`。

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

根据 *flags* 参数读取并执行来自与交互设备相关联的文件的一条语句。用户将得到使用 `sys.ps1` 和 `sys.ps2` 的提示。*filename* 将使用 *filesystem encoding and error handler* 来解码。

当输入被成功执行时返回 0，如果引发异常则返回 -1，或者如果存在解析错误则返回来自作为 Python 的组成部分发布的 `errcode.h` 包括文件的错误代码。（请注意 `errcode.h` 并未被 `Python.h` 所包括，因此如果需要则必须专门地包括。）

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

这是针对下面 `PyRun_InteractiveLoopFlags()` 的简化版接口，将 *flags* 设为 `NULL`。

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

读取并执行来自与交互设备相关联的语句直至到达 EOF。用户将得到使用 `sys.ps1` 和 `sys.ps2` 的提示。*filename* 将使用 *filesystem encoding and error handler* 来解码。当位于 EOF 时将返回 0，或者当失败时将返回一个负数。

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

属于稳定 ABI。可以被设为指向一个原型为 `int func(void)` 的函数。该函数将在 Python 的解释器提示符即将空闲并等待用户从终端输入时被调用。返回值会被忽略。重写这个钩子可被用来将解释器的提示符集成到其他事件循环中，就像 Python 码中 `Modules/_tkinter.c` 所做的那样。

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

可以被设为指向一个原型为 `char *func(FILE *stdin, FILE *stdout, char *prompt)` 的函数，重写被用来读取解释器提示符的一行输入的默认函数。该函数被预期为如果字符串 *prompt* 不为 `NULL` 就输出它，然后从所提供的标准输入文件读取一行输入，并返回结果字符串。例如，`readline` 模块将这个钩子设置为提供行编辑和 `tab` 键补全等功能。



结果必须是一个由 `PyMem_RawMalloc()` 或 `PyMem_RawRealloc()` 分配的字符串，或者如果发生错误则为 `NULL`。

在 3.4 版的變更: 结果必须由 `PyMem_RawMalloc()` 或 `PyMem_RawRealloc()` 分配，而不是由 `PyMem_Malloc()` 或 `PyMem_Realloc()` 分配。

**PyObject \*PyRun\_String** (const char \*str, int start, PyObject \*globals, PyObject \*locals)

回傳值: 新的參照。这是针对下面 `PyRun_StringFlags()` 的简化版接口，将 `flags` 设为 `NULL`。

**PyObject \*PyRun\_StringFlags** (const char \*str, int start, PyObject \*globals, PyObject \*locals, PyCompilerFlags \*flags)

回傳值: 新的參照。在由对象 `globals` 和 `locals` 指定的上下文中执行来自 `str` 的 Python 源代码，并使用以 `flags` 指定的编译器旗标。`globals` 必须是一个字典；`locals` 可以是任何实现了映射协议的对象。形参 `start` 指定了应当被用来解析源代码的起始形符。

返回将代码作为 Python 对象执行的结果，或者如果引发了异常则返回 `NULL`。

**PyObject \*PyRun\_File** (FILE \*fp, const char \*filename, int start, PyObject \*globals, PyObject \*locals)

回傳值: 新的參照。这是针对下面 `PyRun_FileExFlags()` 的简化版接口，将 `closeit` 设为 0 并将 `flags` 设为 `NULL`。

**PyObject \*PyRun\_FileEx** (FILE \*fp, const char \*filename, int start, PyObject \*globals, PyObject \*locals, int closeit)

回傳值: 新的參照。这是针对下面 `PyRun_FileExFlags()` 的简化版接口，将 `flags` 设为 `NULL`。

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

回傳值: 新的參照。这是针对下面 `PyRun_FileExFlags()` 的简化版接口，将 `closeit` 设为 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。类似于 `PyRun_StringFlags()`，但 Python 源代码是从 `fp` 读取而不是一个内存中的字符串。`filename` 应为文件名，它将使用 *filesystem encoding and error handler* 来解码。如果 `closeit` 为真值，则文件将在 `PyRun_FileExFlags()` 返回之前被关闭。

**PyObject \*Py\_CompileString** (const char \*str, const char \*filename, int start)

回傳值: 新的參照。属于稳定 ABI。这是针对下面 `Py_CompileStringFlags()` 的简化版接口，将 `flags` 设为 `NULL`。

**PyObject \*Py\_CompileStringFlags** (const char \*str, const char \*filename, int start, PyCompilerFlags \*flags)

回傳值: 新的參照。这是针对下面 `Py_CompileStringExFlags()` 的简化版接口，将 `optimize` 设为 -1。

**PyObject \*Py\_CompileStringObject** (const char \*str, PyObject \*filename, int start, PyCompilerFlags \*flags, int optimize)

回傳值: 新的參照。解析并编译 `str` 中的 Python 源代码，返回结果代码对象。开始形符由 `start` 给出；这可被用来约束可被编译的代码并且应当为 `Py_eval_input`、`Py_file_input` 或 `Py_single_input`。由 `filename` 指定的文件名会被用来构造代码对象并可能出现在回溯信息或 `SyntaxError` 异常消息中。如果代码无法被解析或编译则此函数将返回 `NULL`。

整数 `optimize` 指定编译器的优化级别；值 -1 将选择与 -O 选项相同的解释器优化级别。显式级别为 0 (无优化；`__debug__` 为真值)、1 (断言被移除，`__debug__` 为假值) 或 2 (文档字符串也被移除)。

在 3.4 版新加入。

**PyObject \*Py\_CompileStringExFlags** (const char \*str, const char \*filename, int start, PyCompilerFlags \*flags, int optimize)

回傳值: 新的參照。与 `Py_CompileStringObject()` 类似，但 `filename` 是以 *filesystem encoding and error handler* 解码出的字节串。

在 3.2 版新加入。

*PyObject* \*PyEval\_EvalCode (*PyObject* \*co, *PyObject* \*globals, *PyObject* \*locals)

回傳值：新的參照。属于稳定 ABI。这是针对 *PyEval\_EvalCodeEx()* 的简化版接口，只附带代码对象，以及全局和局部变量。其他参数均设为 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。属于稳定 ABI。对一个预编译的代码对象求值，为其求值给出特定的环境。此环境由全局变量的字典，局部变量映射对象，参数、关键字和默认值的数组，仅限关键字参数的默认值的字典和单元的封闭元组构成。

*PyObject* \*PyEval\_EvalFrame (*PyFrameObject* \*f)

回傳值：新的參照。属于稳定 ABI。对一个执行帧求值。这是针对 *PyEval\_EvalFrameEx()* 的简化版接口，用于保持向下兼容性。

*PyObject* \*PyEval\_EvalFrameEx (*PyFrameObject* \*f, int throwflag)

回傳值：新的參照。属于稳定 ABI。这是 Python 解释运行不带修饰的主函数。与执行帧 *f* 相关联的代码对象将被执行，解释字节码并根据需要执行调用。额外的 *throwflag* 形参基本可以被忽略——如果为真值，则会导致立即抛出一个异常；这会被用于生成器对象的 *throw()* 方法。

在 3.4 版的變更：该函数现在包含一个调试断言，用以确保不会静默地丢弃活动的异常。

int PyEval\_MergeCompilerFlags (*PyCompilerFlags* \*cf)

此函数会修改当前求值帧的旗标，并在成功时返回真值，失败时返回假值。

int Py\_eval\_input

Python 语法中用于孤立表达式的起始符号；配合 *Py\_CompileString()* 使用。

int Py\_file\_input

Python 语法中用于从文件或其他源读取语句序列的起始符号；配合 *Py\_CompileString()* 使用。这是在编译任意长的 Python 源代码时要使用的符号。

int Py\_single\_input

Python 语法中用于单独语句的起始符号；配合 *Py\_CompileString()* 使用。这是用于交互式解释器循环的符号。

struct PyCompilerFlags

这是用来存放编译器旗标的结构体。对于代码仅被编译的情况，它将作为 *int flags* 传入，而对于代码要被执行的情况，它将作为 *PyCompilerFlags \*flags* 传入。在这种情况下，*from \_\_future\_\_ import* 可以修改 *flags*。

只要 *PyCompilerFlags \*flags* 是 NULL，*cf\_flags* 就会被视为等同于 0，而由于 *from \_\_future\_\_ import* 而产生的任何修改都会被丢弃。

int cf\_flags

编译器旗标。

int cf\_feature\_version

*cf\_feature\_version* 是 Python 的小版本号。它应当被初始化为 *PY\_MINOR\_VERSION*。

该字段默认会被忽略，当且仅当在 *cf\_flags* 中设置了 *PyCF\_ONLY\_AST* 旗标时它才会被使用。

在 3.8 版的變更：新增 *cf\_feature\_version* 欄位。

int CO\_FUTURE\_DIVISION

这个标志位可在 *flags* 中设置以使得除法运算符 / 被解读为 PEP 238 所规定的“真除法”。

---

## 參照計數

---

本節中的巨集用於管理 Python 物件的參照計數。

void **Py\_INCREF** (*PyObject* \*o)

表示為對象 *o* 獲取一個新的 *strong reference*，指明該對象正在被使用且不應被銷毀。

此函式通常用於將借用參照原地 (in-place) 轉為參照。Py\_NewRef() 函式可用於建立新的參照。

當對象使用完畢後，可調用 Py\_DECREF() 釋放它。

該物件不能為 NULL；如果你不確定它不是 NULL，請使用 Py\_XINCREF()。

Do not expect this function to actually modify *o* in any way.

void **Py\_XINCREF** (*PyObject* \*o)

與 Py\_INCREF() 類似，但對象 *o* 可以為 NULL，在這種情況下此函數將沒有任何效果。

另請見 Py\_XNewRef()。

*PyObject* \***Py\_NewRef** (*PyObject* \*o)

屬於穩定 ABI 自 3.10 版起。為對象創建一個新的 *strong reference*：在 *o* 上調用 Py\_INCREF() 並返回對象 *o*。

當不再需要這個 *strong reference* 時，應當在其上調用 Py\_DECREF() 來釋放引用。

物件 *o* 不能為 NULL；如果 *o* 可以為 NULL，則使用 Py\_XNewRef()。

舉例來說：

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

可以寫成：

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

另請參閱 Py\_INCREF()。

在 3.10 版新加入。

*PyObject* \***Py\_XNewRef** (*PyObject* \*o)

属于稳定 ABI 自 3.10 版起, 與 *Py\_NewRef()* 類似, 但物件 *o* 可以為 NULL。

如果物件 *o* 為 NULL, 則該函式僅回傳 NULL。

在 3.10 版新加入。

void **Py\_DECREF** (*PyObject* \*o)

釋放一個指向對象 *o* 的 *strong reference*, 表明該引用不再被使用。

當最後一個 *strong reference* 被釋放時 (即對象的引用計數變為 0), 將會發起調用該對象所屬類型的 deallocation 函數 (它必須不為 NULL)。

此函式通常用於在退出作用域之前刪除參照。

該物件不能為 NULL; 如果你不確定它不是 NULL, 請改用 *Py\_XDECREF()*。

Do not expect this function to actually modify *o* in any way.

**警告:** 釋放函數會導致任意 Python 代碼被發起調用 (例如當一個帶有 `__del__()` 方法的類實例被釋放時就是如此)。雖然這些代碼中的異常不會被傳播, 但被執行的代碼能夠自由訪問所有 Python 全局變量。這意味著在調用 *Py\_DECREF()* 之前任何可通過全局變量獲取的對象都應該處於完好的狀態。例如, 從一個列表中刪除對象的代碼應該將被刪除對象的引用拷貝到一個臨時變量中, 更新列表數據結構, 然後再為臨時變量調用 *Py\_DECREF()*。

void **Py\_XDECREF** (*PyObject* \*o)

與 *Py\_DECREF()* 類似, 但對象 *o* 可以為 NULL, 在這種情況下此函數將沒有任何效果。來自 *Py\_DECREF()* 的警告同樣適用於此處。

void **Py\_CLEAR** (*PyObject* \*o)

釋放一個指向對象 *o* 的 *strong reference*。對象可以為 NULL, 在此情況下該宏將沒有任何效果; 在其他情況下其效果與 *Py\_DECREF()* 相同, 區別在於其參數也會被設為 NULL。針對 *Py\_DECREF()* 的警告不適用於所傳遞的對象, 因為該宏會細心地使用一個臨時變量並在釋放引用之前將參數設為 NULL。

當需要釋放指向一個在垃圾回收期間可能被會遍歷的對象的引用時使用該宏是一個好主意。

void **Py\_IncRef** (*PyObject* \*o)

属于稳定 ABI. 表示获取一个指向对象 *o* 的新 *strong reference*。 *Py\_XINCREF()* 的函数版本。它可被用于 Python 的运行时代嵌入。

void **Py\_DecRef** (*PyObject* \*o)

属于稳定 ABI. 释放一个指向对象 *o* 的 *strong reference*。 *Py\_XDECREF()* 的函数版本。它可被用于 Python 的运行时代嵌入。

以下函式或巨集僅在直譯器核心 使用: `_Py_Dealloc()`、`_Py_ForgetReference()`、`_Py_NewReference()` 以及全域變數 `_Py_RefTotal`。

## 例外處理

本章描述的函数将让你处理和触发 Python 异常。了解一些 Python 异常处理的基础知识是很重要的。它的工作原理有点像 POSIX `errno` 变量: (每个线程) 有一个最近发生的错误的全局指示器。大多数 C API 函数在成功执行时将不理睬它。大多数 C API 函数也会返回一个错误指示器, 如果它们应当返回一个指针则会返回 `NULL`, 或者如果它们应当返回一个整数则会返回 `-1` (例外情况: `PyArg_*` 函数返回 `1` 表示成功而 `0` 表示失败)。

具体地说, 错误指示器由三个对象指针组成: 异常的类型, 异常的值, 和回溯对象。如果没有错误被设置, 这些指针都可以是 `NULL` (尽管一些组合使禁止的, 例如, 如果异常类型是 `NULL`, 你不能有一个非 `NULL` 的回溯)。

当一个函数由于它调用的某个函数失败而必须失败时, 通常不会设置错误指示器; 它调用的那个函数已经设置了它。而它负责处理错误和清理异常, 或在清除其拥有的所有资源后返回 (如对象应用或内存分配)。如果不准备处理异常, 则不应该正常地继续。如果是由于一个错误返回, 那么一定要向调用者表明已经设置了错误。如果错误没有得到处理或小心传播, 对 Python/C API 的其它调用可能不会有预期的行为, 并且可能会以某种神秘的方式失败。

**備註:** 错误指示器 **不是** `sys.exc_info()` 的执行结果。前者对应尚未捕获的异常 (异常还在传播), 而后者在捕获异常后返回这个异常 (异常已经停止传播)。

## 5.1 打印和清理

`void PyErr_Clear()`

属于**稳定 ABI**. 清除错误指示器。如果没有设置错误指示器, 则不会有作用。

`void PyErr_PrintEx(int set_sys_last_vars)`

属于**稳定 ABI**. 将标准回溯打印到 `sys.stderr` 并清除错误指示器。**除非**错误是 `SystemExit`, 这种情况下不会打印回溯进程, 且会退出 Python 进程, 并显示 `SystemExit` 实例指定的错误代码。

只有在错误指示器被设置时才需要调用这个函数, 否则这会导致错误!

如果 `set_sys_last_vars` 非零, 则变量 `sys.last_type`, `sys.last_value` 和 `sys.last_traceback` 将分别设置为打印异常的类型, 值和回溯。

`void PyErr_Print()`

属于**稳定 ABI**. `PyErr_PrintEx(1)` 的同名。



void **PyErr\_WriteUnraisable** (*PyObject* \*obj)

属于稳定 ABI. 使用当前异常和 *obj* 参数调用 `sys.unraisablehook()`。

当异常已被设置但解释器不可能实际引发该异常时，这个工具函数会向 `sys.stderr` 打印一条警告消息。例如，当异常发生在 `__del__()` 方法中时就会使用该函数。

该函数使用单个参数 *obj* 进行调用，该参数标识发生不可触发异常的上下文。如果可能，*obj* 的报告将打印在警告消息中。

调用此函数时必须设置一个异常。

## 5.2 抛出异常

这些函数可帮助你设置当前线程的错误指示器。为了方便起见，一些函数将始终返回 `NULL` 指针，以便于 `return` 语句。

void **PyErr\_SetString** (*PyObject* \*type, const char \*message)

属于稳定 ABI. 这是设置错误指示器最常用的方式。第一个参数指定异常类型；它通常为某个标准异常，例如 `PyExc_RuntimeError`。你无需为其创建新的 *strong reference* (例如使用 `Py_INCREF()`)。第二个参数是一条错误消息；它是用 `'utf-8'` 解码的。

void **PyErr\_SetObject** (*PyObject* \*type, *PyObject* \*value)

属于稳定 ABI. 此函数类似于 `PyErr_SetString()`，但是允许你为异常的“值”指定任意一个 Python 对象。

*PyObject* \***PyErr\_Format** (*PyObject* \*exception, const char \*format, ...)

回傳值：總是 `NULL`。属于稳定 ABI. 这个函数设置了一个错误指示器并且返回了 `NULL`，*exception* 应当是一个 Python 中的异常类。*format* 和随后的形参会帮助格式化这个错误的信息；它们与 `PyUnicode_FromFormat()` 有着相同的含义和值。*format* 是一个 ASCII 编码的字符串。

*PyObject* \***PyErr\_FormatV** (*PyObject* \*exception, const char \*format, va\_list vargs)

回傳值：總是 `NULL`。属于稳定 ABI 自 3.5 版起。和 `PyErr_Format()` 相同，但它接受一个 *va\_list* 类型的参数而不是可变数量的参数集。

在 3.5 版新加入。

void **PyErr\_SetNone** (*PyObject* \*type)

属于稳定 ABI. 这是 `PyErr_SetObject(type, Py_None)` 的简写。

int **PyErr\_BadArgument** ()

属于稳定 ABI. 这是 `PyErr_SetString(PyExc_TypeError, message)` 的简写，其中 *message* 指出使用了非法参数调用内置操作。它主要用于内部使用。

*PyObject* \***PyErr\_NoMemory** ()

回傳值：總是 `NULL`。属于稳定 ABI. 这是 `PyErr_SetNone(PyExc_MemoryError)` 的简写；它返回 `NULL`，以便当内存耗尽时，对象分配函数可以写 `return PyErr_NoMemory();`。

*PyObject* \***PyErr\_SetFromErrno** (*PyObject* \*type)

回傳值：總是 `NULL`。属于稳定 ABI. 这是一个便捷函数，当在 C 库函数返回错误并设置 C 变量 `errno` 时它会引发一个异常。它构造了一个元组对象，其第一项是整数值 `errno` 而第二项是对应的错误信息（从 `strerror()` 获取），然后调用 `PyErr_SetObject(type, object)`。在 Unix 上，当 `errno` 的值为 `EINTR` 时，表示有一个中断的系统调用，这将会调用 `PyErr_CheckSignals()`，如果它设置了错误指示符，则让其保持该设置。该函数总是返回 `NULL`，因此当系统调用返回错误时该系统调用的包装函数可以写入 `return PyErr_SetFromErrno(type);`。

*PyObject* \***PyErr\_SetFromErrnoWithFilenameObject** (*PyObject* \*type, *PyObject* \*filenameObject)

回傳值：總是 `NULL`。属于稳定 ABI. 与 `PyErr_SetFromErrno()` 类似，但如果 *filenameObject* 不为 `NULL`，它将作为第三个参数传递给 *type* 的构造函数。在 `OSError` 异常的情况下，它将被用于定义异常实例的 `filename` 属性。

**PyObject \*PyErr\_SetFromErrnoWithFilenameObjects** (PyObject \*type, PyObject \*filenameObject, PyObject \*filenameObject2)

回傳值：總是 `NULL`。属于稳定 ABI 自 3.7 版起。类似于 `PyErr_SetFromErrnoWithFilenameObject()`，但接受第二个 filename 对象，用于当一个接受两个 filename 的函数失败时触发错误。

在 3.4 版新加入。

**PyObject \*PyErr\_SetFromErrnoWithFilename** (PyObject \*type, const char \*filename)

回傳值：總是 `NULL`。属于稳定 ABI。类似于 `PyErr_SetFromErrnoWithFilenameObject()`，但文件名以 C 字符串形式给出。filename 是用 *filesystem encoding and error handler* 解码的。

**PyObject \*PyErr\_SetFromWindowsError** (int ierr)

回傳值：總是 `NULL`。属于稳定 ABI on Windows 自 3.7 版起。这是一个用于引发 `WindowsError` 的便捷函数。如果在调用传入的 `ierr` 值为 0，则会改用对 `GetLastError()` 的调用所返回的错误代码。它将调用 Win32 函数 `FormatMessage()` 来获取 `ierr` 或 `GetLastError()` 所给出的错误代码的 Windows 描述，然后构造一个元组对象，其第一项为 `ierr` 值而第二项为相应的错误消息（从 `FormatMessage()` 获取），然后调用 `PyErr_SetObject(PyExc_WindowsError, object)`。该函数将总是返回 `NULL`。

適用：Windows。

**PyObject \*PyErr\_SetExcFromWindowsError** (PyObject \*type, int ierr)

回傳值：總是 `NULL`。属于稳定 ABI on Windows 自 3.7 版起。类似于 `PyErr_SetFromWindowsError()`，额外的参数指定要触发的异常类型。

適用：Windows。

**PyObject \*PyErr\_SetFromWindowsErrorWithFilename** (int ierr, const char \*filename)

回傳值：總是 `NULL`。属于稳定 ABI on Windows 自 3.7 版起。与 `PyErr_SetFromWindowsError()` 类似，额外的不同点是如果 `filename` 不为 `NULL`，则会使用文件系统编码格式 (`os.fsdecode()`) 进行解码并作为第三个参数传递给 `OSError` 的构造器用于定义异常实例的 `filename` 属性。

適用：Windows。

**PyObject \*PyErr\_SetExcFromWindowsErrorWithFilenameObject** (PyObject \*type, int ierr, PyObject \*filename)

回傳值：總是 `NULL`。属于稳定 ABI on Windows 自 3.7 版起。与 `PyErr_SetExcFromWindowsError()` 类似，额外的不同点是如果 `filename` 不为 `NULL`，它将作为第三个参数传递给 `OSError` 的构造器用于定义异常实例的 `filename` 属性。

適用：Windows。

**PyObject \*PyErr\_SetExcFromWindowsErrorWithFilenameObjects** (PyObject \*type, int ierr, PyObject \*filename, PyObject \*filename2)

回傳值：總是 `NULL`。属于稳定 ABI on Windows 自 3.7 版起。类似于 `PyErr_SetExcFromWindowsErrorWithFilenameObject()`，但是接受第二个 filename 对象。

適用：Windows。

在 3.4 版新加入。

**PyObject \*PyErr\_SetExcFromWindowsErrorWithFilename** (PyObject \*type, int ierr, const char \*filename)

回傳值：總是 `NULL`。属于稳定 ABI on Windows 自 3.7 版起。类似于 `PyErr_SetFromWindowsErrorWithFilename()`，额外参数指定要触发的异常类型。

適用：Windows。

**PyObject \*PyErr\_SetImportError** (PyObject \*msg, PyObject \*name, PyObject \*path)

回傳值：總是 `NULL`。属于稳定 ABI 自 3.7 版起。这是触发 `ImportError` 的便捷函数。`msg` 将被设为异常的消息字符串。`name` 和 `path`，(都可以为 `NULL`)，将用来被设置 `ImportError` 对应的属性 `name` 和 `path`。

在 3.3 版新加入。

**PyObject\* PyErr\_SetImportErrorSubclass** (PyObject\* exception, PyObject\* msg, PyObject\* name, PyObject\* path)

回傳值：總是 `NULL`。属于稳定 ABI 自 3.6 版起。和 `PyErr_SetImportError()` 很类似，但这个函数允许指定一个 `ImportError` 的子类来触发。

在 3.6 版新加入。

**void PyErr\_SyntaxLocationObject** (PyObject\* filename, int lineno, int col\_offset)

设置当前异常的文件，行和偏移信息。如果当前异常不是 `SyntaxError`，则它设置额外的属性，使异常打印子系统认为异常是 `SyntaxError`。

在 3.4 版新加入。

**void PyErr\_SyntaxLocationEx** (const char\* filename, int lineno, int col\_offset)

属于稳定 ABI 自 3.7 版起。类似于 `PyErr_SyntaxLocationObject()`，但 `filename` 是用 *filesystem encoding and error handler* 解码的字节串。

在 3.2 版新加入。

**void PyErr\_SyntaxLocation** (const char\* filename, int lineno)

属于稳定 ABI。类似于 `PyErr_SyntaxLocationEx()`，但省略了 `col_offset` parameter 形参。

**void PyErr\_BadInternalCall** ()

属于稳定 ABI。这是 `PyErr_SetString(PyExc_SystemError, message)` 的缩写，其中 *message* 表示使用了非法参数调用内部操作（例如，Python/C API 函数）。它主要用于内部使用。

## 5.3 发出警告

这些函数可以从 C 代码中发出警告。它们仿照了由 Python 模块 `warnings` 导出的那些函数。它们通常向 `sys.stderr` 打印一条警告信息；当然，用户也有可能已经指定将警告转换为错误，在这种情况下，它们将触发异常。也有可能由于警告机制出现问题，使得函数触发异常。如果没有触发异常，返回值为 0；如果触发异常，返回值为 -1。（无法确定是否实际打印了警告信息，也无法确定异常触发的原因。这是故意为之）。如果触发了异常，调用者应该进行正常的异常处理（例如，`Py_DECREF()` 持有引用并返回一个错误值）。

**int PyErr\_WarnEx** (PyObject\* category, const char\* message, Py\_ssize\_t stack\_level)

属于稳定 ABI。发出一个警告信息。参数 *category* 是一个警告类别（见下面）或 `NULL`；*message* 是一个 UTF-8 编码的字符串。*stack\_level* 是一个给出栈帧数量的正数；警告将从该栈帧中当前正在执行的代码行发出。*stack\_level* 为 1 的是调用 `PyErr_WarnEx()` 的函数，2 是在此之上的函数，以此类推。

警告类别必须是 `PyExc_Warning` 的子类，`PyExc_Warning` 是 `PyExc_Exception` 的子类；默认警告类别是 `PyExc_RuntimeWarning`。标准 Python 警告类别作为全局变量可用，所有其名称见标准警告类别。

有关警告控制的信息，参见模块文档 `warnings` 和命令行文档中的 `-w` 选项。没有用于警告控制的 C API。

**int PyErr\_WarnExplicitObject** (PyObject\* category, PyObject\* message, PyObject\* filename, int lineno, PyObject\* module, PyObject\* registry)

发出一个对所有警告属性进行显式控制的警告消息。这是位于 Python 函数 `warnings.warn_explicit()` 外层的直接包装；请查看其文档了解详情。*module* 和 *registry* 参数可被设为 `NULL` 以得到相关文档所描述的默认效果。

在 3.4 版新加入。

**int PyErr\_WarnExplicit** (PyObject\* category, const char\* message, const char\* filename, int lineno, const char\* module, PyObject\* registry)



属于稳定 ABI。类似于 `PyErr_WarnExplicitObject()` 不过 `message` 和 `module` 是 UTF-8 编码的字符串，而 `filename` 是由 *filesystem encoding and error handler* 解码的。

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

属于稳定 ABI。类似于 `PyErr_WarnEx()` 的函数，但使用 `PyUnicode_FromFormat()` 来格式化警告消息。`format` 是使用 ASCII 编码的字符串。

在 3.2 版新加入。

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

属于稳定 ABI 自 3.6 版起。类似于 `PyErr_WarnFormat()` 的函数，但 `category` 是 `ResourceWarning` 并且它会将 `source` 传给 `warnings.WarningMessage`。

在 3.6 版新加入。

## 5.4 查询错误指示器

`PyObject *PyErr_Occurred()`

回傳值：借用參照。属于稳定 ABI。测试是否设置了错误指示器。如已设置，则返回异常 `type` (传给对某个 `PyErr_Set*` 函数或 `PyErr_Restore()` 的最后一次调用的第一个参数)。如未设置，则返回 `NULL`。你并不会拥有对返回值的引用，因此你不需要对它执行 `Py_DECREF()`。

调用时必须携带 GIL。

---

**備註：** 不要将返回值与特定的异常进行比较；请改为使用 `PyErr_ExceptionMatches()`，如下所示。（比较很容易失败因为对于类异常来说，异常可能是一个实例而不是类，或者它可能是预期的异常的一个子类。）

---

`int PyErr_ExceptionMatches (PyObject *exc)`

属于稳定 ABI。等价于 `PyErr_GivenExceptionMatches(PyErr_Occurred(), exc)`。此函数应当只在实际设置了异常时才被调用；如果没有任何异常被引发则将发生非法内存访问。

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

属于稳定 ABI。如果 `given` 异常与 `exc` 中的异常类型相匹配则返回真值。如果 `exc` 是一个类对象，则当 `given` 是一个子类的实例时也将返回真值。如果 `exc` 是一个元组，则该元组（以及递归的子元组）中的所有异常类型都将被搜索进行匹配。

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

属于稳定 ABI。将错误指示符提取到三个变量中并传递其地址。如果未设置错误指示符，则将三个变量都设为 `NULL`。如果已设置，则将其清除并且你将得到对所提取的每个对象的引用。值和回溯对象可以为 `NULL` 即使类型对象不为空。

---

**備註：** 此函数通常只被需要捕获异常的代码或需要临时保存和恢复错误指示符的代码所使用，例如：

---

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

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

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

void **PyErr\_Restore** (*PyObject* \*type, *PyObject* \*value, *PyObject* \*traceback)

属于稳定 ABI。基于三个对象设置错误指示符。如果错误指示符已设置，它将首先被清除。如果三个对象均为 NULL，错误指示器将被清除。请不要传入 NULL 类型和非 NULL 值或回溯。异常类型应当是一个类。请不要传入无效的异常类型或值。（违反这些规则将导致微妙的后续问题。）此调用会带走对每个对象的引用：你必须在调用之前拥有对每个对象的引用且在调用之后你将不再拥有这些引用。（如果你不理解这一点，就不要使用此函数。勿谓言之不预。）

備註：此函数通常只被需要临时保存和恢复错误指示符的代码所使用。请使用 *PyErr\_Fetch()* 来保存当前的错误指示符。

void **PyErr\_NormalizeException** (*PyObject* \*\*exc, *PyObject* \*\*val, *PyObject* \*\*tb)

属于稳定 ABI。在特定情况下，下面 *PyErr\_Fetch()* 所返回的值可以是“非正规化的”，即 \*exc 是一个类对象而 \*val 不是同一个类的实例。在这种情况下此函数可以被用来实例化类。如果值已经是正规化的，则不做任何操作。实现这种延迟正规化是为了提升性能。

備註：此函数不会隐式地在异常值上设置 `__traceback__` 属性。如果想要适当地设置回溯，还需要以下附加代码片段：

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

*PyObject* \***PyErr\_GetHandledException** (void)

属于稳定 ABI 自 3.11 版起。提取激活的异常实例，就如 `sys.exception()` 所返回的一样。这是指一个已被捕获的异常，而不是刚被引发的异常。返回一个指向该异常的新引用或者 NULL。不会修改解释器的异常状态。Does not modify the interpreter's exception state.

備註：此函数通常不会被需要处理异常的代码所使用。它可被使用的场合是当代码需要临时保存并恢复异常状态的时候。请使用 *PyErr\_SetHandledException()* 来恢复或清除异常状态。

在 3.11 版新加入。

void **PyErr\_SetHandledException** (*PyObject* \*exc)

属于稳定 ABI 自 3.11 版起。设置激活的异常，就是从 `sys.exception()` 所获得的。这是指一个已被捕获的异常，而不是刚被引发的异常。要清空异常状态，请传入 NULL。

備註：此函数通常不会被需要处理异常的代码所使用。它被使用的场合是在代码需要临时保存并恢复异常状态的时候。请使用 *PyErr\_GetHandledException()* 来获取异常状态。

在 3.11 版新加入。

void **PyErr\_GetExcInfo** (*PyObject* \*\*ptype, *PyObject* \*\*pvalue, *PyObject* \*\*ptraceback)

属于稳定 ABI 自 3.7 版起。提取旧式的异常信息表示形式，就是从 `sys.exc_info()` 所获得的。这是指一个已被捕获的异常，而不是刚被引发的异常。返回分别指向三个对象的新引用，其中任何一个都可以为 NULL。不会修改异常信息的状态。此函数是为了向下兼容而保留的。更推荐使用 *PyErr\_GetHandledException()*。

備註：此函数通常不会被需要处理异常的代码所使用。它被使用的场合是在代码需要临时保存并恢复异常状态的时候。请使用 *PyErr\_SetExcInfo()* 来恢复或清除异常状态。

在 3.3 版新加入。

void **PyErr\_SetExcInfo** (*PyObject* \*type, *PyObject* \*value, *PyObject* \*traceback)

属于稳定 ABI 自 3.7 版起。设置异常信息，就是从 `sys.exc_info()` 所获得的，这是指一个已被捕获的异常，而不是刚被引发的异常。此函数会偷取对参数的引用。要清空异常状态，请为所有三个参数传入 NULL。此函数是为了向下兼容而保留的。更推荐使用 `PyErr_SetHandledException()`。

**備註：** 此函数通常不会被需要处理异常的代码所使用。它被使用的场合是在代码需要临时保存并恢复异常状态的情况。请使用 `PyErr_GetExcInfo()` 来读取异常状态。

在 3.3 版新加入。

在 3.11 版的變更: type 和 traceback 参数已不再被使用并且可以为 NULL。解释器现在会根据异常实例（即 value 参数）来推断出它们。此函数仍然会偷取对所有三个参数的引用。

## 5.5 信号处理

int **PyErr\_CheckSignals** ()

属于稳定 ABI。这个函数与 Python 的信号处理交互。

如果在主 Python 解释器下从主线程调用该函数，它将检查是否向进程发送了信号，如果是，则发起调用相应的信号处理器。如果支持 signal 模块，则可以发起调用以 Python 编写的信号处理器。

该函数会尝试处理所有待处理信号，然后返回 0。但是，如果 Python 信号处理器引发了异常，则设置错误指示符并且函数将立即返回 -1（这样其他待处理信号可能还没有被处理：它们将在下次发起调用 `PyErr_CheckSignals()` 时被处理）。

如果函数从非主线程调用，或在非主 Python 解释器下调用，则它不执行任何操作并返回 0。

这个函数可以由希望被用户请求（例如按 Ctrl-C）中断的长时间运行的 C 代码调用。

**備註：** 针对 SIGINT 的默认 Python 信号处理器会引发 KeyboardInterrupt 异常。

void **PyErr\_SetInterrupt** ()

属于稳定 ABI。模拟一个 SIGINT 信号到达的效果。这等价于 `PyErr_SetInterruptEx(SIGINT)`。

**備註：** 此函数是异步信号安全的。它可以不带 GIL 并由 C 信号处理器来调用。

int **PyErr\_SetInterruptEx** (int signum)

属于稳定 ABI 自 3.10 版起。模拟一个信号到达的效果。当下次 `PyErr_CheckSignals()` 被调用时，将会调用针对指定的信号编号的 Python 信号处理器。

此函数可由自行设置信号处理，并希望 Python 信号处理器会在请求中断时（例如当用户按下 Ctrl-C 来中断操作时）按照预期被发起调用的 C 代码来调用。

如果给定的信号不是由 Python 来处理的（即被设为 `signal.SIG_DFL` 或 `signal.SIG_IGN`），它将会被忽略。

如果 *signum* 在被允许的信号编号范围之外，将返回 -1。在其他情况下，则返回 0。错误指示符绝不会被此函数所修改。

**備註：** 此函数是异步信号安全的。它可以不带 GIL 并由 C 信号处理器来调用。

在 3.10 版新加入。

`int PySignal_SetWakeupFd(int fd)`

这个工具函数指定了一个每当收到信号时将被作为以单个字节的形式写入信号编号的目标的文件描述符。*fd* 必须是非阻塞的。它将返回前一个这样的文件描述符。

设置值 -1 将禁用该特性；这是初始状态。这等价于 Python 中的 `signal.set_wakeup_fd()`，但是没有任何错误检查。*fd* 应当是一个有效的文件描述符。此函数应当只从主线程来调用。

在 3.5 版的變更：在 Windows 上，此函数现在也支持套接字处理。

## 5.6 例外類

*PyObject* \*PyErr\_NewException(const char \*name, *PyObject* \*base, *PyObject* \*dict)

回傳值：新的參照。属于穩定 ABI。这个工具函数会创建并返回一个新的异常类。*name* 参数必须为新异常的名称，是 `module.classname` 形式的 C 字符串。*base* 和 *dict* 参数通常为 NULL。这将创建一个派生自 `Exception` 的类对象（在 C 中可以通过 `PyExc_Exception` 访问）。

新类的 `__module__` 属性将被设为 *name* 参数的前半部分（最后一个点号之前）。*base* 参数可被用来指定替代基类；它可以是一个类或是一个由类组成的元组。*dict* 参数可被用来指定一个由类变量和方法组成的字典。

*PyObject* \*PyErr\_NewExceptionWithDoc(const char \*name, const char \*doc, *PyObject* \*base, *PyObject* \*dict)

回傳值：新的參照。属于穩定 ABI。和 `PyErr_NewException()` 一样，除了可以轻松地给新的异常类一个文档字符串：如果 *doc* 属性非空，它将用作异常类的文档字符串。

在 3.2 版新加入。

## 5.7 例外物件

*PyObject* \*PyException\_GetTraceback(*PyObject* \*ex)

回傳值：新的參照。属于穩定 ABI。将与异常相关联的回溯作为一个新引用返回，可以通过 `__traceback__` 属性在 Python 中访问。如果没有已关联的回溯，则返回 NULL。

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

属于穩定 ABI。将异常关联的回溯设置为 *tb*。使用 `Py_None` 清除它。

*PyObject* \*PyException\_GetContext(*PyObject* \*ex)

回傳值：新的參照。属于穩定 ABI。将与异常相关联的上下文（在处理 *ex* 过程中引发的另一个异常实例）作为一个新引用返回，可以通过 `__context__` 属性在 Python 中访问。如果没有已关联的上下文，则返回 NULL。

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

属于穩定 ABI。将与异常相关联的上下文设置为 *ctx*。使用 NULL 来清空它。没有用来确保 *ctx* 是一个异常实例的类型检查。这将窃取一个指向 *ctx* 的引用。

*PyObject* \*PyException\_GetCause(*PyObject* \*ex)

回傳值：新的參照。属于穩定 ABI。将与异常相关联的原因（一个异常实例，或为 None，由 `raise ... from ...` 设置）作为一个新引用返回，可通过 `__cause__` 属性在 Python 中访问。

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

属于穩定 ABI。将与异常相关联的原因设为 *cause*。使用 NULL 来清空它。不存在类型检查用来确保 *cause* 是一个异常实例或为 None。这个偷取一个指向 *cause* 的引用。

`__suppress_context__` 属性会被此函数隐式地设为 True。

## 5.8 Unicode 异常对象

下列函数被用于创建和修改来自 C 的 Unicode 异常。

*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。属于稳定 ABI。创建一个 UnicodeDecodeError 对象并附带 *encoding*, *object*, *length*, *start*, *end* 和 *reason* 等属性。*encoding* 和 *reason* 为 UTF-8 编码的字符串。

*PyObject* \*PyUnicodeDecodeError\_GetEncoding (*PyObject* \*exc)

*PyObject* \*PyUnicodeEncodeError\_GetEncoding (*PyObject* \*exc)

回傳值：新的參照。属于稳定 ABI。返回给定异常对象的 *encoding* 属性

*PyObject* \*PyUnicodeDecodeError\_GetObject (*PyObject* \*exc)

*PyObject* \*PyUnicodeEncodeError\_GetObject (*PyObject* \*exc)

*PyObject* \*PyUnicodeTranslateError\_GetObject (*PyObject* \*exc)

回傳值：新的參照。属于稳定 ABI。返回给定异常对象的 *object* 属性

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)

属于稳定 ABI。获取给定异常对象的 *start* 属性并将其放入 \*start。start 必须不为 NULL。成功时返回 0，失败时返回 -1。

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)

属于稳定 ABI。将给定异常对象的 *start* 属性设为 start。成功时返回 0，失败时返回 -1。

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)

属于稳定 ABI。获取给定异常对象的 *end* 属性并将其放入 \*end。end 必须不为 NULL。成功时返回 0，失败时返回 -1。

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)

属于稳定 ABI。将给定异常对象的 *end* 属性设为 end。成功时返回 0，失败时返回 -1。

*PyObject* \*PyUnicodeDecodeError\_GetReason (*PyObject* \*exc)

*PyObject* \*PyUnicodeEncodeError\_GetReason (*PyObject* \*exc)

*PyObject* \*PyUnicodeTranslateError\_GetReason (*PyObject* \*exc)

回傳值：新的參照。属于稳定 ABI。返回给定异常对象的 *reason* 属性

int PyUnicodeDecodeError\_SetReason (*PyObject* \*exc, const char \*reason)

int PyUnicodeEncodeError\_SetReason (*PyObject* \*exc, const char \*reason)

int PyUnicodeTranslateError\_SetReason (*PyObject* \*exc, const char \*reason)

属于稳定 ABI。将给定异常对象的 *reason* 属性设为 reason。成功时返回 0，失败时返回 -1。



## 5.9 递归控制

这两个函数提供了一种在 C 层级上进行安全的递归调用的方式，在核心模块与扩展模块中均适用。当递归代码不一定会发起调用 Python 代码（后者会自动跟踪其递归深度）时就需要用到它们。它们对于 `tp_call` 实现来说也无必要因为调用协议会负责递归处理。

**int Py\_EnterRecursiveCall** (const char \*where)

属于稳定 ABI 自 3.9 版起。标记一个递归的 C 层级调用即将被执行的点位。

如果定义了 `USE_STACKCHECK`，此函数会使用 `PyOS_CheckStack()` 来检查 OS 栈是否溢出。如果发生了这种情况，它将设置一个 `MemoryError` 并返回非零值。

随后此函数将检查是否达到递归限制。如果是的话，将设置一个 `RecursionError` 并返回一个非零值。在其他情况下，则返回零。

`where` 应为一个 UTF-8 编码的字符串如 " in instance check"，它将与由递归深度限制所导致的 `RecursionError` 消息相拼接。

在 3.9 版的變更: 此函数现在也在受限 API 中可用。

**void Py\_LeaveRecursiveCall** (void)

属于稳定 ABI 自 3.9 版起。结束一个 `Py_EnterRecursiveCall()`。必须针对 `Py_EnterRecursiveCall()` 的每个成功的发起调用操作执行一次调用。

在 3.9 版的變更: 此函数现在也在受限 API 中可用。

正确地针对容器类型实现 `tp_repr` 需要特别的递归处理。在保护栈之外，`tp_repr` 还需要追踪对象以防止出现循环。以下两个函数将帮助完成此功能。从实际效果来说，这两个函数是 C 中对应 `reprlib.recursive_repr()` 的等价物。

**int Py\_ReprEnter** (PyObject \*object)

属于稳定 ABI。在 `tp_repr` 实现的开头被调用以检测循环。

如果对象已经被处理，此函数将返回一个正整数。在此情况下 `tp_repr` 实现应当返回一个指明发生循环的字符串对象。例如，`dict` 对象将返回 `{...}` 而 `list` 对象将返回 `[...]`。

如果已达到递归限制则此函数将返回一个负正数。在此情况下 `tp_repr` 实现通常应当返回 `NULL`。

在其他情况下，此函数将返回零而 `tp_repr` 实现将可正常继续。

**void Py\_ReprLeave** (PyObject \*object)

属于稳定 ABI。结束一个 `Py_ReprEnter()`。必须针对每个返回零的 `Py_ReprEnter()` 的发起调用操作调用一次。

## 5.10 标准异常

所有的标准 Python 异常都可作为名称为 `PyExc_` 跟上 Python 异常名称的全局变量来访问。这些变量的类型为 `PyObject*`；它们都是类对象。下面完整列出了全部的变量：

C 名称	Python 名称	解
<code>PyExc_BaseException</code>	<code>BaseException</code>	1
<code>PyExc_Exception</code>	<code>Exception</code>	Page 55, 1
<code>PyExc_ArithmeticError</code>	<code>ArithmeticError</code>	Page 55, 1
<code>PyExc_AssertionError</code>	<code>AssertionError</code>	
<code>PyExc_AttributeError</code>	<code>AttributeError</code>	
<code>PyExc_BlockingIOError</code>	<code>BlockingIOError</code>	
<code>PyExc_BrokenPipeError</code>	<code>BrokenPipeError</code>	
<code>PyExc_BufferError</code>	<code>BufferError</code>	
<code>PyExc_ChildProcessError</code>	<code>ChildProcessError</code>	

繼續下一頁

表格 1 – 繼續上一頁

C 名称	Python 名称	解 <sup>F</sup>
PyExc_ConnectionAbortedE	ConnectionAbortedError	
PyExc_ConnectionError	ConnectionError	
PyExc_ConnectionRefusedE	ConnectionRefusedError	
PyExc_ConnectionResetErr	ConnectionResetError	
PyExc_EOFError	EOFError	
PyExc_FileExistsError	FileExistsError	
PyExc_FileNotFoundError	FileNotFoundError	
PyExc_FloatingPointError	FloatingPointError	
PyExc_GeneratorExit	GeneratorExit	
PyExc_ImportError	ImportError	
PyExc_IndentationError	IndentationError	
PyExc_IndexError	IndexError	
PyExc_InterruptedError	InterruptedError	
PyExc_IsADirectoryError	IsADirectoryError	
PyExc_KeyError	KeyError	
PyExc_KeyboardInterrupt	KeyboardInterrupt	
PyExc_LookupError	LookupError	Page 55, 1
PyExc_MemoryError	MemoryError	
PyExc_ModuleNotFoundError	ModuleNotFoundError	
PyExc_NameError	NameError	
PyExc_NotADirectoryError	NotADirectoryError	
PyExc_NotImplementedError	NotImplementedError	
PyExc_OSError	OSError	1
PyExc_OverflowError	OverflowError	
PyExc_PermissionError	PermissionError	
PyExc_ProcessLookupError	ProcessLookupError	
PyExc_RecursionError	RecursionError	
PyExc_ReferenceError	ReferenceError	
PyExc_RuntimeError	RuntimeError	
PyExc_StopAsyncIteration	StopAsyncIteration	
PyExc_StopIteration	StopIteration	
PyExc_SyntaxError	SyntaxError	
PyExc_SystemError	SystemError	
PyExc_SystemExit	SystemExit	
PyExc_TabError	TabError	
PyExc_TimeoutError	TimeoutError	
PyExc_TypeError	TypeError	
PyExc_UnboundLocalError	UnboundLocalError	
PyExc_UnicodeDecodeError	UnicodeDecodeError	
PyExc_UnicodeEncodeError	UnicodeEncodeError	
PyExc_UnicodeError	UnicodeError	
PyExc_UnicodeTranslateEr	UnicodeTranslateError	
PyExc_ValueError	ValueError	
PyExc_ZeroDivisionError	ZeroDivisionError	

在 3.3 版新加入: 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 版新加入: PyExc\_StopAsyncIteration 和 PyExc\_RecursionError。

在 3.6 版新加入: PyExc\_ModuleNotFoundError。

<sup>1</sup> 这是其他标准异常的基类。

这些是兼容性别名 PyExc\_OSError:

C 名称	解
PyExc_EnvironmentError	
PyExc_IOError	
PyExc_WindowsError	<sup>2</sup>

在 3.3 版的變更: 这些别名曾经是单独的异常类型。

解:

## 5.11 标准警告类别

所有的标准 Python 警告类别都可以用作全局变量, 其名称为 PyExc\_ 加上 Python 异常名称。这些类型是 *PyObject\** 类型; 它们都是类对象。以下列出了全部的变量名称:

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

在 3.2 版新加入: PyExc\_ResourceWarning.

解:

<sup>2</sup> 仅在 Windows 中定义; 检测是否定义了预处理程序宏 MS\_WINDOWS, 以便保护用到它的代码。

<sup>3</sup> 这是其他标准警告类别的基类。



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移植性 (portable)、在 C 中使用 Python module (模組)、以及剖析函式引數以基於 C 中的值來構建 Python 中的值等。

## 6.1 作業系統工具

*PyObject* \*PyOS\_FSPath (*PyObject* \*path)

回傳值：新的參照。属于稳定 ABI 自 3.6 版起。返回 *path* 在文件系统中的表示形式。如果该对象是一个 str 或 bytes 对象，则返回一个新的 *strong reference*。如果对象实现了 os.PathLike 接口，则只要它是一个 str 或 bytes 对象就将返回 \_\_fspath\_\_ ()。在其他情况下将引发 TypeError 并返回 NULL。

在 3.6 版新加入。

int Py\_FdIsInteractive (FILE \*fp, const char \*filename)

如果名称为 *filename* 的标准 I/O 文件 *fp* 被确认为可交互的则返回真 (非零) 值。isatty (fileno (fp)) 为真值的文件均属于这种情况。如果全局旗标 *Py\_InteractiveFlag* 为真值，此函数在 *filename* 指针为 NULL 或者其名称等于字符串 '<stdin>' 或 '???' 时也将返回真值。

void PyOS\_BeforeFork ()

属于稳定 ABI on platforms with fork() 自 3.7 版起。在进程分叉之前准备某些内部状态的函数。此函数应当在调用 fork () 或者任何类似的克隆当前进程的函数之前被调用。只适用于定义了 fork () 的系统。

**警告：** C fork () 调用应当只在“main”线程 (位于“main”解释器) 中进行。对于 PyOS\_BeforeFork () 来说也是如此。

在 3.7 版新加入。

void PyOS\_AfterFork\_Parent ()

属于稳定 ABI on platforms with fork() 自 3.7 版起。在进程分叉之后更新某些内部状态的函数。此函数应当在调用 fork () 或任何类似的克隆当前进程的函数之后被调用，无论进程克隆是否成功。只适用于定义了 fork () 的系统。

**警告：** C `fork()` 调用应当只在“*main*”线程（位于“*main*”解释器）中进行。对于 `PyOS_AfterFork_Parent()` 来说也是如此。

在 3.7 版新加入。

void **PyOS\_AfterFork\_Child()**

属于稳定 ABI on platforms with `fork()` 自 3.7 版起。在进程分叉之后更新内部解释器状态的函数。此函数必须在调用 `fork()` 或任何类似的克隆当前进程的函数之后在子进程中被调用，如果该进程有机会回调到 Python 解释器的话。只适用于定义了 `fork()` 的系统。

**警告：** C `fork()` 调用应当只在“*main*”线程（位于“*main*”解释器）中进行。对于 `PyOS_AfterFork_Child()` 来说也是如此。

在 3.7 版新加入。

也参考：

`os.register_at_fork()` 允许注册可被 `PyOS_BeforeFork()`, `PyOS_AfterFork_Parent()` 和 `PyOS_AfterFork_Child()` 调用的自定义 Python 函数。

void **PyOS\_AfterFork()**

属于稳定 ABI on platforms with `fork()`。在进程分叉之后更新某些内部状态的函数；如果要继续使用 Python 解释器则此函数应当在新进程中被调用。如果已将一个新的可执行文件载入到新进程中，则不需要调用此函数。

在 3.7 版之後被用：此函数已被 `PyOS_AfterFork_Child()` 取代。

int **PyOS\_CheckStack()**

属于稳定 ABI on platforms with `USE_STACKCHECK` 自 3.7 版起。当解释器耗尽栈空间时返回真值。这是一个可靠的检测，但仅在定义了 `USE_STACKCHECK` 时可用（目前是在使用 Microsoft Visual C++ 编译器的特定 Windows 版本上）。`USE_STACKCHECK` 将被自动定义；你绝不应该在你自己的代码中改变此定义。

typedef void (\***PyOS\_sighandler\_t**)(int)

属于稳定 ABI。

**PyOS\_sighandler\_t PyOS\_getsig**(int i)

属于稳定 ABI。返回信号 *i* 当前的信号处理器。这是一个对 `sigaction()` 或 `signal()` 的简单包装器。请不要直接调用这两个函数！

**PyOS\_sighandler\_t PyOS\_setsig**(int i, **PyOS\_sighandler\_t** h)

属于稳定 ABI。将信号 *i* 的信号处理器设为 *h*；返回原来的信号处理器。这是一个对 `sigaction()` 或 `signal()` 的简单包装器。请不要直接调用这两个函数！

wchar\_t \***Py\_DecodeLocale**(const char \*arg, size\_t \*size)

属于稳定 ABI 自 3.7 版起。

**警告：** 此函数不应当被直接调用：请使用 `PyConfig` API 以及可确保对 *Python* 进行预初始化的 `PyConfig_SetBytesString()` 函数。

此函数不可在 This function must not be called before 对 *Python* 进行预初始化之前被调用以便正确地配置 `LC_CTYPE` 语言区域：请参阅 `Py_PreInitialize()` 函数。

使用 *filesystem encoding and error handler* 来解码一个字节串。如果错误处理器为 `surrogateescape` 错误处理器，则不可解码的字节将被解码为 `U+DC80..U+DCFF` 范围内的字符；而如果一个字节序列可被解码为代理字符，则其中的字节会使用 `surrogateescape` 错误处理器来转义而不是解码它们。

返回一个指向新分配的由宽字符组成的字符串的指针，使用 `PyMem_RawFree()` 来释放内存。如果 `size` 不为 `NULL`，则将排除了 `null` 字符的宽字符数量写入到 `*size`

在解码错误或内存分配错误时返回 `NULL`。如果 `size` 不为 `NULL`，则 `*size` 将在内存错误时设为 `(size_t)-1` 或在解码错误时设为 `(size_t)-2`。

*filesystem encoding and error handler* 是由 `PyConfig_Read()` 来选择的：参见 `PyConfig` 的 *filesystem\_encoding* 和 *filesystem\_errors* 等成员。

解码错误绝对不应当发生，除非 C 库有程序缺陷。

请使用 `Py_EncodeLocale()` 函数来将字符串编码回字节串。

#### 也参考：

`PyUnicode_DecodeFSDefaultAndSize()` 和 `PyUnicode_DecodeLocaleAndSize()` 函数。

在 3.5 版新加入。

在 3.7 版的變更：现在此函数在 Python UTF-8 模式下将使用 UTF-8 编码格式。

在 3.8 版的變更：现在如果在 Windows 上 `Py_LegacyWindowsFSEncodingFlag` 为零则此函数将使用 UTF-8 编码格式；

char **\*Py\_EncodeLocale** (const wchar\_t \*text, size\_t \*error\_pos)

属于稳定 ABI 自 3.7 版起。将一个由宽字符组成的字符串编码为 *filesystem encoding and error handler*。如果错误处理器为 `surrogateescape` 错误处理器，则在 `U+DC80..U+DCFF` 范围内的代理字符会被转换为字节值 `0x80..0xFF`。

返回一个指向新分配的字节串的指针，使用 `PyMem_Free()` 来释放内存。当发生编码错误或内存分配错误时返回 `NULL`。

如果 `error_pos` 不为 `NULL`，则成功时会将 `*error_pos` 设为 `(size_t)-1`，或是在发生编码错误时设为无效字符的索引号。

*filesystem encoding and error handler* 是由 `PyConfig_Read()` 来选择的：参见 `PyConfig` 的 *filesystem\_encoding* 和 *filesystem\_errors* 等成员。

请使用 `Py_DecodeLocale()` 函数来将字节串解码回由宽字符组成的字符串。

**警告：** 此函数不可在 `This function must not be called before` 对 *Python* 进行预初始化之前被调用以便正确地配置 `LC_CTYPE` 语言区域：请参阅 `Py_PreInitialize()` 函数。

#### 也参考：

`PyUnicode_EncodeFSDefault()` 和 `PyUnicode_EncodeLocale()` 函数。

在 3.5 版新加入。

在 3.7 版的變更：现在此函数在 Python UTF-8 模式下将使用 UTF-8 编码格式。

在 3.8 版的變更：现在如果在 Windows 上 `Py_LegacyWindowsFSEncodingFlag` 为零则此函数将使用 UTF-8 编码格式。

## 6.2 系統函式

这些是使来自 `sys` 模块的功能可以让 C 代码访问的工具函数。它们都可用于当前解释器线程的 `sys` 模块的字典，该字典包含在内部线程状态结构体中。

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

回傳值：借用參照。属于稳定 ABI。返回来自 `sys` 模块的对象 `name` 或者如果它不存在则返回 `NULL`，并且不会设置异常。

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

属于稳定 ABI。将 `sys` 模块中的 `name` 设为 `v` 除非 `v` 为 `NULL`，在此情况下 `name` 将从 `sys` 模块中被删除。成功时返回 0，发生错误时返回 -1。

**`void PySys_ResetWarnOptions ()`**

属于稳定 ABI。将 `sys.warnoptions` 重置为空列表。此函数可在 `Py_Initialize()` 之前被调用。

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

属于稳定 ABI。此 API 被保留用于向下兼容：应当改为采用设置 `PyConfig.warnoptions`，参见 *Python 初始化配置*。

将 `s` 添加到 `sys.warnoptions`。此函数必须在 `Py_Initialize()` 之前被调用以便影响警告过滤器列表。

在 3.11 版之後被弃用。

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

属于稳定 ABI。此 API 被保留用于向下兼容：应当改为采用设置 `PyConfig.warnoptions`，参见 *Python 初始化配置*。

将 `unicode` 添加到 `sys.warnoptions`。

注意：目前此函数不可在 CPython 实现之外使用，因为它必须在 `Py_Initialize()` 中的 `warnings` 显式导入之前被调用，但是要等运行时已初始化到足以允许创建 `Unicode` 对象时才能被调用。

在 3.11 版之後被弃用。

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

属于稳定 ABI。此 API 被保留用于向下兼容：应当改为采用设置 `PyConfig.module_search_paths` 和 `PyConfig.module_search_paths_set`，参见 *Python 初始化配置*。

将 `sys.path` 设为由在 `path` 中找到的路径组成的列表对象，该参数应为使用特定平台的搜索路径分隔符（在 Unix 上为 `:`，在 Windows 上为 `;`）分隔的路径的列表。

在 3.11 版之後被弃用。

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

属于稳定 ABI。将以 `format` 描述的输出字符串写入到 `sys.stdout`。不会引发任何异常，即使发生了截断（见下文）。

`format` 应当将已格式化的输出字符串的总大小限制在 1000 字节以下 -- 超过 1000 字节后，输出字符串会被截断。特别地，这意味着不应出现不受限制的 `"%s"` 格式；它们应当使用 `"%.<N>s"` 来限制，其中 `<N>` 是一个经计算使得 `<N>` 与其他已格式化文本的最大尺寸之和不会超过 1000 字节的十进制数字。还要注意 `"%f"`，它可能为非常大的数字打印出数以百计的数位。

如果发生了错误，`sys.stdout` 会被清空，已格式化的消息将被写入到真正的 (C 层级) `stdout`。

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

属于稳定 ABI。类似 `PySys_WriteStdout()`，但改为写入到 `sys.stderr` 或 `stderr`。

void **PySys\_FormatStdout** (const char \*format, ...)

属于稳定 ABI。类似 `PySys_WriteStdout()` 的函数将会使用 `PyUnicode_FromFormatV()` 来格式化消息并且不会将消息截短至任意长度。

在 3.2 版新加入。

void **PySys\_FormatStderr** (const char \*format, ...)

属于稳定 ABI。类似 `PySys_FormatStdout()`，但改为写入到 `sys.stderr` 或 `stderr`。

在 3.2 版新加入。

void **PySys\_AddXOption** (const wchar\_t \*s)

属于稳定 ABI 自 3.7 版起。此 API 被保留用于向下兼容：应当改为采用设置 `PyConfig.xoptions`，参见 *Python 初始化配置*。

将 `s` 解析为一个由 `-X` 选项组成的集合并将它们添加到 `PySys_GetXOptions()` 所返回的当前选项映射。此函数可以在 `Py_Initialize()` 之前被调用。

在 3.2 版新加入。

在 3.11 版之後被用。

*PyObject* \***PySys\_GetXOptions** ()

回傳值：借用參照。属于稳定 ABI 自 3.7 版起。返回当前 `-X` 选项的字典，类似于 `sys._xoptions`。发生错误时，将返回 `NULL` 并设置一个异常。

在 3.2 版新加入。

int **PySys\_Audit** (const char \*event, const char \*format, ...)

引发一个审计事件并附带任何激活的钩子。成功时返回零值或在失败时返回非零值并设置一个异常。

如果已添加了任何钩子，则将使用 `format` 和其他参数来构造一个传入的元组。除 `N` 以外，在 `Py_BuildValue()` 中使用的格式字符均可使用。如果构建的值不是一个元组，它将被添加到一个单元元素元组中。（格式选项 `N` 会消耗一个引用，但是由于没有办法知道此函数的参数是否将被消耗，因此使用它可能导致引用泄漏。）

请注意 # 格式字符应当总是被当作 `Py_ssize_t` 来处理，无论是否定义了 `PY_SSIZE_T_CLEAN`。

`sys.audit()` 会执行与来自 Python 代码的函数相同的操作。

在 3.8 版新加入。

在 3.8.2 版的變更：要求 `Py_ssize_t` 用于 # 格式字符。在此之前，会引发一个不可避免的弃用警告。

int **PySys\_AddAuditHook** (*Py\_AuditHookFunction* hook, void \*userData)

将可调用对象 `hook` 添加到激活的审计钩子列表。在成功时返回零而在失败时返回非零值。如果运行时已经被初始化，还会在失败时设置一个错误。通过此 API 添加的钩子会针对在运行时创建的所有解释器被调用。

`userData` 指针会被传入钩子函数。因于钩子函数可能由不同的运行时调用，该指针不应直接指向 Python 状态。

此函数可在 `Py_Initialize()` 之前被安全地调用。如果在运行时初始化之后被调用，现有的审计钩子将得到通知并可能通过引发一个从 `Exception` 子类化的错误静默地放弃操作（其他错误将不会被静默）。

钩子函数总是会由引发异常的 Python 解释器在持有 GIL 的情况下调用。

请参阅 **PEP 578** 了解有关审计的详细描述。在运行时和标准库中会引发审计事件的函数清单见 审计事件表。更多细节见每个函数的文档。

引發一個不附帶引數的稽核事件 `sys.addaudithook`。



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

钩子函数的类型。*event* 是传给 `PySys_Audit()` 的 C 字符串事件参数。*args* 会确保为一个 `PyTupleObject`。*userData* 是传给 `PySys_AddAuditHook()` 的参数。

在 3.8 版新加入。

## 6.3 行程 (Process) 控制

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

属于稳定 ABI。打印一个致命错误消息并杀死进程。不会执行任何清理。此函数应当仅在检测到可能令继续使用 Python 解释器会有危险的情况时被发起调用；例如对象管理已被破坏的时候。在 Unix 上，会调用标准 C 库函数 `abort()` 并将由它来尝试生成一个 core 文件。

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: 自动记录函数名称。

```
void Py_Exit (int status)
```

属于稳定 ABI。退出当前进程。这将调用 `Py_FinalizeEx()` 然后再调用标准 C 库函数 `exit(status)`。如果 `Py_FinalizeEx()` 提示错误，退出状态将被设为 120。

在 3.6 版的變更: 来自最终化的错误不会再被忽略。

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

属于稳定 ABI。注册一个由 `Py_FinalizeEx()` 调用的清理函数。调用清理函数将不传入任何参数且不应返回任何值。最多可以注册 32 个清理函数。当注册成功时，`Py_AtExit()` 将返回 0；失败时，它将返回 -1。最后注册的清理函数会最先被调用。每个清理函数将至多被调用一次。由于 Python 的内部最终化将在清理函数之前完成，因此 Python API 不应被 *func* 调用。

## 6.4 引入模組

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

回傳值: 新的參照。属于稳定 ABI。这是一个对 `PyImport_Import()` 的包装器，它接受一个 `const char*` 作为参数而不是 `PyObject*`。

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

回傳值: 新的參照。属于稳定 ABI。该函数是 `PyImport_ImportModule()` 的一个被遗弃的别名。

在 3.3 版的變更: 在导入锁被另一线程掌控时此函数会立即失败。但是从 Python 3.3 起，锁方案在大多数情况下都已切换为针对每个模块加锁，所以此函数的特殊行为已无必要。

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

回傳值: 新的參照。导入一个模块。请参阅内置 Python 函数 `__import__()` 获取完善的相关描述。

返回值是一个对所导入模块或最高层级包的新引用，或是在导入失败时则为 NULL 并设置一个异常。与 `__import__()` 类似，当请求一个包的子模块时返回值通常为该最高层级包，除非给出了一个非空的 *fromlist*。

导入失败将移动不完整的模块对象，就像 `PyImport_ImportModule()` 那样。

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

回傳值: 新的參照。属于稳定 ABI 自 3.7 版起。导入一个模块。关于此函数的最佳说明请参考内置 Python 函数 `__import__()`，因为标准 `__import__()` 函数会直接调用此函数。

返回值是一个对所导入模块或最高层级包的新引用，或是在导入失败时则为 `NULL` 并设置一个异常。与 `__import__()` 类似，当请求一个包的子模块时返回值通常为该最高层级包，除非给出了一个非空的 `fromlist`。

在 3.3 版新加入。

**PyObject\*PyImport\_ImportModuleLevel** (const char \*name, PyObject \*globals, PyObject \*locals, PyObject \*fromlist, int level)

回傳值：新的參照。属于稳定 ABI。类似于 `PyImport_ImportModuleLevelObject()`，但其名称为 UTF-8 编码的字符串而不是 Unicode 对象。

在 3.3 版的變更：不再接受 `level` 为负数值。

**PyObject\*PyImport\_Import** (PyObject \*name)

回傳值：新的參照。属于稳定 ABI。这是一个调用了当前“导入钩子函数”的更高层级接口（显式指定 `level` 为 0，表示绝对导入）。它将发起调用当前全局作用域下 `__builtins__` 中的 `__import__()` 函数。这意味着将使用当前环境下安装的任何导入钩子来完成导入。

该函数总是使用绝对路径导入。

**PyObject\*PyImport\_ReloadModule** (PyObject \*m)

回傳值：新的參照。属于稳定 ABI。重载一个模块。返回一个指向被重载模块的新引用，或者在失败时返回 `NULL` 并设置一个异常（在此情况下模块仍然会存在）。

**PyObject\*PyImport\_AddModuleObject** (PyObject \*name)

回傳值：借用參照。属于稳定 ABI 自 3.7 版起。返回对应于某个模块名称的模块对象。`name` 参数的形式可以为 `package.module`。如果存在 `modules` 字典则首先检查该字典，如果找不到，则创建一个新模块并将其插入到 `modules` 字典。在失败时返回 `NULL` 并设置一个异常。

---

**備註：** 此函数不会加载或导入指定模块；如果模块还未被加载，你将得到一个空的模块对象。请使用 `PyImport_ImportModule()` 或它的某个变体形式来导入模块。`name` 使用带点号名称的包结构如果尚不存在则不会被创建。

---

在 3.3 版新加入。

**PyObject\*PyImport\_AddModule** (const char \*name)

回傳值：借用參照。属于稳定 ABI。类似于 `PyImport_AddModuleObject()`，但其名称为 UTF-8 编码的字符串而不是 Unicode 对象。`object`。

**PyObject\*PyImport\_ExecCodeModule** (const char \*name, PyObject \*co)

回傳值：新的參照。属于稳定 ABI。给定一个模块名称（可能为 `package.module` 形式）和一个从 Python 字节码文件读取或从内置函数 `compile()` 获取的代码对象，加载该模块。返回对该模块对象的新引用，或者如果发生错误则返回 `NULL` 并设置一个异常。在发生错误的情况下 `name` 会从 `sys.modules` 中被移除，即使 `name` 在进入 `PyImport_ExecCodeModule()` 时已存在于 `sys.modules` 中。在 `sys.modules` 中保留未完全初始化的模块是危险的，因为导入这样的模块没有办法知道模块对象是否处于一种未知的（对于模块作者的意图来说可能是已损坏的）状态。

模块的 `__spec__` 和 `__loader__` 如果尚未设置的话，将被设为适当的值。相应 `spec` 的加载器（如果已设置）将被设为模块的 `__loader__` 而在其他情况下将被设为 `SourceFileLoader` 的实例。

模块的 `__file__` 属性将被设为代码对象的 `co_filename`。如果适用，还将设置 `__cached__`。

如果模块已被导入则此函数将重载它。请参阅 `PyImport_ReloadModule()` 了解重载模块的预定方式。

如果 `name` 指向一个形式为 `package.module` 的带点号的名称，则任何尚未创建的包结构仍然不会被创建。

另请参阅 `PyImport_ExecCodeModuleEx()` 和 `PyImport_ExecCodeModuleWithPathnames()`。

*PyObject* \*PyImport\_ExecCodeModuleEx (const char \*name, *PyObject* \*co, const char \*pathname)

回傳值: 新的參照。属于稳定 ABI。类似于 `PyImport_ExecCodeModule()`，但如果 `pathname` 不为 NULL 则会被设为模块对象的 `__file__` 属性的值。

也請見 `PyImport_ExecCodeModuleWithPathnames()`。

*PyObject* \*PyImport\_ExecCodeModuleObject (*PyObject* \*name, *PyObject* \*co, *PyObject* \*pathname, *PyObject* \*cpathname)

回傳值: 新的參照。属于稳定 ABI 自 3.7 版起。类似于 `PyImport_ExecCodeModuleEx()`，但如果 `cpathname` 不为 NULL 则会被设为模块对象的 `__cached__` 值。在三个函数中，这是推荐使用的一个。

在 3.3 版新加入。

*PyObject* \*PyImport\_ExecCodeModuleWithPathnames (const char \*name, *PyObject* \*co, const char \*pathname, const char \*cpathname)

回傳值: 新的參照。属于稳定 ABI。类似于 `PyImport_ExecCodeModuleObject()`，但 `name`, `pathname` 和 `cpathname` 为 UTF-8 编码的字符串。如果 `pathname` 也被设为 NULL 则还会尝试根据 `cpathname` 推断出前者的值。

在 3.2 版新加入。

在 3.3 版的變更: 如果只提供了字节码路径则会使用 `imp.source_from_cache()` 来计算源路径。

long PyImport\_GetMagicNumber ()

属于稳定 ABI。返回 Python 字节码文件 (即 `.pyc` 文件) 的魔数。此魔数应当存在于字节码文件的开头四个字节中，按照小端字节序。出错时返回 -1。

在 3.3 版的變更: 當失敗時回傳 -1。

const char \*PyImport\_GetMagicTag ()

属于稳定 ABI。针对 **PEP 3147** 格式的 Python 字节码文件名返回魔术标签字符串。请记住在 `sys.implementation.cache_tag` 上的值是应当被用来代替此函数的更权威的值。

在 3.2 版新加入。

*PyObject* \*PyImport\_GetModuleDict ()

回傳值: 借用參照。属于稳定 ABI。返回用于模块管理的字典 (即 `sys.modules`)。请注意这是针对每个解释器的变量。

*PyObject* \*PyImport\_GetModule (*PyObject* \*name)

回傳值: 新的參照。属于稳定 ABI 自 3.8 版起。返回给定名称的已导入模块。如果模块尚未导入则返回 NULL 但不会设置错误。如果查找失败则返回 NULL 并设置错误。

在 3.7 版新加入。

*PyObject* \*PyImport\_GetImporter (*PyObject* \*path)

回傳值: 新的參照。属于稳定 ABI。返回针对一个 `sys.path/pkg.__path__` 中条目 `path` 的查找器对象，可能会从 `sys.path_importer_cache` 字典中获取。如果它尚未被缓存，则会遍历 `sys.path_hooks` 直至找到一个能处理该路径条目的钩子。如果没有可用的钩子则返回 None；这将告知调用方 *path based finder* 无法为该路径条目找到查找器。结果将缓存到 `sys.path_importer_cache` 中。返回一个指向查找器对象的新引用。

int PyImport\_ImportFrozenModuleObject (*PyObject* \*name)

属于稳定 ABI 自 3.7 版起。加载名称为 `name` 的已冻结模块。成功时返回 1，如果未找到模块则返回 0，如果初始化失败则返回 -1 并设置一个异常。要在加载成功后访问被导入的模块，请使用 `PyImport_ImportModule()`。(请注意此名称有误导性 --- 如果模块已被导入此函数将重载它。)

在 3.3 版新加入。

在 3.4 版的變更: `__file__` 属性将不再在模块上设置。



**int PyImport\_ImportFrozenModule** (const char \*name)

属于稳定 ABI。类似于 `PyImport_ImportFrozenModuleObject()`，但其名称为 UTF-8 编码的字符串而不是 Unicode 对象。

**struct \_frozen**

这是针对已冻结模块描述器的结构类型定义，与由 **freeze** 工具所生成的一致（请参看 Python 源代码发行版中的 `Tools/freeze/`）。其定义可在 `Include/import.h` 中找到：

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

在 3.11 版的變更：新的 `is_package` 字段指明模块是否为一个包。这替代了将 `size` 设为负值的做法。

**const struct \_frozen \*PyImport\_FrozenModules**

该指针被初始化为指向一个 `_frozen` 记录的数组，以一个所有成员均为 NULL 或零的记录表示结束。当一个冻结模块被导入时，它将在此表中被搜索。第三方代码可以利用此方式来提供动态创建的冻结模块集。

**int PyImport\_AppendInittab** (const char \*name, PyObject \*(\*initfunc)(void))

属于稳定 ABI。向现有的内置模块表添加一个模块。这是对 `PyImport_ExtendInittab()` 的便捷包装，如果无法扩展表则返回 -1。新的模块可使用名称 `name` 来导入，并使用函数 `initfunc` 作为在第一次尝试导入时调用的初始化函数。此函数应当在 `Py_Initialize()` 之前调用。

**struct \_inittab**

描述内置模块列表中一个单独条目的结构体。嵌入 Python 的程序可以将这些结构体的数组与 `PyImport_ExtendInittab()` 结合使用以提供额外的内置模块。该结构体由两个成员组成：

**const char \*name**

模块名称，为一个 ASCII 编码的字符串。

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

针对内置于解释器的模块的初始化函数。

**int PyImport\_ExtendInittab** (struct \_inittab \*newtab)

向内置模块表添加一组模块。`newtab` 数组必须以一个包含 NULL 作为 `name` 字段的哨兵条目结束；未提供哨兵值可能导致内存错误。成功时返回 0 或者如果无法分配足够内存来扩展内部表则返回 -1。当失败时，将不会向内部表添加任何模块。该函数必须在 `Py_Initialize()` 之前调用。

如果 Python 要被多次初始化，则 `PyImport_AppendInittab()` 或 `PyImport_ExtendInittab()` 必须在每次 Python 初始化之前调用。

## 6.5 数据 marshal 操作支持

这些例程允许 C 代码处理与 `marshal` 模块所用相同数据格式的序列化对象。其中有些函数可用来将数据写入这种序列化格式，另一些函数则可用来读取并恢复数据。用于存储 `marshal` 数据的文件必须以二进制模式打开。

数字值在存储时会最低位字节放在开头。

此模块支持两种数据格式版本：第 0 版为历史版本，第 1 版本会在文件和 `marshal` 反序列化中共享固化的字符串。第 2 版本会对浮点数使用二进制格式。`Py_MARSHAL_VERSION` 指明了当前文件的格式（当前取值为 2）。

void **PyMarshal\_WriteLongToFile** (long value, FILE \*file, int version)

Marshal a long integer, *value*, to *file*. This will only write the least-significant 32 bits of *value*; regardless of the size of the native long type. *version* indicates the file format.

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

void **PyMarshal\_WriteObjectToFile** (*PyObject* \*value, FILE \*file, int version)

将一个 Python 对象 *value* 以 marshal 格式写入 *file*。 *version* 指明文件格式的版本。

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

*PyObject* \***PyMarshal\_WriteObjectToString** (*PyObject* \*value, int version)

回傳值：新的參照。 返回一个包含 *value* 的 marshal 表示形式的字节串对象。 *version* 指明文件格式的版本。

以下函数允许读取并恢复存储为 marshal 格式的值。

long **PyMarshal\_ReadLongFromFile** (FILE \*file)

Return a C long from the data stream in a FILE\* opened for reading. Only a 32-bit value can be read in using this function, regardless of the native size of long.

发生错误时，将设置适当的异常 (EOFError) 并返回 -1。

int **PyMarshal\_ReadShortFromFile** (FILE \*file)

Return a C short from the data stream in a FILE\* opened for reading. Only a 16-bit value can be read in using this function, regardless of the native size of short.

发生错误时，将设置适当的异常 (EOFError) 并返回 -1。

*PyObject* \***PyMarshal\_ReadObjectFromFile** (FILE \*file)

回傳值：新的參照。 Return a Python object from the data stream in a FILE\* opened for reading.

发生错误时，将设置适当的异常 (EOFError, ValueError 或 TypeError) 并返回 NULL。

*PyObject* \***PyMarshal\_ReadLastObjectFromFile** (FILE \*file)

回傳值：新的參照。 Return a Python object from the data stream in a FILE\* opened for reading. Unlike *PyMarshal\_ReadObjectFromFile()*, this function assumes that no further objects will be read from the file, allowing it to aggressively load file data into memory so that the de-serialization can operate from data in memory rather than reading a byte at a time from the file. Only use these variant if you are certain that you won't be reading anything else from the file.

发生错误时，将设置适当的异常 (EOFError, ValueError 或 TypeError) 并返回 NULL。

*PyObject* \***PyMarshal\_ReadObjectFromString** (const char \*data, *Py\_ssize\_t* len)

回傳值：新的參照。 从包含指向 *data* 的 *len* 个字节的字节缓冲区对应的数据流返回一个 Python 对象。

发生错误时，将设置适当的异常 (EOFError, ValueError 或 TypeError) 并返回 NULL。

## 6.6 剖析引數與建置數值

在创建你自己的扩展函数和方法时，这些函数是有用的。其它的信息和样例见 *extending-index*。

这些函数描述的前三个，*PyArg\_ParseTuple()*，*PyArg\_ParseTupleAndKeywords()*，以及 *PyArg\_Parse()*，它们都使用 格式化字符串 来将函数期待的参数告知函数。这些函数都使用相同语法规则的格式化字符串。

## 6.6.1 解析参数

一个格式化字符串包含 0 或者更多的格式单元。一个格式单元用来描述一个 Python 对象；它通常是一个字符或者由括号括起来的格式单元序列。除了少数例外，一个非括号序列的格式单元通常对应这些函数的具有单一地址的参数。在接下来的描述中，双引号内的表达式是格式单元；圆括号 () 内的是对应这个格式单元的 Python 对象类型；方括号 [] 内的是传递的 C 变量 (变量集) 类型。

### 字符串和缓存区

这些格式允许将对象按照连续的内存块形式进行访问。你没必要提供返回的 unicode 字符或者字节区的原始数据存储。

除非另有说明，缓冲区是不会以空终止的。

有三种办法可以将字符串和缓冲区转换到 C 类型：

- 像 `y*` 和 `s*` 这样的格式会填充一个 `Py_buffer` 结构体。这将锁定下层缓冲区以便调用者随后使用这个缓冲区即使是在 `Py_BEGIN_ALLOW_THREADS` 块中也不会有可变数据因大小调整或销毁所带来的风险。因此，在你结束处理数据（或任何更早的中止场景）之前 **你必须调用** `PyBuffer_Release()`。
- `es`, `es#`, `et` 和 `et#` 等格式会分配结果缓冲区。在你结束处理数据（或任何更早的中止场景）之后 **你必须调用** `PyMem_Free()`。
- 其他格式接受一个 `str` 或只读的 *bytes-like object*，如 `bytes`，并向其缓冲区提供一个 `const char *` 指针。在缓冲区是“被借入”的情况下：它将由对应的 Python 对象来管理，并共享此对象的生命期。你不需要自行释放任何内存。

为确保下层缓冲区可以安全地被借入，对象的 `PyBufferProcs.bf_releasebuffer` 字段必须为 `NULL`。这将不允许普通的可变对象如 `bytearray`，以及某些只读对象如 `bytes` 的 `memoryview`。

在这个 `bf_releasebuffer` 要求以外，没有用于验证输入对象是否为不可变对象的检查（例如它是否会接受可写缓冲区的请求，或者另一个线程是否能改变此数据）。

---

**備註：** 对于所有 # 格式的变体 (`s#`、`y#` 等)，宏 `PY_SSIZE_T_CLEAN` 必须在包含 `Python.h` 之前定义。在 Python 3.9 及更早版本上，如果定义了 `PY_SSIZE_T_CLEAN` 宏，则长度参数的类型为 `Py_ssize_t`，否则为 `int`。

---

#### `s(str) [const char *]`

将一个 Unicode 对象转换成一个指向字符串的 C 指针。一个指针指向一个已经存在的字符串，这个字符串存储的是传如的字符指针变量。C 字符串是已空结束的。Python 字符串不能包含嵌入的无效的代码点；如果由，一个 `ValueError` 异常会被引发。Unicode 对象被转化成 'utf-8' 编码的 C 字符串。如果转换失败，一个 `UnicodeError` 异常被引发。

---

**備註：** 这个表达式不接受 *bytes-like objects*。如果你想接受文件系统路径并将它们转化成 C 字符串，建议使用 `O&` 表达式配合 `PyUnicode_FSConverter()` 作为转化函数。

---

在 3.5 版的變更：以前，当 Python 字符串中遇到了嵌入的 `null` 代码点会引发 `TypeError`。

#### `s*(str 或 bytes-like object) [Py_buffer]`

这个表达式既接受 Unicode 对象也接受类字节类型对象。它为由调用者提供的 `Py_buffer` 结构赋值。这里结果的 C 字符串可能包含嵌入的 `NUL` 字节。Unicode 对象通过 'utf-8' 编码转化成 C 字符串。

#### `s#(str, read-only bytes-like object) [const char *, Py_ssize_t]`

像是 `s*`，区别在于它提供了一个借入的缓冲区。结果存储在两个 C 变量中，第一个是指向 C 字符串的指针，第二个是其长度。该字符串可能包含嵌入的空字节。Unicode 对象会使用 'utf-8' 编码格式转换为 C 字符串。

**z (str 或 None) [const char \*]**

与 s 类似，但 Python 对象也可能为 None，在这种情况下，C 指针设置为 NULL。

**z\* (str、bytes-like object 或 None) [Py\_buffer]**

与 s\* 类似，但 Python 对象也可能为 None，在这种情况下，*Py\_buffer* 结构的 buf 成员设置为 NULL。

**z# (str, read-only bytes-like object 或者 None) [const char \*, Py\_ssize\_t]**

与 s# 类似，但 Python 对象也可能为 None，在这种情况下，C 指针设置为 NULL。

**y (唯讀 bytes-like object) [const char \*]**

这个格式会将一个类字节对象转换为一个指向借入的字符串的 C 指针；它不接受 Unicode 对象。字节缓冲区不可包含嵌入的空字节；如果包含这样的内容，将会引发 ValueError 异常。exception is raised.

在 3.5 版的變更: 以前，当字节缓冲区中遇到了嵌入的 null 字节会引发 TypeError。

**y\* (bytes-like object) [Py\_buffer]**

s\* 的变式，不接受 Unicode 对象，只接受类字节类型变量。这是接受二进制数据的推荐方法。

**y# (read-only bytes-like object) [const char \*, Py\_ssize\_t]**

s# 的变式，不接受 Unicode 对象，只接受类字节类型变量。

**S (bytes) [PyBytesObject \*]**

要求 Python 对象为 bytes 对象，不尝试进行任何转换。如果该对象不为 bytes 对象则会引发 TypeError。C 变量也可被声明为 *PyObject\**。

**Y (bytearray) [PyByteArrayObject \*]**

要求 Python 对象为 bytearray 对象，不尝试进行任何转换。如果该对象不为 bytearray 对象则会引发 TypeError。C 变量也可被声明为 *PyObject\**。

**u (str) [const Py\_UNICODE \*]**

将一个 Python Unicode 对象转化成指向一个以空终止的 Unicode 字符缓冲区的指针。你必须传入一个 *Py\_UNICODE* 指针变量的地址，存储了一个指向已经存在的 Unicode 缓冲区的指针。请注意一个 *Py\_UNICODE* 类型的字符宽度取决于编译选项 (16 位或者 32 位)。Python 字符串必须不能包含嵌入的 null 代码点；如果有，引发一个 ValueError 异常。

在 3.5 版的變更: 以前，当 Python 字符串中遇到了嵌入的 null 代码点会引发 TypeError。

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: 这是旧版样式 *Py\_UNICODE* API; 请迁移至 *PyUnicode\_AsWideCharString()*。

**u# (str) [const Py\_UNICODE \*, Py\_ssize\_t]**

u 的变式，存储两个 C 变量，第一个指针指向一个 Unicode 数据缓存区，第二个是它的长度。它允许 null 代码点。

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: 这是旧版样式 *Py\_UNICODE* API; 请迁移至 *PyUnicode\_AsWideCharString()*。

**Z (str 或 None) [const Py\_UNICODE \*]**

与 u 类似，但 Python 对象也可能为 None，在这种情况下 *Py\_UNICODE* 指针设置为 NULL。

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: 这是旧版样式 *Py\_UNICODE* API; 请迁移至 *PyUnicode\_AsWideCharString()*。

**Z# (str 或 None) [const Py\_UNICODE \*, Py\_ssize\_t]**

与 u# 类似，但 Python 对象也可能为 None，在这种情况下 *Py\_UNICODE* 指针设置为 NULL。

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: 这是旧版样式 *Py\_UNICODE* API; 请迁移至 *PyUnicode\_AsWideCharString()*。

**U (str) [PyObject \*]**

要求 Python 对象为 Unicode 对象，不尝试进行任何转换。如果该对象不为 Unicode 对象则会引发 TypeError。C 变量也可被声明为 *PyObject\**。

**w\* (可讀寫 bytes-like object) [Py\_buffer]**

这个表达式接受任何实现可读写缓存区接口的对象。它为调用者提供的 *Py\_buffer* 结构赋值。缓冲区可能存在嵌入的 null 字节。当缓冲区使用完后调用者需要调用 *PyBuffer\_Release()*。



**es (str) [const char \*encoding, char \*\*buffer]**

s 的变式，它将编码后的 Unicode 字符存入字符缓冲区。它只处理没有嵌 NUL 字节的已编码数据。

此格式需要两个参数。第一个仅用作输入，并且必须为 `const char*`，它指向一个以 NUL 结束的字符串表示的编码格式名称，或者为 `NULL`，这种情况会使用 'utf-8' 编码格式。如果 Python 无法识别指定的编码格式则会引发异常。第二个参数必须为 `char**`；它所引用的指针值将被设为带有参数文本内容的缓冲区。文本将以第一个参数所指定的编码格式进行编码。

`PyArg_ParseTuple()` 会分配一个足够大小的缓冲区，将编码后的数据拷贝进这个缓冲区并且设置 `*buffer` 引用这个新分配的内存空间。调用者有责任在使用后调用 `PyMem_Free()` 去释放已经分配的缓冲区。

**et (str, bytes or bytearray) [const char \*encoding, char \*\*buffer]**

和 es 相同，除了不用重编码传入的字符串对象。相反，它假设传入的参数是编码后的字符串类型。

**es# (str) [const char \*encoding, char \*\*buffer, Py\_ssize\_t \*buffer\_length]**

s# 的变式，它将已编码的 Unicode 字符存入字符缓冲区。不像 es 表达式，它允许传入的数据包含 NUL 字符。

它需要三个参数。第一个仅用作输入，并且必须为 `const char*`，它指向一个以 NUL 结束的字符串表示的编码格式名称，或者为 `NULL`，这种情况会使用 'utf-8' 编码格式。如果 Python 无法识别指定的编码格式则会引发异常。第二个参数必须为 `char**`；它所引用的指针值将被设为带有参数文本内容的缓冲区。文本将以第一个参数所指定的编码格式进行编码。第三个参数必须为指向一个整数的指针；被引用的整数将被设为输出缓冲区中的字节数。

有两种操作方式：

如果 `*buffer` 指向 `NULL` 指针，则函数将分配所需大小的缓冲区，将编码的数据复制到此缓冲区，并设置 `*buffer` 以引用新分配的存储。呼叫者负责调用 `PyMem_Free()` 以在使用后释放分配的缓冲区。

如果 `*buffer` 指向非 `NULL` 指针（已分配的缓冲区），则 `PyArg_ParseTuple()` 将使用此位置作为缓冲区，并将 `*buffer_length` 的初始值解释为缓冲区大小。然后，它会将编码的数据复制到缓冲区，并终止它。如果缓冲区不够大，将设置一个 `ValueError`。

在这两个例子中，`*buffer_length` 被设置为编码后结尾不为 NUL 的数据的长度。

**et# (str, bytes 或 bytearray) [const char \*encoding, char \*\*buffer, Py\_ssize\_t \*buffer\_length]**

和 es# 相同，除了不用重编码传入的字符串对象。相反，它假设传入的参数是编码后的字符串类型。

**數字****b (int) [unsigned char]**

将非负的 Python 整数转换为无符号的微整数，存储为一个 `C unsigned char`。

**B (int) [unsigned char]**

将 Python 整数转换为微整数并且不进行溢出检查，存储为一个 `C unsigned char`。

**h (int) [short int]**

将一个 Python 整数转成 C 的 `short int`。

**H (int) [unsigned short int]**

将一个 Python 整数转成 C 的 `unsigned short int`，转过程無溢位檢查。

**i (int) [int]**

将一个 Python 整数转成 C 的 `int`。

**I (int) [unsigned int]**

将一个 Python 整数转成 C 的 `unsigned int`，转过程無溢位檢查。

**l (int) [long int]**

将一个 Python 整数转成 C 的 `long int`。

**k (int) [unsigned long]**

将一个 Python 整数转成 C 的 `unsigned long`，转过程無溢位檢查。

**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。

**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]**

將一個 Python 字節類型, 如一個長度為 1 的 bytes 或 bytearray 對象, 轉換為 C char。

在 3.3 版的變更: 允許 bytearray 物件。

**C (長度 1 的 str) [int]**

將一個 Python 字符, 如一個長度為 1 的 str 對象, 轉換為 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 複數轉成 C 的 Py\_complex 結構。

**其他物件****o (物件) [PyObject\*]**

將 Python 對象 (未經任何轉換) 存儲到一個 C 對象指針中。這樣 C 程序就能接收到實際傳遞的對象。對象的 *strong reference* 不會被創建 (即其引用計數不會增加)。存儲的指針將不為 NULL。

**o! (物件) [PyObject\*, PyObject\*]**

將一個 Python 對象存入一個 C 對象指針。這類似於 o, 但是接受兩個 C 參數: 第一個是 Python 類型對象的地址, 第二個是存儲對象指針的 C 變量 (類型為 PyObject\*)。如果 Python 對象不具有所要求的類型, 則會引發 TypeError。

**o& (物件) [converter, anything]**

通過 converter 函數將 Python 對象轉換為 C 變量。這需要兩個參數: 第一個是函數, 第二個是 C 變量 (任意類型) 的地址, 轉換為 void\*。轉換器函數依次調用如下:

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

其中 object 是待轉換的 Python 對象而 address 為傳給 PyArg\_Parse\* 函數的 void\* 參數。返回的 status 應當以 1 代表轉換成功而以 0 代表轉換失敗。當轉換失敗時, converter 函數應當引發異常並讓 address 的內容保持未修改狀態。

如果 converter 返回 Py\_CLEANUP\_SUPPORTED, 則如果參數解析最終失敗, 它可能會再次調用該函數, 從而使轉換器有機會釋放已分配的任何內存。在第二個調用中, object 參數將為 NULL; 因此, 該參數將為 NULL; 因此, 該參數將為 NULL, 因此, 該參數將為 NULL (如果值) 為 NULL address 的值與原始呼叫中的值相同。

在 3.1 版的變更: 加入 Py\_CLEANUP\_SUPPORTED。

**p(bool) [int]**

測試傳入的值是否為真 (一個布林判斷) 並且將結果轉化為相對應的 C true/false 整型值。如果表達式為真置 1, 假則置 0。它接受任何合法的 Python 值。參見 truth 獲取更多關於 Python 如何測試值為真的信息。

在 3.3 版新加入。

**(items) (tuple) [matching-items]**

對象必須是 Python 序列, 它的長度是 items 中格式單元的數量。C 參數必須對應 items 中每一個獨立的格式單元。序列中的格式單元可能有嵌套。

传递“long”整型(取值超出平台的 `LONG_MAX` 限制的整形)是可能的,然而不会进行适当的范围检测 --- 当接受字段太小而接收不到值时,最高有效比特位会被静默地截断(实际上,该语义是继承自 C 的向下转换 --- 你的计数可能会发生变化)。

格式化字符串中还有一些其他的字符具有特殊的涵义。这些可能并不嵌套在圆括号中。它们是:

- | 表明在 Python 参数列表中剩下的参数都是可选的。C 变量对应的可选参数需要初始化为默认值 --- 当一个可选参数没有指定时, `PyArg_ParseTuple()` 不能访问相应的 C 变量(变量集)的内容。
- \$ `PyArg_ParseTupleAndKeywords()` only: 表明在 Python 参数列表中剩下的参数都是强制关键字参数。当前,所有强制关键字参数都必须也是可选参数,所以格式化字符串中 | 必须一直在 \$ 前面。  
在 3.3 版新加入。
- :
- 格式单元的列表结束标志;冒号后的字符串被用来作为错误消息中的函数名(`PyArg_ParseTuple()` 函数引发的“关联值”异常)。
- ;
- 格式单元的列表结束标志;分号后的字符串被用来作为错误消息取代默认的错误消息。: 和 ; 相互排斥。

请注意提供给调用者的任何 Python 对象引用都是 借入引用;不要释放它们(即不要递减它们的引用计数)!

传递给这些函数的附加参数必须是由格式化字符串确定的变量的地址;这些都是用来存储输入元组的值。有一些情况,如上面的格式单元列表中所描述的,这些参数作为输入值使用;在这种情况下,它们应该匹配指定的相应的格式单元。

为了让转换成功, `arg` 对象必须匹配格式并且格式必须被用尽。当成功时, `PyArg_Parse*` 函数将返回真值,否则将返回假值并引发适当的异常。当 `PyArg_Parse*` 函数由于某个格式单元转换出错而失败时,该格式单元及其后续格式单元对应的地址上的变量都将保持原样。

## API 函式

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

属于稳定 ABI. 解析一个函数的参数,表达式中的参数按参数位置顺序存入局部变量中。成功返回 true; 失败返回 false 并且引发相应的异常。

**int `PyArg_VaParse`** (*PyObject* \*args, const char \*format, va\_list vargs)

属于稳定 ABI. 和 `PyArg_ParseTuple()` 相同,然而它接受一个 `va_list` 类型的参数而不是可变数量的参数集。

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

属于稳定 ABI. 分析将位置参数和关键字参数同时转换为局部变量的函数的参数。`keywords` 参数是关键字参数名称的 NULL 终止数组。空名称表示 *positional-only parameters*。成功时返回 true; 发生故障时,它将返回 false 并引发相应的异常。

在 3.6 版的變更: 添加了 *positional-only parameters* 的支持。

**int `PyArg_VaParseTupleAndKeywords`** (*PyObject* \*args, *PyObject* \*kw, const char \*format, char \*keywords[], va\_list vargs)

属于稳定 ABI. 和 `PyArg_ParseTupleAndKeywords()` 相同,然而它接受一个 `va_list` 类型的参数而不是可变数量的参数集。

`int PyArg_ValidateKeywordArguments (PyObject*)`

属于稳定 ABI。确保字典中的关键字参数都是字符串。这个函数只被使用于 `PyArg_ParseTupleAndKeywords()` 不被使用的情况下，后者已经不再做这样的检查。

在 3.2 版新加入。

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

属于稳定 ABI。函数被用来析构“旧类型”函数的参数列表——这些函数使用的 `METH_OLDARGS` 参数解析方法已从 Python 3 中移除。这不被推荐用于新代码的参数解析，并且在标准解释器中的大多数代码已被修改，已不再用于该目的。它仍然方便于分解其他元组，然而可能因为这个目的被继续使用。

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

属于稳定 ABI。一个更简单的形参提取形式，它不使用格式字符串来指定参数类型。使用此方法来提取其形参的函数应当在函数或方法表中声明为 `METH_VARARGS`。包含实际形参的元组应当作为 `args` 传入；它必须确实是一个元组。该元组的长度必须至少为 `min` 且不超过 `max`；`min` 和 `max` 可能相等。额外的参数必须被传给函数，每个参数应当是一个指向 `PyObject*` 变量的指针；它们将来自 `args` 的值来填充；它们将包含借入引用。对应于 `args` 未给出的可选形参的变量不会被填充；它们应当由调用方来初始化。此函数在执行成功时返回真值而在 `args` 不为元组或包含错误数量的元素时返回假值；如果执行失败则还将设置一个异常。

这是一个使用该函数的示例，取自 `_weakref` 弱引用辅助模块的源代码：

```
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;
}
```

这个例子中调用 `PyArg_UnpackTuple()` 完全等价于调用 `PyArg_ParseTuple()`：

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

## 6.6.2 创建变量

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

回傳值：新的参照。属于稳定 ABI。基于类似 `PyArg_Parse*` 函数族所接受内容的格式字符串和一个值序列来创建一个新值。返回该值或在发生错误的情况下返回 `NULL`；如果返回 `NULL` 则将引发一个异常。

`Py_BuildValue()` 并不一直创建一个元组。只有当它的格式化字符串包含两个或更多的格式单元才会创建一个元组。如果格式化字符串是空，它返回 `None`；如果它包含一个格式单元，它返回由格式单元描述的任一对象。用圆括号包裹格式化字符串可以强制它返回一个大小为 0 或者 1 的元组。

当内存缓存区的数据以参数形式传递用来构建对象时，如 `s` 和 `s#` 格式单元，会拷贝需要的数据。调用者提供的缓冲区从来都不会被由 `Py_BuildValue()` 创建的对象来引用。换句话说，如果你的代码调用 `malloc()` 并且将分配的内存空间传递给 `Py_BuildValue()`，你的代码就有责任在 `Py_BuildValue()` 返回时调用 `free()`。

在下面的描述中，双引号的表达式使格式单元；圆括号 () 内的是格式单元将要返回的 Python 对象类型；方括号 [] 内的是传递的 C 变量 (变量集) 的类型。



字符例如空格，制表符，冒号和逗号在格式化字符串中会被忽略 (但是不包括格式单元，如 `s#`)。这可以使很长的格式化字符串具有更好的可读性。

**s (str 或 None) [const char \*]**

使用 'utf-8' 编码将空终止的 C 字符串转换为 Python str 对象。如果 C 字符串指针为 NULL，则使用 None。

**s# (str 或 None) [const char \*, Py\_ssize\_t]**

使用 'utf-8' 编码将 C 字符串及其长度转换为 Python str 对象。如果 C 字符串指针为 NULL，则长度将被忽略，并返回 None。

**y (bytes) [const char \*]**

这将 C 字符串转换为 Python bytes 对象。如果 C 字符串指针为 NULL，则返回 None。

**y# (bytes) [const char \*, Py\_ssize\_t]**

这会将 C 字符串及其长度转换为一个 Python 对象。如果该 C 字符串指针为 NULL，则返回 None。

**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 \*]**

将空终止的 wchar\_t 的 Unicode (UTF-16 或 UCS-4) 数据缓冲区转换为 Python Unicode 对象。如果 Unicode 缓冲区指针为 NULL，则返回 None。

**u# (str) [const wchar\_t \*, Py\_ssize\_t]**

将 Unicode (UTF-16 或 UCS-4) 数据缓冲区及其长度转换为 Python Unicode 对象。如果 Unicode 缓冲区指针为 NULL，则长度将被忽略，并返回 None。

**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。

**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 1 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 1 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 複數。

**o (物件) [PyObject \*]**

原封不動地傳遞一個 Python 對象，但為其創建一個新的 *strong reference* (即其引用計數加一)。如果傳入的對象是一個 `NULL` 指針，則會假定這是因為產生該參數的調用發現了錯誤並設置了異常。因此，`Py_BuildValue()` 將返回 `NULL` 但不會引發異常。如果尚未引發異常，則會設置 `SystemError`。

**s (物件) [PyObject \*]**

和 `o` 相同。

**N (物件) [PyObject \*]**

與 `o` 相同，但它不會創建新的 *strong reference*。如果對象是通過調用參數列表中的對象構造器來創建的則該方法將很有用處。

**o& (物件) [converter, anything]**

通過 `converter` 函數將 *anything* 轉換為 Python 對象。該函數在調用時附帶 *anything* (它應當兼容 `void*`) 作為其參數並且應返回一個“新的” Python 對象，或者如果發生錯誤則返回 `NULL`。

**(items) (tuple) [matching-items]**

將一個 C 變量序列轉換成 Python 元組並保持相同的元素數量。

**[items] (list) [matching-items]**

將一個 C 變量序列轉換成 Python 列表並保持相同的元素數量。

**{items} (dict) [matching-items]**

將一個 C 變量序列轉換成 Python 字典。每一對連續的 C 變量對作為一個元素插入字典中，分別作為關鍵字和值。

如果格式字符串中出現錯誤，則設置 `SystemError` 異常並返回 `NULL`。

*PyObject \****Py\_VaBuildValue** (const char \*format, va\_list args)

回傳值：新的參照。屬於穩定 ABI。和 `Py_BuildValue()` 相同，然而它接受一個 `va_list` 類型的參數而不是可變數量的參數集。

## 6.7 字串轉與格式化

數字轉函式和被格式化的字串輸出。

**int PyOS\_snprintf** (char \*str, size\_t size, const char \*format, ...)

屬於穩定 ABI。根據格式字符串 *format* 和額外參數，輸出不超過 *size* 個字節到 *str*。參見 Unix 手冊頁面 `snprintf(3)`。

**int PyOS\_vsnprintf** (char \*str, size\_t size, const char \*format, va\_list va)

屬於穩定 ABI。根據格式字符串 *format* 和變量參數列表 *va*，輸出不超過 *size* 個字節到 *str*。參見 Unix 手冊頁面 `vsnprintf(3)`。

`PyOS_snprintf()` 和 `PyOS_vsnprintf()` 包装 C 标准库函数 `snprintf()` 和 `vsnprintf()`。它们的目的是保证在极端情况下的一致行为，而标准 C 的函数则不然。

此包装器会确保 `str[size-1]` 在返回时始终为 `'\0'`。它们从不写入超过 `size` 字节 (包括末尾的 `'\0'`) 到 `str`。两个函数都要求 `str != NULL`, `size > 0`, `format != NULL` 且 `size < INT_MAX`。请注意这意味着不存在可确定所需缓冲区大小的 C99 `n = snprintf(NULL, 0, ...)` 的等价物。

當回傳值 (`rv`) 給這些函式應該被編譯如下：

- 当  $0 \leq rv < size$  时，输出转换即成功并将 `rv` 个字符写入到 `str` (不包括末尾 `str[rv]` 位置的 `'\0'` 字节)。
- 当 `rv >= size` 时，输出转换会被截断并且需要一个具有 `rv + 1` 字节的缓冲区才能成功执行。在此情况下 `str[size-1]` 为 `'\0'`。
- 当 `rv < 0` 时，”会发生不好的事情。”在此情况下 `str[size-1]` 也为 `'\0'`，但 `str` 的其余部分是未定义的。错误的确切原因取决于底层平台。

以下函数提供与语言环境无关的字符串到数字转换。

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

属于稳定 ABI。根据给定的 `base` 将 `str` 中字符串的初始部分转换为 unsigned long 值，该值必须在 2 至 36 的开区间内，或者为特殊值 0。

空白前缀和字符大小写将被忽略。如果 `base` 为零则会查找 `0b`、`0o` 或 `0x` 前缀以确定基数。如果没有则默认基数为 10。基数必须为 0 或在 2 和 36 之间（包括边界值）。如果 `ptr` 不为 `NULL` 则它将包含一个指向扫描结束位置的指针。

如果转换后的值不在对应返回类型的取值范围之内，则会发生取值范围错误 (`errno` 被设为 `ERANGE`) 并返回 `ULONG_MAX`。如果无法执行转换，则返回 0。

另请参阅 Unix 指南页 `strtoul(3)`。

在 3.2 版新加入。

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

属于稳定 ABI。根据给定的 `base` 将 `str` 中字符串的初始部分转换为 long 值，该值必须在 2 至 36 的开区间内，或者为特殊值 0。

类似于 `PyOS_strtoul()`，但在溢出时将返回一个 long 值而不是 `LONG_MAX`。

另请参阅 Unix 指南页 `strtol(3)`。

在 3.2 版新加入。

**double `PyOS_string_to_double` (const char \*s, char \*\*endptr, *PyObject* \*overflow\_exception)**

属于稳定 ABI。将字符串 `s` 转换为 double 类型，失败时会引发 Python 异常。接受的字符串集合对应于可被 Python 的 `float()` 构造器所接受的字符集集合，除了 `s` 必须没有前导或尾随空格。转换必须独立于当前的语言区域。

如果 `endptr` 是 `NULL`，转换整个字符串。引发 `ValueError` 并且返回 `-1.0` 如果字符串不是浮点数的有效的表达方式。

如果 `endptr` 不是 `NULL`，尽可能多的转换字符串并将 `*endptr` 设置为指向第一个未转换的字符。如果字符串的初始段不是浮点数的有效的表达方式，将 `*endptr` 设置为指向字符串的开头，引发 `ValueError` 异常，并且返回 `-1.0`。

如果 `s` 表示一个太大而不能存储在一个浮点数中的值（比方说，`"1e500"` 在许多平台上是一个字符串）然后如果 `overflow_exception` 是 `NULL` 返回 `Py_HUGE_VAL`（用适当的符号）并且不设置任何异常。在其他方面，`overflow_exception` 必须指向一个 Python 异常对象；引发异常并返回 `-1.0`。在这两种情况下，设置 `*endptr` 指向转换值之后的第一个字符。

如果在转换期间发生任何其他错误（比如一个内存不足的错误），设置适当的 Python 异常并且返回 `-1.0`。

在 3.1 版新加入。

char **PyOS\_double\_to\_string** (double val, char format\_code, int precision, int flags, int \*ptype)

属于稳定 ABI。将 double val 转换为一个使用给定的 *format\_code*, *precision* 和 *flags* 的字符串。

格式码必须是以下其中之一, 'e', 'E', 'f', 'F', 'g', 'G' 或者 'r'。对于 'r', 提供的精度必须是 0。'r' 格式码指定了标准函数 repr() 格式。

*flags* 可以为零或者其他值 Py\_DTST\_SIGN, Py\_DTST\_ADD\_DOT\_0 或 Py\_DTST\_ALT 或其组合:

- Py\_DTST\_SIGN 表示总是在返回的字符串前附加一个符号字符, 即使 val 为非负数。
- Py\_DTST\_ADD\_DOT\_0 表示确保返回的字符串看起来不像是个整数。
- Py\_DTST\_ALT 表示应用“替代的”格式化规则。相关细节请参阅 `PyOS_snprintf()` '#' 定义文档。

如果 *ptype* 不为 NULL, 则它指向的值将被设为 Py\_DTST\_FINITE, Py\_DTST\_INFINITE 或 Py\_DTST\_NAN 中的一个, 分别表示 val 是一个有限数字、无限数字或非数字。

返回值是一个指向包含转换后字符串的 *buffer* 的指针, 如果转换失败则为 NULL。调用方要负责调用 `PyMem_Free()` 来释放返回的字符串。

在 3.1 版新加入。

int **PyOS\_stricmp** (const char \*s1, const char \*s2)

不区分大小写的字符串比较。除了忽略大小写之外, 该函数的工作方式与 strcmp() 相同。

int **PyOS\_strnicmp** (const char \*s1, const char \*s2, Py\_ssize\_t size)

不区分大小写的字符串比较。除了忽略大小写之外, 该函数的工作方式与 strncmp() 相同。

## 6.8 PyHash API

另请参阅 `PyTypeObject.tp_hash` 成员。

type **Py\_hash\_t**

哈希值类型: 有符号整数。

在 3.2 版新加入。

type **Py\_uhash\_t**

哈希值类型: 无符号整数。

在 3.2 版新加入。

type **PyHash\_FuncDef**

`PyHash_GetFuncDef()` 使用的哈希函数定义。

const char \***name**

哈希函数名称 (UTF-8 编码的字符串)。

const int **hash\_bits**

以比特位表示的哈希值内部大小。

const int **seed\_bits**

以比特位表示的输入种子值大小。

在 3.4 版新加入。

`PyHash_FuncDef` \***PyHash\_GetFuncDef** (void)

获取哈希函数定义。

**也参考:**

**PEP 456** “安全且可互换的哈希算法”。

在 3.4 版新加入。

## 6.9 反射

*PyObject* \*PyEval\_GetBuiltins (void)

回傳值：借用參照。属于稳定 ABI。返回当前执行帧中内置函数的字典，如果当前没有帧正在执行，则返回线程状态的解释器。

*PyObject* \*PyEval\_GetLocals (void)

回傳值：借用參照。属于稳定 ABI。返回当前执行帧中局部变量的字典，如果没有当前执行的帧则返回 NULL。

*PyObject* \*PyEval\_GetGlobals (void)

回傳值：借用參照。属于稳定 ABI。返回当前执行帧中全局变量的字典，如果没有当前执行的帧则返回 NULL。

*PyFrameObject* \*PyEval\_GetFrame (void)

回傳值：借用參照。属于稳定 ABI。返回当前线程状态的帧，如果没有当前执行的帧则返回 NULL。

另請見 *PyThreadState\_GetFrame()*。

const char \*PyEval\_GetFuncName (*PyObject* \*func)

属于稳定 ABI。如果 *func* 是函数、类或实例对象，则返回它的名称，否则返回 *func* 的类型的名称。

const char \*PyEval\_GetFuncDesc (*PyObject* \*func)

属于稳定 ABI。根据 *func* 的类型返回描述字符串。返回值包括函数和方法的“()”，“constructor”，“instance”和“object”。与 *PyEval\_GetFuncName()* 的结果连接，结果将是 *func* 的描述。

## 6.10 编解码器注册与支持功能

int PyCodec\_Register (*PyObject* \*search\_function)

属于稳定 ABI。注册一个新的编解码器搜索函数。

作为其附带影响，如果 *encodings* 包尚未加载，则会尝试加载它，以确保它在搜索函数列表中始终排在第一位。

int PyCodec\_Unregister (*PyObject* \*search\_function)

属于稳定 ABI 自 3.10 版起，注销一个编解码器搜索函数并清空注册表缓存。如果指定搜索函数未被注册，则不做任何操作。成功时返回 0。出错时引发一个异常并返回 -1。

在 3.10 版新加入。

int PyCodec\_KnownEncoding (const char \*encoding)

属于稳定 ABI。根据注册的给定 *encoding* 的编解码器是否已存在而返回 1 或 0。此函数总能成功。

*PyObject* \*PyCodec\_Encode (*PyObject* \*object, const char \*encoding, const char \*errors)

回傳值：新的參照。属于稳定 ABI。泛型编解码器基本编码 API。

*object* 使用由 *errors* 所定义的错误处理方法传递给 *encoding* 的编码器函数。*errors* 可以为 NULL 表示使用为编码器所定义的默认方法。如果找不到编码器则会引发 *LookupError*。

*PyObject* \*PyCodec\_Decode (*PyObject* \*object, const char \*encoding, const char \*errors)

回傳值：新的參照。属于稳定 ABI。泛型编解码器基本解码 API。

*object* 使用由 *errors* 所定义的错误处理方法传递给 *encoding* 的解码器函数。*errors* 可以为 NULL 表示使用为编解码器所定义的默认方法。如果找不到编解码器则会引发 *LookupError*。



## 6.10.1 Codec 查找 API

在下列函数中, *encoding* 字符串会被查找并转换为小写字母形式, 这使得通过此机制查找编码格式实际上对大小写不敏感。如果未找到任何编解码器, 则将设置 `KeyError` 并返回 `NULL`。

*PyObject* \***PyCodec\_Encoder** (const char \*encoding)

回傳值: 新的參照。属于稳定 ABI。为给定的 *encoding* 获取一个编码器函数。

*PyObject* \***PyCodec\_Decoder** (const char \*encoding)

回傳值: 新的參照。属于稳定 ABI。为给定的 *encoding* 获取一个解码器函数。

*PyObject* \***PyCodec\_IncrementalEncoder** (const char \*encoding, const char \*errors)

回傳值: 新的參照。属于稳定 ABI。为给定的 *encoding* 获取一个 `IncrementalEncoder` 对象。

*PyObject* \***PyCodec\_IncrementalDecoder** (const char \*encoding, const char \*errors)

回傳值: 新的參照。属于稳定 ABI。为给定的 *encoding* 获取一个 `IncrementalDecoder` 对象。

*PyObject* \***PyCodec\_StreamReader** (const char \*encoding, *PyObject* \*stream, const char \*errors)

回傳值: 新的參照。属于稳定 ABI。为给定的 *encoding* 获取一个 `StreamReader` 工厂函数。

*PyObject* \***PyCodec\_StreamWriter** (const char \*encoding, *PyObject* \*stream, const char \*errors)

回傳值: 新的參照。属于稳定 ABI。为给定的 *encoding* 获取一个 `StreamWriter` 工厂函数。

## 6.10.2 用于 Unicode 编码错误处理程序的注册表 API

int **PyCodec\_RegisterError** (const char \*name, *PyObject* \*error)

属于稳定 ABI。在给定的 *name* 之下注册错误处理回调函数 *error*。该回调函数将在一个编解码器遇到无法编码的字符/无法解码的字节数据并且 *name* 被指定为 `encode/decode` 函数调用的 *error* 形参时由该编解码器来调用。

该回调函数会接受一个 `UnicodeEncodeError`, `UnicodeDecodeError` 或 `UnicodeTranslateError` 的实例作为单独参数, 其中包含关于有问题字符或字节序列及其在原始序列的偏移量信息 (请参阅 [Unicode 异常对象](#) 了解提取此信息的函数详情)。该回调函数必须引发给定的异常, 或者返回一个包含有问题序列及相应替换序列的二元组, 以及一个表示偏移量的整数, 该整数指明应在什么位置上恢复编码/解码操作。

成功则返回 0, 失败则返回 -1。

*PyObject* \***PyCodec\_LookupError** (const char \*name)

回傳值: 新的參照。属于稳定 ABI。查找在 *name* 之下注册的错误处理回调函数。作为特例还可以传入 `NULL`, 在此情况下将返回针对“strict”的错误处理回调函数。

*PyObject* \***PyCodec\_StrictErrors** (*PyObject* \*exc)

回傳值: 總是 `NULL`。属于稳定 ABI。引发 *exc* 作为异常。

*PyObject* \***PyCodec\_IgnoreErrors** (*PyObject* \*exc)

回傳值: 新的參照。属于稳定 ABI。忽略 unicode 错误, 跳过错误的输入。

*PyObject* \***PyCodec\_ReplaceErrors** (*PyObject* \*exc)

回傳值: 新的參照。属于稳定 ABI。使用 ? 或 U+FFFD 替换 unicode 编码错误。

*PyObject* \***PyCodec\_XMLCharRefReplaceErrors** (*PyObject* \*exc)

回傳值: 新的參照。属于稳定 ABI。使用 XML 字符引用替换 unicode 编码错误。

*PyObject* \***PyCodec\_BackslashReplaceErrors** (*PyObject* \*exc)

回傳值: 新的參照。属于稳定 ABI。使用反斜杠转义符 (\x, \u 和 \U) 替换 unicode 编码错误。

*PyObject* \***PyCodec\_NameReplaceErrors** (*PyObject* \*exc)

回傳值: 新的參照。属于稳定 ABI 自 3.7 版起, 使用 `\N{...}` 转义符替换 unicode 编码错误。

在 3.5 版新加入。

---

## 抽象物件層 (Abstract Objects Layer)

---

本章中的函式與 Python 物件相互作用，無論其型別、或具有廣泛類別的物件型別（例如所有數值型別或所有序列型別）。當使用於不適用的物件型別時，他們會引發一個 Python 異常 (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定一些非 NULL 的值。

### 7.1 对象协议

#### `PyObject *Py_NotImplemented`

`NotImplemented` 单例，用于标记某个操作没有针对给定类型组合的实现。

#### `Py_RETURN_NOTIMPLEMENTED`

正确处理从 C 语言函数中返回 `Py_NotImplemented` 的问题（即新建一个指向 `NotImplemented` 的 *strong reference* 并返回它）。

#### `Py_PRINT_RAW`

要与多个打印对象的函数（如 `PyObject_Print()` 和 `PyFile_WriteObject()`）一起使用的标志。如果传入，这些函数应当使用对象的 `str()` 而不是 `repr()`。

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

打印对象 `o` 到文件 `fp`。出错时返回 -1。flags 参数被用于启用特定的打印选项。目前唯一支持的选项是 `Py_PRINT_RAW`；如果给出该选项，则将写入对象的 `str()` 而不是 `repr()`。

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

属于稳定 ABI。如果 `o` 带有属性 `attr_name`，则返回 1，否则返回 0。这相当于 Python 表达式 `hasattr(o, attr_name)`。此函数总是成功。

---

**備註：** 在调用 `__getattr__()` 和 `__getattribute__()` 方法时发生的异常将被静默地忽略。想要进行适当的错误处理，请改用 `PyObject_GetAttr()`。

---

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

属于稳定 ABI。这与 `PyObject_HasAttr()` 相同，但 `attr_name` 被指定为 `const char*` UTF-8 编码的字节串，而不是 `PyObject*`。

備註：在调用 `__getattr__()` 和 `__getattribute__()` 方法时或者在创建临时 `str` 对象期间发生的异常将被静默地忽略。想要进行适当的处理，请改用 `PyObject_GetAttrString()`。

**PyObject\*PyObject\_GetAttr (PyObject \*o, PyObject \*attr\_name)**

回傳值：新的參照。属于穩定 ABI。从对象 `o` 中读取名为 `attr_name` 的属性。成功返回属性值，失败则返回 `NULL`。这相当于 Python 表达式 `o.attr_name`。

**PyObject\*PyObject\_GetAttrString (PyObject \*o, const char \*attr\_name)**

回傳值：新的參照。属于穩定 ABI。这与 `PyObject_GetAttr()` 相同，但 `attr_name` 被指定为 `const char*` UTF-8 编码的字节串，而不是 `PyObject*`。

**PyObject\*PyObject\_GenericGetAttr (PyObject \*o, PyObject \*name)**

回傳值：新的參照。属于穩定 ABI。通用的属性获取函数，用于放入类型对象的 `tp_getattro` 槽中。它在类的字典中（位于对象的 MRO 中）查找某个描述符，并在对象的 `__dict__` 中查找某个属性。正如 `descriptors` 所述，数据描述符优先于实例属性，而非数据描述符则不优先。失败则会触发 `AttributeError`。

**int PyObject\_SetAttr (PyObject \*o, PyObject \*attr\_name, PyObject \*v)**

属于穩定 ABI。将对象 `o` 中名为 `attr_name` 的属性值设为 `v`。失败时引发异常并返回 `-1`；成功时返回 `0`。这相当于 Python 语句 `o.attr_name = v`。

如果 `v` 为 `NULL`，该属性将被删除。此行为已被弃用而应改用 `PyObject_DelAttr()`，但目前还没有移除它的计划。

**int PyObject\_SetAttrString (PyObject \*o, const char \*attr\_name, PyObject \*v)**

属于穩定 ABI。这与 `PyObject_SetAttr()` 相同，但 `attr_name` 被指定为 `const char*` UTF-8 编码的字节串，而不是 `PyObject*`。

如果 `v` 为 `NULL`，该属性将被删除，但是此功能已被弃用而应改用 `PyObject_DelAttrString()`。

**int PyObject\_GenericSetAttr (PyObject \*o, PyObject \*name, PyObject \*value)**

属于穩定 ABI。通用的属性设置和删除函数，用于放入类型对象的 `tp_setattro` 槽。它在类的字典中（位于对象的 MRO 中）查找数据描述器，如果找到，则将在实例字典中设置或删除属性优先执行。否则，该属性将在对象的 `__dict__` 中设置或删除。如果成功将返回 `0`，否则将引发 `AttributeError` 并返回 `-1`。

**int PyObject\_DelAttr (PyObject \*o, PyObject \*attr\_name)**

删除对象 `o` 中名为 `attr_name` 的属性。失败时返回 `-1`。这相当于 Python 语句 `del o.attr_name`。

**int PyObject\_DelAttrString (PyObject \*o, const char \*attr\_name)**

这与 `PyObject_DelAttr()` 相同，但 `attr_name` 被指定为 `const char*` UTF-8 编码的字节串，而不是 `PyObject*`。

**PyObject\*PyObject\_GenericGetDict (PyObject \*o, void \*context)**

回傳值：新的參照。属于穩定 ABI 自 3.10 版起。 `__dict__` 描述符的获取函数的一种通用实现。必要时会创建该字典。

此函数还可能被调用以获取对象 `o` 的 `__dict__`。当调用它时可传入 `NULL` 作为 `context`。由于此函数可能需要为字典分配内存，所以在访问对象上的属性时调用 `PyObject_GetAttr()` 可能会更为高效。

当失败时，将返回 `NULL` 并设置一个异常。

在 3.3 版新加入。

**int PyObject\_GenericSetDict (PyObject \*o, PyObject \*value, void \*context)**

属于穩定 ABI 自 3.7 版起。 `__dict__` 描述符设置函数的一种通用实现。这里不允许删除该字典。

在 3.3 版新加入。



*PyObject* \***PyObject\_GetDictPtr** (*PyObject* \*obj)

返回一个指向对象 *obj* 的 `__dict__` 的指针。如果不存在 `__dict__`，则返回 NULL 并且不设置异常。

此函数可能需要为字典分配内存，所以在访问对象上的属性时调用 `PyObject_GetAttr()` 可能会更为高效。

*PyObject* \***PyObject\_RichCompare** (*PyObject* \*o1, *PyObject* \*o2, int opid)

回傳值：新的参照。属于稳定 ABI。使用由 *opid* 指定的操作来比较 *o1* 和 *o2* 的值，操作必须为 `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT` 或 `Py_GE` 中的一个，分别对应于 `<`, `<=`, `==`, `!=`, `>` 或 `>=`。这等价于 Python 表达式 `o1 op o2`，其中 `op` 是与 *opid* 对应的运算符。成功时返回比较结果值，失败时返回 NULL。

int **PyObject\_RichCompareBool** (*PyObject* \*o1, *PyObject* \*o2, int opid)

属于稳定 ABI。使用 *opid* 所指定的操作，例如 `PyObject_RichCompare()` 来比较 *o1* 和 *o2* 的值，但在出错时返回 -1，在结果为假值时返回 0，在其他情况下返回 1。

---

**備註：** 如果 *o1* 和 *o2* 是同一个对象，`PyObject_RichCompareBool()` 将总是为 `Py_EQ` 返回 1 并为 `Py_NE` 返回 0。

---

*PyObject* \***PyObject\_Format** (*PyObject* \*obj, *PyObject* \*format\_spec)

属于稳定 ABI。格式 *obj* 使用 *format\_spec*。这等价于 Python 表达式 `format(obj, format_spec)`。

*format\_spec* 可以为 NULL。在此情况下调用将等价于 `format(obj)`。成功时返回已格式化的字符串，失败时返回 NULL。

*PyObject* \***PyObject\_Repr** (*PyObject* \*o)

回傳值：新的参照。属于稳定 ABI。计算对象 *o* 的字符串形式。成功时返回字符串，失败时返回 NULL。这相当于 Python 表达式 `repr(o)`。由内置函数 `repr()` 调用。

在 3.4 版的變更：该函数现在包含一个调试断言，用以确保不会静默地丢弃活动的异常。

*PyObject* \***PyObject\_ASCII** (*PyObject* \*o)

回傳值：新的参照。属于稳定 ABI。与 `PyObject_Repr()` 一样，计算对象 *o* 的字符串形式，但在 `PyObject_Repr()` 返回的字符串中用 `\x`、`\u` 或 `\U` 转义非 ASCII 字符。这将生成一个类似于 Python 2 中由 `PyObject_Repr()` 返回的字符串。由内置函数 `ascii()` 调用。

*PyObject* \***PyObject\_Str** (*PyObject* \*o)

回傳值：新的参照。属于稳定 ABI。计算对象 *o* 的字符串形式。成功时返回字符串，失败时返回 NULL。这相当于 Python 表达式 `str(o)`。由内置函数 `str()` 调用，因此也由 `print()` 函数调用。

在 3.4 版的變更：该函数现在包含一个调试断言，用以确保不会静默地丢弃活动的异常。

*PyObject* \***PyObject\_Bytes** (*PyObject* \*o)

回傳值：新的参照。属于稳定 ABI。计算对象 *o* 的字节形式。失败时返回 NULL，成功时返回一个字节串对象。这相当于 *o* 不是整数时的 Python 表达式 `bytes(o)`。与 `bytes(o)` 不同的是，当 *o* 是整数而不是初始为 0 的字节串对象时，会触发 `TypeError`。

int **PyObject\_IsSubclass** (*PyObject* \*derived, *PyObject* \*cls)

属于稳定 ABI。如果 *derived* 类与 *cls* 类相同或为其派生类，则返回 1，否则返回 0。如果出错则返回 -1。

如果 *cls* 是元组，则会对 *cls* 进行逐项检测。如果至少有一次检测返回 1，结果将为 1，否则将是 0。

正如 **PEP 3119** 所述，如果 *cls* 带有 `__subclasscheck__()` 方法，将会被调用以确定子类的状态。否则，如果 *derived* 是个直接或间接子类，即包含在 `cls.__mro__` 中，那么它就是 *cls* 的一个子类。

通常只有类对象（即 `type` 或派生类的实例）才被视为类。但是，对象可以通过设置 `__bases__` 属性（必须是基类的元组）来覆盖这一点。

**int PyObject\_IsInstance** (*PyObject* \*inst, *PyObject* \*cls)

属于稳定 ABI。如果 *inst* 是 *cls* 类或其子类的实例，则返回 1，如果不是则返回 0。如果出错则返回 -1 并设置一个异常。

如果 *cls* 是元组，则会对 *cls* 进行逐项检测。如果至少有一次检测返回 1，结果将为 1，否则将是 0。

正如 PEP 3119 所述，如果 *cls* 带有 `__subclasscheck__()` 方法，将会被调用以确定子类的状态。否则，如果 *derived* 是 *cls* 的子类，那么它就是 *cls* 的一个实例。

实例 *inst* 可以通过 `__class__` 属性来覆盖其所属的类。

对象 *cls* 可以通过设置 `__bases__` 属性（该属性必须是基类的元组）来覆盖其是否会被视为类，及其有哪些基类。

**Py\_hash\_t PyObject\_Hash** (*PyObject* \*o)

属于稳定 ABI。计算并返回对象的哈希值 *o*。失败时返回 -1。这相当于 Python 表达式 `hash(o)`。

在 3.2 版的變更：现在的返回类型是 `Py_hash_t`。这是一个大小与 `Py_ssize_t` 相同的有符号整数。

**Py\_hash\_t PyObject\_HashNotImplemented** (*PyObject* \*o)

属于稳定 ABI。设置一个 `TypeError` 来指明 `type(o)` 不是 *hashable* 并返回 -1。此函数在存储于 `tp_hash` 槽位内时会获得特别对待，允许某个类型显式地向解释器指明它是不可哈希对象。

**int PyObject\_IsTrue** (*PyObject* \*o)

属于稳定 ABI。如果对象 *o* 被认为是 `true`，则返回 1，否则返回 0。这相当于 Python 表达式 `not not o`。失败则返回 -1。

**int PyObject\_Not** (*PyObject* \*o)

属于稳定 ABI。如果对象 *o* 被认为是 `true`，则返回 1，否则返回 0。这相当于 Python 表达式 `not not o`。失败则返回 -1。

**PyObject\* PyObject\_Type** (*PyObject* \*o)

回傳值：新的參照。属于稳定 ABI。当 *o* 不为 NULL 时，返回一个与对象 *o* 的类型相对应的类型对象。当失败时，将引发 `SystemError` 并返回 NULL。这等同于 Python 表达式 `type(o)`。该函数会新建一个指向返回值的 *strong reference*。实际上没有多少理由使用此函数来替代 `Py_TYPE()` 函数，后者将返回一个 `PyTypeObject*` 类型的指针，除非是需要一个新的 *strong reference*。

**int PyObject\_TypeCheck** (*PyObject* \*o, *PyTypeObject* \*type)

如果对象 *o* 是 *type* 类型或其子类型，则返回非零，否则返回 0。两个参数都必须非 NULL。

**Py\_ssize\_t PyObject\_Size** (*PyObject* \*o)

**Py\_ssize\_t PyObject\_Length** (*PyObject* \*o)

属于稳定 ABI。返回对象 *o* 的长度。如果对象 *o* 支持序列和映射协议，则返回序列长度。出错时返回 -1。这等同于 Python 表达式 `len(o)`。

**Py\_ssize\_t PyObject\_LengthHint** (*PyObject* \*o, *Py\_ssize\_t* defaultvalue)

返回对象 *o* 的估计长度。首先尝试返回实际长度，然后用 `__length_hint__()` 进行估计，最后返回默认值。出错时返回 -1。这等同于 Python 表达式 `operator.length_hint(o, defaultvalue)`。

在 3.4 版新加入。

**PyObject\* PyObject\_GetItem** (*PyObject* \*o, *PyObject* \*key)

回傳值：新的參照。属于稳定 ABI。返回对象 *key* 对应的 *o* 元素，或在失败时返回 NULL。这等同于 Python 表达式 `o[key]`。

**int PyObject\_SetItem** (*PyObject* \*o, *PyObject* \*key, *PyObject* \*v)

属于稳定 ABI。将对象 *key* 映射到值 *v*。失败时引发异常并返回 -1；成功时返回 0。这相当于 Python 语句 `o[key] = v`。该函数不会偷取 *v* 的引用计数。

**int PyObject\_DelItem** (*PyObject* \*o, *PyObject* \*key)

属于稳定 ABI。从对象 *o* 中移除对象 *key* 的映射。失败时返回 -1。这相当于 Python 语句 `del o[key]`。

*PyObject* \*PyObject\_Dir(PyObject \*o)

回傳值：新的參照。屬於穩定 ABI。相當於 Python 表达式 `dir(o)`，返回一个（可能为空）适合对象参数的字符串列表，如果出错则返回 `NULL`。如果参数为 `NULL`，类似 Python 的 `dir()`，则返回当前 `locals` 的名字；这时如果没有活动的执行框架，则返回 `NULL`，但 `PyErr_Occurred()` 将返回 `false`。

*PyObject* \*PyObject\_GetIter(PyObject \*o)

回傳值：新的參照。屬於穩定 ABI。等同於 Python 表达式 `iter(o)`。为对象参数返回一个新的迭代器，如果该对象已经是一个迭代器，则返回对象本身。如果对象不能被迭代，会引发 `TypeError`，并返回 `NULL`。

*PyObject* \*PyObject\_GetAIter(PyObject \*o)

回傳值：新的參照。屬於穩定 ABI 自 3.10 版起。等同於 Python 表达式 `aiter(o)`。接受一个 `AsyncIterable` 对象，并为其返回一个 `AsyncIterator`。通常返回的是一个新迭代器，但如果参数是一个 `AsyncIterator`，将返回其自身。如果该对象不能被迭代，会引发 `TypeError`，并返回 `NULL`。

在 3.10 版新加入。

## 7.2 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）。

### 7.2.1 `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` 必須不為 `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。

### 7.2.2 `Vectorcall` 協定

在 3.9 版新加入。

`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 於內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` 設定為 `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`。

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為 `args tuple`（引數元組）和 `kwargs dict`（關鍵字引數字典），那麼實作 `vectorcall` 就沒有意義。

类可以通过启用 `Py_TPFLAGS_HAVE_VECTORCALL` 旗标并将 `tp_vectorcall_offset` 设为对象结构中 `vectorcallfunc` 出现位置偏移量来实现 `vectorcall` 协议。这是一个指向具有以下签名的函数的指针：

```
typedef PyObject *(*vectorcallfunc)(PyObject *callable, PyObject *const *args, size_t nargsf, PyObject *kwnames)
```

- *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。如果沒有引數，這個值可以是 *NULL*。
- *nargsf* 是位置引數的數量加上可能會有 *PY\_VECTORCALL\_ARGUMENTS\_OFFSET* 旗標。要从 *nargsf* 获得位置参数的实际数量，请使用 *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)，且它們必須是不重複的。如果沒有關鍵字引數，那 *kwnames* 可以用 *NULL* 代替。

#### PY\_VECTORCALL\_ARGUMENTS\_OFFSET

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。

只要调用方能以低代价 (不额外分配内存) 这样做，就推荐使用 *PY\_VECTORCALL\_ARGUMENTS\_OFFSET*。这样做将允许诸如绑定方法之类的可调用对象非常高效地执行前向调用 (这种调用将包括一个加在开头的 *self* 参数)。

在 3.8 版新加入。

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。

---

**備註：** 在 CPython 3.8 中，*vectorcall* API 和相關函式暫定以帶開頭底 *\_* 的名稱提供：*\_PyObject\_Vectorcall*、*\_Py\_TPFLAGS\_HAVE\_VECTORCALL*、*\_PyObject\_VectorcallMethod*、*\_PyVectorcall\_Function*、*\_PyObject\_CallOneArg*、*\_PyObject\_CallMethodNoArgs*、*\_PyObject\_CallMethodOneArg*。此外，*PyObject\_VectorcallDict* 也以 *\_PyObject\_FastCallDict* 名稱提供。這些舊名稱仍有被定義，做不帶底的新名稱的。  

---

#### 遞歸控制

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)

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 版新加入。

*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 版新加入。

*PyObject* \***PyVectorcall\_Call** (*PyObject* \*callable, *PyObject* \*tuple, *PyObject* \*dict)

呼叫 *callable* 的 *vectorcallfunc*，其位置引數和關鍵字引數分別以 `tuple` 和 `dict` 格式給定。

這是一個專用函數，用於放入 `tp_call` 槽位或是用於 `tp_call` 的實現。它不會檢查 `Py_TPFLAGS_HAVE_VECTORCALL` 旗標並且它也不會回退到 `tp_call`。

在 3.8 版新加入。

## 7.2.3 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成被呼叫物件所支援的慣用形式—可以是 `tp_call` 或 `vectorcall`。為了可能因少轉的進行，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；請參閱各個說明文件以瞭解詳情。

函式	callable	args	kwargs
<code>PyObject_Call()</code>	<code>PyObject *</code>	<code>tuple</code>	<code>dict/NULL</code>
<code>PyObject_CallNoArgs()</code>	<code>PyObject *</code>	---	---
<code>PyObject_CallOneArg()</code>	<code>PyObject *</code>	一個物件	---
<code>PyObject_CallObject()</code>	<code>PyObject *</code>	<code>tuple/NULL</code>	---
<code>PyObject_CallFunction()</code>	<code>PyObject *</code>	<code>format</code>	---
<code>PyObject_CallMethod()</code>	物件 + <code>char*</code>	<code>format</code>	---
<code>PyObject_CallFunctionObjArgs()</code>	<code>PyObject *</code>	可變引數	---
<code>PyObject_CallMethodObjArgs()</code>	物件 + 名稱	可變引數	---
<code>PyObject_CallMethodNoArgs()</code>	物件 + 名稱	---	---
<code>PyObject_CallMethodOneArg()</code>	物件 + 名稱	一個物件	---
<code>PyObject_Vectorcall()</code>	<code>PyObject *</code>	<code>vectorcall</code>	<code>vectorcall</code>
<code>PyObject_VectorcallDict()</code>	<code>PyObject *</code>	<code>vectorcall</code>	<code>dict/NULL</code>
<code>PyObject_VectorcallMethod()</code>	引數 + 名稱	<code>vectorcall</code>	<code>vectorcall</code>

*PyObject* \***PyObject\_Call** (*PyObject* \*callable, *PyObject* \*args, *PyObject* \*kwargs)

回傳值：新的參照。屬於穩定 ABI。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* 必須不能 `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並回傳 `NULL`。

這等價於 Python 運算式 `callable(*args, **kwargs)`。

*PyObject* \***PyObject\_CallNoArgs** (*PyObject* \*callable)

回傳值：新的參照。屬於穩定 ABI 自 3.10 版起。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並回傳 `NULL`。

在 3.9 版新加入。



*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* 而沒有關鍵字引數。

成功時回傳結果，或在失敗時引發一個例外回傳 *NULL*。

在 3.9 版新加入。

*PyObject* \***PyObject\_CallObject** (*PyObject* \*callable, *PyObject* \*args)

回傳值：新的參照。屬於穩定 ABI。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* 可以為 *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。屬於穩定 ABI。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。格式可以為 *NULL*，表示沒有提供任何引數。

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。屬於穩定 ABI。呼叫 *obj* 物件中名為 *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，應生成一個 tuple。

格式可以為 *NULL*，表示沒有提供任何引數。

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。屬於穩定 ABI。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。屬於穩定 ABI。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 版新加入。

*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 版新加入。

*PyObject* \***PyObject\_Vectorcall** (*PyObject* \*callable, *PyObject* \*const \*args, size\_t nargsf, *PyObject* \*kwnames)

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 版新加入。

*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。

無論何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 版新加入。

*PyObject* \***PyObject\_VectorcallMethod** (*PyObject* \*name, *PyObject* \*const \*args, size\_t nargsf, *PyObject* \*kwnames)

使用 *vectorcall* 调用惯例来调用一个方法。方法的名称以 Python 字符串 *name* 的形式给出。调用方法的对象为 *args*[0]，而 *args* 数组从 *args*[1] 开始的部分则代表调用的参数。必须传入至少一个位置参数。*nargsf* 为包括 *args*[0] 在内的位置参数的数量，如果 *args*[0] 的值可能被临时改变则还要加上 *PY\_VECTORCALL\_ARGUMENTS\_OFFSET*。关键字参数可以像在 *PyObject\_Vectorcall()* 中那样传入。

如果对象具有 *Py\_TPFLAGS\_METHOD\_DESCRIPTOR* 特性，此函数将调用未绑定的方法对象并传入完整的 *args* vector 作为参数。

成功時回傳結果，或在失敗時引發一個例外回傳 *NULL*。

在 3.9 版新加入。

## 7.2.4 呼叫支援 API

int **PyCallable\_Check** (*PyObject* \*o)

属于稳定 ABI。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，其他情況回傳 0。這個函式不會呼叫失敗。

## 7.3 数字协议

int **PyNumber\_Check** (*PyObject* \*o)

属于稳定 ABI。如果对象 *o* 提供数字的协议，返回真 1，否则返回假。这个函数不会调用失败。

在 3.8 版的變更：如果 *o* 是一个索引整数则返回 1。

*PyObject* \***PyNumber\_Add** (*PyObject* \*o1, *PyObject* \*o2)

回傳值：新的參照。属于稳定 ABI。返回 *o1*、*o2* 相加的结果，如果失败，返回 *NULL*。等价于 Python 表达式 *o1* + *o2*。

*PyObject* \*PyNumber\_Subtract (*PyObject* \*o1, *PyObject* \*o2)

回傳值：新的參照。屬於穩定 ABI。返回 *o1* 減去 *o2* 的結果，如果失敗，返回 NULL。等價於 Python 表达式 *o1* - *o2*。

*PyObject* \*PyNumber\_Multiply (*PyObject* \*o1, *PyObject* \*o2)

回傳值：新的參照。屬於穩定 ABI。返回 *o1*、*o2* 相乘的結果，如果失敗，返回 NULL。等價於 Python 表达式 *o1* \* *o2*。

*PyObject* \*PyNumber\_MatrixMultiply (*PyObject* \*o1, *PyObject* \*o2)

回傳值：新的參照。屬於穩定 ABI 自 3.7 版起。返回 *o1*、*o2* 做矩陣乘法的結果，如果失敗，返回 NULL。等價於 Python 表达式 *o1* @ *o2*。

在 3.5 版新加入。

*PyObject* \*PyNumber\_FloorDivide (*PyObject* \*o1, *PyObject* \*o2)

回傳值：新的參照。屬於穩定 ABI。返回 *o1* 除以 *o2* 向下取整的值，失敗時返回 NULL。這等價於 Python 表达式 *o1* // *o2*。

*PyObject* \*PyNumber\_TrueDivide (*PyObject* \*o1, *PyObject* \*o2)

回傳值：新的參照。屬於穩定 ABI。返回 *o1* 除以 *o2* 的數學值的合理近似值，或失敗時返回 NULL。返回的是“近似值”因為二進制浮點數本身就是近似值；不可能以二進制精確表示所有實數。此函數可以在傳入兩個整數時返回一個浮點值。此函數等價於 Python 表达式 *o1* / *o2*。

*PyObject* \*PyNumber\_Remainder (*PyObject* \*o1, *PyObject* \*o2)

回傳值：新的參照。屬於穩定 ABI。返回 *o1* 除以 *o2* 得到的余數，如果失敗，返回 NULL。等價於 Python 表达式 *o1* % *o2*。

*PyObject* \*PyNumber\_Divmod (*PyObject* \*o1, *PyObject* \*o2)

回傳值：新的參照。屬於穩定 ABI。參考內置函數 divmod()。如果失敗，返回 NULL。等價於 Python 表达式 divmod(*o1*, *o2*)。

*PyObject* \*PyNumber\_Power (*PyObject* \*o1, *PyObject* \*o2, *PyObject* \*o3)

回傳值：新的參照。屬於穩定 ABI。請參閱內置函數 pow()。如果失敗，返回 NULL。等價於 Python 中的表达式 pow(*o1*, *o2*, *o3*)，其中 *o3* 是可選的。如果要忽略 *o3*，則需傳入 *Py\_None* 作為代替（如果傳入 NULL 會導致非法內存訪問）。

*PyObject* \*PyNumber\_Negative (*PyObject* \*o)

回傳值：新的參照。屬於穩定 ABI。返回 *o* 的負值，如果失敗，返回 NULL。等價於 Python 表达式 -*o*。

*PyObject* \*PyNumber\_Positive (*PyObject* \*o)

回傳值：新的參照。屬於穩定 ABI。返回 *o*，如果失敗，返回 NULL。等價於 Python 表达式 +*o*。

*PyObject* \*PyNumber\_Absolute (*PyObject* \*o)

回傳值：新的參照。屬於穩定 ABI。返回 *o* 的絕對值，如果失敗，返回 NULL。等價於 Python 表达式 abs(*o*)。

*PyObject* \*PyNumber\_Invert (*PyObject* \*o)

回傳值：新的參照。屬於穩定 ABI。返回 *o* 的按位取反後的結果，如果失敗，返回 NULL。等價於 Python 表达式 ~*o*。

*PyObject* \*PyNumber\_Lshift (*PyObject* \*o1, *PyObject* \*o2)

回傳值：新的參照。屬於穩定 ABI。返回 *o1* 左移 *o2* 個比特後的結果，如果失敗，返回 NULL。等價於 Python 表达式 *o1* << *o2*。

*PyObject* \*PyNumber\_Rshift (*PyObject* \*o1, *PyObject* \*o2)

回傳值：新的參照。屬於穩定 ABI。返回 *o1* 右移 *o2* 個比特後的結果，如果失敗，返回 NULL。等價於 Python 表达式 *o1* >> *o2*。

*PyObject* \*PyNumber\_And (*PyObject* \*o1, *PyObject* \*o2)

回傳值：新的參照。屬於穩定 ABI。返回 *o1* 和 *o2* “按位與”的結果，如果失敗，返回 NULL。等價於 Python 表达式 *o1* & *o2*。



**PyObject \*PyNumber\_Xor (PyObject \*o1, PyObject \*o2)**

回傳值：新的參照。属于穩定 ABI。返回 *o1* 和 *o2* “按位异或”的结果，如果失败，返回 NULL。等价于 Python 表达式 *o1* ^ *o2*。

**PyObject \*PyNumber\_Or (PyObject \*o1, PyObject \*o2)**

回傳值：新的參照。属于穩定 ABI。返回 *o1* 和 *o2* “按位或”的结果，如果失败，返回 NULL。等价于 Python 表达式 *o1* | *o2*。

**PyObject \*PyNumber\_InPlaceAdd (PyObject \*o1, PyObject \*o2)**

回傳值：新的參照。属于穩定 ABI。返回 *o1*、*o2* 相加的结果，如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 *o1* += *o2*。

**PyObject \*PyNumber\_InPlaceSubtract (PyObject \*o1, PyObject \*o2)**

回傳值：新的參照。属于穩定 ABI。返回 *o1*、*o2* 相减的结果，如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 *o1* -= *o2*。

**PyObject \*PyNumber\_InPlaceMultiply (PyObject \*o1, PyObject \*o2)**

回傳值：新的參照。属于穩定 ABI。返回 *o1*、*o2*\* 相乘的结果，如果失败，返回 “NULL”。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 *o1* \*= *o2*。

**PyObject \*PyNumber\_InPlaceMatrixMultiply (PyObject \*o1, PyObject \*o2)**

回傳值：新的參照。属于穩定 ABI 自 3.7 版起。返回 *o1*、*o2* 做矩阵乘法后的结果，如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 *o1* @= *o2*。

在 3.5 版新加入。

**PyObject \*PyNumber\_InPlaceFloorDivide (PyObject \*o1, PyObject \*o2)**

回傳值：新的參照。属于穩定 ABI。返回 *o1* 除以 *o2* 后向下取整的结果，如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 *o1* //= *o2*。

**PyObject \*PyNumber\_InPlaceTrueDivide (PyObject \*o1, PyObject \*o2)**

回傳值：新的參照。属于穩定 ABI。返回 *o1* 除以 *o2* 的数学值的合理近似值，或失败时返回 NULL。返回的是“近似值”因为二进制浮点数本身就是近似值；不可能以二进制精确表示所有实数。此函数可以在传入两个整数时返回一个浮点数。此运算在 *o1* 支持的时候会 原地执行。此函数等价于 Python 语句 *o1* /= *o2*。

**PyObject \*PyNumber\_InPlaceRemainder (PyObject \*o1, PyObject \*o2)**

回傳值：新的參照。属于穩定 ABI。返回 *o1* 除以 *o2* 得到的余数，如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 *o1* %= *o2*。

**PyObject \*PyNumber\_InPlacePower (PyObject \*o1, PyObject \*o2, PyObject \*o3)**

回傳值：新的參照。属于穩定 ABI。请参阅内置函数 pow()。如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。当 *o3* 是 *Py\_None* 时，等价于 Python 语句 *o1* \*\*= *o2*；否则等价于在原来位置储存结果的 pow(*o1*, *o2*, *o3*)。如果要忽略 *o3*，则需传入 *Py\_None*（传入 NULL 会导致非法内存访问）。

**PyObject \*PyNumber\_InPlaceLshift (PyObject \*o1, PyObject \*o2)**

回傳值：新的參照。属于穩定 ABI。返回 *o1* 左移 *o2* 个比特后的结果，如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 *o1* <<= *o2*。

**PyObject \*PyNumber\_InPlaceRshift (PyObject \*o1, PyObject \*o2)**

回傳值：新的參照。属于穩定 ABI。返回 *o1* 右移 *o2* 个比特后的结果，如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 *o1* >>= *o2*。

**PyObject \*PyNumber\_InPlaceAnd (PyObject \*o1, PyObject \*o2)**

回傳值：新的參照。属于穩定 ABI。成功时返回 *o1* 和 *o2* “按位与”的结果，失败时返回 NULL。在 *o1* 支持的前提下该操作将 原地执行。等价于 Python 语句 *o1* &= *o2*。

**PyObject \*PyNumber\_InPlaceXor (PyObject \*o1, PyObject \*o2)**

回傳值：新的參照。属于穩定 ABI。成功时返回 *o1* 和 *o2* “按位异或”的结果，失败时返回 NULL。在 *o1* 支持的前提下该操作将 原地执行。等价于 Python 语句 *o1* ^= *o2*。

*PyObject* \*PyNumber\_InPlaceOr (*PyObject* \*o1, *PyObject* \*o2)

回傳值：新的參照。屬於穩定 ABI。成功時返回 *o1* 和 *o2* “按位或” 的結果，失敗時返回 NULL。在 *o1* 支持的前提下該操作將 原地執行。等價於 Python 語句 *o1* |= *o2*。

*PyObject* \*PyNumber\_Long (*PyObject* \*o)

回傳值：新的參照。屬於穩定 ABI。成功時返回 *o* 轉換為整數對象後的結果，失敗時返回 NULL。等價於 Python 表达式 `int(o)`。

*PyObject* \*PyNumber\_Float (*PyObject* \*o)

回傳值：新的參照。屬於穩定 ABI。成功時返回 *o* 轉換為浮點對象後的結果，失敗時返回 NULL。等價於 Python 表达式 `float(o)`。

*PyObject* \*PyNumber\_Index (*PyObject* \*o)

回傳值：新的參照。屬於穩定 ABI。成功時返回 *o* 轉換為 Python `int` 類型後的結果，失敗時返回 NULL 並引發 `TypeError` 異常。

在 3.10 版的變更：結果總是為 `int` 類型。在之前版本中，結果可能為 `int` 的子類的實例。

*PyObject* \*PyNumber\_ToBase (*PyObject* \*n, int base)

回傳值：新的參照。屬於穩定 ABI。返回整數 *n* 轉換成以 *base* 為基數的字符串後的結果。這個 *base* 參數必須是 2, 8, 10 或者 16。對於基數 2, 8, 或 16，返回的字符串將分別加上基數標識 `'0b'`, `'0o'`, 或 `'0x'`。如果 *n* 不是 Python 中的整數 `int` 類型，就先通過 `PyNumber_Index()` 將它轉換成整數類型。

*Py\_ssize\_t* PyNumber\_AsSsize\_t (*PyObject* \*o, *PyObject* \*exc)

屬於穩定 ABI。如果 *o* 可以被解讀為一個整數則返回 *o* 轉換成的 *Py\_ssize\_t* 值。如果調用失敗，則會引發一個異常並返回 -1。

如果 *o* 可以被轉換為 Python 的 `int` 值但嘗試轉換為 *Py\_ssize\_t* 值則會引發 `OverflowError`，則 *exc* 參數將為所引發的異常類型（通常為 `IndexError` 或 `OverflowError`）。如果 *exc* 為 NULL，則異常會被清除並且值會在為負整數時被裁剪為 `PY_SSIZE_T_MIN` 而在為正整數時被裁剪為 `PY_SSIZE_T_MAX`。

int PyIndex\_Check (*PyObject* \*o)

屬於穩定 ABI 自 3.8 版起。返回 1 如果 *o* 是一個索引整數（將 `nb_index` 槽位填充到 `tp_as_number` 結構體），或者在其他情況下返回 0。此函數總是會成功執行。

## 7.4 序列協議

int PySequence\_Check (*PyObject* \*o)

屬於穩定 ABI。如果對象提供了序列協議則返回 1，否則返回 0。請注意對於具有 `__getitem__()` 方法的 Python 類返回 1，除非它們是 `dict` 的子類，因為在通常情況下無法確定這種類支持哪種鍵類型。該函數總是會成功執行。

*Py\_ssize\_t* PySequence\_Size (*PyObject* \*o)

*Py\_ssize\_t* PySequence\_Length (*PyObject* \*o)

屬於穩定 ABI。成功時返回序列中 \**o*\* 的對象數，失敗時返回 -1。相當於 Python 的 `len(o)` 表达式。

*PyObject* \*PySequence\_Concat (*PyObject* \*o1, *PyObject* \*o2)

回傳值：新的參照。屬於穩定 ABI。成功時返回 *o1* 和 *o2* 的拼接，失敗時返回 NULL。這等價於 Python 表达式 `o1 + o2`。

*PyObject* \*PySequence\_Repeat (*PyObject* \*o, *Py\_ssize\_t* count)

回傳值：新的參照。屬於穩定 ABI。返回序列對象 *o* 重複 *count* 次的結果，失敗時返回 NULL。這等價於 Python 表达式 `o * count`。

*PyObject* \*PySequence\_InPlaceConcat (*PyObject* \*o1, *PyObject* \*o2)

回傳值：新的參照。屬於穩定 ABI。成功時返回 *o1* 和 *o2* 的拼接，失敗時返回 NULL。在 *o1* 支持的情況下操作將 原地完成。這等價於 Python 表达式 `o1 += o2`。

*PyObject* \*PySequence\_InPlaceRepeat (*PyObject* \*o, *Py\_ssize\_t* count)

回傳值：新的參照。属于穩定 ABI。Return the result of repeating sequence object 返回序列对象 *o* 重复 *count* 次的结果，失败时返回 NULL。在 *o* 支持的情况下该操作会 原地完成。这等价于 Python 表达式 *o* \*= *count*。

*PyObject* \*PySequence\_GetItem (*PyObject* \*o, *Py\_ssize\_t* i)

回傳值：新的參照。属于穩定 ABI。返回 *o* 中的第 *i* 号元素，失败时返回 NULL。这等价于 Python 表达式 *o*[*i*]。

*PyObject* \*PySequence\_GetSlice (*PyObject* \*o, *Py\_ssize\_t* i1, *Py\_ssize\_t* i2)

回傳值：新的參照。属于穩定 ABI。返回序列对象 *o* 的 *i1* 到 *i2* 的切片，失败时返回 NULL。这等价于 Python 表达式 *o*[*i1*:*i2*]。

int PySequence\_SetItem (*PyObject* \*o, *Py\_ssize\_t* i, *PyObject* \*v)

属于穩定 ABI。将对象 *v* 赋值给 *o* 的第 *i* 号元素。失败时会引发异常并返回 -1；成功时返回 0。这相当于 Python 语句 *o*[*i*] = *v*。此函数 不会改变对 *v* 的引用。

如果 *v* 为 NULL，元素将被删除，但是此特性已被弃用而应改用 *PySequence\_DelItem*()。

int PySequence\_DelItem (*PyObject* \*o, *Py\_ssize\_t* i)

属于穩定 ABI。删除对象 *o* 的第 *i* 号元素。失败时返回 -1。这相当于 Python 语句 *del o*[*i*]。

int PySequence\_SetSlice (*PyObject* \*o, *Py\_ssize\_t* i1, *Py\_ssize\_t* i2, *PyObject* \*v)

属于穩定 ABI。将序列对象 *v* 赋值给序列对象 *o* 的从 *i1* 到 *i2* 切片。这相当于 Python 语句 *o*[*i1*:*i2*] = *v*。

int PySequence\_DelSlice (*PyObject* \*o, *Py\_ssize\_t* i1, *Py\_ssize\_t* i2)

属于穩定 ABI。删除序列对象 *o* 的从 *i1* 到 *i2* 的切片。失败时返回 -1。这相当于 Python 语句 *del o*[*i1*:*i2*]。

*Py\_ssize\_t* PySequence\_Count (*PyObject* \*o, *PyObject* \*value)

属于穩定 ABI。返回 *value* 在 *o* 中出现的次数，即返回使得 *o*[*key*] == *value* 的键的数量。失败时返回 -1。这相当于 Python 表达式 *o*.count(*value*)。

int PySequence\_Contains (*PyObject* \*o, *PyObject* \*value)

属于穩定 ABI。确定 *o* 是否包含 *value*。如果 *o* 中的某一项等于 *value*，则返回 1，否则返回 0。出错时，返回 -1。这相当于 Python 表达式 *value* in *o*。

*Py\_ssize\_t* PySequence\_Index (*PyObject* \*o, *PyObject* \*value)

属于穩定 ABI。返回第一个索引 \**i*，其中 *o*[*i*] == *value*。出错时，返回 -1。相当于 Python 的 *o*.index(*value*) 表达式。

*PyObject* \*PySequence\_List (*PyObject* \*o)

回傳值：新的參照。属于穩定 ABI。返回一个列表对象，其内容与序列或可迭代对象 *o* 相同，失败时返回 NULL。返回的列表保证是一个新对象。这等价于 Python 表达式 list(*o*)。

*PyObject* \*PySequence\_Tuple (*PyObject* \*o)

回傳值：新的參照。属于穩定 ABI。返回一个元组对象，其内容与序列或可迭代对象 *o* 相同，失败时返回 NULL。如果 *o* 为元组，则将返回一个新的引用，在其他情况下将使用适当的内容构造一个元组。这等价于 Python 表达式 tuple(*o*)。

*PyObject* \*PySequence\_Fast (*PyObject* \*o, const char \*m)

回傳值：新的參照。属于穩定 ABI。将序列或可迭代对象 *o* 作为其他 *PySequence\_Fast\** 函数族可用的对象返回。如果该对象不是序列或可迭代对象，则会引发 *TypeError* 并将 *m* 作为消息文本。失败时返回 NULL。

*PySequence\_Fast\** 函数之所以这样命名，是因为它们会假定 *o* 是一个 *PyTupleObject* 或 *PyListObject* 并直接访问 *o* 的数据字段。

作为 CPython 的实现细节，如果 *o* 已经是一个序列或列表，它将被直接返回。

*Py\_ssize\_t* **PySequence\_Fast\_GET\_SIZE** (*PyObject* \*o)

在 *o* 由 *PySequence\_Fast()* 返回且 *o* 不为 NULL 的情况下返回 *o* 长度。也可以通过在 *o* 上调用 *PySequence\_Size()* 来获取大小，但是 *PySequence\_Fast\_GET\_SIZE()* 的速度更快因为它可以假定 *o* 为列表或元组。

*PyObject* \***PySequence\_Fast\_GET\_ITEM** (*PyObject* \*o, *Py\_ssize\_t* i)

回傳值：借用參照。在 *o* 由 *PySequence\_Fast()* 返回且 *o* 不为 NULL，并且 *i* 在索引范围内的情况下返回 *o* 的第 *i* 号元素。

*PyObject* \*\***PySequence\_Fast\_ITEMS** (*PyObject* \*o)

返回 *PyObject* 指针的底层数组。假设 *o* 由 *PySequence\_Fast()* 返回且 *o* 不为 NULL。

请注意，如果列表调整大小，重新分配可能会重新定位 *items* 数组。因此，仅在序列无法更改的上下文中使用基础数组指针。

*PyObject* \***PySequence\_ITEM** (*PyObject* \*o, *Py\_ssize\_t* i)

回傳值：新的參照。返回 *o* 的第 *i* 个元素或在失败时返回 NULL。此形式比 *PySequence\_GetItem()* 理饌，但不会检查 *o* 上的 *PySequence\_Check()* 是否为真值，也不会对负序号进行调整。

## 7.5 映射协议

参见 *PyObject\_GetItem()*、*PyObject\_SetItem()* 与 *PyObject\_DelItem()*。

int **PyMapping\_Check** (*PyObject* \*o)

属于稳定 ABI。如果对象提供了映射协议或是支持切片则返回 1，否则返回 0。请注意它将为具有 *\_\_getitem\_\_()* 方法的 Python 类返回 1，因为在通常情况下无法确定该类支持哪种键类型。此函数总是会成功执行。

*Py\_ssize\_t* **PyMapping\_Size** (*PyObject* \*o)

*Py\_ssize\_t* **PyMapping\_Length** (*PyObject* \*o)

属于稳定 ABI。成功时返回对象 *o* 中键的数量，失败时返回 -1。这相当于 Python 表达式 *len(o)*。

*PyObject* \***PyMapping\_GetItemString** (*PyObject* \*o, const char \*key)

回傳值：新的參照。属于稳定 ABI。这与 *PyObject\_GetItem()* 相同，但 *key* 被指定为 const char\* UTF-8 编码的字节串，而不是 *PyObject\**。

int **PyMapping\_SetItemString** (*PyObject* \*o, const char \*key, *PyObject* \*v)

属于稳定 ABI。这与 *PyObject\_SetItem()* 相同，但 *key* 被指定为 const char\* UTF-8 编码的字节串，而不是 *PyObject\**。

int **PyMapping\_DelItem** (*PyObject* \*o, *PyObject* \*key)

这是 *PyObject\_DelItem()* 的一个别名。

int **PyMapping\_DelItemString** (*PyObject* \*o, const char \*key)

这与 *PyObject\_DelItem()* 相同，但 *key* 被指定为 const char\* UTF-8 编码的字节串，而不是 *PyObject\**。

int **PyMapping\_HasKey** (*PyObject* \*o, *PyObject* \*key)

属于稳定 ABI。如果映射对象具有键 *key* 则返回 1，否则返回 0。这相当于 Python 表达式 *key in o*。此函数总是会成功执行。

---

**備註：** 在调用 *\_\_getitem\_\_()* 方法时发生的异常将被静默地忽略。想要进行适当的错误处理，请改用 *PyObject\_GetItem()*。

---

int **PyMapping\_HasKeyString** (*PyObject* \*o, const char \*key)

属于稳定 ABI。这与 *PyMapping\_HasKey()* 相同，但 *key* 被指定为 const char\* UTF-8 编码的字节串，而不是 *PyObject\**。



備 F: 在调用 `__getitem__()` 方法或创建临时 `str` 对象时发生的异常将被忽略。想要进行适当的错误处理, 请改用 `PyMapping_GetItemString()`。

**PyObject \*PyMapping\_Keys (PyObject \*o)**

回傳值: 新的參照。属于稳定 ABI。成功时, 返回对象 *o* 中的键的列表。失败时, 返回 NULL。

在 3.7 版的變更: 在之前版本中, 此函数返回一个列表或元组。

**PyObject \*PyMapping\_Values (PyObject \*o)**

回傳值: 新的參照。属于稳定 ABI。成功时, 返回对象 *o* 中的值的列表。失败时, 返回 NULL。

在 3.7 版的變更: 在之前版本中, 此函数返回一个列表或元组。

**PyObject \*PyMapping\_Items (PyObject \*o)**

回傳值: 新的參照。属于稳定 ABI。成功时, 返回对象 *o* 中条目的列表, 其中每个条目是一个包含键值对的元组。失败时, 返回 NULL。

在 3.7 版的變更: 在之前版本中, 此函数返回一个列表或元组。

## 7.6 F 代器協議

有兩個專門用於 F 代器的函式。

**int PyIter\_Check (PyObject \*o)**

属于稳定 ABI 自 3.8 版起。如果物件 *o* 可以安全地傳遞給 `PyIter_Next()` 則回傳非零 (non-zero), 否則回傳 0。這個函式一定會執行成功。

**int PyAsyncIter\_Check (PyObject \*o)**

属于稳定 ABI 自 3.10 版起。如果物件 *o* 有提供 `AsyncIterator` 協議, 則回傳非零, 否則回傳 0。這個函式一定會執行成功。

在 3.10 版新加入。

**PyObject \*PyIter\_Next (PyObject \*o)**

回傳值: 新的參照。属于稳定 ABI。回傳 F 代器 *o* 的下一個值。根據 `PyIter_Check()`, 該物件必須是一個 F 代器 (由呼叫者檢查)。如果 F 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 且傳遞例外。

要編寫一個 F 代於 F 代器的 F 圈, C 程式碼應該會像這樣:

```
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;

if (iterator == NULL) {
    /* propagate error */
}

while ((item = PyIter_Next(iterator))) {
    /* do something with item */
    ...
    /* release reference when done */
    Py_DECREF(item);
}

Py_DECREF(iterator);

if (PyErr_Occurred()) {
    /* propagate error */
}
```

(繼續下一頁)

```
else {
    /* continue doing useful work */
}
```

### type `PySendResult`

用於表示 `PyIter_Send()` 不同結果的列舉 (enum) 值。

在 3.10 版新加入。

`PySendResult PyIter_Send(PyObject *iter, PyObject *arg, PyObject **presult)`

屬於穩定 ABI 自 3.10 版起。將 `arg` 值發送到代器 `iter` 中。回傳：

- 如果代器有回傳則 `PySendResult` `PyIter_RETURN`。回傳值透過 `presult` 回傳。
- 如果代器有生成 (yield) 則 `PySendResult` `PyIter_NEXT`。生成值透過 `presult` 回傳。
- 如果代器引發例外則 `PySendResult` `PyIter_ERROR`。 `presult` 被設定為 `NULL`。

在 3.10 版新加入。

## 7.7 緩沖協定 (Buffer Protocol)

在 Python 中可使用一些對象來包裝對底層內存數組或稱緩沖的訪問。此類對象包括內置的 `bytes` 和 `bytearray` 以及一些如 `array.array` 這樣的擴展類型。第三方庫也可能会為了特殊的目的而定義它們自己的類型，例如用於圖像處理和數值分析等。

雖然這些類型中的每一種都有自己的語義，但它們具有由可能較大的內存緩沖區支持的特徵。在某些情況下，希望直接訪問該緩沖區而無需中間複製。

Python 以緩沖協議的形式在 C 層級上提供這樣的功能。此協議包括兩個方面：

- 在生產者這一方面，該類型的協議可以導出一個“緩沖區接口”，允許公開它的底層緩沖區信息。該接口的描述信息在緩沖區對象結構體一節中；
- 在消費者一側，有幾種方法可用於獲得指向對象的原始底層數據的指針（例如一個方法的形參）。

一些簡單的對象例如 `bytes` 和 `bytearray` 會以面向字節的形式公開它們的底層緩沖區。也可能会用其他形式；例如 `array.array` 所公開的元素可以是多字節值。

緩沖區接口的消費者的一個例子是文件對象的 `write()` 方法：任何可以輸出為一系列字節流的對象都可以被寫入文件。然而 `write()` 只需要對傳入對象內容的只讀權限，其他的方法如 `readinto()` 需要對參數內容的寫入權限。緩沖區接口使用對象可以選擇性地允許或拒絕讀寫或只讀緩沖區的導出。

對於緩沖區接口的使用者而言，有兩種方式來獲取一個目的對象的緩沖：

- 使用正確的參數來調用 `PyObject_GetBuffer()` 函數；
- 調用 `PyArg_ParseTuple()` (或其同級對象之一) 並傳入 `y*`, `w*` or `s*` 格式代碼中的一個。

在這兩種情況下，當不再需要緩沖區時必須調用 `PyBuffer_Release()`。如果此操作失敗，可能會導致各種問題，例如資源洩漏。

### 7.7.1 缓冲区结构

缓冲区结构 (或者简单地称为 “buffers”) 对于将二进制数据从另一个对象公开给 Python 程序员非常有用。它们还可以用作零拷贝机制。使用它们引用内存块的能力, 可以很容易地将任何数据公开给 Python 程序员。内存可以是 C 扩展中的一个大的常量数组, 也可以是在传递到操作系统库之前用于操作的原始内存块, 或者可以用来传递本机内存格式的结构化数据。

与 Python 解释器公开的大多数数据类型不同, 缓冲区不是 `PyObject` 指针而是简单的 C 结构。这使得它们可以非常简单地创建和复制。当需要为缓冲区加上泛型包装器时, 可以创建一个内存视图对象。

有关如何编写并导出对象的简短说明, 请参阅缓冲区对象结构。要获取缓冲区对象, 请参阅 `PyObject_GetBuffer()`。

type **Py\_buffer**

属于稳定 ABI (包括所有成员) 自 3.11 版起。

void \***buf**

指向由缓冲区字段描述的逻辑结构开始的指针。这可以是导出程序底层物理内存块中的任何位置。例如, 使用负的 `strides` 值可能指向内存块的末尾。

对于 *contiguous*, ‘邻接’ 数组, 值指向内存块的开头。

`PyObject` \***obj**

对导出对象的新引用。该引用由消费方拥有, 并由 `PyBuffer_Release()` 自动释放 (即引用计数递减) 并设置为 NULL。该字段相当于任何标准 C-API 函数的返回值。

作为一种特殊情况, 对于由 `PyMemoryView_FromBuffer()` 或 `PyBuffer_FillInfo()` 包装的 *temporary* 缓冲区, 此字段为 NULL。通常, 导出对象不得使用此方案。

`Py_ssize_t` **len**

`product(shape) * itemsize`。对于连续数组, 这是基础内存块的长度。对于非连续数组, 如果逻辑结构复制到连续表示形式, 则该长度将具有该长度。

仅当缓冲区是通过保证连续性的请求获取时, 才访问 `((char *)buf)[0]` up to `((char *)buf)[len-1]` 时才有效。在大多数情况下, 此类请求将为 `PyBUF_SIMPLE` 或 `PyBUF_WRITABLE`。

int **readonly**

缓冲区是否为只读的指示器。此字段由 `PyBUF_WRITABLE` 标志控制。

`Py_ssize_t` **itemsize**

单个元素的项大小 (以字节为单位)。与 `struct.calcsize()` 调用非 NULL *format* 的值相同。

重要例外: 如果使用者请求的缓冲区没有 `PyBUF_FORMAT` 标志, *format* 将设置为 NULL, 但 *itemsize* 仍具有原始格式的值。

如果 *shape* 存在, 则相等的 `product(shape) * itemsize == len` 仍然存在, 使用者可以使用 *itemsize* 来导航缓冲区。

如果 *shape* 是 NULL, 因为结果为 `PyBUF_SIMPLE` 或 `PyBUF_WRITABLE` 请求, 则使用者必须忽略 *itemsize*, 并假设 `itemsize == 1`。

const char \***format**

在 `struct` 模块样式语法中 *NUL* 字符串, 描述单个项的内容。如果这是 NULL, 则假定为 "B" (无符号字节)。

此字段由 `PyBUF_FORMAT` 标志控制。

int **ndim**

内存表示为 *n* 维数组形式对应的维度数。如果为 0, 则 *buf* 指向表示标量的单个条目。在这种情况下, *shape*, *strides* 和 *suboffsets* 必须为 NULL。最大维度数由 `PyBUF_MAX_NDIM` 给出。

**`Py_ssize_t *shape`**

一个长度为 `Py_ssize_t` 的数组 `ndim` 表示作为 `n` 维数组的内存形状。请注意, `shape[0] * ... * shape[ndim-1] * itemsize` 必须等于 `len`。

Shape 形状数组中的值被限定在 `shape[n] >= 0`。 `shape[n] == 0` 这一情形需要特别注意。更多信息请参阅 [complex arrays](#)。

`shape` 数组对于使用者来说是只读的。

**`Py_ssize_t *strides`**

一个长度为 `Py_ssize_t` 的数组 `ndim` 给出要跳过的字节数以获取每个尺寸中的新元素。

Stride 步幅数组中的值可以为任何整数。对于常规数组, 步幅通常为正数, 但是使用者必须能够处理 `strides[n] <= 0` 的情况。更多信息请参阅 [complex arrays](#)。

`strides` 数组对用户来说是只读的。

**`Py_ssize_t *suboffsets`**

一个长度为 `ndim` 类型为 `Py_ssize_t` 的数组。如果 `suboffsets[n] >= 0`, 则第 `n` 维存储的是指针, `suboffset` 值决定了解除引用时要给指针增加多少字节的偏移。`suboffset` 为负值, 则表示不应解除引用 (在连续内存块中移动)。

如果所有子偏移均为负 (即无需取消引用), 则此字段必须为 `NULL` (默认值)。

Python Imaging Library (PIL) 中使用了这种类型的数组表达方式。请参阅 [complex arrays](#) 来了解如何从这样一个数组中访问元素。

`suboffsets` 数组对于使用者来说是只读的。

**`void *internal`**

供输出对象内部使用。比如可能被输出程序重组为一个整数, 用于存储一个标志, 标明在缓冲区释放时是否必须释放 `shape`、`strides` 和 `suboffsets` 数组。消费者程序 不得修改该值。

常量:

**`PyBUF_MAX_NDIM`**

内存表示的最大维度数。导出程序必须遵守这个限制, 多维缓冲区的使用者应该能够处理最多 `PyBUF_MAX_NDIM` 个维度。目前设置为 64。

## 7.7.2 缓冲区请求的类型

通常, 通过 `PyObject_GetBuffer()` 向输出对象发送缓冲区请求, 即可获得缓冲区。由于内存的逻辑结构复杂, 可能会有很大差异, 缓冲区使用者可用 `flags` 参数指定其能够处理的缓冲区具体类型。

所有 `Py_buffer` 字段均由请求类型无歧义地定义。

### 与请求无关的字段

以下字段不会被 `flags` 影响, 并且必须总是用正确的值填充: `obj`, `buf`, `len`, `itemsize`, `ndim`。

### 只读, 格式

**`PyBUF_WRITABLE`**

控制 `readonly` 字段。如果设置了, 输出程序 必须提供一个可写的缓冲区, 否则报告失败。若未设置, 输出程序 可以提供只读或可写的缓冲区, 但对所有消费者程序 必须保持一致。

**`PyBUF_FORMAT`**

控制 `format` 字段。如果设置, 则必须正确填写此字段。其他情况下, 此字段必须为 `NULL`。



`PyBUF_WRITABLE` 可以和下一节的所有标志联用。由于 `PyBUF_SIMPLE` 定义为 0，所以 `PyBUF_WRITABLE` 可以作为一个独立的标志，用于请求一个简单的可写缓冲区。

`PyBUF_FORMAT` 可以被设为除了 `PyBUF_SIMPLE` 之外的任何标志。后者已经按暗示了 B (无符号字节串) 格式。

### 形状，步幅，子偏移量

控制内存逻辑结构的标志按照复杂度的递减顺序列出。注意，每个标志包含它下面的所有标志。

请求	形状	步幅	子偏移量
<code>PyBUF_INDIRECT</code>	是	是	如果需要的话
<code>PyBUF_STRIDES</code>	是	是	NULL
<code>PyBUF_ND</code>	是	NULL	NULL
<code>PyBUF_SIMPLE</code>	NULL	NULL	NULL

### 连续性的请求

可以显式地请求 C 或 Fortran 连续，不管有没有步幅信息。若没有步幅信息，则缓冲区必须是 C-连续的。

请求	形状	步幅	子偏移量	邻接
<code>PyBUF_C_CONTIGUOUS</code>	是	是	NULL	C
<code>PyBUF_F_CONTIGUOUS</code>	是	是	NULL	F
<code>PyBUF_ANY_CONTIGUOUS</code>	是	是	NULL	C 或 F
<code>PyBUF_ND</code>	是	NULL	NULL	C

### 复合请求

所有可能的请求都由上一节中某些标志的组合完全定义。为方便起见，缓冲区协议提供常用的组合作为单个标志。

在下表中，*U* 代表连续性未定义。消费者程序必须调用 `PyBuffer_IsContiguous()` 以确定连续性。

请求	形状	步幅	子偏移量	邻接	readonly	format
<code>PyBUF_FULL</code>	是	是	如果需要的话	U	0	是
<code>PyBUF_FULL_RO</code>	是	是	如果需要的话	U	1 或 0	是
<code>PyBUF_RECORDS</code>	是	是	NULL	U	0	是
<code>PyBUF_RECORDS_RO</code>	是	是	NULL	U	1 或 0	是
<code>PyBUF_STRIDED</code>	是	是	NULL	U	0	NULL
<code>PyBUF_STRIDED_RO</code>	是	是	NULL	U	1 或 0	NULL
<code>PyBUF_CONTIG</code>	是	NULL	NULL	C	0	NULL
<code>PyBUF_CONTIG_RO</code>	是	NULL	NULL	C	1 或 0	NULL

### 7.7.3 复杂数组

#### NumPy-风格：形状和步幅

NumPy 风格数组的逻辑结构由 `itemsize`、`ndim`、`shape` 和 `strides` 定义。

如果 `ndim == 0`，`buf` 指向的内存位置被解释为大小为 `itemsize` 的标量。这时，`shape` 和 `strides` 都为 NULL。

如果 `strides` 为 NULL，则数组将被解释为一个标准的 `n` 维 C 语言数组。否则，消费者程序必须按如下方式访问 `n` 维数组：

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

如上所述，`buf` 可以指向实际内存块中的任意位置。输出者程序可以用该函数检查缓冲区的有效性。

```
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
```

(繼續下一頁)

(繼續上一頁)

```

if ndim <= 0:
    return ndim == 0 and not shape and not strides
if 0 in shape:
    return True

imin = sum(strides[j]*(shape[j]-1) for j in range(ndim)
            if strides[j] <= 0)
imax = sum(strides[j]*(shape[j]-1) for j in range(ndim)
            if strides[j] > 0)

return 0 <= offset+imin and offset+imax+itemsizememlen

```

### PIL-风格：形状，步幅和子偏移量

除了常规项之外，PIL 风格的数组还可以包含指针，必须跟随这些指针才能到达维度的下一个元素。例如，常规的三维 C 语言数组 `char v[2][2][3]` 可以看作是一个指向 2 个二维数组的 2 个指针：`char (*v[2])[2][3]`。在子偏移表示中，这两个指针可以嵌入在 `buf` 的开头，指向两个可以位于内存任何位置的 `char x[2][3]` 数组。

这是一个函数，当 `n` 维索引所指向的 N-D 数组中有 NULL 步长和子偏移量时，它返回一个指针

```

void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
                      Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >= 0) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}

```

## 7.7.4 缓冲区相关函数

`int PyObject_CheckBuffer(PyObject *obj)`

属于稳定 ABI 自 3.11 版起。如果 `obj` 支持缓冲区接口，则返回 1，否则返回 0。返回 1 时不保证 `PyObject_GetBuffer()` 一定成功。本函数一定调用成功。

`int PyObject_GetBuffer(PyObject *exporter, Py_buffer *view, int flags)`

属于稳定 ABI 自 3.11 版起。向 `exporter` 发送请求以按照 `flags` 指定的内容填充 `view`。如果 `exporter` 无法提供要求类型的缓冲区，则它必须引发 `BufferError`，将 `view->obj` 设为 NULL 并返回 -1。

成功时，填充 `view`，将 `view->obj` 设为对 `exporter` 的新引用，并返回 0。当链式缓冲区提供程序将请求重定向到一个对象时，`view->obj` 可以引用该对象而不是 `exporter` (参见缓冲区对象结构)。

`PyObject_GetBuffer()` 必须与 `PyBuffer_Release()` 同时调用成功，类似于 `malloc()` 和 `free()`。因此，消费者程序用完缓冲区后，`PyBuffer_Release()` 必须保证被调用一次。

`void PyBuffer_Release(Py_buffer *view)`

属于稳定 ABI 自 3.11 版起。释放缓冲区 `view` 并释放对视图的支持对象 `view->obj` 的 *strong reference* (即递减引用计数)。该函数必须在缓冲区不再使用时调用，否则可能会发生引用泄漏。

若该函数针对的缓冲区不是通过 `PyObject_GetBuffer()` 获得的，将会出错。

`Py_ssize_t PyBuffer_SizeFromFormat(const char *format)`

属于稳定 ABI 自 3.11 版起。从 `format` 返回隐含的 `itemsizememlen`。如果出错，则引发异常并返回 -1。

在 3.9 版新加入。

`int PyBuffer_IsContiguous (const Py_buffer *view, char order)`

属于稳定 ABI 自 3.11 版起。如果 *view* 定义的内存是 C 风格 (*order* 为 'C') 或 Fortran 风格 (*order* 为 'F') *contiguous* 或其中之一 (*order* 是 'A')，则返回 1。否则返回 0。该函数总会成功。

`void *PyBuffer_GetPointer (const Py_buffer *view, const Py_ssize_t *indices)`

属于稳定 ABI 自 3.11 版起。获取给定 *view* 内的 *indices* 所指向的内存区域。*indices* 必须指向一个 *view->ndim* 索引的数组。

`int PyBuffer_FromContiguous (const Py_buffer *view, const void *buf, Py_ssize_t len, char fort)`

属于稳定 ABI 自 3.11 版起。从 *buf* 复制连续的 *len* 字节到 *view*。*fort* 可以是 'C' 或 'F' (对应于 C 风格或 Fortran 风格的顺序)。成功时返回 0，错误时返回 -1。

`int PyBuffer_ToContiguous (void *buf, const Py_buffer *src, Py_ssize_t len, char order)`

属于稳定 ABI 自 3.11 版起。从 *src* 复制 *len* 字节到 *buf*，成为连续字节串的形式。*order* 可以是 'C' 或 'F' 或 'A' (对应于 C 风格、Fortran 风格的顺序或其中任意一种)。成功时返回 0，出错时返回 -1。

如果 *len* != *src->len* 则此函数将报错。

`int PyObject_CopyData (PyObject *dest, PyObject *src)`

属于稳定 ABI 自 3.11 版起。将数据从 *src* 拷贝到 *dest* 缓冲区。可以在 C 风格或 Fortran 风格的缓冲区之间进行转换。

成功时返回 0，出错时返回 -1。

`void PyBuffer_FillContiguousStrides (int ndims, Py_ssize_t *shape, Py_ssize_t *strides, int itemsize, char order)`

属于稳定 ABI 自 3.11 版起。用给定形状的 *contiguous* 字节串数组 (如果 *order* 为 'C' 则为 C 风格，如果 *order* 为 'F' 则为 Fortran 风格) 来填充 *strides* 数组，每个元素具有给定的字节数。

`int PyBuffer_FillInfo (Py_buffer *view, PyObject *exporter, void *buf, Py_ssize_t len, int readonly, int flags)`

属于稳定 ABI 自 3.11 版起。处理导出程序的缓冲区请求，该导出程序要公开大小为 *len* 的 *buf*，并根据 *readonly* 设置可写性。*bug* 被解释为一个无符号字节序列。

参数 *flags* 表示请求的类型。该函数总是按照 *flag* 指定的内容填入 *view*，除非 *buf* 设为只读，并且 *flag* 中设置了 *PyBUF\_WRITABLE* 标志。

成功时，将 *view->obj* 设为对 *exporter* 的新引用并返回 0。否则，引发 *BufferError*，将 *view->obj* 设为 NULL 并返回 -1；

如果此函数用作 *getbufferproc* 的一部分，则 *exporter* 必须设置为导出对象，并且必须在未修改的情况下传递 *flags*。否则，*exporter* 必须是 NULL。

## 7.8 舊式緩衝協定 (Buffer Protocol)

在 3.0 版之後被弃用。

這些函式是 Python 2 中「舊式緩衝區協定」API 的一部分。在 Python 3 中，該協議已經不存在，但這些函式仍有公開以供移植 2.x 程式碼。它們充當新式緩衝區協定的相容性包裝器，但它們無法讓你控制匯出 (export) 緩衝區時所獲取資源的生命週期。

因此，建議你呼叫 *PyObject\_GetBuffer()* (或是以 *y\** 或 *w\** 格式碼 (*format code*) 呼叫 *PyArg\_ParseTuple()* 系列函式) 獲取物件的緩衝區視圖 (buffer view)，以及緩衝區視圖可被釋放時呼叫 *PyBuffer\_Release()*。

`int PyObject_AsCharBuffer (PyObject *obj, const char **buffer, Py_ssize_t *buffer_len)`

属于稳定 ABI。回傳一個指向可用作基於字元輸入之唯讀記憶體位置的指標。*obj* 引數必須支援單一片段 (single-segment) 字元緩衝區介面。成功時回傳 0，將 *buffer* 設定為記憶體位置、將 *buffer\_len* 設定為緩衝區長度。回傳 -1 在錯誤時設定 *TypeError*。

`int PyObject_AsReadBuffer (PyObject *obj, const void **buffer, Py_ssize_t *buffer_len)`

属于稳定 ABI。回傳一個指向包含任意資料之唯讀記憶體位置的指標。*obj* 引數必須支援單一片段可讀緩衝區介面。成功時回傳 0，`&buffer` 將 *buffer* 設定為記憶體位置、將 *buffer\_len* 設定為緩衝區長度。回傳 -1 在錯誤時設定 `TypeError`。

`int PyObject_CheckReadBuffer (PyObject *o)`

属于稳定 ABI。如果 *o* 支援單一片段可讀緩衝區介面，則回傳 1，否則回傳 0。這個函式一定會執行成功的。

請注意，該函式嘗試獲取和釋放緩衝區，且呼叫相應函式時發生的例外將被抑制。要獲取錯誤報告，請改用 `PyObject_GetBuffer()`。

`int PyObject_AsWriteBuffer (PyObject *obj, void **buffer, Py_ssize_t *buffer_len)`

属于稳定 ABI。回傳指向可寫記憶體位置的指標。*obj* 引數必須支援單一片段字元緩衝區介面。成功時回傳 0，`&buffer` 將 *buffer* 設定為記憶體位置，且將 *buffer\_len* 設定為緩衝區長度。回傳 -1 在錯誤時設定 `TypeError`。



## 具體物件層

此章節列出的函式僅能接受某些特定的 Python 物件型別，將錯誤型別的物件傳遞給它們不是什麼好事，如果你從 Python 程式當中接收到一個不確定是否正確型別的物件，那麼請一定要先做型別檢查。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 物件型別的“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別，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。

## 8.1 基礎物件

此段落描述 Python 型別物件與單例 (singleton) 物件 `None`。

### 8.1.1 类型对象

**type `PyTypeObject`**

属于受限 API（作为不透明的结构体）。对象的 C 结构用于描述 built-in 类型。

**`PyTypeObject` `PyType_Type`**

属于稳定 ABI。这是属于 type 对象的 type object，它在 Python 层面和 `type` 是相同的对象。

**`int` `PyType_Check` (`PyObject *`*o*)**

如果对象 *o* 是一个类型对象，包括派生自标准类型对象的类型实例则返回非零值。在所有其它情况下都返回 0。此函数将总是成功执行。

**`int` `PyType_CheckExact` (`PyObject *`*o*)**

如果对象 *o* 是一个类型对象，但不是标准类型对象的子类型则返回非零值。在所有其它情况下都返回 0。此函数将总是成功执行。

**`unsigned int` `PyType_ClearCache` ()**

属于稳定 ABI。清空内部查找缓存。返回当前版本标签。



unsigned long **PyType\_GetFlags** (*PyTypeObject* \*type)

属于稳定 ABI. 返回 *type* 的 *tp\_flags* 成员。此函数主要是配合 `Py_LIMITED_API` 使用；单独的旗标位会确保在各个 Python 发布版之间保持稳定，但对 *tp\_flags* 本身的访问并不是受限 API 的一部分。

在 3.2 版新加入。

在 3.4 版的變更: 返回类型现在是 unsigned long 而不是 long。

void **PyType\_Modified** (*PyTypeObject* \*type)

属于稳定 ABI. 使该类型及其所有子类型的内部查找缓存失效。此函数必须在对该类型的属性或基类进行任何手动修改之后调用。

int **PyType\_HasFeature** (*PyTypeObject* \*o, int feature)

如果类型对象 *o* 设置了特性 *feature* 则返回非零值。类型特性是用单个比特位旗标来表示的。

int **PyType\_IS\_GC** (*PyTypeObject* \*o)

如果类型对象包括了对循环检测器的支持则返回真值；这将测试类型旗标 `Py_TPFLAGS_HAVE_GC`。

int **PyType\_IsSubtype** (*PyTypeObject* \*a, *PyTypeObject* \*b)

属于稳定 ABI. 如果 *a* 是 *b* 的子类型则返回真值。

此函数只检查实际的子类型，这意味着 `__subclasscheck__()` 不会在 *b* 上被调用。请调用 `PyObject_IsSubclass()` 来执行与 `issubclass()` 所做的相同检查。

*PyObject* \***PyType\_GenericAlloc** (*PyTypeObject* \*type, *Py\_ssize\_t* nitems)

回傳值: 新的参照。属于稳定 ABI. 类型对象的 *tp\_alloc* 槽位的通用处理器。请使用 Python 的默认内存分配机制来分配一个新的实例并将其所有内容初始化为 NULL。

*PyObject* \***PyType\_GenericNew** (*PyTypeObject* \*type, *PyObject* \*args, *PyObject* \*kwargs)

回傳值: 新的参照。属于稳定 ABI. 类型对象的 *tp\_new* 槽位的通用处理器。请使用类型的 *tp\_alloc* 槽位来创建一个新的实例。

int **PyType\_Ready** (*PyTypeObject* \*type)

属于稳定 ABI. 最终化一个类型对象。这应当在所有类型对象上调用以完成它们的初始化。此函数会负责从一个类型的基类添加被继承的槽位。成功时返回 0，或是在出错时返回 -1 并设置一个异常。

---

**備註:** 如果某些基类实现了 GC 协议并且所提供的类型的旗标中未包括 `Py_TPFLAGS_HAVE_GC`，则将自动从其父类实现 GC 协议。相反地，如果被创建的类型的旗标中确实包含 `Py_TPFLAGS_HAVE_GC` 则它 **必须** 自己实现 GC 协议，至少要实现 *tp\_traverse* 句柄。

---

*PyObject* \***PyType\_GetName** (*PyTypeObject* \*type)

回傳值: 新的参照。属于稳定 ABI 自 3.11 版起。返回类型名称。等同于获取类型的 `__name__` 属性。

在 3.11 版新加入。

*PyObject* \***PyType\_GetQualName** (*PyTypeObject* \*type)

回傳值: 新的参照。属于稳定 ABI 自 3.11 版起。返回类型的限定名称。等同于获取类型的 `__qualname__` 属性。

在 3.11 版新加入。

void \***PyType\_GetSlot** (*PyTypeObject* \*type, int slot)

属于稳定 ABI 自 3.4 版起。返回存储在给定槽位中的函数指针。如果结果为 NULL，则表示或者该槽位为 NULL，或者该函数调用传入了无效的形参。调用方通常要将结果指针转换到适当的函数类型。

请参阅 `PyType_Slot.slot` 查看可用的 *slot* 参数值。

在 3.4 版新加入。

在 3.10 版的變更: `PyType_GetSlot()` 现在可以接受所有类型。在此之前，它被限制为堆类型。

*PyObject* \*PyType\_GetModule (*PyTypeObject* \*type)

属于稳定 ABI 自 3.10 版起。返回当使用 *PyType\_FromModuleAndSpec()* 创建类型时关联到给定类型的模块对象。

如果没有关联到给定类型的模块，则设置 *TypeError* 并返回 *NULL*。

此函数通常被用于获取方法定义所在的模块。请注意在这样的方法中，*PyType\_GetModule(Py\_TYPE(self))* 可能不会返回预期的结果。*Py\_TYPE(self)* 可以是目标类的一个子类，而子类并不一定是在与其超类相同的模块中定义的。请参阅 *PyCMethod* 了解如何获取方法定义所在的类。请参阅 *PyType\_GetModuleByDef()* 了解有关无法使用 *PyCMethod* 的情况。

在 3.9 版新加入。

void \*PyType\_GetModuleState (*PyTypeObject* \*type)

属于稳定 ABI 自 3.10 版起。返回关联到给定类型的模块对象的状态。这是一个在 *PyType\_GetModule()* 的结果上调用 *PyModule\_GetState()* 的快捷方式。

如果没有关联到给定类型的模块，则设置 *TypeError* 并返回 *NULL*。

如果 *type* 有关联的模块但其状态为 *NULL*，则返回 *NULL* 且不设置异常。

在 3.9 版新加入。

*PyObject* \*PyType\_GetModuleByDef (*PyTypeObject* \*type, struct *PyModuleDef* \*def)

找到所属模块基于给定的 *PyModuleDef def* 创建的第一个上级类，并返回该模块。

如果未找到模块，则会引发 *TypeError* 并返回 *NULL*。

此函数预期会与 *PyModule\_GetState()* 一起使用以便从槽位方法 (如 *tp\_init* 或 *nb\_add*) 及其他定义方法的类无法使用 *PyCMethod* 调用惯例来传递的场合获取模块状态。

在 3.11 版新加入。

## 创建堆分配类型

下列函数和结构体可被用来创建堆类型。

*PyObject* \*PyType\_FromModuleAndSpec (*PyObject* \*module, *PyType\_Spec* \*spec, *PyObject* \*bases)

回傳值：新的參照。属于稳定 ABI 自 3.10 版起。Creates and returns a *heap type* from the *spec* (*Py\_TPFLAGS\_HEAPTYPE*).

*bases* 参数可被用来指定基类；它可以是单个类或由多个类组成的元组。如果 *bases* 为 *NULL*，则会改用 *Py\_tp\_bases* 槽位。如果该槽位也为 *NULL*，则会改用 *Py\_tp\_base* 槽位。如果该槽位同样为 *NULL*，则新类型将派生自 *object*。

*module* 参数可被用来记录新类定义所在的模块。它必须是一个模块对象或为 *NULL*。如果不为 *NULL*，则该模块会被关联到新类型并且可在之后通过 *PyType\_GetModule()* 来获取。这个关联模块不可被子类继承；它必须为每个类单独指定。

此函数会在新类型上调用 *PyType\_Ready()*。

在 3.9 版新加入。

在 3.10 版的變更：此函数现在接受一个单独类作为 *bases* 参数并接受 *NULL* 作为 *tp\_doc* 槽位。

*PyObject* \*PyType\_FromSpecWithBases (*PyType\_Spec* \*spec, *PyObject* \*bases)

回傳值：新的參照。属于稳定 ABI 自 3.3 版起。等價於 *PyType\_FromModuleAndSpec(NULL, spec, bases)*。

在 3.3 版新加入。

*PyObject* \*PyType\_FromSpec (*PyType\_Spec* \*spec)

回傳值：新的參照。属于稳定 ABI。等價於 *PyType\_FromSpecWithBases(spec, NULL)*。

**type PyType\_Spec**

属于稳定 ABI（包括所有成员）。定义一个类型的行为的结构体。

`const char *PyType_Spec.name`

类型的名称，用来设置 `PyTypeObject.tp_name`。

`int PyType_Spec.basicsize`

`int PyType_Spec.itemsize`

以字节数表示的实例大小，用来设置 `PyTypeObject.tp_basicsize` 和 `PyTypeObject.tp_itemsize`。

`int PyType_Spec.flags`

类型旗标，用来设置 `PyTypeObject.tp_flags`。

如果未设置 `Py_TPFLAGS_HEAPTYPE` 旗标，则 `PyType_FromSpecWithBases()` 会自动设置它。

`PyType_Slot *PyType_Spec.slots`

`PyType_Slot` 结构体的数组。以特殊槽位值 {0, NULL} 来结束。

**type PyType\_Slot**

属于稳定 ABI（包括所有成员）。定义一个类型的可选功能的结构体，包含一个槽位 ID 和一个值指针。

`int PyType_Slot.slot`

槽位 ID。

槽位 ID 的类名像是结构体 `PyTypeObject`, `PyNumberMethods`, `PySequenceMethods`, `PyMappingMethods` 和 `PyAsyncMethods` 的字段名附加一个 `Py_` 前缀。举例来说，使用：

- `Py_tp_dealloc` 设置 `PyTypeObject.tp_dealloc`
- `Py_nb_add` 设置 `PyNumberMethods.nb_add`
- `Py_sq_length` 设置 `PySequenceMethods.sq_length`

下列字段完全无法使得 `PyType_Spec` 和 `PyType_Slot` 来设置：

- `tp_dict`
- `tp_mro`
- `tp_cache`
- `tp_subclasses`
- `tp_weaklist`
- `tp_vectorcall`
- `tp_weaklistoffset` (参见 `PyMemberDef`)
- `tp_dictoffset` (参见 `PyMemberDef`)
- `tp_vectorcall_offset` (参见 `PyMemberDef`)

在某些平台上设置 `Py_tp_bases` 或 `Py_tp_base` 可能会有问题。为了避免问题，请改用 `PyType_FromSpecWithBases()` 的 `bases` 参数。

在 3.9 版的變更: `PyBufferProcs` 中的槽位可能会在不受限 API 中被设置。

在 3.11 版的變更: 现在 `bf_getbuffer` 和 `bf_releasebuffer` 将在受限 API 中可用。

`void *PyType_Slot.pfunc`

该槽位的预期值。在大多数情况下，这将是一个指向函数的指针。

`Py_tp_doc` 以外的槽位均不可为 NULL。

## 8.1.2 None 物件

请注意，Python/C API 中并没有直接公开 `None` 的 `PyTypeObject`。由于 `None` 是一个单例，测试对象标识号（在 C 语言中使用 `==` 运算符）就足够了。出于同样的原因也没有 `PyNone_Check()` 函数。

`PyObject *Py_None`

Python `None` 对象，表示缺乏值。这个对象没有方法。它需要像引用计数一样处理任何其他对象。

`Py_RETURN_NONE`

正确处理来自 C 函数内的 `Py_None` 返回（也就是说，增加 `None` 的引用计数并返回它。）

## 8.2 数值物件

### 8.2.1 整数物件

所有整数都实现为长度任意的长整数对象。

在出错时，大多数 `PyLong_As*` API 都会返回 `(return type)-1`，这与数字无法区分开。请采用 `PyErr_Occurred()` 来加以区分。

type `PyLongObject`

属于受限 ABI（作为不透明的结构体）。表示 Python 整数对象的 `PyObject` 子类型。

`PyTypeObject PyLong_Type`

属于稳定 ABI。这个 `PyTypeObject` 的实例表示 Python 的整数类型。与 Python 语言中的 `int` 相同。

int `PyLong_Check(PyObject *p)`

如果参数是 `PyLongObject` 或 `PyLongObject` 的子类型，则返回 `True`。该函数一定能够执行成功。

int `PyLong_CheckExact(PyObject *p)`

如果其参数属于 `PyLongObject`，但不是 `PyLongObject` 的子类型则返回真值。此函数总是会成功执行。

`PyObject *PyLong_FromLong(long v)`

回傳值：新的参照。属于稳定 ABI。由 `v` 返回一个新的 `PyLongObject` 对象，失败时返回 `NULL`。

当前的实现维护着一个整数对象数组，包含 `-5` 和 `256` 之间的所有整数对象。若创建一个位于该区间的 `int` 时，实际得到的将是对已有对象的引用。

`PyObject *PyLong_FromUnsignedLong(unsigned long v)`

回傳值：新的参照。属于稳定 ABI。基于 C `unsigned long` 返回一个新的 `PyLongObject` 对象，失败时返回 `NULL`。

`PyObject *PyLong_FromSsize_t(Py_ssize_t v)`

回傳值：新的参照。属于稳定 ABI。由 C `Py_ssize_t` 返回一个新的 `PyLongObject` 对象，失败时返回 `NULL`。

`PyObject *PyLong_FromSize_t(size_t v)`

回傳值：新的参照。属于稳定 ABI。由 C `size_t` 返回一个新的 `PyLongObject` 对象，失败则返回 `NULL`。

`PyObject *PyLong_FromLongLong(long long v)`

回傳值：新的参照。属于稳定 ABI。基于 C `long long` 返回一个新的 `PyLongObject`，失败时返回 `NULL`。

`PyObject *PyLong_FromUnsignedLongLong(unsigned long long v)`

回傳值：新的参照。属于稳定 ABI。基于 C `unsigned long long` 返回一个新的 `PyLongObject` 对象，失败时返回 `NULL`。

*PyObject* \*PyLong\_FromDouble (double v)

回傳值：新的參照。屬於穩定 ABI。由 *v* 的整數部分返回一個新的 *PyLongObject* 對象，失敗則返回 NULL。

*PyObject* \*PyLong\_FromString (const char \*str, char \*\*pend, int base)

回傳值：新的參照。屬於穩定 ABI。根據 *str* 字符串值返回一個新的 *PyLongObject*，*base* 指定了整數的基。如果 *pend* 不為 NULL，則 *\*pend* 將指向 *str* 中表示數字部分后面的第一個字符。如果 *base* 為 0，*str* 將採用 *integers* 的定義進行解釋；這時非零十進制數的前導零會觸發 *ValueError*。如果 *base* 不為 0，則須位於 2 和 36 之間（含 2 和 36）。基之後及數字之間的前導空格、單下劃線將被忽略。如果不存在數字，將觸發 *ValueError*。

也參考：

Python 方法 *int.to\_bytes()* 和 *int.from\_bytes()* 用於 *PyLongObject* 到/從字節數組之間以 256 為基數進行轉換。你可以使用 *PyObject\_CallMethod()* 從 C 調用它們。

*PyObject* \*PyLong\_FromUnicodeObject (*PyObject* \*u, int base)

回傳值：新的參照。將字符串 *u* 中的 Unicode 數字序列轉換為 Python 整數值。

在 3.3 版新加入。

*PyObject* \*PyLong\_FromVoidPtr (void \*p)

回傳值：新的參照。屬於穩定 ABI。從指針 *p* 創建一個 Python 整數。可以使用 *PyLong\_AsVoidPtr()* 返回的指針值。

long PyLong\_AsLong (*PyObject* \*obj)

屬於穩定 ABI。返回 *obj* 的 C long 表示形式。如果 *obj* 不是 *PyLongObject* 的實例，則會先調用其 *\_\_index\_\_()* 方法（如果存在）將其轉換為 *PyLongObject*。

如果 *obj* 的值超出了 long 的取值範圍則會引發 *OverflowError*。

出錯則返回 -1。請用 *PyErr\_Occurred()* 找出具體問題。

在 3.8 版的變更：如果可能將使用 *\_\_index\_\_()*。

在 3.10 版的變更：此函數將不再使用 *\_\_int\_\_()*。

long PyLong\_AsLongAndOverflow (*PyObject* \*obj, int \*overflow)

屬於穩定 ABI。返回 *obj* 的 C long 表示形式。如果 *obj* 不是 *PyLongObject* 的實例，則會先調用其 *\_\_index\_\_()* 方法（如果存在）將其轉換為 *PyLongObject*。

如果 *obj* 的值大於 LONG\_MAX 或小於 LONG\_MIN，則會把 *\*overflow* 分別置為 1 或 -1，並返回 -1；否則，將 *\*overflow* 置為 0。如果發生其他異常則按常規把 *\*overflow* 置為 0 並返回 -1。

出錯則返回 -1。請用 *PyErr\_Occurred()* 找出具體問題。

在 3.8 版的變更：如果可能將使用 *\_\_index\_\_()*。

在 3.10 版的變更：此函數將不再使用 *\_\_int\_\_()*。

long long PyLong\_AsLongLong (*PyObject* \*obj)

屬於穩定 ABI。返回 *obj* 的 C long long 表示形式。如果 *obj* 不是 *PyLongObject* 的實例，則會先調用其 *\_\_index\_\_()* 方法（如果存在）將其轉換為 *PyLongObject*。

如果 *obj* 值超出 long long 的取值範圍則會引發 *OverflowError*。

出錯則返回 -1。請用 *PyErr\_Occurred()* 找出具體問題。

在 3.8 版的變更：如果可能將使用 *\_\_index\_\_()*。

在 3.10 版的變更：此函數將不再使用 *\_\_int\_\_()*。

long long PyLong\_AsLongLongAndOverflow (*PyObject* \*obj, int \*overflow)

屬於穩定 ABI。返回 *obj* 的 C long long 表示形式。如果 *obj* 不是 *PyLongObject* 的實例，則會先調用其 *\_\_index\_\_()* 方法（如果存在）將其轉換為 *PyLongObject*。

如果 *obj* 的值大於 LLONG\_MAX 或小於 LLONG\_MIN，則會把 *\*overflow* 分別置為 1 或 -1，並返回 -1；否則，將 *\*overflow* 置為 0。如果發生其他異常則按常規把 *\*overflow* 置為 0 並返回 -1。



出错则返回 -1。请用 `PyErr_Occurred()` 找出具体问题。

在 3.2 版新加入。

在 3.8 版的變更: 如果可能将使用 `__index__()`。

在 3.10 版的變更: 此函数将不再使用 `__int__()`。

**`Py_ssize_t PyLong_AsSsize_t (PyObject *pylong)`**

属于稳定 ABI。返回 `pylong` 的 C 语言 `Py_ssize_t` 形式。`pylong` 必须是 `PyLongObject` 的实例。

如果 `pylong` 的值超出了 `Py_ssize_t` 的取值范围则会引发 `OverflowError`。

出错则返回 -1。请用 `PyErr_Occurred()` 找出具体问题。

**`unsigned long PyLong_AsUnsignedLong (PyObject *pylong)`**

属于稳定 ABI。返回 `pylong` 的 C `unsigned long` 表示形式。`pylong` 必须是 `PyLongObject` 的实例。

如果 `pylong` 的值超出了 `unsigned long` 的取值范围则会引发 `OverflowError`。

出错时返回 `(unsigned long)-1`，请利用 `PyErr_Occurred()` 辨别具体问题。

**`size_t PyLong_AsSize_t (PyObject *pylong)`**

属于稳定 ABI。返回 `pylong` 的 C 语言 `size_t` 形式。`pylong` 必须是 `PyLongObject` 的实例。

如果 `pylong` 的值超出了 `size_t` 的取值范围则会引发 `OverflowError`。

出错时返回 `(size_t)-1`，请利用 `PyErr_Occurred()` 辨别具体问题。

**`unsigned long long PyLong_AsUnsignedLongLong (PyObject *pylong)`**

属于稳定 ABI。返回 `pylong` 的 C `unsigned long long` 表示形式。`pylong` 必须是 `PyLongObject` 的实例。

如果 `pylong` 的值超出 `unsigned long long` 的取值范围则会引发 `OverflowError`。

出错时返回 `(unsigned long long)-1`，请利用 `PyErr_Occurred()` 辨别具体问题。

在 3.1 版的變更: 现在 `pylong` 为负值会触发 `OverflowError`，而不是 `TypeError`。

**`unsigned long PyLong_AsUnsignedLongMask (PyObject *obj)`**

属于稳定 ABI。返回 `obj` 的 C `unsigned long` 表示形式。如果 `obj` 不是 `PyLongObject` 的实例，则会先调用其 `__index__()` 方法（如果存在）将其转换为 `PyLongObject`。

如果 `obj` 的值超出了 `unsigned long` 的取值范围，则返回该值对 `ULONG_MAX + 1` 求模的余数。

出错时返回 `(unsigned long)-1`，请利用 `PyErr_Occurred()` 辨别具体问题。

在 3.8 版的變更: 如果可能将使用 `__index__()`。

在 3.10 版的變更: 此函数将不再使用 `__int__()`。

**`unsigned long long PyLong_AsUnsignedLongLongMask (PyObject *obj)`**

属于稳定 ABI。返回 `obj` 的 C `unsigned long long` 表示形式。如果 `obj` 不是 `PyLongObject` 的实例，则会先调用其 `__index__()` 方法（如果存在）将其转换为 `PyLongObject`。

如果 `obj` 的值超出了 `unsigned long long` 的取值范围，则返回该值对 `ULLONG_MAX + 1` 求模的余数。

出错时返回 `(unsigned long long)-1`，请利用 `PyErr_Occurred()` 辨别具体问题。

在 3.8 版的變更: 如果可能将使用 `__index__()`。

在 3.10 版的變更: 此函数将不再使用 `__int__()`。

**`double PyLong_AsDouble (PyObject *pylong)`**

属于稳定 ABI。返回 `pylong` 的 C `double` 表示形式。`pylong` 必须是 `PyLongObject` 的实例。

如果 `pylong` 的值超出了 `double` 的取值范围则会引发 `OverflowError`。

出错时返回 `-1.0`，请利用 `PyErr_Occurred()` 辨别具体问题。

`void *PyLong_AsVoidPtr (PyObject *pylong)`

属于稳定 ABI。将一个 Python 整数 *pylong* 转换为 C void 指针。如果 *pylong* 无法被转换，则将引发 `OverflowError`。这只是为了保证将通过 `PyLong_FromVoidPtr()` 创建的值产生一个可用的 void 指针。

出错时返回 NULL，请利用 `PyErr_Occurred()` 辨别具体问题。

## 8.2.2 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。

*PyObject* **PyBool\_Type**

属于稳定 ABI。这个 *PyObject* 的实例代表一个 Python 布尔类型；它与 Python 层面的 `bool` 是相同的对象。

`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。在參照 (reference) 計數上必須有著和其他物件一樣的處理方式。

*PyObject* \***Py\_True**

Python 的 True 物件。此物件有任何方法。在參照計數上必須有著和其他物件一樣的處理方式。

**Py\_RETURN\_FALSE**

從函式回傳 `Py_False`，適當的增加它的參照計數。

**Py\_RETURN\_TRUE**

從函式回傳 `Py_True`，適當的增加它的參照計數。

*PyObject* \***PyBool\_FromLong** (long v)

回傳值：新的參照。属于稳定 ABI。根據 *v* 的實際值來回傳一個 `Py_True` 或者 `Py_False` 的新參照。

## 8.2.3 浮點數 (Floating Point) 物件

`type PyFloatObject`

这个 C 类型 *PyObject* 的子类型代表一个 Python 浮点数对象。

*PyObject* **PyFloat\_Type**

属于稳定 ABI。这是个属于 C 类型 *PyObject* 的代表 Python 浮点类型的实例。在 Python 层面的类型 `float` 是同一个对象。

`int PyFloat_Check (PyObject *p)`

如果它的参数是一个 *PyFloatObject* 或者 *PyFloatObject* 的子类型则返回真值。此函数总是会成功执行。

`int PyFloat_CheckExact (PyObject *p)`

如果它的参数是一个 *PyFloatObject* 但不是 *PyFloatObject* 的子类型则返回真值。此函数总是会成功执行。

*PyObject* \***PyFloat\_FromString** (*PyObject* \*str)

回傳值：新的參照。属于稳定 ABI。根据字符串 *str* 的值创建一个 *PyFloatObject*，失败时返回 NULL。

*PyObject* \***PyFloat\_FromDouble** (double v)

回傳值：新的參照。属于稳定 ABI。根据 *v* 创建一个 *PyFloatObject* 对象，失败时返回 NULL。



double **PyFloat\_AsDouble** (*PyObject* \*pyfloat)

属于稳定 ABI。Return a C double representation of the contents of *pyfloat*. If *pyfloat* is not a Python floating point object but has a `__float__()` method, this method will first be called to convert *pyfloat* into a float. If `__float__()` is not defined then it falls back to `__index__()`. This method returns `-1.0` upon failure, so one should call *PyErr\_Occurred()* to check for errors.

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。属于稳定 ABI。返回一个 structseq 实例, 其中包含有关 float 的精度、最小值和最大值的信息。它是头文件 `float.h` 的一个简单包装。

double **PyFloat\_GetMax** ()

属于稳定 ABI。Return the maximum representable finite float *DBL\_MAX* as C double.

double **PyFloat\_GetMin** ()

属于稳定 ABI。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.

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 版新加入。

## 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` and `+0.0` produce the same bytes string.

int **PyFloat\_Pack2** (double *x*, unsigned char \**p*, int *le*)

Pack a C double as the IEEE 754 binary16 half-precision format.

int **PyFloat\_Pack4** (double *x*, unsigned char \**p*, int *le*)

Pack a C double as the IEEE 754 binary32 single precision format.

int **PyFloat\_Pack8** (double *x*, unsigned 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 unsigned char \*p, int le)

Unpack the IEEE 754 binary16 half-precision format as a C double.

double **PyFloat\_Unpack4** (const unsigned char \*p, int le)

Unpack the IEEE 754 binary32 single precision format as a C double.

double **PyFloat\_Unpack8** (const unsigned char \*p, int le)

Unpack the IEEE 754 binary64 double precision format as a C double.

## 8.2.4 复数对象

从 C API 看, Python 的复数对象由两个不同的部分实现: 一个是在 Python 程序使用的 Python 对象, 另外的是一个代表真正复数值的 C 结构体。API 提供了函数共同操作两者。

### 表示复数的 C 结构体

需要注意的是接受这些结构体的作为参数并当做结果返回的函数, 都是传递“值”而不是引用指针。此规则适用于整个 API。

type **Py\_complex**

这是一个对应 Python 复数对象的值部分的 C 结构体。绝大部分处理复数对象的函数都用这类型的结构体作为输入或者输出值, 它可近似地定义为:

```
typedef struct {
    double real;
    double imag;
} Py_complex;
```

**Py\_complex\_Py\_c\_sum** (*Py\_complex* left, *Py\_complex* right)

返回两个复数的和, 用 C 类型 *Py\_complex* 表示。

**Py\_complex\_Py\_c\_diff** (*Py\_complex* left, *Py\_complex* right)

返回两个复数的差, 用 C 类型 *Py\_complex* 表示。

**Py\_complex\_Py\_c\_neg** (*Py\_complex* num)

返回复数 *num* 的负值, 用 C *Py\_complex* 表示。

**Py\_complex\_Py\_c\_prod** (*Py\_complex* left, *Py\_complex* right)

返回两个复数的乘积, 用 C 类型 *Py\_complex* 表示。

**Py\_complex\_Py\_c\_quot** (*Py\_complex* dividend, *Py\_complex* divisor)

返回两个复数的商, 用 C 类型 *Py\_complex* 表示。

如果 *divisor* 为空, 则此方法将返回零并将 `errno` 设为 `EDOM`。

**Py\_complex\_Py\_c\_pow** (*Py\_complex* num, *Py\_complex* exp)

返回 *num* 的 *exp* 次幂, 用 C 类型 *Py\_complex* 表示。

如果 *num* 为空且 *exp* 不是正实数, 则此方法将返回零并将 `errno` 设为 `EDOM`。

## 表示复数的 Python 对象

**type `PyComplexObject`**

这个 C 类型 `PyObject` 的子类型代表一个 Python 复数对象。

**`PyTypeObject` `PyComplex_Type`**

属于稳定 ABI。这是个属于 `PyTypeObject` 的代表 Python 复数类型的实例。在 Python 层面的类型 `complex` 是同一个对象。

**int `PyComplex_Check` (`PyObject` \*p)**

如果它的参数是一个 `PyComplexObject` 或者 `PyComplexObject` 的子类型则返回真值。此函数总是会成功执行。

**int `PyComplex_CheckExact` (`PyObject` \*p)**

如果它的参数是一个 `PyComplexObject` 但不是 `PyComplexObject` 的子类型则返回真值。此函数总是会成功执行。

**`PyObject` \*`PyComplex_FromCComplex` (`Py_complex` v)**

回傳值：新的參照。根据 C 类型 `Py_complex` 的值生成一个新的 Python 复数对象。

**`PyObject` \*`PyComplex_FromDoubles` (double real, double imag)**

回傳值：新的參照。属于稳定 ABI。根据 `real` 和 `imag` 返回一个新的 C 类型 `PyComplexObject` 对象。

**double `PyComplex_RealAsDouble` (`PyObject` \*op)**

属于稳定 ABI。以 C 类型 `double` 返回 `op` 的实部。

**double `PyComplex_ImagAsDouble` (`PyObject` \*op)**

属于稳定 ABI。以 C 类型 `double` 返回 `op` 的虚部。

**`Py_complex` `PyComplex_AsCComplex` (`PyObject` \*op)**

返回复数 `op` 的 C 类型 `Py_complex` 值。

如果 `op` 不是一个 Python 复数对象但是具有 `__complex__()` 方法，则会先调用该方法将 `op` 转换为 Python 复数对象。如果 `__complex__()` 未定义则将回退至 `__float__()`。如果 `__float__()` 未定义则将回退至 `__index__()`。当失败时，该方法将返回实数值 `-1.0`。

在 3.8 版的變更：如果可用則使用 `__index__()`。

## 8.3 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。

### 8.3.1 位元組物件 (Bytes Objects)

这些函数在期望附带一个字节串形参但却附带了一个非字节串形参被调用时会引发 `TypeError`。

**type `PyBytesObject`**

这种 `PyObject` 的子类型表示一个 Python 字节对象。

**`PyTypeObject` `PyBytes_Type`**

属于稳定 ABI。 `PyTypeObject` 的实例代表一个 Python 字节类型，在 Python 层面它与 `bytes` 是相同的对象。

**int `PyBytes_Check` (`PyObject` \*o)**

如果对象 `o` 是一个 `bytes` 对象或者 `bytes` 类型的子类型的实例则返回真值。此函数总是会成功执行。

**int PyBytes\_CheckExact** (*PyObject* \*o)

如果对象 *o* 是一个 bytes 对象但不是 bytes 类型的子类型的实例则返回真值。此函数总是会成功执行。

**PyObject\* PyBytes\_FromString** (const char \*v)

回傳值：新的參照。属于稳定 ABI。成功时返回一个以字符串 *v* 的副本为值的新字节串对象，失败时返回 NULL。形参 *v* 不可为 NULL；它不会被检查。

**PyObject\* PyBytes\_FromStringAndSize** (const char \*v, *Py\_ssize\_t* len)

回傳值：新的參照。属于稳定 ABI。成功时返回一个以字符串 *v* 的副本为值且长度为 *len* 的新字节串对象，失败时返回 NULL。如果 *v* 为 NULL，则不初始化字节串对象的内容。

**PyObject\* PyBytes\_FromFormat** (const char \*format, ...)

回傳值：新的參照。属于稳定 ABI。接受一个 C printf() 风格的 *format* 字符串和可变数量的参数，计算结果 Python 字节串对象的大小并返回参数值经格式化后的字节串对象。可变数量的参数必须均为 C 类型并且必须恰好与 *format* 字符串中的格式字符相对应。允许使用下列格式字符串：

格式字符	类型	注释
%%	<i>n/a</i>	文字% 字符。
%c	int	一个字节，被表示为一个 C 语言的整型
%d	int	等價於 printf("%d")。 <sup>1</sup>
%u	unsigned int	等價於 printf("%u")。 <sup>1</sup>
%ld	long	等價於 printf("%ld")。 <sup>1</sup>
%lu	unsigned long	等價於 printf("%lu")。 <sup>1</sup>
%zd	<i>Py_ssize_t</i>	等價於 printf("%zd")。 <sup>1</sup>
%zu	size_t	等價於 printf("%zu")。 <sup>1</sup>
%i	int	等價於 printf("%i")。 <sup>1</sup>
%x	int	等價於 printf("%x")。 <sup>1</sup>
%s	const char*	以 null 为终止符的 C 字符串组。
%p	const void*	一个 C 指针的十六进制表示形式。基本等价于 printf("%p") 但它会确保以字面值 0x 开头，不论系统平台上 printf 的输出是什么。

无法识别的格式字符会导致将格式字符串的其余所有内容原样复制到结果对象，并丢弃所有多余的参数。

**PyObject\* PyBytes\_FromFormatV** (const char \*format, va\_list vargs)

回傳值：新的參照。属于稳定 ABI。与 *PyBytes\_FromFormat()* 完全相同，除了它需要两个参数。

**PyObject\* PyBytes\_FromObject** (*PyObject* \*o)

回傳值：新的參照。属于稳定 ABI。返回字节表示实现缓冲区协议的对象 \*o\*。

**Py\_ssize\_t PyBytes\_Size** (*PyObject* \*o)

属于稳定 ABI。返回字节对象 \*o\* 中字节的长度。

**Py\_ssize\_t PyBytes\_GET\_SIZE** (*PyObject* \*o)

类似于 *PyBytes\_Size()*，但是不带错误检测。

**char\* PyBytes\_AsString** (*PyObject* \*o)

属于稳定 ABI。返回对应 *o* 的内容的指针。该指针指向 *o* 的内部缓冲区，其中包含 *len(o) + 1* 个字节。缓冲区的最后一个字节总是为空，不论是否存在其他空字节。该数据不可通过任何形式来修改，除非是刚使用 *PyBytes\_FromStringAndSize(NULL, size)* 创建该对象。它不可被撤销分配。如果 *o* 根本不是一个字节串对象，则 *PyBytes\_AsString()* 将返回 NULL 并引发 *TypeError*。

**char\* PyBytes\_AS\_STRING** (*PyObject* \*string)

类似于 *PyBytes\_AsString()*，但是不带错误检测。

<sup>1</sup> 对于整数说明符 (d, u, ld, lu, zd, zu, i, x)：当给出精度时，0 转换标志是有效的。

**int PyBytes\_AsStringAndSize** (*PyObject* \*obj, char \*\*buffer, *Py\_ssize\_t* \*length)

属于稳定 ABI。通过输出变量 *buffer* 和 *length* 返回对象 *obj* 以空值作为结束的内容。成功时返回 0。

如果 *length* 为 NULL，字节串对象就不包含嵌入的空字节；如果包含，则该函数将返回 -1 并引发 `ValueError`。

该缓冲区指向 *obj* 的内部缓冲，它的末尾包含一个额外的空字节（不算在 *length* 当中）。该数据不可通过任何方式来修改，除非是刚使用 `PyBytes_FromStringAndSize(NULL, size)` 创建该对象。它不可被撤销分配。如果 *obj* 根本不是一个字节串对象，则 `PyBytes_AsStringAndSize()` 将返回 -1 并引发 `TypeError`。

在 3.5 版的變更：以前，当字节串对象中出现嵌入的空字节时将引发 `TypeError`。

**void PyBytes\_Concat** (*PyObject* \*\*bytes, *PyObject* \*newpart)

属于稳定 ABI。在 *\*bytes* 中创建新的字节串对象，其中包含添加到 *bytes* 的 *newpart* 的内容；调用者将获得新的引用。对 *bytes* 原值的引用将被收回。如果无法创建新对象，对 *bytes* 的旧引用仍将被丢弃且 *\*bytes* 的值将被设为 NULL；并将设置适当的异常。

**void PyBytes\_ConcatAndDel** (*PyObject* \*\*bytes, *PyObject* \*newpart)

属于稳定 ABI。在 *\*bytes* 中创建一个新的字节串对象，其中包含添加到 *bytes* 的 *newpart* 的内容。此版本将释放对 *newpart* 的 *strong reference*（即递减其引用计数）。

**int \_PyBytes\_Resize** (*PyObject* \*\*bytes, *Py\_ssize\_t* newsize)

改变字节串大小的一种方式，即使其为“不可变对象”。此方式仅用于创建全新的字节串对象；如果字节串在代码的其他部分已知则不可使用此方式。如果输入字节串对象的引用计数不为 1 则调用此函数将报错。传入一个现有字节串对象的地址作为 *lvalue*（它可能会被写入），并传入希望的新大小。当成功时，*\*bytes* 将存放改变大小后的字节串对象并返回 0；*\*bytes* 中的地址可能与其输入值不同。如果重新分配失败，则 *\*bytes* 上的原字节串对象将被撤销分配，*\*bytes* 会被设为 NULL，同时设置 `MemoryError` 并返回 -1。

## 8.3.2 位元組串列物件 (Byte Array Objects)

**type PyByteArrayObject**

这个 *PyObject* 的子类型表示一个 Python 字节数组对象。

**PyTypeObject PyByteArray\_Type**

属于稳定 ABI。Python bytearray 类型表示为 *PyTypeObject* 的实例；这与 Python 层面的 bytearray 是相同的对象。

### 类型检查宏

**int PyByteArray\_Check** (*PyObject* \*o)

如果对象 *o* 是一个 bytearray 对象或者 bytearray 类型的子类型的实例则返回真值。此函数总是会成功执行。

**int PyByteArray\_CheckExact** (*PyObject* \*o)

如果对象 *o* 是一个 bytearray 对象但不是 bytearray 类型的子类型的实例则返回真值。此函数总是会成功执行。



## 直接 API 函数

*PyObject* \***PyByteArray\_FromObject** (*PyObject* \*o)

回傳值：新的參照。属于稳定 ABI。根据任何实现了缓冲区协议的对象 *o*，返回一个新的字节数组对象。

*PyObject* \***PyByteArray\_FromStringAndSize** (const char \*string, *Py\_ssize\_t* len)

回傳值：新的參照。属于稳定 ABI。根据 *string* 及其长度 *len* 创建一个新的 bytearray 对象。当失败时返回 NULL。

*PyObject* \***PyByteArray\_Concat** (*PyObject* \*a, *PyObject* \*b)

回傳值：新的參照。属于稳定 ABI。连接字节数组 *a* 和 *b* 并返回一个带有结果的新的字节数组。

*Py\_ssize\_t* **PyByteArray\_Size** (*PyObject* \*bytearray)

属于稳定 ABI。在检查 NULL 指针后返回 bytearray 的大小。

char \***PyByteArray\_AsString** (*PyObject* \*bytearray)

属于稳定 ABI。在检查 NULL 指针后返回将 bytearray 返回为一个字符数组。返回的数组总是会附加一个额外的空字节。

int **PyByteArray\_Resize** (*PyObject* \*bytearray, *Py\_ssize\_t* len)

属于稳定 ABI。将 bytearray 的内部缓冲区的大小调整为 *len*。

## 巨集

这些宏减低安全性以换取性能，它们不检查指针。

char \***PyByteArray\_AS\_STRING** (*PyObject* \*bytearray)

Similar to *PyByteArray\_AsString()*, but without error checking.

*Py\_ssize\_t* **PyByteArray\_GET\_SIZE** (*PyObject* \*bytearray)

Similar to *PyByteArray\_Size()*, but without error checking.

## 8.3.3 Unicode 物件與編碼

### Unicode 对象

自从 python3.3 中实现了 **PEP 393** 以来，Unicode 对象在内部使用各种表示形式，以便在保持内存效率的同时处理完整范围的 Unicode 字符。对于所有代码点都低于 128、256 或 65536 的字符串，有一些特殊情况；否则，代码点必须低于 1114112（这是完整的 Unicode 范围）。

*Py\_UNICODE\** 和 UTF-8 表示形式将按需创建并缓存至 Unicode 对象。*Py\_UNICODE\** 表示形式是已弃用且低效率的。

由于旧 API 和新 API 之间的转换，Unicode 对象内部可以处于两种状态，这取决于它们的创建方式：

- “规范” Unicode 对象是由非弃用的 Unicode API 创建的所有对象。它们使用实现所允许的最有效的表达方式。
- “legacy” Unicode objects have been created through one of the deprecated APIs (typically *PyUnicode\_FromUnicode()*) and only bear the *Py\_UNICODE\** representation; you will have to call *PyUnicode\_READY()* on them before calling any other API.

---

備註：The “legacy” Unicode object will be removed in Python 3.12 with deprecated APIs. All Unicode objects will be “canonical” since then. See **PEP 623** for more information.

---



## Unicode 类型

以下是用于 Python 中 Unicode 实现的基本 Unicode 对象类型：

type **Py\_UCS4**

type **Py\_UCS2**

type **Py\_UCS1**

属于稳定 ABI。这些类型是无符号整数类型的类型定义，其宽度足以分别包含 32 位、16 位和 8 位字符。当需要处理单个 Unicode 字符时，请使用 `Py_UCS4`。

在 3.3 版新加入。

type **Py\_UNICODE**

这是 `wchar_t` 的类型定义，根据平台的不同它可能为 16 位类型或 32 位类型。

在 3.3 版的變更：在以前的版本中，这是 16 位类型还是 32 位类型，这取决于您在构建时选择的是“窄”还是“宽” Unicode 版本的 Python。

type **PyASCIIObject**

type **PyCompactUnicodeObject**

type **PyUnicodeObject**

这些关于 `PyObject` 的子类型表示了一个 Python Unicode 对象。在几乎所有情形下，它们不应该被直接使用，因为所有处理 Unicode 对象的 API 函数都接受并返回 `PyObject` 类型的指针。

在 3.3 版新加入。

*PyTypeObject* **PyUnicode\_Type**

属于稳定 ABI。这个 `PyTypeObject` 实例代表 Python Unicode 类型。它作为 `str` 公开给 Python 代码。

以下 API 是 C 宏和静态内联函数，用于快速检查和访问 Unicode 对象的内部只读数据：

int **PyUnicode\_Check** (*PyObject* \*obj)

如果对象 *obj* 是 Unicode 对象或 Unicode 子类型的实例则返回真值。此函数总是会成功执行。

int **PyUnicode\_CheckExact** (*PyObject* \*obj)

如果对象 *obj* 是一个 Unicode 对象，但不是某个子类型的实例则返回真值。此函数总是会成功执行。

int **PyUnicode\_READY** (*PyObject* \*unicode)

确保字符串对象 \*o\* 处于“规范的”表达方式。在使用下面描述的任何访问宏之前，这是必需的。

Returns 0 on success and -1 with an exception set on failure, which in particular happens if memory allocation fails.

在 3.3 版新加入。

自從版本 3.10 後不推薦使用，將會自版本 3.12 中移除。： This API will be removed with `PyUnicode_FromUnicode()`.

*Py\_ssize\_t* **PyUnicode\_GET\_LENGTH** (*PyObject* \*unicode)

返回以码位点数量表示的 Unicode 字符串长度。*unicode* 必须为“规范”表示的 Unicode 对象（不会检查这一点）。

在 3.3 版新加入。

*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 macro. Make sure `PyUnicode_READY()` has been called before accessing this.

在 3.3 版新加入。

**PyUnicode\_WCHAR\_KIND**

**PyUnicode\_1BYTE\_KIND**

**PyUnicode\_2BYTE\_KIND**

**PyUnicode\_4BYTE\_KIND**

返回 `PyUnicode_KIND()` 宏的值。

在 3.3 版新加入。

自從版本 3.10 後不推薦使用，將會自版本 3.12 中移除。: `PyUnicode_WCHAR_KIND` 已用。

**int PyUnicode\_KIND (PyObject \*unicode)**

返回一个 PyUnicode 类型的常量 (见上文), 指明此 see above) that indicate how many bytes per character this Unicode 对象用来存储每个字符所使用的字节数。unicode 必须为“规范”表示的 Unicode 对象 (不会检查这一点)。

在 3.3 版新加入。

**void \*PyUnicode\_DATA (PyObject \*unicode)**

返回一个指向原始 Unicode 缓冲区的空指针。unicode 必须为“规范”表示的 Unicode 对象 (不会检查这一点)。

在 3.3 版新加入。

**void PyUnicode\_WRITE (int kind, void \*data, Py\_ssize\_t index, Py\_UCS4 value)**

写入一个规范表示的 data (如同用 `PyUnicode_DATA()` 获取)。此函数不会执行正确性检查，被设计为在循环中使用。调用者应当如同从其他调用中获取一样缓存 kind 值和 data 指针。index 是字符串中的索引号 (从 0 开始) 而 value 是应写入该位置的新码位值。

在 3.3 版新加入。

**Py\_UCS4 PyUnicode\_READ (int kind, void \*data, Py\_ssize\_t index)**

从规范表示的 data (如同用 `PyUnicode_DATA()` 获取) 中读取一个码位。不会执行检查或就绪调用。

在 3.3 版新加入。

**Py\_UCS4 PyUnicode\_READ\_CHAR (PyObject \*unicode, Py\_ssize\_t index)**

从 Unicode 对象 unicode 读取一个字符，必须为“规范”表示形式。如果你执行多次连续读取则此函数的效率将低于 `PyUnicode_READ()`。

在 3.3 版新加入。

**Py\_UCS4 PyUnicode\_MAX\_CHAR\_VALUE (PyObject \*unicode)**

返回适合基于 unicode 创建另一个字符串的最大码位点，该参数必须为“规范”表示形式。这始终是一种近似但比在字符串上执行迭代更高效。

在 3.3 版新加入。

**Py\_ssize\_t PyUnicode\_GET\_SIZE (PyObject \*unicode)**

Return the size of the deprecated `Py_UNICODE` representation, in code units (this includes surrogate pairs as 2 units). unicode has to be a Unicode object (not checked).

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: 旧式 Unicode API 的一部分，请迁移到使用 `PyUnicode_GET_LENGTH()`。

**Py\_ssize\_t PyUnicode\_GET\_DATA\_SIZE (PyObject \*unicode)**

Return the size of the deprecated `Py_UNICODE` representation in bytes. unicode has to be a Unicode object (not checked).

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: 旧式 Unicode API 的一部分，请迁移到使用 `PyUnicode_GET_LENGTH()`。

**Py\_UNICODE \*PyUnicode\_AS\_UNICODE (PyObject \*unicode)**

`const char *PyUnicode_AS_DATA (PyObject *unicode)`

Return a pointer to a `Py_UNICODE` representation of the object. The returned buffer is always terminated with an extra null code point. It may also contain embedded null code points, which would cause the string to be truncated when used in most C functions. The `AS_DATA` form casts the pointer to `const char*`. The `unicode` argument has to be a Unicode object (not checked).

在 3.3 版的變更: This function is now inefficient -- because in many cases the `Py_UNICODE` representation does not exist and needs to be created -- and can fail (return NULL with an exception set). Try to port the code to use the new `PyUnicode_nBYTE_DATA()` macros or use `PyUnicode_WRITE()` or `PyUnicode_READ()`.

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: 旧式 Unicode API 的一部分，请迁移到使用 `PyUnicode_nBYTE_DATA()` 宏族。

`int PyUnicode_IsIdentifier (PyObject *unicode)`

属于稳定 ABI。如果字符串按照语言定义是合法的标识符则返回 1，参见 `identifiers` 小节。否则返回 0。

在 3.9 版的變更: 如果字符串尚未就绪则此函数不会再调用 `Py_FatalError()`。

## Unicode 字符属性

Unicode 提供了许多不同的字符特性。最常需要的宏可以通过这些宏获得，这些宏根据 Python 配置映射到 C 函数。

`int Py_UNICODE_ISSPACE (Py_UCS4 ch)`

根据 `ch` 是否为空白字符返回 1 或 0。

`int Py_UNICODE_ISLOWER (Py_UCS4 ch)`

根据 `ch` 是否为小写字符返回 1 或 0。

`int Py_UNICODE_ISUPPER (Py_UCS4 ch)`

根据 `ch` 是否为大写字符返回 1 或 0

`int Py_UNICODE_ISTITLE (Py_UCS4 ch)`

根据 `ch` 是否为标题化的大小写返回 1 或 0。

`int Py_UNICODE_ISLINEBREAK (Py_UCS4 ch)`

根据 `ch` 是否为换行类字符返回 1 或 0。

`int Py_UNICODE_ISDECIMAL (Py_UCS4 ch)`

根据 `ch` 是否为十进制数字返回 1 或 0。

`int Py_UNICODE_ISDIGIT (Py_UCS4 ch)`

根据 `ch` 是否为数码类字符返回 1 或 0。

`int Py_UNICODE_ISNUMERIC (Py_UCS4 ch)`

根据 `ch` 是否为数值类字符返回 1 或 0。

`int Py_UNICODE_ISALPHA (Py_UCS4 ch)`

根据 `ch` 是否为字母类字符返回 1 或 0。

`int Py_UNICODE_ISALNUM (Py_UCS4 ch)`

根据 `ch` 是否为字母数字类字符返回 1 或 0。

`int Py_UNICODE_ISPRINTABLE (Py_UCS4 ch)`

根据 `ch` 是否为可打印字符返回 1 或 “0”。不可打印字符是指在 Unicode 字符数据库中被定义为“Other”或“Separator”的字符，例外情况是 ASCII 空格 (0x20) 被视为可打印字符。(请注意在此语境下可打印字符是指当在字符串上发起调用 `repr()` 时不应被转义的字符。它们字符串写入 `sys.stdout` 或 `sys.stderr` 时所需的处理无关)。

这些 API 可用于快速直接的字符转换：

*Py\_UCS4* **Py\_UNICODE\_TOLOWER** (*Py\_UCS4* ch)

返回转换为小写形式的字符 *ch*。

在 3.3 版之後被用: 此函数使用简单的大小写映射。

*Py\_UCS4* **Py\_UNICODE\_TOUPPER** (*Py\_UCS4* ch)

返回转换为大写形式的字符 *ch*。

在 3.3 版之後被用: 此函数使用简单的大小写映射。

*Py\_UCS4* **Py\_UNICODE\_TOTITLE** (*Py\_UCS4* ch)

返回转换为标题大小写形式的字符 *ch*。

在 3.3 版之後被用: 此函数使用简单的大小写映射。

int **Py\_UNICODE\_TODECIMAL** (*Py\_UCS4* ch)

Return the character *ch* converted to a decimal positive integer. Return  $-1$  if this is not possible. This macro does not raise exceptions.

int **Py\_UNICODE\_TODIGIT** (*Py\_UCS4* ch)

Return the character *ch* converted to a single digit integer. Return  $-1$  if this is not possible. This macro does not raise exceptions.

double **Py\_UNICODE\_TONUMERIC** (*Py\_UCS4* ch)

Return the character *ch* converted to a double. Return  $-1.0$  if this is not possible. This macro does not raise exceptions.

这些 API 可被用来操作代理项:

**Py\_UNICODE\_IS\_SURROGATE** (ch)

检测 *ch* 是否为代理项 ( $0xD800 \leq ch \leq 0xDFFF$ )。

**Py\_UNICODE\_IS\_HIGH\_SURROGATE** (ch)

检测 *ch* 是否为高代理项 ( $0xD800 \leq ch \leq 0xDBFF$ )。

**Py\_UNICODE\_IS\_LOW\_SURROGATE** (ch)

检测 *ch* 是否为低代理项 ( $0xDC00 \leq ch \leq 0xDFFF$ )。

**Py\_UNICODE\_JOIN\_SURROGATES** (high, low)

Join two surrogate characters and return a single *Py\_UCS4* value. *high* and *low* are respectively the leading and trailing surrogates in a surrogate pair.

## 创建和访问 Unicode 字符串

要创建 Unicode 对象和访问其基本序列属性, 请使用这些 API:

*PyObject* \***PyUnicode\_New** (*Py\_ssize\_t* size, *Py\_UCS4* maxchar)

回傳值: 新的參照。创建一个新的 Unicode 对象。*maxchar* 应为可放入字符串的实际最大码位。作为一个近似值, 它可被向上舍入到序列 127, 255, 65535, 1114111 中最接近的值。

这是分配新的 Unicode 对象的推荐方式。使用此函数创建的对象不可改变大小。

在 3.3 版新加入。

*PyObject* \***PyUnicode\_FromKindAndData** (int kind, const void \*buffer, *Py\_ssize\_t* size)

回傳值: 新的參照。以给定的 *kind* 创建一个新的 Unicode 对象 (可能的值为 *PyUnicode\_1BYTE\_KIND* 等, 即 *PyUnicode\_KIND()* 所返回的值)。*buffer* 必须指向由此分类所给出的, 以每字符 1, 2 或 4 字节单位的 *size* 大小的数组。

如有必要, 输入 *buffer* 将被拷贝并转换为规范表示形式。例如, 如果 *buffer* 是一个 UCS4 字符串 (*PyUnicode\_4BYTE\_KIND*) 且仅由 UCS1 范围内的码位组成, 它将被转换为 UCS1 (*PyUnicode\_1BYTE\_KIND*)。

在 3.3 版新加入。

**PyObject\*PyUnicode\_FromStringAndSize** (const char \*str, Py\_ssize\_t size)

回傳值：新的參照。属于穩定 ABI。Create a Unicode object from the char buffer *str*. The bytes will be interpreted as being UTF-8 encoded. The buffer is copied into the new object. If the buffer is not NULL, the return value might be a shared object, i.e. modification of the data is not allowed.

If *str* is NULL, this function behaves like *PyUnicode\_FromUnicode()* with the buffer set to NULL. This usage is deprecated in favor of *PyUnicode\_New()*, and will be removed in Python 3.12.

**PyObject\*PyUnicode\_FromString** (const char \*str)

回傳值：新的參照。属于穩定 ABI。根据 UTF-8 编码的以空值结束的字符缓冲区 *str* 创建一个 Unicode 对象。

**PyObject\*PyUnicode\_FromFormat** (const char \*format, ...)

回傳值：新的參照。属于穩定 ABI。Take a C *printf()*-style *format* string and a variable number of arguments, calculate the size of the resulting Python Unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* ASCII-encoded string. The following format characters are allowed:

格式字符	类型	注释
%%	<i>n/a</i>	文字% 字符。
%c	int	单个字符，表示为 C 语言的整型。
%d	int	等價於 <code>printf("%d")</code> 。 <sup>1</sup>
%u	unsigned int	等價於 <code>printf("%u")</code> 。 <sup>1</sup>
%ld	long	等價於 <code>printf("%ld")</code> 。 <sup>1</sup>
%li	long	等價於 <code>printf("%li")</code> 。 <sup>1</sup>
%lu	unsigned long	等價於 <code>printf("%lu")</code> 。 <sup>1</sup>
%lld	long long	等價於 <code>printf("%lld")</code> 。 <sup>1</sup>
%lli	long long	等價於 <code>printf("%lli")</code> 。 <sup>1</sup>
%llu	unsigned long long	等價於 <code>printf("%llu")</code> 。 <sup>1</sup>
%zd	<i>Py_ssize_t</i>	等價於 <code>printf("%zd")</code> 。 <sup>1</sup>
%zi	<i>Py_ssize_t</i>	等價於 <code>printf("%zi")</code> 。 <sup>1</sup>
%zu	<i>size_t</i>	等價於 <code>printf("%zu")</code> 。 <sup>1</sup>
%i	int	等價於 <code>printf("%i")</code> 。 <sup>1</sup>
%x	int	等價於 <code>printf("%x")</code> 。 <sup>1</sup>
%s	const char*	以 null 为终止符的 C 字符数组。
%p	const void*	一个 C 指针的十六进制表示形式。基本等价于 <code>printf("%p")</code> 但它会确保以字面值 0x 开头，不论系统平台上 <code>printf</code> 的输出是什么。
%A	PyObject*	<code>ascii()</code> 调用的结果。
%U	PyObject*	一 Unicode 物件。
%V	PyObject*, const char*	一个 Unicode 对象 (可以为 NULL) 和一个以空值结束的 C 字符数组作为第二个形参 (如果第一个形参为 NULL, 第二个形参将被使用)。
%S	PyObject*	调用 <code>PyObject_Str()</code> 的结果。
%R	PyObject*	调用 <code>PyObject_Repr()</code> 的结果。

An unrecognized format character causes all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

備註：The width formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes 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).

在 3.2 版的變更：增加了对 "%lld" 和 "%llu" 的支持。

在 3.3 版的變更：增加了对 "%li", "%lli" 和 "%zi" 的支持。

<sup>1</sup> For integer specifiers (d, u, ld, li, lu, lld, lli, llu, zd, zi, zu, i, x): the 0-conversion flag has effect even when a precision is given.



在 3.4 版的變更: 增加了对 "%s", "%A", "%U", "%V", "%S", "%R" 的宽度和精度格式符支持。

*PyObject* \*PyUnicode\_FromFormatV (const char \*format, va\_list args)

回傳值: 新的參照。属于稳定 ABI。等同于 `PyUnicode_FromFormat()` 但它将接受恰好两个参数。

*PyObject* \*PyUnicode\_FromObject (*PyObject* \*obj)

回傳值: 新的參照。属于稳定 ABI。如有必要将把一个 Unicode 子类型的实例拷贝为新的真实 Unicode 对象。如果 *obj* 已经是一个真实 Unicode 对象 (而非子类型), 则返回一个新的指向该对象的 *strong reference*。

非 Unicode 或其子类型的对象将导致 `TypeError`。

*PyObject* \*PyUnicode\_FromEncodedObject (*PyObject* \*obj, const char \*encoding, const char \*errors)

回傳值: 新的參照。属于稳定 ABI。将一个已编码的对象 *obj* 解码为 Unicode 对象。

`bytes`, `bytearray` 和其他字节类对象 将按照给定的 *encoding* 来解码并使用由 *errors* 定义的错误处理方式。两者均可为 `NULL` 即让接口使用默认值 (请参阅内建编解码器 了解详情)。

所有其他对象, 包括 Unicode 对象, 都将导致设置 `TypeError`。

如有错误该 API 将返回 `NULL`。调用方要负责递减指向所返回对象的引用。

*Py\_ssize\_t* PyUnicode\_GetLength (*PyObject* \*unicode)

属于稳定 ABI 自 3.7 版起。返回 Unicode 对象码位的长度。

在 3.3 版新加入。

*Py\_ssize\_t* PyUnicode\_CopyCharacters (*PyObject* \*to, *Py\_ssize\_t* to\_start, *PyObject* \*from, *Py\_ssize\_t* from\_start, *Py\_ssize\_t* how\_many)

将一个 Unicode 对象中的字符拷贝到另一个对象中。此函数会在必要时执行字符转换并会在可能的情况下回退到 `memcpy()`。在出错时将返回 -1 并设置一个异常, 在其他情况下将返回拷贝的字符数量。

在 3.3 版新加入。

*Py\_ssize\_t* PyUnicode\_Fill (*PyObject* \*unicode, *Py\_ssize\_t* start, *Py\_ssize\_t* length, *Py\_UCS4* fill\_char)

使用一个字符填充字符串: 将 *fill\_char* 写入 `unicode[start:start+length]`。

如果 *fill\_char* 值大于字符串最大字符值, 或者如果字符串有 1 以上的引用将执行失败。

返回写入的字符数量, 或者在出错时返回 -1 并引发一个异常。

在 3.3 版新加入。

int PyUnicode\_WriteChar (*PyObject* \*unicode, *Py\_ssize\_t* index, *Py\_UCS4* character)

属于稳定 ABI 自 3.7 版起。将一个字符写入到字符串。字符串必须通过 `PyUnicode_New()` 创建。由于 Unicode 字符串应当是不可变的, 因此该字符串不能被共享, 或是被哈希。

该函数将检查 *unicode* 是否为 Unicode 对象, 索引是否未越界, 并且对象是否可被安全地修改 (即其引用计数为一)。

在 3.3 版新加入。

*Py\_UCS4* PyUnicode\_ReadChar (*PyObject* \*unicode, *Py\_ssize\_t* index)

属于稳定 ABI 自 3.7 版起。从字符串读取一个字符。该函数将检查 *unicode* 是否为 Unicode 对象且索引是否未越界, 这不同于 `PyUnicode_READ_CHAR()`, 后者不会执行任何错误检查。

在 3.3 版新加入。

*PyObject* \*PyUnicode\_Substring (*PyObject* \*unicode, *Py\_ssize\_t* start, *Py\_ssize\_t* end)

回傳值: 新的參照。属于稳定 ABI 自 3.7 版起。返回 *unicode* 的一个子串, 从字符索引 *start* (包括) 到字符索引 *end* (不包括)。不支持负索引号。

在 3.3 版新加入。



*PyObject* \*PyUnicode\_AsUCS4 (*PyObject* \*unicode, *Py\_UCS4* \*buffer, *Py\_ssize\_t* buflen, int copy\_null)

属于稳定 ABI 自 3.7 版起. 将字符串 *unicode* 拷贝到一个 UCS4 缓冲区, 包括一个空字符, 如果设置了 *copy\_null* 的话. 出错时返回 NULL 并设置一个异常 (特别是当 *buflen* 小于 *unicode* 的长度时, 将设置 `SystemError` 异常). 成功时返回 *buffer*.

在 3.3 版新加入.

*Py\_UCS4* \*PyUnicode\_AsUCS4Copy (*PyObject* \*unicode)

属于稳定 ABI 自 3.7 版起. 将字符串 *unicode* 拷贝到使用 `PyMem_Malloc()` 分配的新 UCS4 缓冲区. 如果执行失败, 将返回 NULL 并设置 `MemoryError`. 返回的缓冲区将总是会添加一个额外的空码位.

在 3.3 版新加入.

## 已弃用的 Py\_UNICODE API

自從版本 3.3 後不推薦使用, 將會自版本 3.12 中移除.

These API functions are deprecated with the implementation of [PEP 393](#). Extension modules can continue using them, as they will not be removed in Python 3.x, but need to be aware that their use can now cause performance and memory hits.

*PyObject* \*PyUnicode\_FromUnicode (const *Py\_UNICODE* \*u, *Py\_ssize\_t* size)

回傳值: 新的參照. Create a Unicode object from the *Py\_UNICODE* buffer *u* of the given size. *u* may be NULL which causes the contents to be undefined. It is the user's responsibility to fill in the needed data. The buffer is copied into the new object.

If the buffer is not NULL, the return value might be a shared object. Therefore, modification of the resulting Unicode object is only allowed when *u* is NULL.

If the buffer is NULL, `PyUnicode_READY()` must be called once the string content has been filled before using any of the access macros such as `PyUnicode_KIND()`.

自從版本 3.3 後不推薦使用, 將會自版本 3.12 中移除.: Part of the old-style Unicode API, please migrate to using `PyUnicode_FromKindAndData()`, `PyUnicode_FromWideChar()`, or `PyUnicode_New()`.

*Py\_UNICODE* \*PyUnicode\_AsUnicode (*PyObject* \*unicode)

Return a read-only pointer to the Unicode object's internal *Py\_UNICODE* buffer, or NULL on error. This will create the *Py\_UNICODE\** representation of the object if it is not yet available. The buffer is always terminated with an extra null code point. Note that the resulting *Py\_UNICODE* string may also contain embedded null code points, which would cause the string to be truncated when used in most C functions.

自從版本 3.3 後不推薦使用, 將會自版本 3.12 中移除.: Part of the old-style Unicode API, please migrate to using `PyUnicode_AsUCS4()`, `PyUnicode_AsWideChar()`, `PyUnicode_ReadChar()` or similar new APIs.

*Py\_UNICODE* \*PyUnicode\_AsUnicodeAndSize (*PyObject* \*unicode, *Py\_ssize\_t* \*size)

Like `PyUnicode_AsUnicode()`, but also saves the *Py\_UNICODE()* array length (excluding the extra null terminator) in *size*. Note that the resulting *Py\_UNICODE\** string may contain embedded null code points, which would cause the string to be truncated when used in most C functions.

在 3.3 版新加入.

自從版本 3.3 後不推薦使用, 將會自版本 3.12 中移除.: Part of the old-style Unicode API, please migrate to using `PyUnicode_AsUCS4()`, `PyUnicode_AsWideChar()`, `PyUnicode_ReadChar()` or similar new APIs.

*Py\_ssize\_t* PyUnicode\_GetSize (*PyObject* \*unicode)

属于稳定 ABI. Return the size of the deprecated *Py\_UNICODE* representation, in code units (this includes surrogate pairs as 2 units).

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。：旧式 Unicode API 的一部分，请迁移到使用 `PyUnicode_GET_LENGTH()`。

## 语言区域编码格式

当前语言区域编码格式可被用来解码来自操作系统的文本。

**PyObject \*PyUnicode\_DecodeLocaleAndSize** (const char \*str, *Py\_ssize\_t* length, const char \*errors)

回傳值：新的參照。属于稳定 ABI 自 3.7 版起。解码字符串在 Android 和 VxWorks 上使用 UTF-8，在其他平台上则使用当前语言区域编码格式。支持的错误处理器有 "strict" 和 "surrogateescape" (PEP 383)。如果 *errors* 为 NULL 则解码器将使用 "strict" 错误处理器。*str* 必须以一个空字符结束但不可包含嵌入的空字符。

Use `PyUnicode_DecodeFSDefaultAndSize()` to decode a string from `Py_FileSystemDefaultEncoding` (the locale encoding read at Python startup).

此函数将忽略 Python UTF-8 模式。

### 也参考：

`Py_DecodeLocale()` 函式。

在 3.3 版新加入。

在 3.7 版的變更：此函数现在也会为 surrogateescape 错误处理器使用当前语言区域编码格式，但在 Android 上例外。在之前版本中，`Py_DecodeLocale()` 将被用于 surrogateescape，而当前语言区域编码格式将被用于 strict。

**PyObject \*PyUnicode\_DecodeLocale** (const char \*str, const char \*errors)

回傳值：新的參照。属于稳定 ABI 自 3.7 版起。类似于 `PyUnicode_DecodeLocaleAndSize()`，但会使用 `strlen()` 来计算字符串长度。

在 3.3 版新加入。

**PyObject \*PyUnicode\_EncodeLocale** (*PyObject \**unicode, const char \*errors)

回傳值：新的參照。属于稳定 ABI 自 3.7 版起。编码 Unicode 对象在 Android 和 VxWorks 上使用 UTF-8，在其他平台上使用当前语言区域编码格式。支持的错误处理器有 "strict" 和 "surrogateescape" (PEP 383)。如果 *errors* 为 NULL 则编码器将使用 "strict" 错误处理器。返回一个 bytes 对象。*unicode* 不可包含嵌入的空字符。

Use `PyUnicode_EncodeFSDefault()` to encode a string to `Py_FileSystemDefaultEncoding` (the locale encoding read at Python startup).

此函数将忽略 Python UTF-8 模式。

### 也参考：

`Py_EncodeLocale()` 函式。

在 3.3 版新加入。

在 3.7 版的變更：此函数现在也会为 surrogateescape 错误处理器使用当前语言区域编码格式，但在 Android 上例外。在之前版本中，`Py_EncodeLocale()` 将被用于 surrogateescape，而当前语言区域编码格式将被用于 strict。

## 文件系统编码格式

To encode and decode file names and other environment strings, `Py_FileSystemDefaultEncoding` should be used as the encoding, and `Py_FileSystemDefaultEncodeErrors` should be used as the 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)

属于稳定 ABI. ParseTuple 转换器: 编码 str 对象 -- 直接获取或是通过 `os.PathLike` 接口 -- 使用 `PyUnicode_EncodeFSDefault()` 转为 bytes; bytes 对象将被原样输出。result 必须为 *PyBytesObject*\* 并将在其不再被使用时释放。

在 3.1 版新加入。

在 3.6 版的變更: 接受一个 *path-like object*。

要在参数解析期间将文件名解码为 str, 应当使用 "O&" 转换器, 传入 `PyUnicode_FSDecoder()` 作为转换函数:

**int PyUnicode\_FSDecoder** (*PyObject* \*obj, void \*result)

属于稳定 ABI. ParseTuple 转换器: 解码 bytes 对象 -- 直接获取或是通过 `os.PathLike` 接口间接获取 -- 使用 `PyUnicode_DecodeFSDefaultAndSize()` 转为 str; str 对象将被原样输出。result 必须为 *PyUnicodeObject*\* 并将在其不再被使用时释放。

在 3.2 版新加入。

在 3.6 版的變更: 接受一个 *path-like object*。

*PyObject*\* **PyUnicode\_DecodeFSDefaultAndSize** (const char \*str, *Py\_ssize\_t* size)

回傳值: 新的參照。属于稳定 ABI. 使用 *filesystem encoding and error handler* 解码字符串。

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

`Py_FileSystemDefaultEncoding` is initialized at startup from the locale encoding and cannot be modified later. If you need to decode a string from the current locale encoding, use `PyUnicode_DecodeLocaleAndSize()`.

也参考:

`Py_DecodeLocale()` 函式。

在 3.6 版的變更: Use `Py_FileSystemDefaultEncodeErrors` error handler.

*PyObject*\* **PyUnicode\_DecodeFSDefault** (const char \*str)

回傳值: 新的參照。属于稳定 ABI. 使用 *filesystem encoding and error handler* 解码以空值结尾的字符串。

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

Use `PyUnicode_DecodeFSDefaultAndSize()` if you know the string length.

在 3.6 版的變更: Use `Py_FileSystemDefaultEncodeErrors` error handler.

*PyObject*\* **PyUnicode\_EncodeFSDefault** (*PyObject* \*unicode)

回傳值: 新的參照。属于稳定 ABI. Encode a Unicode object to `Py_FileSystemDefaultEncoding` with the `Py_FileSystemDefaultEncodeErrors` error handler, and return bytes. Note that the resulting bytes object may contain null bytes.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

`Py_FileSystemDefaultEncoding` is initialized at startup from the locale encoding and cannot be modified later. If you need to encode a string to the current locale encoding, use `PyUnicode_EncodeLocale()`.

也参考:

`Py_EncodeLocale()` 函式。

在 3.2 版新加入。

在 3.6 版的變更: Use `Py_FileSystemDefaultEncodeErrors` error handler.

## wchar\_t 支持

在受支持的平台上支持 `wchar_t`:

*PyObject* \***PyUnicode\_FromWideChar** (const `wchar_t` \*wstr, *Py\_ssize\_t* size)

回傳值: 新的參照。屬於穩定 ABI。根據給定的 *size* 的 `wchar_t` 緩衝區 *wstr* 創建一個 Unicode 對象。傳入 -1 作為 *size* 表示該函數必須使用 `wcslen()` 自動計算緩衝區長度。失敗時將返回 NULL。

*Py\_ssize\_t* **PyUnicode\_AsWideChar** (*PyObject* \*unicode, `wchar_t` \*wstr, *Py\_ssize\_t* size)

屬於穩定 ABI。將 Unicode 對象的內容拷貝到 `wchar_t` 緩衝區 *wstr* 中。至多拷貝 *size* 個 `wchar_t` 字符 (不包括可能存在的末尾空結束字符)。返回拷貝的 `wchar_t` 字符數或在出錯時返回 -1。

當 *wstr* 為 NULL 時, 則改為返回存儲包括結束空值在內的所有 *unicode* 內容所需的 *size*。

請注意結果 `wchar_t`\* 字符串可能是以空值結束也可能不是。調用方要負責確保 `wchar_t`\* 字符串以空值結束以防應用有此要求。此外, 請注意 `wchar_t`\* 字符串有可能包含空字符, 這將導致字符串在與大多數 C 函數一起使用時被截斷。

`wchar_t` \***PyUnicode\_AsWideCharString** (*PyObject* \*unicode, *Py\_ssize\_t* \*size)

屬於穩定 ABI 自 3.7 版起。將 Unicode 對象轉換為寬字符串。輸出字符串將總是以空字符結尾。如果 *size* 不為 NULL, 則會將寬字符的數量 (不包括結尾空字符) 寫入到 \**size* 中。請注意結果 `wchar_t` 字符串可能包含空字符, 這將導致在大多數 C 函數中使用時字符串被截斷。如果 *size* 為 NULL 並且 `wchar_t`\* 字符串包含空字符則將引發 `ValueError`。

成功時返回由 `PyMem_New` 分配的緩衝區 (使用 `PyMem_Free()` 來釋放它)。發生錯誤時, 則返回 NULL 並且 \**size* 將是未定義的。如果內存分配失敗則會引發 `MemoryError`。

在 3.2 版新加入。

在 3.7 版的變更: 如果 *size* 為 NULL 且 `wchar_t`\* 字符串包含空字符則會引發 `ValueError`。

## 內置編解碼器

Python 提供了一組以 C 編寫以保證運行速度的內置編解碼器。所有這些編解碼器均可通過下列函數直接使用。

下列 API 大都接受 `encoding` 和 `errors` 兩個參數, 它們具有與在內置 `str()` 字符串對象構造器中同名參數相同的語義。

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 variable `Py_FileSystemDefaultEncoding` internally. This variable should be treated as read-only: on some systems, it will be a pointer to a static string, on others, it will change at run-time (such as when the application invokes `setlocale`).

錯誤處理方式由 `errors` 設置並且也可以設為 NULL 表示使用為編解碼器定義的默認處理方式。所有內置編解碼器的默認錯誤處理方式是 "strict" (會引發 `ValueError`)。

編解碼器都使用類似的接口。為了保持簡單只有與下列泛型編解碼器的差異才會記錄在文檔中。

## 泛型编解码器

以下是泛型编解码器的 API:

**PyObject \*PyUnicode\_Decode** (const char \*str, *Py\_ssize\_t* size, const char \*encoding, const char \*errors)

回傳值: 新的参照。属于稳定 ABI。通过解码已编码字节串 *str* 的 *size* 个字节创建一个 Unicode 对象。*encoding* 和 *errors* 具有与 `str()` 内置函数中同名形参相同的含义。要使用的编解码器将使用 Python 编解码器注册表来查找。如果编解码器引发了异常则返回 NULL。

**PyObject \*PyUnicode\_AsEncodedString** (PyObject \*unicode, const char \*encoding, const char \*errors)

回傳值: 新的参照。属于稳定 ABI。编码一个 Unicode 对象并将结果作为 Python 字节串对象返回。*encoding* 和 *errors* 具有与 Unicode `encode()` 方法中同名形参相同的含义。要使用的编解码器将使用 Python 编解码器注册表来查找。如果编解码器引发了异常则返回 NULL。

## UTF-8 编解码器

以下是 UTF-8 编解码器 API:

**PyObject \*PyUnicode\_DecodeUTF8** (const char \*str, *Py\_ssize\_t* size, const char \*errors)

回傳值: 新的参照。属于稳定 ABI。通过解码 UTF-8 编码的字节串 *str* 的 *size* 个字节创建一个 Unicode 对象。如果编解码器引发了异常则返回 NULL。

**PyObject \*PyUnicode\_DecodeUTF8Stateful** (const char \*str, *Py\_ssize\_t* size, const char \*errors, *Py\_ssize\_t* \*consumed)

回傳值: 新的参照。属于稳定 ABI。如果 *consumed* 为 NULL, 则行为类似于 `PyUnicode_DecodeUTF8()`。如果 *consumed* 不为 NULL, 则末尾的不完整 UTF-8 字节序列将不被视为错误。这些字节将不会被解码并且已被解码的字节数将存储在 *consumed* 中。

**PyObject \*PyUnicode\_AsUTF8String** (PyObject \*unicode)

回傳值: 新的参照。属于稳定 ABI。使用 UTF-8 编码 Unicode 对象并将结果作为 Python 字节串对象返回。错误处理方式为“strict”。如果编解码器引发了异常则将返回 NULL。

**const char \*PyUnicode\_AsUTF8AndSize** (PyObject \*unicode, *Py\_ssize\_t* \*size)

属于稳定 ABI 自 3.10 版起。返回一个指向 Unicode 对象的 UTF-8 编码格式数据的指针, 并将已编码数据的大小 (以字节为单位) 存储在 *size* 中。*size* 参数可以为 NULL; 在此情况下数据的大小将不会被存储。返回的缓冲区总是会添加一个额外的空字节 (不包括在 *size* 中), 无论是否存在任何其他的空码位。

在发生错误的情况下, 将返回 NULL 附带设置一个异常并且不会存储 *size* 值。

这将缓存 Unicode 对象中字符串的 UTF-8 表示形式, 并且后续调用将返回指向同一缓存区的指针。调用方不必负责释放该缓冲区。缓冲区会在 Unicode 对象被作为垃圾回收时被释放并使指向它的指针失效。

在 3.3 版新加入。

在 3.7 版的變更: 返回类型现在是 `const char *` 而不是 `char *`。

在 3.10 版的變更: 此函数是受限 API 的组成部分。

**const char \*PyUnicode\_AsUTF8** (PyObject \*unicode)

类似于 `PyUnicode_AsUTF8AndSize()`, 但不会存储大小值。

在 3.3 版新加入。

在 3.7 版的變更: 返回类型现在是 `const char *` 而不是 `char *`。



## UTF-32 编解码器

以下是 UTF-32 编解码器 API:

*PyObject* **\*PyUnicode\_DecodeUTF32** (const char \*str, *Py\_ssize\_t* size, const char \*errors, int \*byteorder)

回傳值: 新的參照。属于稳定 ABI。从 UTF-32 编码的缓冲区数据解码 *size* 个字节并返回相应的 Unicode 对象。*errors* (如果不为 NULL) 定义了错误处理方式。默认为“strict”。

如果 *byteorder* 不为 NULL, 解码器将使用给定的字节序进行解码:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

如果 *\*byteorder* 为零, 且输入数据的前四个字节为字节序标记 (BOM), 则解码器将切换为该字节序并且 BOM 将不会被拷贝到结果 Unicode 字符串中。如果 *\*byteorder* 为 -1 或 1, 则字节序标记会被拷贝到输出中。

在完成后, *\*byteorder* 将在输入数据的末尾被设为当前字节序。

如果 *byteorder* 为 NULL, 编解码器将使用本机字节序。

如果编解码器引发了异常则返回 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。属于稳定 ABI。如果 *consumed* 为 NULL, 则行为类似于 *PyUnicode\_DecodeUTF32()*。如果 *consumed* 不为 NULL, 则 *PyUnicode\_DecodeUTF32Stateful()* 将不把末尾的不完整 UTF-32 字节序列 (如字节数不可被四整除) 视为错误。这些字节将不会被解码并且已被解码的字节数将存储在 *consumed* 中。

*PyObject* **\*PyUnicode\_AsUTF32String** (*PyObject* \*unicode)

回傳值: 新的參照。属于稳定 ABI。返回使用 UTF-32 编码格式本机字节序的 Python 字节串。字节串将总是以 BOM 标记打头。错误处理方式为“strict”。如果编解码器引发了异常则返回 NULL。

## UTF-16 编解码器

以下是 UTF-16 编解码器的 API:

*PyObject* **\*PyUnicode\_DecodeUTF16** (const char \*str, *Py\_ssize\_t* size, const char \*errors, int \*byteorder)

回傳值: 新的參照。属于稳定 ABI。从 UTF-16 编码的缓冲区数据解码 *size* 个字节并返回相应的 Unicode 对象。*errors* (如果不为 NULL) 定义了错误处理方式。默认为“strict”。

如果 *byteorder* 不为 NULL, 解码器将使用给定的字节序进行解码:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

如果 *\*byteorder* 为零, 且输入数据的前两个字节为字节序标记 (BOM), 则解码器将切换为该字节序并且 BOM 将不会被拷贝到结果 Unicode 字符串中。如果 *\*byteorder* 为 -1 或 1, 则字节序标记会被拷贝到输出中 (它将是一个 \ufeff 或 \ufffe 字符)。

在完成后, *\*byteorder* 将在输入数据的末尾被设为当前字节序。

如果 *byteorder* 为 NULL, 编解码器将使用本机字节序。

如果编解码器引发了异常则返回 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。屬於穩定 ABI。如果 *consumed* 為 NULL，則行為類似於 *PyUnicode\_DecodeUTF16()*。如果 *consumed* 不為 NULL，則 *PyUnicode\_DecodeUTF16Stateful()* 將不把末尾的不完整 UTF-16 字节序列（如為奇數个字节或為分開的替代對）視為錯誤。這些字节將不會被解碼並且已被解碼的字节數將存儲在 *consumed* 中。

*PyObject* \*PyUnicode\_AsUTF16String (*PyObject* \*unicode)

回傳值：新的參照。屬於穩定 ABI。返回使用 UTF-16 編碼格式本机字节序的 Python 字节串。字节串將总是以 BOM 标记打頭。錯誤處理方式為“strict”。如果編解碼器引發了異常則返回 NULL。

## UTF-7 編解碼器

以下是 UTF-7 編解碼器 API:

*PyObject* \*PyUnicode\_DecodeUTF7 (const char \*str, *Py\_ssize\_t* size, const char \*errors)

回傳值：新的參照。屬於穩定 ABI。通過解碼 UTF-7 編碼的字节串 *str* 的 *size* 个字节創建一個 Unicode 對象。如果編解碼器引發了異常則返回 NULL。

*PyObject* \*PyUnicode\_DecodeUTF7Stateful (const char \*str, *Py\_ssize\_t* size, const char \*errors, *Py\_ssize\_t* \*consumed)

回傳值：新的參照。屬於穩定 ABI。如果 *consumed* 為 NULL，則行為類似於 *PyUnicode\_DecodeUTF7()*。如果 *consumed* 不為 NULL，則末尾的不完整 UTF-7 base-64 部分將不被視為錯誤。這些字节將不會被解碼並且已被解碼的字节數將存儲在 *consumed* 中。

## Unicode-Escape 編解碼器

以下是“Unicode Escape”編解碼器的 API:

*PyObject* \*PyUnicode\_DecodeUnicodeEscape (const char \*str, *Py\_ssize\_t* size, const char \*errors)

回傳值：新的參照。屬於穩定 ABI。通過解碼 Unicode-Escape 編碼的字节串 *str* 的 *size* 个字节創建一個 Unicode 對象。如果編解碼器引發了異常則返回 NULL。

*PyObject* \*PyUnicode\_AsUnicodeEscapeString (*PyObject* \*unicode)

回傳值：新的參照。屬於穩定 ABI。使用 Unicode-Escape 編碼 Unicode 對象並將結果作為字节串對象返回。錯誤處理方式為“strict”。如果編解碼器引發了異常則將返回 NULL。

## Raw-Unicode-Escape 編解碼器

以下是“Raw Unicode Escape”編解碼器的 API:

*PyObject* \*PyUnicode\_DecodeRawUnicodeEscape (const char \*str, *Py\_ssize\_t* size, const char \*errors)

回傳值：新的參照。屬於穩定 ABI。通過解碼 Raw-Unicode-Escape 編碼的字节串 *str* 的 *size* 个字节創建一個 Unicode 對象。如果編解碼器引發了異常則返回 NULL。

*PyObject* \*PyUnicode\_AsRawUnicodeEscapeString (*PyObject* \*unicode)

回傳值：新的參照。屬於穩定 ABI。使用 Raw-Unicode-Escape 編碼 Unicode 對象並將結果作為字节串對象返回。錯誤處理方式為“strict”。如果編解碼器引發了異常則將返回 NULL。

## Latin-1 编解码器

以下是 Latin-1 编解码器的 API: Latin-1 对应于前 256 个 Unicode 码位且编码器在编码期间只接受这些码位。

*PyObject* \*PyUnicode\_DecodeLatin1 (const char \*str, *Py\_ssize\_t* size, const char \*errors)

回傳值: 新的参照。属于稳定 ABI。通过解码 Latin-1 编码的字节串 *str* 的 *size* 个字节创建一个 Unicode 对象。如果编解码器引发了异常则返回 NULL。

*PyObject* \*PyUnicode\_AsLatin1String (*PyObject* \*unicode)

回傳值: 新的参照。属于稳定 ABI。使用 Latin-1 编码 Unicode 对象并将结果作为 Python 字节串对象返回。错误处理方式为“strict”。如果编解码器引发了异常则将返回 NULL。

## ASCII 编解码器

以下是 ASCII 编解码器的 API。只接受 7 位 ASCII 数据。任何其他编码的数据都将导致错误。

*PyObject* \*PyUnicode\_DecodeASCII (const char \*str, *Py\_ssize\_t* size, const char \*errors)

回傳值: 新的参照。属于稳定 ABI。通过解码 ASCII 编码的字节串 *str* 的 *size* 个字节创建一个 Unicode 对象。如果编解码器引发了异常则返回 NULL。

*PyObject* \*PyUnicode\_AsASCIIString (*PyObject* \*unicode)

回傳值: 新的参照。属于稳定 ABI。使用 ASCII 编码 Unicode 对象并将结果作为 Python 字节串对象返回。错误处理方式为“strict”。如果编解码器引发了异常则将返回 NULL。

## 字符映射编解码器

此编解码器的特殊之处在于它可被用来实现许多不同的编解码器 (而且这实际上就是包括在 `encodings` 包中的大部分标准编解码器的实现方式)。此编解码器使用映射来编码和解码字符。所提供的映射对象必须支持 `__getitem__()` 映射接口; 字典和序列均可胜任。

以下是映射编解码器的 API:

*PyObject* \*PyUnicode\_DecodeCharmap (const char \*str, *Py\_ssize\_t* length, *PyObject* \*mapping, const char \*errors)

回傳值: 新的参照。属于稳定 ABI。通过使用给定的 *mapping* 对象解码已编码字节串 *str* 的 *size* 个字节创建一个 Unicode 对象。如果编解码器引发了异常则返回 NULL。

如果 *mapping* 为 NULL, 则将应用 Latin-1 编码格式。否则 *mapping* 必须为字节码位值 (0 至 255 范围内的整数) 到 Unicode 字符串的映射、整数 (将被解读为 Unicode 码位) 或 None。未映射的数据字节 -- 这样的数据将导致 `LookupError`, 以及被映射到 None 的数据, `0xFFFE` 或 `'\ufffe'`, 将被视为未定义的映射并导致报错。

*PyObject* \*PyUnicode\_AsCharmapString (*PyObject* \*unicode, *PyObject* \*mapping)

回傳值: 新的参照。属于稳定 ABI。使用给定的 *mapping* 对象编码 Unicode 对象并将结果作为字节串对象返回。错误处理方式为“strict”。如果编解码器引发了异常则将返回 NULL。

*mapping* 对象必须将整数 Unicode 码位映射到字节串对象、0 至 255 范围内的整数或 None。未映射的字符码位 (将导致 `LookupError` 的数据) 以及映射到 None 的数据将被视为“未定义的映射”并导致报错。

以下特殊的编解码器 API 会将 Unicode 映射至 Unicode。

*PyObject* \*PyUnicode\_Translate (*PyObject* \*unicode, *PyObject* \*table, const char \*errors)

回傳值: 新的参照。属于稳定 ABI。通过应用字符映射表来转写字符串并返回结果 Unicode 对象。如果编解码器引发了异常则返回 NULL。

字符映射表必须将整数 Unicode 码位映射到整数 Unicode 码位或 None (这将删除相应的字符)。

映射表只需提供 `__getitem__()` 接口; 字典和序列均可胜任。未映射的字符码位 (将导致 `LookupError` 的数据) 将保持不变并被原样拷贝。

*errors* 具有用于编解码器的通常含义。它可以为 `NULL` 表示使用默认的错误处理方式。

## Windows 中的 MBCS 编解码器

以下是 MBCS 编解码器的 API。目前它们仅在 Windows 中可用并使用 Win32 MBCS 转换器来实现转换。请注意 MBCS（或 DBCS）是一类编码格式，而非只有一个。目标编码格式是由运行编解码器的机器上的用户设置定义的。

***PyObject* \*PyUnicode\_DecodeMBCS** (const char \*str, *Py\_ssize\_t* size, const char \*errors)

回傳值：新的參照。属于稳定 ABI on Windows 自 3.7 版起。通过解码 MBCS 编码的字节串 *str* 的 *size* 个字节创建一个 Unicode 对象。如果编解码器引发了异常则返回 `NULL`。

***PyObject* \*PyUnicode\_DecodeMBCSStateful** (const char \*str, *Py\_ssize\_t* size, const char \*errors, *Py\_ssize\_t* \*consumed)

回傳值：新的參照。属于稳定 ABI on Windows 自 3.7 版起。如果 *consumed* 为 `NULL`，则行为类似于 `PyUnicode_DecodeMBCS()`。如果 *consumed* 不为 `NULL`，则 `PyUnicode_DecodeMBCSStateful()` 将不会解码末尾的不完整字节并且已被解码的字节数将存储在 *consumed* 中。

***PyObject* \*PyUnicode\_AsMBCSString** (*PyObject* \*unicode)

回傳值：新的參照。属于稳定 ABI on Windows 自 3.7 版起。使用 MBCS 编码 Unicode 对象并将结果作为 Python 字节串对象返回。错误处理方式为“strict”。如果编解码器引发了异常则将返回 `NULL`。

***PyObject* \*PyUnicode\_EncodeCodePage** (int code\_page, *PyObject* \*unicode, const char \*errors)

回傳值：新的參照。属于稳定 ABI on Windows 自 3.7 版起。使用指定的代码页编码 Unicode 对象并返回一个 Python 字节串对象。如果编解码器引发了异常则返回 `NULL`。使用 `CP_ACP` 代码页来获取 MBCS 解码器。

在 3.3 版新加入。

## 方法和槽位

### 方法与槽位函数

以下 API 可以处理输入的 Unicode 对象和字符串（在描述中我们称其为字符串）并返回适当的 Unicode 对象或整数值。

如果发生异常它们都将返回 `NULL` 或 `-1`。

***PyObject* \*PyUnicode\_Concat** (*PyObject* \*left, *PyObject* \*right)

回傳值：新的參照。属于稳定 ABI。拼接两个字符串得到一个新的 Unicode 字符串。

***PyObject* \*PyUnicode\_Split** (*PyObject* \*unicode, *PyObject* \*sep, *Py\_ssize\_t* maxsplit)

回傳值：新的參照。属于稳定 ABI。拆分一个字符串得到一个 Unicode 字符串的列表。如果 *sep* 为 `NULL`，则将根据空格来拆分所有子字符串。否则，将根据指定的分隔符来拆分。最多拆分数为 *maxsplit*。如为负值，则没有限制。分隔符不包括在结果列表中。

***PyObject* \*PyUnicode\_Splitlines** (*PyObject* \*unicode, int keepends)

回傳值：新的參照。属于稳定 ABI。根据分行符来拆分 Unicode 字符串，返回一个 Unicode 字符串的列表。CRLF 将被视为一个分行符。如果 *keepends* 为 0，则行分隔符将不包括在结果字符串中。

***PyObject* \*PyUnicode\_Join** (*PyObject* \*separator, *PyObject* \*seq)

回傳值：新的參照。属于稳定 ABI。使用给定的 *separator* 合并一个字符串列表并返回结果 Unicode 字符串。

***Py\_ssize\_t* PyUnicode\_Tailmatch** (*PyObject* \*unicode, *PyObject* \*substr, *Py\_ssize\_t* start, *Py\_ssize\_t* end, int direction)

属于稳定 ABI。如果 *substr* 在给定的端点 (*direction* == -1 表示前缀匹配, *direction* == 1 表示后缀匹配) 与 `unicode[start:end]` 相匹配则返回 1，否则返回 0。如果发生错误则返回 -1。

*Py\_ssize\_t* **PyUnicode\_Find** (*PyObject* \*unicode, *PyObject* \*substr, *Py\_ssize\_t* start, *Py\_ssize\_t* end, int direction)

属于稳定 ABI。返回使用给定的 *direction* (*direction* == 1 表示前向搜索, *direction* == -1 表示后向搜索) 时 *substr* 在 `unicode[start:end]` 中首次出现的位置。返回值为首个匹配的索引号; 值为 -1 表示未找到匹配, -2 则表示发生了错误并设置了异常。

*Py\_ssize\_t* **PyUnicode\_FindChar** (*PyObject* \*unicode, *Py\_UCS4* ch, *Py\_ssize\_t* start, *Py\_ssize\_t* end, int direction)

属于稳定 ABI 自 3.7 版起。返回使用给定的 *direction* (*direction* == 1 表示前向搜索, *direction* == -1 表示后向搜索) 时字符 *ch* 在 `unicode[start:end]` 中首次出现的位置。返回值为首个匹配的索引号; 值为 -1 表示未找到匹配, -2 则表示发生错误并设置了异常。

在 3.3 版新加入。

在 3.7 版的變更: 现在 *start* 和 *end* 被调整为与 `unicode[start:end]` 类似的行为。

*Py\_ssize\_t* **PyUnicode\_Count** (*PyObject* \*unicode, *PyObject* \*substr, *Py\_ssize\_t* start, *Py\_ssize\_t* end)

属于稳定 ABI。返回 *substr* 在 `unicode[start:end]` 中不重叠出现的次数。如果发生错误则返回 -1。

*PyObject* \***PyUnicode\_Replace** (*PyObject* \*unicode, *PyObject* \*substr, *PyObject* \*replstr, *Py\_ssize\_t* maxcount)

回傳值: 新的參照。属于稳定 ABI。将 *unicode* 中 *substr* 替换为 *replstr* 至多 *maxcount* 次并返回结果 Unicode 对象。*maxcount* == -1 表示全部替换。

int **PyUnicode\_Compare** (*PyObject* \*left, *PyObject* \*right)

属于稳定 ABI。比较两个字符串并返回 -1, 0, 1 分别表示小于、等于和大于。

此函数执行失败时返回 -1, 因此应当调用 *PyErr\_Occurred()* 来检查错误。

int **PyUnicode\_CompareWithASCIIString** (*PyObject* \*unicode, const char \*string)

属于稳定 ABI。将 Unicode 对象 *unicode* 与 *string* 进行比较并返回 -1, 0, 1 分别表示小于、等于和大于。最好只传入 ASCII 编码的字符串, 但如果输入字符串包含非 ASCII 字符则此函数会将其按 ISO-8859-1 编码格式来解读。

此函数不会引发异常。

*PyObject* \***PyUnicode\_RichCompare** (*PyObject* \*left, *PyObject* \*right, int op)

回傳值: 新的參照。属于稳定 ABI。对两个 Unicode 字符串执行富比较并返回以下值之一:

- NULL 用于引发了异常的情况
- *Py\_True* 或 *Py\_False* 用于成功完成比较的情况
- *Py\_NotImplemented* 用于类型组合未知的情况

可能的 *op* 值有 *Py\_GT*, *Py\_GE*, *Py\_EQ*, *Py\_NE*, *Py\_LT*, 和 *Py\_LE*。

*PyObject* \***PyUnicode\_Format** (*PyObject* \*format, *PyObject* \*args)

回傳值: 新的參照。属于稳定 ABI。根据 *format* 和 *args* 返回一个新的字符串对象; 这等同于 `format % args`。

int **PyUnicode\_Contains** (*PyObject* \*unicode, *PyObject* \*substr)

属于稳定 ABI。检查 *substr* 是否包含在 *unicode* 中并相应返回真值或假值。

*substr* 必须强制转为一个单元素 Unicode 字符串。如果发生错误则返回 -1。

void **PyUnicode\_InternInPlace** (*PyObject* \*\*p\_unicode)

属于稳定 ABI。原地内部化参数 *\*p\_unicode*。该参数必须是一个指向 Python Unicode 字符串对象的指针变量的地址。如果已存在与 *\*p\_unicode* 相同的内部化字符串, 则将其设为 *\*p\_unicode* (释放对旧字符串的引用并新建一个指向内部化字符串对象的 *strong reference*), 否则将保持 *\*p\_unicode* 不变并将其内部化 (新建一个 *strong reference*)。 (澄清说明: 虽然这里大量提及了引用, 但请将此函数视为引用无关的; 当且仅当你在调用之前就已拥有该对象时你才会在调用之后也拥有它。)



*PyObject* \*PyUnicode\_InternFromString (const char \*str)

回傳值：新的參照。屬於穩定 ABI。PyUnicode\_FromString() 和 PyUnicode\_InternInPlace() 的組合操作，返回一個已內部化的新 Unicode 字符串對象，或一個指向具有相同值的原有內部化字符串對象的新的（“擁有的”）引用。

### 8.3.4 元組 (Tuple) 物件

type **PyTupleObject**

這個 *PyObject* 的子類型代表一個 Python 的元組對象。

*PyTypeObject* **PyTuple\_Type**

屬於穩定 ABI。PyTypeObject 的實例代表一個 Python 元組類型，這與 Python 層面的 tuple 是相同的對象。

int **PyTuple\_Check** (*PyObject* \*p)

如果 *p* 是一個 tuple 對象或者 tuple 類型的子類型的實例則返回真值。此函數總是會成功執行。

int **PyTuple\_CheckExact** (*PyObject* \*p)

如果 *p* 是一個 tuple 對象但不是 tuple 類型的子類型的實例則返回真值。此函數總是會成功執行。

*PyObject* \*PyTuple\_New (*Py\_ssize\_t* len)

回傳值：新的參照。屬於穩定 ABI。成功時返回一個新的元組對象，長度為 *len*，失敗時返回 NULL。

*PyObject* \*PyTuple\_Pack (*Py\_ssize\_t* n, ...)

回傳值：新的參照。屬於穩定 ABI。成功時返回一個新的元組對象，大小為 *n*，失敗時返回 NULL。元組值初始化為指向 Python 對象的後續 *n* 個 C 參數。PyTuple\_Pack(2, *a*, *b*) 和 Py\_BuildValue("(OO)", *a*, *b*) 相等。

*Py\_ssize\_t* **PyTuple\_Size** (*PyObject* \*p)

屬於穩定 ABI。獲取指向元組對象的指針，並返回該元組的大小。

*Py\_ssize\_t* **PyTuple\_GET\_SIZE** (*PyObject* \*p)

返回元組 *p* 的大小，它必須為非 NULL 並且指向一個元組；不執行錯誤檢查。

*PyObject* \*PyTuple\_GetItem (*PyObject* \*p, *Py\_ssize\_t* pos)

回傳值：借用參照。屬於穩定 ABI。返回 *p* 所指向上的元組中位於 *pos* 處的對象。如果 *pos* 為負值或超出範圍，則返回 NULL 並設置一個 IndexError 異常。

*PyObject* \*PyTuple\_GET\_ITEM (*PyObject* \*p, *Py\_ssize\_t* pos)

回傳值：借用參照。類似於 *PyTuple\_GetItem()*，但不檢查其參數。

*PyObject* \*PyTuple\_GetSlice (*PyObject* \*p, *Py\_ssize\_t* low, *Py\_ssize\_t* high)

回傳值：新的參照。屬於穩定 ABI。返回 *p* 所指向上的元組的切片，在 *low* 和 *high* 之間，或者在失敗時返回 NULL。這等同於 Python 表达式 *p*[*low*:*high*]。不支持從列表末尾索引。

int **PyTuple\_SetItem** (*PyObject* \*p, *Py\_ssize\_t* pos, *PyObject* \*o)

屬於穩定 ABI。在 *p* 指向上的元組的 *pos* 位置插入對對象 *o* 的引用。成功時返回 0；如果 *pos* 越界，則返回 -1，並拋出一個 IndexError 異常。

---

備註：此函數會“竊取”對 *o* 的引用，並丟棄對元組中已在受影響位置的條目的引用。

---

void **PyTuple\_SET\_ITEM** (*PyObject* \*p, *Py\_ssize\_t* pos, *PyObject* \*o)

類似於 *PyTuple\_SetItem()*，但不進行錯誤檢查，並且應該只是被用來填充全新的元組。

---

備註：這個函數會“竊取”一個對 *o* 的引用，但是，不與 *PyTuple\_SetItem()* 不同，它不會丟棄對任何被替換項的引用；元組中位於 *pos* 位置的任何引用都將被洩漏。

---

**int `_PyTuple_Resize` (*PyObject* \*\*p, *Py\_ssize\_t* newsize)**

可以用于调整元组的大小。*newsize* 将是元组的新长度。因为元组被认为是不可变的，所以只有在对象仅有一个引用时，才应该使用它。如果元组已经被代码的其他部分所引用，请不要使用此项。元组在最后总是会增长或缩小。把它看作是销毁旧元组并创建一个新元组，只会更有效。成功时返回 0。客户端代码不应假定 \*p 的结果值将与调用此函数之前的值相同。如果替换了 \*p 引用的对象，则原始的 \*p 将被销毁。失败时，返回 -1，将 \*p 设置为 NULL，并引发 `MemoryError` 或者 `SystemError`。

### 8.3.5 结构序列对象

结构序列对象是等价于 `namedtuple()` 的 C 对象，即一个序列，其中的条目也可以通过属性访问。要创建结构序列，你首先必须创建特定的结构序列类型。

***PyObject* \*`PyStructSequence_NewType` (*PyStructSequence\_Desc* \*desc)**

回傳值：新的參照。属于稳定 ABI。根据 *desc* 中的数据创建一个新的结构序列类型，如下所述。可以使用 `PyStructSequence_New()` 创建结果类型的实例。

**void `PyStructSequence_InitType` (*PyTypeObject* \*type, *PyStructSequence\_Desc* \*desc)**

从 *desc* 就地初始化结构序列类型 *type*。

**int `PyStructSequence_InitType2` (*PyTypeObject* \*type, *PyStructSequence\_Desc* \*desc)**

与 `PyStructSequence_InitType` 相同，但成功时返回 0，失败时返回 -1。

在 3.4 版新加入。

**type `PyStructSequence_Desc`**

属于稳定 ABI（包括所有成员）。包含要创建的结构序列类型的元信息。

**const char \*`name`**

结构序列类型的名称。

**const char \*`doc`**

指向类型的文档字符串的指针或以 NULL 表示忽略。

***PyStructSequence\_Field* \*`fields`**

指向以 NULL 结尾的数组的指针，该数组包含新类型的字段名。

**int `n_in_sequence`**

Python 端可见的字段数（如果用作元组）。

**type `PyStructSequence_Field`**

属于稳定 ABI（包括所有成员）。描述结构序列的一个字段。由于结构序列是以元组为模型的，因此所有字段的类型都是 *PyObject*\*。 *PyStructSequence\_Desc* 的 *fields* 数组中的索引决定了描述结构序列的是哪个字段。

**const char \*`name`**

字段的名称或 NULL 表示结束已命名字段列表，设为 *PyStructSequence\_UnnamedField* 则保持未命名状态。

**const char \*`doc`**

字段文档字符串或 NULL 表示省略。

**const char \*const `PyStructSequence_UnnamedField`**

属于稳定 ABI 自 3.11 版起。字段名的特殊值将保持未命名状态。

在 3.9 版的變更：这个类型已从 `char *` 更改。

***PyObject* \*`PyStructSequence_New` (*PyTypeObject* \*type)**

回傳值：新的參照。属于稳定 ABI。创建 *type* 的实例，该实例必须使用 `PyStructSequence_NewType()` 创建。



*PyObject* \*PyStructSequence\_GetItem(*PyObject* \*p, *Py\_ssize\_t* pos)

回傳值：借用參照。属于稳定 ABI。返回 *p* 所指向的结构序列中，位于 *pos* 处的对象。不需要进行边界检查。

*PyObject* \*PyStructSequence\_GET\_ITEM(*PyObject* \*p, *Py\_ssize\_t* pos)

回傳值：借用參照。PyStructSequence\_GetItem() 的宏版本。

void PyStructSequence\_SetItem(*PyObject* \*p, *Py\_ssize\_t* pos, *PyObject* \*o)

属于稳定 ABI。将结构序列 *p* 的索引 *pos* 处的字段设置为值 *o*。与 PyTuple\_SET\_ITEM() 一样，它应该只用于填充全新的实例。

---

備註：这个函数“窃取”了指向 *o* 的一个引用。

---

void PyStructSequence\_SET\_ITEM(*PyObject* \*p, *Py\_ssize\_t* \*pos, *PyObject* \*o)

类似于 PyStructSequence\_SetItem()，但是被实现为一个静态内联函数。

---

備註：这个函数“窃取”了指向 *o* 的一个引用。

---

### 8.3.6 List（串列）物件

type *PyListObject*

这个 C 类型 *PyObject* 的子类型代表一个 Python 列表对象。

*PyTypeObject* *PyList\_Type*

属于稳定 ABI。这是个属于 *PyTypeObject* 的代表 Python 列表类型的实例。在 Python 层面和类型 *list* 是同一个对象。

int *PyList\_Check*(*PyObject* \*p)

如果 *p* 是一个 *list* 物件或者是 *list* 型 的实例，就回傳 true。這個函式永遠會成功執行。

int *PyList\_CheckExact*(*PyObject* \*p)

如果 *p* 是一个 *list* 物件但不是 *list* 型 的子型 的实例，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。属于稳定 ABI。成功时返回一个长度为 *len* 的新列表，失败时返回 NULL。

---

備註：当 *len* 大于零时，被返回的列表对象项目被设成 NULL。因此你不能用类似 C 函数 *PySequence\_SetItem*() 的抽象 API 或者用 C 函数 *PyList\_SetItem*() 将所有项目设置成真实对象前对 Python 代码公开这个对象。

---

*Py\_ssize\_t* *PyList\_Size*(*PyObject* \*list)

属于稳定 ABI。返回 *list* 中列表对象的长度；这等于在列表对象调用 *len(list)*。

*Py\_ssize\_t* *PyList\_GET\_SIZE*(*PyObject* \*list)

與 *PyList\_Size*() 類似，但 有錯誤檢查。

*PyObject* \*PyList\_GetItem(*PyObject* \*list, *Py\_ssize\_t* index)

回傳值：借用參照。属于稳定 ABI。返回 *list* 所指向列表中 *index* 位置上的对象。位置值必须为非负数；不支持从列表末尾进行索引。如果 *index* 超出边界 (*<0* or *>=len(list)*)，则返回 NULL 并设置 *IndexError* 异常。

*PyObject* \*PyList\_GET\_ITEM(*PyObject* \*list, *Py\_ssize\_t* i)

回傳值：借用參照。Similar to *PyList\_GetItem*(), but without error checking.

`int PyList_SetItem(PyObject *list, Py_ssize_t index, PyObject *item)`

属于稳定 ABI。将列表中索引为 *index* 的项设为 *item*。成功时返回 0。如果 *index* 超出范围则返回 -1 并设定 `IndexError` 异常。

備註：此函数会“偷走”一个对 *item* 的引用并丢弃一个对列表中受影响位置上的已有条目的引用。

`void PyList_SET_ITEM(PyObject *list, Py_ssize_t i, PyObject *o)`

不带错误检测的宏版本 `PyList_SetItem()`。这通常只被用于新列表中之前没有内容的位置进行填充。

備註：该宏会“偷走”一个对 *item* 的引用，但与 `PyList_SetItem()` 不同的是它不会丢弃对被替换条目的引用；在 *list* 的 *i* 位置上的任何引用都将被泄露。

`int PyList_Insert(PyObject *list, Py_ssize_t index, PyObject *item)`

属于稳定 ABI。将条目 *item* 插入到列表 *list* 索引号 *index* 之前的位置。如果成功将返回 0；如果不成功则返回 -1 并设置一个异常。相当于 `list.insert(index, item)`。

`int PyList_Append(PyObject *list, PyObject *item)`

属于稳定 ABI。将对象 *item* 添加到列表 *list* 的末尾。如果成功将返回 0；如果不成功则返回 -1 并设置一个异常。相当于 `list.append(item)`。

`PyObject *PyList_GetSlice(PyObject *list, Py_ssize_t low, Py_ssize_t high)`

回傳值：新的参照。属于稳定 ABI。返回一个对象列表，包含 *list* 当中位于 *low* 和 *high* 之间的对象。如果不成功则返回 `NULL` 并设置异常。相当于 `list[low:high]`。不支持从列表末尾进行索引。

`int PyList_SetSlice(PyObject *list, Py_ssize_t low, Py_ssize_t high, PyObject *itemlist)`

属于稳定 ABI。将 *list* 当中 *low* 与 *high* 之间的切片设为 *itemlist* 的内容。相当于 `list[low:high] = itemlist`。*itemlist* 可以为 `NULL`，表示赋值为一个空列表（删除切片）。成功时返回 0，失败时返回 -1。这里不支持从列表末尾进行索引。

`int PyList_Sort(PyObject *list)`

属于稳定 ABI。对 *list* 中的条目进行原地排序。成功时返回 0，失败时返回 -1。这等价于 `list.sort()`。

`int PyList_Reverse(PyObject *list)`

属于稳定 ABI。对 *list* 中的条目进行原地反转。成功时返回 0，失败时返回 -1。这等价于 `list.reverse()`。

`PyObject *PyList_AsTuple(PyObject *list)`

回傳值：新的参照。属于稳定 ABI。返回一个新的元组对象，其中包含 *list* 的内容；等价于 `tuple(list)`。

## 8.4 容器物件

### 8.4.1 字典物件

`type PyDictObject`

`PyObject` 子型態代表一个 Python 字典物件。

`PyTypeObject PyDict_Type`

属于稳定 ABI。`PyTypeObject` 实例代表一个 Python 字典类型。此与 Python 层中的 `dict` 同一物件。

`int PyDict_Check(PyObject *p)`

若 *p* 是一个字典物件或字典的子型態实例则会回傳 `true`。此函数式每次都会执行成功。

`int PyDict_CheckExact (PyObject *p)`

若  $p$  是一個字典物件但  $\nabla$  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。属于穩定 ABI。返回一个新的空字典，失败时返回 `NULL`。

`PyObject *PyDictProxy_New (PyObject *mapping)`

回傳值：新的參照。属于穩定 ABI。返回 `types.MappingProxyType` 对象，用于强制执行只读行为的映射。这通常用于创建视图以防止修改非动态类类型的字典。

`void PyDict_Clear (PyObject *p)`

属于穩定 ABI。清空现有字典的所有键值对。

`int PyDict_Contains (PyObject *p, PyObject *key)`

属于穩定 ABI。确定  $key$  是否包含在字典  $p$  中。如果  $key$  匹配上  $p$  的某一项，则返回 `1`，否则返回 `0`。返回 `-1` 表示出错。这等同于 Python 表达式 `key in p`。

`PyObject *PyDict_Copy (PyObject *p)`

回傳值：新的參照。属于穩定 ABI。返回与  $p$  包含相同键值对的新字典。

`int PyDict_SetItem (PyObject *p, PyObject *key, PyObject *val)`

属于穩定 ABI。使用  $key$  作为键将  $val$  插入字典  $p$ 。 $key$  必须为 *hashable*；如果不是，则将引发 `TypeError`。成功时返回 `0`，失败时返回 `-1`。此函数 不会附带对  $val$  的引用。

`int PyDict_SetItemString (PyObject *p, const char *key, PyObject *val)`

属于穩定 ABI。这与 `PyDict_SetItem()` 相同，但  $key$  被指定为 `const char*` UTF-8 编码的字节串，而不是 `PyObject*`。

`int PyDict_DelItem (PyObject *p, PyObject *key)`

属于穩定 ABI。移除字典  $p$  中键为  $key$  的条目。 $key$  必须是 *hashable*；如果不是，则会引发 `TypeError`。如果字典中没有  $key$ ，则会引发 `KeyError`。成功时返回 `0` 或者失败时返回 `-1`。

`int PyDict_DelItemString (PyObject *p, const char *key)`

属于穩定 ABI。这与 `PyDict_DelItem()` 相同，但  $key$  被指定为 `const char*` UTF-8 编码的字节串，而不是 `PyObject*`。

`PyObject *PyDict_GetItem (PyObject *p, PyObject *key)`

回傳值：借用參照。属于穩定 ABI。从字典  $p$  中返回以  $key$  为键的对象。如果键名  $key$  不存在但没有设置一个异常则返回 `NULL`。

---

備 註： 在调用 `__hash__()` 和 `__eq__()` 方法时发生的异常将被静默地忽略。建议改用 `PyDict_GetItemWithError()` 函数。

---

在 3.10 版的變更：在不保持 *GIL* 的情况下调用此 API 曾因历史原因而被允许。现在已不再被允许。

`PyObject *PyDict_GetItemWithError (PyObject *p, PyObject *key)`

回傳值：借用參照。属于穩定 ABI。`PyDict_GetItem()` 的变种，它不会屏蔽异常。当异常发生时将返回 `NULL` 并且设置一个异常。如果键不存在则返回 `NULL` 并且不会设置一个异常。

`PyObject *PyDict_GetItemString (PyObject *p, const char *key)`

回傳值：借用參照。属于穩定 ABI。这与 `PyDict_GetItem()` 一样，但  $key$  是由一个 `const char*` UTF-8 编码的字节串来指定的，而不是 `PyObject*`。

---

備 註： 在调用 `__hash__()` 和 `__eq__()` 方法时或者在创建临时 `str` 对象期间发生的异常将被静默地忽略。建议改用 `PyDict_GetItemWithError()` 函数并附帶你自己的 `PyUnicode_FromString()`  $key$ 。

---

*PyObject* \*PyDict\_SetDefault (*PyObject* \*p, *PyObject* \*key, *PyObject* \*defaultobj)

回傳值：借用參照。這跟 Python 層面的 `dict.setdefault()` 一樣。如果鍵 *key* 存在，它返回在字典 *p* 里面對應的值。如果鍵不存在，它會和值 *defaultobj* 一起插入並返回 *defaultobj*。這個函數只計算 *key* 的哈希函數一次，而不是在查找和插入時分別計算它。

在 3.4 版新加入。

*PyObject* \*PyDict\_Items (*PyObject* \*p)

回傳值：新的參照。屬於穩定 ABI。返回一個包含字典中所有鍵值項的 *PyListObject*。

*PyObject* \*PyDict\_Keys (*PyObject* \*p)

回傳值：新的參照。屬於穩定 ABI。返回一個包含字典中所有鍵 (keys) 的 *PyListObject*。

*PyObject* \*PyDict\_Values (*PyObject* \*p)

回傳值：新的參照。屬於穩定 ABI。返回一個包含字典中所有值 (values) 的 *PyListObject*。

*Py\_ssize\_t* PyDict\_Size (*PyObject* \*p)

屬於穩定 ABI。返回字典中項目數，等價於對字典 *p* 使用 `len(p)`。

int PyDict\_Next (*PyObject* \*p, *Py\_ssize\_t* \*ppos, *PyObject* \*\*pkey, *PyObject* \*\*pvalue)

屬於穩定 ABI。迭代字典 *p* 中的所有鍵值對。在第一次調用此函數開始迭代之前，由 *ppos* 所引用的 *Py\_ssize\_t* 必須被初始化為 0；該函數將為字典中的每個鍵值對返回真值，一旦所有鍵值對都報告完畢則返回假值。形參 *pkey* 和 *pvalue* 應當指向 *PyObject\** 變量，它們將分別使用每個鍵和值來填充，或者也可以為 NULL。通過它們返回的任何引用都是暫借的。*ppos* 在迭代期間不應被更改。它的值表示內部字典結構中的偏移量，並且由於結構是稀疏的，因此偏移量並不連續。

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... */
    ...
}
```

字典 *p* 不應該在遍歷期間發生改變。在遍歷字典時，改變鍵中的值是安全的，但僅限於鍵的集合不發生改變。例如：

```
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);
}
```

int PyDict\_Merge (*PyObject* \*a, *PyObject* \*b, int override)

屬於穩定 ABI。對映射對象 *b* 進行迭代，將鍵值對添加到字典 *a*。*b* 可以是一個字典，或任何支持 *PyMapping\_Keys()* 和 *PyObject\_GetItem()* 的對象。如果 *override* 為真值，則如果在 *b* 中找到相同的鍵則 *a* 中已存在的相應鍵值對將被替換，否則如果在 *a* 中没有相同的鍵則只是添加鍵值對。當成功時返回 0 或者當引發異常時返回 -1。

**int PyDict\_Update** (*PyObject* \*a, *PyObject* \*b)

属于稳定 ABI。这与 C 中的 `PyDict_Merge(a, b, 1)` 一样，也类似于 Python 中的 `a.update(b)`，差别在于 `PyDict_Update()` 在第二个参数没有“keys”属性时不会回退到迭代键值对的序列。当成功时返回 0 或者当引发异常时返回 -1。

**int PyDict\_MergeFromSeq2** (*PyObject* \*a, *PyObject* \*seq2, int override)

属于稳定 ABI。将 `seq2` 中的键值对更新或合并到字典 `a`。`seq2` 必须为产生长度为 2 的用作键值对的元素的可迭代对象。当存在重复的键时，如果 `override` 真值则最后出现的键胜出。当成功时返回 0 或者当引发异常时返回 -1。等价的 Python 代码（返回值除外）：

```
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value
```

## 8.4.2 集合对象

这一节详细介绍了针对 `set` 和 `frozenset` 对象的公共 API。任何未在下面列出的功能最好是使用抽象对象协议（包括 `PyObject_CallMethod()`, `PyObject_RichCompareBool()`, `PyObject_Hash()`, `PyObject_Repr()`, `PyObject_IsTrue()`, `PyObject_Print()` 以及 `PyObject_GetIter()`）或者抽象数字协议（包括 `PyNumber_And()`, `PyNumber_Subtract()`, `PyNumber_Or()`, `PyNumber_Xor()`, `PyNumber_InPlaceAnd()`, `PyNumber_InPlaceSubtract()`, `PyNumber_InPlaceOr()` 以及 `PyNumber_InPlaceXor()`）。

**type PySetObject**

这个 *PyObject* 的子类型被用来保存 `set` 和 `frozenset` 对象的内部数据。它类似于 *PyDictObject* 的地方在于对小尺寸集合来说它是固定大小的（很像元组的存储方式），而对于中等和大尺寸集合来说它将指向单独的可变大小的内存块（很像列表的存储方式）。此结构体的字段不应被视为公有并且可能发生改变。所有访问都应当通过已写入文档的 API 来进行而不可通过直接操纵结构体中的值。

**PyTypeObject PySet\_Type**

属于稳定 ABI。这是一个 *PyTypeObject* 实例，表示 Python `set` 类型。

**PyTypeObject PyFrozenSet\_Type**

属于稳定 ABI。这是一个 *PyTypeObject* 实例，表示 Python `frozenset` 类型。

下列类型检查宏适用于指向任意 Python 对象的指针。类似地，这些构造函数也适用于任意可迭代的 Python 对象。

**int PySet\_Check** (*PyObject* \*p)

如果 `p` 是一个 `set` 对象或者是其子类型的实例则返回真值。此函数总是会成功执行。

**int PyFrozenSet\_Check** (*PyObject* \*p)

如果 `p` 是一个 `frozenset` 对象或者是其子类型的实例则返回真值。此函数总是会成功执行。

**int PyAnySet\_Check** (*PyObject* \*p)

如果 `p` 是一个 `set` 对象、`frozenset` 对象或者是其子类型的实例则返回真值。此函数总是会成功执行。

**int PySet\_CheckExact** (*PyObject* \*p)

如果 `p` 是一个 `set` 对象但不是其子类型的实例则返回真值。此函数总是会成功执行。

在 3.10 版新加入。

**int PyAnySet\_CheckExact** (*PyObject* \*p)

如果 `p` 是一个 `set` 或 `frozenset` 对象但不是其子类型的实例则返回真值。此函数总是会成功执行。



**int PyFrozenSet\_CheckExact (PyObject \*p)**

如果 *p* 是一个 frozenset 对象但不是其子类型的实例则返回真值。此函数总是会成功执行。

**PyObject \*PySet\_New (PyObject \*iterable)**

回傳值：新的參照。属于稳定 ABI。返回一个新的 set，其中包含 *iterable* 所返回的对象。*iterable* 可以为 NULL 表示创建一个新的空集合。成功时返回新的集合，失败时返回 NULL。如果 *iterable* 实际上不是可迭代对象则引发 TypeError。该构造器也适用于拷贝集合 (*c*=set(*s*))。

**PyObject \*PyFrozenSet\_New (PyObject \*iterable)**

回傳值：新的參照。属于稳定 ABI。返回一个新的 frozenset，其中包含 *iterable* 所返回的对象。*iterable* 可以为 NULL 表示创建一个新的空冻结集合。成功时返回新的冻结集合，失败时返回 NULL。如果 *iterable* 实际上不是可迭代对象则引发 TypeError。

下列函数和宏适用于 set 或 frozenset 的实例或是其子类型的实例。

**Py\_ssize\_t PySet\_Size (PyObject \*anyset)**

属于稳定 ABI。返回 set 或 frozenset 对象的长度。等同于 len(*anyset*)。如果 *anyset* 不是 set, frozenset 或其子类型的实例，则会引发 SystemError。

**Py\_ssize\_t PySet\_GET\_SIZE (PyObject \*anyset)**

宏版本的 *PySet\_Size()*，不带错误检测。

**int PySet\_Contains (PyObject \*anyset, PyObject \*key)**

属于稳定 ABI。如果找到则返回 1，如果未找到则返回 0，如果遇到错误则返回 -1。与 Python `__contains__()` 方法不同，该函数不会自动将不可哈希的集合转换为临时冻结集合。如果 *key* 是不可哈希对象则会引发 TypeError。如果 *anyset* 不是 set, frozenset 或其子类型的实例则会引发 SystemError。

**int PySet\_Add (PyObject \*set, PyObject \*key)**

属于稳定 ABI。添加 *key* 到一个 set 实例。也可用于 frozenset 实例（与 *PyTuple\_SetItem()* 的类似之处是它也可被用来为全新的冻结集合在公开给其他代码之前填充全新的值）。成功时返回 0 而失败时返回 -1。如果 *key* 为不可哈希对象则会引发 TypeError。如果没有增长空间则会引发 MemoryError。如果 *set* 不是 set 或其子类型的实例则会引发 SystemError。

下列函数适用于 set 或其子类型的实例，但不可用于 frozenset 或其子类型的实例。

**int PySet\_Discard (PyObject \*set, PyObject \*key)**

属于稳定 ABI。如果找到并已删除则返回 1，如未找到（无操作）则返回 0，如果遇到错误则返回 -1。对于不存在的键不会引发 KeyError。如果 *key* 为不可哈希对象则会引发 TypeError。与 Python `discard()` 方法不同，该函数不会自动将不可哈希的集合转换为临时的冻结集合。如果 *set* 不是 set 或其子类的实例则会引发 SystemError。

**PyObject \*PySet\_Pop (PyObject \*set)**

回傳值：新的參照。属于稳定 ABI。返回 *set* 中任意对象的新引用，并从 *set* 中移除该对象。失败时返回 NULL。如果集合为空则会引发 KeyError。如果 *set* 不是 set 或其子类型的实例则会引发 SystemError。

**int PySet\_Clear (PyObject \*set)**

属于稳定 ABI。清空现有的所有元素的集合。成功时返回 0。如果 *set* 不是 set 或其子类型的实际则返回 -1 并引发 SystemError。



## 8.5 函式物件

### 8.5.1 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 `F`，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 `F`）則回傳 `true`。參數必須不為 `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。

函数的文档字符串和名称是从代码对象中提取的。`__module__` 是从 *globals* 中提取的。参数 `defaults`, `annotations` 和 `closure` 被设为 `NULL`。`__qualname__` 被设为与代码对象的 `co_qualname` 字段相同的值。

**`PyObject` \*`PyFunction_NewWithQualName` (`PyObject` \*code, `PyObject` \*globals, `PyObject` \*qualname)**

回傳值：新的參照。类似 `PyFunction_New()`，但还允许设置函数对象的 `__qualname__` 属性。*qualname* 应当是一个 `unicode` 对象或为 `NULL`；如为 `NULL`，则 `__qualname__` 属性会被设为与代码对象的 `co_qualname` 字段相同的值。

在 3.3 版新加入。

**`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。向 函数对象 *op* 的 `__module__` 属性返回一个 *borrowed reference*。该值可以为 `NULL`。

这通常为一个包含模块名称的字符串，但可以通过 Python 代码设为任何其他对象。

**`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`。

**`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。

## 8.5.2 實例方法物件 (Instance Method Objects)

實例方法是 *PyCFunction* 的包裝器，也是將 *PyCFunction* 與類對象綁定的新方法。它取代了以前的調用 *PyMethod\_New(func, NULL, class)*。

*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。

## 8.5.3 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。

### 8.5.4 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 **PyObject** **PyCellObject**

Cell 物件所用之 C 結構。

*PyObject* **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 必須不為 NULL。此函式總是會成功執行。

*PyObject* \***PyCell\_New** (*PyObject* \*ob)

回傳值：新的參照。建立一個回傳一個包含 ob 的新 cell 物件。參數可以為 NULL。

*PyObject* \***PyCell\_Get** (*PyObject* \*cell)

回傳值：新的參照。回傳 cell 內容中的 cell。

*PyObject* \***PyCell\_GET** (*PyObject* \*cell)

回傳值：借用參照。回傳 cell 物件 cell 的內容，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 的內容設為 value。這將釋放任何對 cell 物件當前內容的參照。value 可以為 NULL。cell 必須不為 NULL；如果它不是一個 cell 物件則將回傳 -1。如果設定成功則將回傳 0。

void **PyCell\_SET** (*PyObject* \*cell, *PyObject* \*value)

將 cell 物件 cell 的值設為 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。

### 8.5.5 程式碼物件

代碼對象是 CPython 實現的低層級細節。每個代表一塊尚未綁定到函數中的可執行代碼。

type **PyObject** **PyCodeObject**

用於描述代碼對象的對象的 C 結構。此類型字段可隨時更改。

*PyObject* **PyCode\_Type**

這一個代表 Python 代碼對象的 *PyObject* 實例。

int **PyCode\_Check** (*PyObject* \*co)

如果 co 是一個代碼對象則返回真值。此函數總是會成功執行。

int **PyCode\_GetNumFree** (*PyCodeObject* \*co)

返回 co 中的自由變量數。

*PyCodeObject* \***PyCode\_New** (int argcount, int kwnonlyargcount, int nlocals, int stacksize, int flags, *PyObject* \*code, *PyObject* \*consts, *PyObject* \*names, *PyObject* \*varnames, *PyObject* \*freevars, *PyObject* \*cellvars, *PyObject* \*filename, *PyObject* \*name, *PyObject* \*qualname, int firstlineno, *PyObject* \*linetable, *PyObject* \*exceptiontable)

回傳值：新的參照。返回一個新的代碼對象。如果你需要一個空白代碼對象來創建幀，請改用 *PyCode\_NewEmpty*()。直接調用 *PyCode\_New*() 將會綁定一個具體 Python 版本，因為字節碼經常會變動。此函數的各種參數存在複雜的相互依賴，這意味著對值的細微改變可能會導致不正確的執行或 VM 的崩潰。必須非常小心地使用這個函數。

在 3.11 版的變更：添加了 qualname 和 exceptiontable 形參。

*PyCodeObject* \***PyCode\_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)

回傳值：新的參照。與 *PyCode\_New()* 類似，但還有一個針對僅限位置參數的額外的“posonlyargcount”。適用於 *PyCode\_New* 的注意事項同樣適用於這個函數。

在 3.8 版新加入。

在 3.11 版的變更：增加了 *qualname* 和 *exceptiontable* 形參。

*PyCodeObject* \***PyCode\_NewEmpty** (const char \*filename, const char \*funcname, int firstlineno)

回傳值：新的參照。返回一個具有指定用戶名、函數名和首行行號的空代碼對象。結果代碼對象如果被執行則將引發一個 *Exception*。

int **PyCode\_Addr2Line** (*PyCodeObject* \*co, int byte\_offset)

返回在 *byte\_offset* 位置或之前以及之後發生的指令的行號。如果你只需要一個幀的行號，請改用 *PyFrame\_GetLineNumber()*。

要高效地對代碼對象中的行號進行迭代，請使用在 [PEP 626](#) 中描述的 API。

int **PyCode\_Addr2Location** (*PyObject* \*co, int byte\_offset, int \*start\_line, int \*start\_column, int \*end\_line, int \*end\_column)

將傳入的 int 指針設為 *byte\_offset* 處的指令的源代碼行編號和列編號。當沒有任何特定元素的信息時則將值設為 0。

如果函數執行成功則返回 1 否則返回 0。

在 3.11 版新加入。

*PyObject* \***PyCode\_GetCode** (*PyCodeObject* \*co)

等價於 Python 代碼 *getattr(co, 'co\_code')*。返回一個指向表示代碼對象中的字節碼的 *PyBytesObject* 的強引用。當出錯時，將返回 NULL 並引發一個異常。

這個 *PyBytesObject* 可以由解釋器按需創建並且不必代表 CPython 所實際執行的字節碼。此函數的主要用途是調試器和性能分析工具。

在 3.11 版新加入。

*PyObject* \***PyCode\_GetVarnames** (*PyCodeObject* \*co)

等價於 Python 代碼 *getattr(co, 'co\_varnames')*。返回一個指向包含局部變量名稱的 *PyTupleObject* 的新引用。當出錯時，將返回 NULL 並引發一個異常。

在 3.11 版新加入。

*PyObject* \***PyCode\_GetCellvars** (*PyCodeObject* \*co)

等價於 Python 代碼 *getattr(co, 'co\_cellvars')*。返回一個包含被嵌套的函數所引用的局部變量名稱的 *PyTupleObject* 的新引用。當出錯時，將返回 NULL 並引發一個異常。

在 3.11 版新加入。

*PyObject* \***PyCode\_GetFreevars** (*PyCodeObject* \*co)

等價於 Python 代碼 *getattr(co, 'co\_freevars')*。返回一個指向包含自由變量名稱的 *PyTupleObject* 的新引用。當出錯時，將返回 NULL 並引發一個異常。

在 3.11 版新加入。

## 8.6 其他物件

### 8.6.1 檔案物件 (File Objects)

這些 API 是用於建立檔案物件的 Python 2 C API 的最小模擬 (minimal emulation)，它過去依賴於 C 標準函式庫對於緩衝 I/O (FILE\*) 的支援。在 Python 3 中，檔案和串流使用新的 io 模組，它在操作系統的低階無緩衝 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內部錯誤報告；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。屬於穩定 ABI。根據已打開文件 *fd* 的文件描述符創建一個 Python 文件對象。參數 *name*, *encoding*, *errors* 和 *newline* 可以為 NULL 表示使用默认值；*buffering* 可以為 -1 表示使用默认值。*name* 會被忽略僅保留用於向下兼容。失敗時返回 NULL。有關參數的更全面描述，請參閱 `io.open()` 函數的文檔。

**警告：** 由於 Python 串流有自己的緩衝層，將它們與操作系統層級檔案描述器混合使用會產生各種問題（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)

屬於穩定 ABI。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。屬於穩定 ABI。等價於 `p.readline([n])`，這個函數從對象 *p* 中讀取一行。*p* 可以是文件對象或具有 `readline()` 方法的任何對象。如果 *n* 是 0，則無論該行的長度如何，都會讀取一行。如果 *n* 大於 0，則從文件中讀取不超過 *n* 個字節；可以返回行的一部分。在這兩種情況下，如果立即到達文件末尾，則返回空字符串。但是，如果 *n* 小於 0，則無論長度如何都會讀取一行，但是如果立即到達文件末尾，則引發 `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。

*handler* 函數的類型為：

**typedef PyObject\* (\*Py\_OpenCodeHookFunction)** (PyObject\*, void\*)

等價於 `PyObject* (*)(PyObject *path, void *userData)`，其中 *path* 會確保為 `PyUnicodeObject`。

*userData* 指標被傳遞到 `hook function` 中。由於可能會從不同的執行環境 (runtime) 呼叫 `hook function`，因此該指標不應直接指向 Python 狀態。

由於此 `hook function` 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。

一旦鉤子被設定，它就不能被移除或替換，之後對 `PyFile_SetOpenCodeHook()` 的調用也將失敗，如果解釋器已經被初始化，函數將返回 -1 並設置一個異常。

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 版新加入。

**int PyFile\_WriteObject** (PyObject \*obj, PyObject \*p, int flags)

屬於穩定 ABI。將對象 *obj* 寫入文件對象 *p*。*flags* 唯一支持的旗標是 `Py_PRINT_RAW`；如果給定，則寫入對象的 `str()` 而不是 `repr()`。成功時返回 0，失敗時返回 -1；將設置適當的異常。



`int PyFile_WriteString (const char *s, PyObject *p)`

属于稳定 ABI. 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，`PyFile_WriteString` 會設定合適的例外狀況。

## 8.6.2 模組物件模組

*PyObject* **PyModule\_Type**

属于稳定 ABI. 这个 C 类型实例 *PyModule\_Type* 用来表示 Python 中的模块类型。在 Python 程序中该实例被暴露为 `types.ModuleType`。

`int PyModule_Check (PyObject *p)`

当 *p* 为模块类型的对象，或是模块子类型的对象时返回真值。该函数永远有返回值。

`int PyModule_CheckExact (PyObject *p)`

当 *p* 为模块类型的对象且不是 *PyModule\_Type* 的子类型的对象时返回真值。该函数永远有返回值。

*PyObject* \***PyModule\_NewObject** (*PyObject* \*name)

回傳值：新的參照。属于稳定 ABI 自 3.7 版起. 返回新的模块对象，其属性 `__name__` 为 *name*。模块的如下属性 `__name__`, `__doc__`, `__package__`, and `__loader__` 都会被自动填充。（所有属性除了 `__name__` 都被设为 `None`）。调用时应当提供 `__file__` 属性。

在 3.3 版新加入。

在 3.4 版的變更: `__package__` 和 `__loader__` 被設 `None`。

*PyObject* \***PyModule\_New** (const char \*name)

回傳值：新的參照。属于稳定 ABI. 这类似于 *PyModule\_NewObject* ()，但其名称为 UTF-8 编码的字符串而不是 Unicode 对象。

*PyObject* \***PyModule\_GetDict** (*PyObject* \*module)

回傳值：借用參照。属于稳定 ABI. 返回实现 *module* 的命名空间的字典对象；此对象与模块对象的 `__dict__` 属性相同。如果 *module* 不是一个模块对象（或模块对象的子类型），则会引发 `SystemError` 并返回 `NULL`。

建议扩展使用其他 *PyModule\_\** 和 *PyObject\_\** 函数而不是直接操纵模块的 `__dict__`。

*PyObject* \***PyModule\_GetNameObject** (*PyObject* \*module)

回傳值：新的參照。属于稳定 ABI 自 3.7 版起. 返回 *module* 的 `__name__` 值。如果模块未提供该值，或者如果它不是一个字符串，则会引发 `SystemError` 并返回 `NULL`。

在 3.3 版新加入。

const char \***PyModule\_GetName** (*PyObject* \*module)

属于稳定 ABI. 类似于 *PyModule\_GetNameObject* () 但返回 'utf-8' 编码的名称。

void \***PyModule\_GetState** (*PyObject* \*module)

属于稳定 ABI. 返回模块的“状态”，也就是说，返回指向在模块创建时分配的内存块的指针，或者 `NULL`。参见 *PyModuleDef.m\_size*。

*PyModuleDef* \***PyModule\_GetDef** (*PyObject* \*module)

属于稳定 ABI. 返回指向模块创建所使用的 *PyModuleDef* 结构体的指针，或者如果模块不是使用结构体定义创建的则返回 `NULL`。

*PyObject* \***PyModule\_GetFilenameObject** (*PyObject* \*module)

回傳值：新的參照。属于稳定 ABI. 返回使用 *module* 的 `__file__` 属性所加载的模块的文件名。如果属性未定义，或者如果它不是一个 Unicode 字符串，则会引发 `SystemError` 并返回 `NULL`；在其他情况下将返回一个指向 Unicode 对象的引用。

在 3.2 版新加入。



```
const char *PyModule_GetFilename(PyObject *module)
```

属于稳定 ABI。类似于 `PyModule_GetFilenameObject()` 但会返回编码为 'utf-8' 的文件名。

在 3.2 版之後被 用: `PyModule_GetFilename()` 对于不可编码的文件名会引发 `UnicodeEncodeError`, 请改用 `PyModule_GetFilenameObject()`。

## 初始化 C 模块

模块对象通常是基于扩展模块（导出初始化函数的共享库），或内部编译模块（其中使用 `PyImport_AppendInittab()` 添加初始化函数）。请参阅 [building](#) 或 [extending-with-embedding](#) 了解详情。

初始化函数可以向 `PyModule_Create()` 传入一个模块定义实例，并返回结果模块对象，或者通过返回定义结构体本身来请求“多阶段初始化”。

type **PyModuleDef**

属于稳定 ABI（包括所有成员）。模块定义结构，它保存创建模块对象所需的所有信息。每个模块通常只有一个这种类型的静态初始化变量

**PyModuleDef\_Base m\_base**

始终将此成员初始化为 `PyModuleDef_HEAD_INIT`。

**const char \*m\_name**

新模块的名称。

**const char \*m\_doc**

模块的文档字符串；一般会使用通过 `PyDoc_STRVAR` 创建的文档字符串变量。

**Py\_ssize\_t m\_size**

可以把模块的状态保存在为单个模块分配的内存区域中，使用 `PyModule_GetState()` 检索，而不是保存在静态全局区。这使得模块可以在多个子解释器中安全地使用。

这个内存区域将在创建模块时根据 `m_size` 分配，并在调用 `m_free` 函数（如果存在）在取消分配模块对象时释放。

将 `m_size` 设置为 -1，意味着这个模块具有全局状态，因此不支持子解释器。

将其设置为非负值，意味着模块可以重新初始化，并指定其状态所需要的额外内存大小。多阶段初始化需要非负的 `m_size`。

更多詳情請見 [PEP 3121](#)。

**PyMethodDef \*m\_methods**

一个指向模块函数表的指针，由 `PyMethodDef` 描述。如果模块没有函数，可以为 NULL。

**PyModuleDef\_Slot \*m\_slots**

由针对多阶段初始化的槽位定义组成的数组，以一个 {0, NULL} 条目结束。当使用单阶段初始化时，`m_slots` 必须为 NULL。

在 3.5 版的變更: 在 3.5 版之前，此成员总是被设为 NULL，并被定义为:

*inquiry* **m\_reload**

*traverseproc* **m\_traverse**

在模块对象的垃圾回收遍历期间所调用的遍历函数，如果不需要则为 NULL。

如果模块状态已被请求但尚未分配则不会调用此函数。在模块创建之后至模块执行之前（调用 `Py_mod_exec` 函数）就属于这种情况。更确切地说，如果 `m_size` 大于 0 且模块状态（由 `PyModule_GetState()` 返回）为 NULL 则不会调用此函数。

在 3.9 版的變更: 在模块状态被分配之前不再调用。

***inquiry* m\_clear**

在模块对象的垃圾回收清理期间所调用的清理函数，如果不需要则为 NULL。

如果模块状态已被请求但尚未分配则不会调用此函数。在模块创建之后至模块执行之前（调用 `Py_mod_exec` 函数）就属于这种情况。更确切地说，如果 `m_size` 大于 0 且模块状态（由 `PyModule_GetState()` 返回）为 NULL 则不会调用此函数。

就像 `PyTypeObject.tp_clear` 那样，这个函数并不总是在模块被释放前被调用。例如，当引用计数足以确定一个对象不再被使用时，就会直接调用 `m_free`，而不使用循环垃圾回收器。

在 3.9 版的變更：在模块状态被分配之前不再调用。

***freefunc* m\_free**

在模块对象的释放期间所调用的函数，如果不需要则为 NULL。

如果模块状态已被请求但尚未分配则不会调用此函数。在模块创建之后至模块执行之前（调用 `Py_mod_exec` 函数）就属于这种情况。更确切地说，如果 `m_size` 大于 0 且模块状态（由 `PyModule_GetState()` 返回）为 NULL 则不会调用此函数。

在 3.9 版的變更：在模块状态被分配之前不再调用。

**单阶段初始化**

模块初始化函数可以直接创建并返回模块对象，称为“单阶段初始化”，使用以下两个模块创建函数中的一个：

***PyObject* \*PyModule\_Create (PyModuleDef \*def)**

回傳值：新的參照。根据在 `def` 中给出的定义创建一个新的模块对象。它的行为类似于 `PyModule_Create2()` 将 `module_api_version` 设为 `PYTHON_API_VERSION`。

***PyObject* \*PyModule\_Create2 (PyModuleDef \*def, int module\_api\_version)**

回傳值：新的參照。属于稳定 ABI。创建一个新的模块对象，在参数 `def` 中给出定义，设定 API 版本为参数 `module_api_version`。如果该版本与正在运行的解释器版本不匹配，则会触发 `RuntimeWarning`。

---

**備註：**大多数时候应该使用 `PyModule_Create()` 代替使用此函数，除非你确定需要使用它。

---

在初始化函数返回之前，生成的模块对象通常使用 `PyModule_AddObjectRef()` 等函数进行填充。

**多阶段初始化**

指定扩展的另一种方式是请求“多阶段初始化”。以这种方式创建的扩展模块的行为更类似 Python 模块：初始化分为创建阶段即创建模块对象时和 执行阶段即填充模块对象时。这种区分类似于类的 `__new__()` 和 `__init__()` 方法。

与使用单阶段初始化创建的模块不同，这些模块不是单例：如果移除 `sys.modules` 条目并重新导入模块，将会创建一个新的模块对象，而旧的模块则会成为常规的垃圾回收目标——就像 Python 模块那样。默认情况下，根据同一个定义创建的多个模块应该是相互独立的：对其中一个模块的更改不应影响其他模块。这意味着所有状态都应该是模块对象（例如使用 `PyModule_GetState()`）或其内容（例如模块的 `__dict__` 或使用 `PyType_FromSpec()` 创建的单独类）的特定状态。

所有使用多阶段初始化创建的模块都应该支持子解释器。保证多个模块之间相互独立，通常就可以实现这一点。

要请求多阶段初始化，初始化函数 (`PyInit_modulename`) 返回一个包含非空的 `m_slots` 属性的 `PyModuleDef` 实例。在它被返回之前，这个 `PyModuleDef` 实例必须先使用以下函数初始化：

*PyObject* \*PyModuleDef\_Init (PyModuleDef \*def)

回傳值：借用參照。属于稳定 ABI 自 3.5 版起。确保模块定义是一个正确初始化的 Python 对象，拥有正确的类型和引用计数。

返回转换为 PyObject\* 的 def，如果发生错误，则返回 NULL。

在 3.5 版新加入。

模块定义的 m\_slots 成员必须指向一个 PyModuleDef\_Slot 结构体数组：

type PyModuleDef\_Slot

int slot

槽位 ID，从下面介绍的可用值中选择。

void \*value

槽位值，其含义取决于槽位 ID。

在 3.5 版新加入。

m\_slots 数组必须以一个 id 为 0 的槽位结束。

可用的槽位类型是：

Py\_mod\_create

指定一个函数供调用以创建模块对象本身。该槽位的 value 指针必须指向一个具有如下签名的函数：

*PyObject* \*create\_module (PyObject \*spec, PyModuleDef \*def)

该函数接受一个 ModuleSpec 实例，如 PEP 451 所定义的，以及模块定义。它应当返回一个新的模块对象，或者设置一个错误并返回 NULL。

此函数应当保持最小化。特别地，它不应当调用任意 Python 代码，因为尝试再次导入同一个模块可能会导致无限循环。

多个 Py\_mod\_create 槽位不能在一个模块定义中指定。

如果未指定 Py\_mod\_create，导入机制将使用 PyModule\_New() 创建一个普通的模块对象。名称是获取自 spec 而非定义，以允许扩展模块动态地调整它们在模块层级结构中的位置并通过符号链接以不同的名称被导入，同时共享同一个模块定义。

不要求返回的对象必须为 PyModule\_Type 的实例。任何类型均可使用，只要它支持设置和获取导入相关的属性。但是，如果 PyModuleDef 具有非 NULL 的 m\_traverse, m\_clear, m\_free；非零的 m\_size；或者 Py\_mod\_create 以外的槽位则只能返回 PyModule\_Type 的实例。

Py\_mod\_exec

指定一个供调用以执行模块的函数。这造价于执行一个 Python 模块的代码：通常，此函数会向模块添加类和常量。此函数的签名为：

int exec\_module (PyObject \*module)

如果指定了多个 Py\_mod\_exec 槽位，将按照它们在 \*m\_slots\* 数组中出现的顺序进行处理。

有关多阶段初始化的更多细节，请参阅 PEP:489

## 底层模块创建函数

当使用多阶段初始化时，将会调用以下函数。例如，在动态创建模块对象的时候，可以直接使用它们。注意，必须调用 PyModule\_FromDefAndSpec 和 PyModule\_ExecDef 来完整地初始化一个模块。

*PyObject* \*PyModule\_FromDefAndSpec (PyModuleDef \*def, PyObject \*spec)

回傳值：新的參照。根据在 def 中给出的定义和 ModuleSpec spec 创建一个新的模块对象。它的行为类似于 PyModule\_FromDefAndSpec2() 将 module\_api\_version 设为 PYTHON\_API\_VERSION。

在 3.5 版新加入。

*PyObject* \*PyModule\_FromDefAndSpec2 (*PyModuleDef* \*def, *PyObject* \*spec, int module\_api\_version)

回傳值：新的參照。屬於穩定 ABI 自 3.7 版起。創建一個新的模塊對象，在參數 *def* 和 *spec* 中給出定義，設置 API 版本為參數 *module\_api\_version*。如果該版本與正在運行的解釋器版本不匹配，則會觸發 RuntimeWarning。

備註：大多數時候應該使用 *PyModule\_FromDefAndSpec()* 代替使用此函數，除非你確定需要使用它。

在 3.5 版新加入。

int PyModule\_ExecDef (*PyObject* \*module, *PyModuleDef* \*def)

屬於穩定 ABI 自 3.7 版起。執行參數 \*def\* 中給出的任意執行槽 (*Py\_mod\_exec*)。

在 3.5 版新加入。

int PyModule\_SetDocString (*PyObject* \*module, const char \*docstring)

屬於穩定 ABI 自 3.7 版起。將 \*module\* 的文档字符串設置為 \*docstring\*。當使用 *PyModule\_Create* 或 *PyModule\_FromDefAndSpec* 從 *PyModuleDef* 創建模塊時，會自動調用此函數。

在 3.5 版新加入。

int PyModule\_AddFunctions (*PyObject* \*module, *PyMethodDef* \*functions)

屬於穩定 ABI 自 3.7 版起。將以 NULL 結尾的 \*functions\* 數組中的函數添加到 \*module\* 模塊中。有關單個條目的更多細節，請參閱 *PyMethodDef* 文档（由於缺少共享的模塊命名空間，在 C 中實現的模塊級“函數”通常將模塊作為它的第一個參數，與 Python 類的實例方法類似）。當使用 *PyModule\_Create* 或 *PyModule\_FromDefAndSpec* 從 *PyModuleDef* 創建模塊時，會自動調用此函數。

在 3.5 版新加入。

## 支持函數

模塊初始化函數（單階段初始化）或通過模塊的執行槽位調用的函數（多階段初始化），可以使用以下函數，來幫助初始化模塊的狀態：

int PyModule\_AddObjectRef (*PyObject* \*module, const char \*name, *PyObject* \*value)

屬於穩定 ABI 自 3.10 版起。將一個名為 \*name\* 的對象添加到 \*module\* 模塊中。這是一個方便的函數，可以在模塊的初始化函數中使用。

如果成功，返回 0。如果發生錯誤，引發異常並返回 -1。

如果 \*value\* 為 NULL，返回 NULL。在調用它時發生這種情況，必須拋出異常。

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;
}
```

這個例子也可以寫成不顯式地檢查 *obj* 是否為 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;
}
```

注意在此情况下应当使用 `Py_XDECREF()` 而不是 `Py_DECREF()`，因为 `obj` 可能为 `NULL`。

在 3.10 版新加入。

`int PyModule_AddObject(PyObject *module, const char *name, PyObject *value)`

属于稳定 ABI。类似于 `PyModule_AddObjectRef()`，但会在成功时偷取一个对 `value` 的引用（如果它返回 0 值）。

推荐使用新的 `PyModule_AddObjectRef()` 函数，因为误用 `PyModule_AddObject()` 函数很容易导致引用泄漏。

---

**備註：** 与其他窃取引用的函数不同，`PyModule_AddObject()` 只在 **成功** 时释放对 `value` 的引用。这意味着必须检查它的返回值，调用方必须在发生错误时手动为 `*value*` 调用 `Py_DECREF()`。

---

用法範例：

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    if (obj == NULL) {
        return -1;
    }
    if (PyModule_AddObject(module, "spam", obj) < 0) {
        Py_DECREF(obj);
        return -1;
    }
    // PyModule_AddObject() stole a reference to obj:
    // Py_DECREF(obj) is not needed here
    return 0;
}
```

这个例子也可以写成不显式地检查 `obj` 是否为 `NULL`：

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    if (PyModule_AddObject(module, "spam", obj) < 0) {
        Py_XDECREF(obj);
        return -1;
    }
    // PyModule_AddObject() stole a reference to obj:
    // Py_DECREF(obj) is not needed here
    return 0;
}
```

注意在此情况下应当使用 `Py_XDECREF()` 而不是 `Py_DECREF()`，因为 `obj` 可能为 `NULL`。

`int PyModule_AddIntConstant(PyObject *module, const char *name, long value)`

属于稳定 ABI。将一个名称为 `*name*` 的整型常量添加到 `*module*` 模块中。这个方便的函数可以在模块的初始化函数中使用。如果发生错误，返回 -1，成功返回 0。



**int PyModule\_AddStringConstant** (*PyObject* \*module, const char \*name, const char \*value)

属于稳定 ABI。将一个名称为 \*name\* 的字符串常量添加到 \*module\* 模块中。这个方便的函数可以在模块的初始化函数中使用。字符串 \*value\* 必须以 NULL 结尾。如果发生错误，返回 -1，成功返回 0。

**PyModule\_AddIntMacro** (module, macro)

将一个整型常量添加到 \*module\* 模块中。名称和值取自 \*macro\* 参数。例如，`PyModule_AddIntMacro(module, AF_INET)` 将值为 \*AF\_INET\* 的整型常量 \*AF\_INET\* 添加到 \*module\* 模块中。如果发生错误，返回 -1，成功返回 0。

**PyModule\_AddStringMacro** (module, macro)

将一个字符串常量添加到 \*module\* 模块中。

**int PyModule\_AddType** (*PyObject* \*module, *PyTypeObject* \*type)

属于稳定 ABI 自 3.10 版起。将一个类型对象添加到 *module* 模块中。类型对象通过在函数内部调用 `PyType_Ready()` 完成初始化。类型对象的名称取自 `tp_name` 最后一个点号之后的部分。如果发生错误，返回 -1，成功返回 0。

在 3.9 版新加入。

## 查找模块

单阶段初始化创建可以在当前解释器上下文中被查找的单例模块。这使得仅通过模块定义的引用，就可以检索模块对象。

这些函数不适用于通过多阶段初始化创建的模块，因为可以从一个模块定义创建多个模块对象。

*PyObject* \***PyState\_FindModule** (*PyModuleDef* \*def)

返回值：借用参照。属于稳定 ABI。返回当前解释器中由 *def* 创建的模块对象。此方法要求模块对象此前已通过 `PyState_AddModule()` 函数附加到解释器状态中。如果找不到相应的模块对象，或模块对象还未附加到解释器状态，返回 NULL。

**int PyState\_AddModule** (*PyObject* \*module, *PyModuleDef* \*def)

属于稳定 ABI 自 3.3 版起。将传给函数的模块对象附加到解释器状态。这将允许通过 `PyState_FindModule()` 来访问该模块对象。

仅在使用单阶段初始化创建的模块上有效。

Python 会在导入一个模块后自动调用 `PyState_AddModule`，因此从模块初始化代码中调用它是没有必要的（但也没有害处）。显式的调用仅在模块自己的初始化代码后继调用了 `PyState_FindModule` 的情况下才是必要的。此函数主要是为了实现替代导入机制（或是通过直接调用它，或是通过引用它的实现来获取所需的状态更新详情）。

调用时必须携带 GIL。

成功是返回 0 或者失败时返回 -1。

在 3.3 版新加入。

**int PyState\_RemoveModule** (*PyModuleDef* \*def)

属于稳定 ABI 自 3.3 版起。从解释器状态中移除由 *def* 创建的模块对象。成功时返回 0，者失败时返回 -1。

调用时必须携带 GIL。

在 3.3 版新加入。



### 8.6.3 迭代器 (Iterator) 物件

Python 提供了两个通用迭代器对象。第一个是序列迭代器，它可与支持 `__getitem__()` 方法的任意序列一起使用。第二个迭代器使用一个可调对象和一个哨兵值，为序列中的每个项目调用可调对象，并在返回哨兵值时结束迭代。

#### *PyTypeObject* **PySeqIter\_Type**

属于稳定 ABI. *PySeqIter\_New()* 返回迭代器对象的类型对象和内置序列类型内置函数 *iter()* 的单参数形式。

#### **int PySeqIter\_Check** (*PyObject* \*op)

如果 *op* 的类型为 *PySeqIter\_Type* 则返回真值。此函数总是会成功执行。

#### *PyObject* \***PySeqIter\_New** (*PyObject* \*seq)

回傳值：新的參照。属于稳定 ABI. 返回一个与常规序列对象一起使用的迭代器 *seq*。当序列订阅操作引发 *IndexError* 时，迭代结束。

#### *PyTypeObject* **PyCallIter\_Type**

属于稳定 ABI. 由函数 *PyCallIter\_New()* 和 *iter()* 内置函数的双参数形式返回的迭代器对象类型对象。

#### **int PyCallIter\_Check** (*PyObject* \*op)

如果 *op* 的类型为 *PyCallIter\_Type* 则返回真值。此函数总是会成功执行。

#### *PyObject* \***PyCallIter\_New** (*PyObject* \*callable, *PyObject* \*sentinel)

回傳值：新的參照。属于稳定 ABI. 返回一个新的迭代器。第一个参数 *callable* 可以是任何可以在没有参数的情况下调用的 Python 可调对象；每次调用都应该返回迭代中的下一个项目。当 *callable* 返回等于 *sentinel* 的值时，迭代将终止。

### 8.6.4 Descriptor (描述器) 物件

“Descriptor” 是描述物件某些属性的物件，它們存在於型物件的 *dictionary* (字典) 中。

#### *PyTypeObject* **PyProperty\_Type**

属于稳定 ABI. 创建 descriptor 型的物件。

#### *PyObject* \***PyDescr\_NewGetSet** (*PyTypeObject* \*type, struct *PyGetSetDef* \*getset)

回傳值：新的參照。属于稳定 ABI.

#### *PyObject* \***PyDescr\_NewMember** (*PyTypeObject* \*type, struct *PyMemberDef* \*meth)

回傳值：新的參照。属于稳定 ABI.

#### *PyObject* \***PyDescr\_NewMethod** (*PyTypeObject* \*type, struct *PyMethodDef* \*meth)

回傳值：新的參照。属于稳定 ABI.

#### *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。属于稳定 ABI.

#### **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* 必須是一個 descriptor 物件；有錯誤檢查。

#### *PyObject* \***PyWrapper\_New** (*PyObject* \*, *PyObject* \*)

回傳值：新的參照。属于稳定 ABI.

## 8.6.5 切片物件

### *PyObject* **PySlice\_Type**

属于稳定 ABI. 切片对象的类型对象。它与 Python 层面的 `slice` 是相同的对象。

### **int** **PySlice\_Check** (*PyObject* \*ob)

如果 *ob* 是一个 `slice` 对象则返回真值；*ob* 必须不为 `NULL`。此函数总是会成功执行。

### *PyObject* \***PySlice\_New** (*PyObject* \*start, *PyObject* \*stop, *PyObject* \*step)

回傳值：新的參照。属于稳定 ABI. 返回一个具有给定值的新切片对象。*start*, *stop* 和 *step* 形参会被用作 `slice` 对象相应名称的属性的值。这些值中的任何一个都可以为 `NULL`，在这种情况下将使用 `None` 作为对应属性的值。如果新对象无法被分配则返回 `NULL`。

### **int** **PySlice\_GetIndices** (*PyObject* \*slice, *Py\_ssize\_t* length, *Py\_ssize\_t* \*start, *Py\_ssize\_t* \*stop, *Py\_ssize\_t* \*step)

属于稳定 ABI. 从切片对象 *slice* 提取 *start*, *stop* 和 *step* 索引号，将序列长度视为 *length*。大于 *length* 的序列号将被当作错误。

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).

你可能不会打算使用此函数。

在 3.2 版的變更: 之前 *slice* 形参的形参类型是 `PySliceObject*`。

### **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)

属于稳定 ABI. *PySlice\_GetIndices()* 的可用替代。从切片对象 *slice* 提取 *start*, *stop* 和 *step* 索引号，将序列长度视为 *length*，并将切片的长度保存在 *slicelength* 中，超出范围的索引号会以与普通切片一致的方式进行剪切。

成功时返回 0，出错时返回 -1 并且不设置异常。

**備註：** 此函数对于可变大序列来说是不安全的。对它的调用应被替换为 *PySlice\_Unpack()* 和 *PySlice\_AdjustIndices()* 的组合，其中

```
if (PySlice_GetIndicesEx(slice, length, &start, &stop, &step, &slicelength) < 0) {
    // return error
}
```

会被替换为

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {
    // return error
}
slicelength = PySlice_AdjustIndices(length, &start, &stop, step);
```

在 3.2 版的變更: 之前 *slice* 形参的形参类型是 `PySliceObject*`。

在 3.6.1 版的變更: 如果 `Py_LIMITED_API` 未设置或设置为 `0x03050400` 与 `0x03060000` 之间的值（不包括边界）或 `0x03060100` 或更大则 *PySlice\_GetIndicesEx()* 会被实现为一个使用 *PySlice\_Unpack()* 和 *PySlice\_AdjustIndices()* 的宏。参数 *start*, *stop* 和 *step* 会被多被求值。

在 3.6.1 版之後被弃用: 如果 `Py_LIMITED_API` 设置为小于 `0x03050400` 或 `0x03060000` 与 `0x03060100` 之间的值（不包括边界）则 *PySlice\_GetIndicesEx()* 为已弃用的函数。

### **int** **PySlice\_Unpack** (*PyObject* \*slice, *Py\_ssize\_t* \*start, *Py\_ssize\_t* \*stop, *Py\_ssize\_t* \*step)

属于稳定 ABI 自 3.7 版起。从切片对象中将 *start*, *stop* 和 *step* 数据成员提取为 C 整数。会静默地将大于 `PY_SSIZE_T_MAX` 的值减小为 `PY_SSIZE_T_MAX`，静默地将小于 `PY_SSIZE_T_MIN` 的

start 和 stop 值增大为 PY\_SSIZE\_T\_MIN, 并静默地将小于 -PY\_SSIZE\_T\_MAX 的 step 值增大为 -PY\_SSIZE\_T\_MAX。

出错时返回 -1, 成功时返回 0。

在 3.6.1 版新加入。

*Py\_ssize\_t* **PySlice\_AdjustIndices** (*Py\_ssize\_t* length, *Py\_ssize\_t* \*start, *Py\_ssize\_t* \*stop, *Py\_ssize\_t* step)

属于稳定 ABI 自 3.7 版起。将 start/end 切片索引号根据指定的序列长度进行调整。超出范围的索引号会以与普通切片一致的方式进行剪切。

返回切片的长度。此操作总是会成功。不会调用 Python 代码。

在 3.6.1 版新加入。

## Ellipsis 对象

*PyObject* \***Py\_Ellipsis**

Python 的 Ellipsis 对象。该对象没有任何方法。它必须以与任何其他对象一样的方式遵循引用计数。它与 *Py\_None* 一样属于单例对象。

## 8.6.6 MemoryView 物件

一个 memoryview 对象 C 级别的缓冲区接口 暴露为一个可以像任何其他对象一样传递的 Python 对象。

*PyObject* \***PyMemoryView\_FromObject** (*PyObject* \*obj)

回傳值: 新的参照。属于稳定 ABI。从提供缓冲区接口的对象创建 memoryview 对象。如果 obj 支持可写缓冲区导出, 则 memoryview 对象将可以被读/写, 否则它可能是只读的, 也可以是导出器自行决定的读/写。

**PyBUF\_READ**

用于请求只读缓冲区的旗标。

**PyBUF\_WRITE**

用于请求可写缓冲区的旗标。

*PyObject* \***PyMemoryView\_FromMemory** (char \*mem, *Py\_ssize\_t* size, int flags)

回傳值: 新的参照。属于稳定 ABI 自 3.7 版起。使用 mem 作为底层缓冲区创建一个 memoryview 对象。flags 可以是 *PyBUF\_READ* 或者 *PyBUF\_WRITE* 之一。

在 3.3 版新加入。

*PyObject* \***PyMemoryView\_FromBuffer** (const *Py\_buffer* \*view)

回傳值: 新的参照。属于稳定 ABI 自 3.11 版起。创建一个包含给定缓冲区结构 view 的 memoryview 对象。对于简单的字节缓冲区, *PyMemoryView\_FromMemory()* 是首选函数。

*PyObject* \***PyMemoryView\_GetContiguous** (*PyObject* \*obj, int buffertype, char order)

回傳值: 新的参照。属于稳定 ABI。从定义缓冲区接口的对象创建一个 memoryview 对象 contiguous 内存块 (在 'C' 或 'Fortran order' 中)。如果内存是连续的, 则 memoryview 对象指向原始内存。否则, 复制并且 memoryview 指向新的 bytes 对象。

buffertype 可以为 *PyBUF\_READ* 或 *PyBUF\_WRITE* 中的一个。

int **PyMemoryView\_Check** (*PyObject* \*obj)

如果 obj 是一个 memoryview 对象则返回真值。目前不允许创建 memoryview 的子类。此函数总是会成功执行。

*Py\_buffer* \***PyMemoryView\_GET\_BUFFER** (*PyObject* \*mview)

返回指向 memoryview 的导出缓冲区私有副本的指针。mview 必须是一个 memoryview 实例; 这个宏不检查它的类型, 你必须自己检查, 否则你将面临崩溃风险。

*PyObject* \*PyMemoryView\_GET\_BASE (*PyObject* \*mview)

返回 memoryview 所基于的导出对象的指针，或者如果 memoryview 已由函数 *PyMemoryView\_FromMemory()* 或 *PyMemoryView\_FromBuffer()* 创建则返回 NULL。*mview* 必须是一个 memoryview 实例。

## 8.6.7 弱参照物件

Python 支持“弱引用”作为一类对象。具体来说，有两种直接实现弱引用的对象。第一种就是简单的引用对象，第二种尽可能地作用为一个原对象的代理。

int PyWeakref\_Check (*PyObject* \*ob)

如果 *ob* 是一个引用或代理对象则返回真值。此函数总是会成功执行。

int PyWeakref\_CheckRef (*PyObject* \*ob)

如果 *ob* 是一个引用对象则返回真值。此函数总是会成功执行。

int PyWeakref\_CheckProxy (*PyObject* \*ob)

如果 *ob* 是一个代理对象则返回真值。此函数总是会成功执行。

*PyObject* \*PyWeakref\_NewRef (*PyObject* \*ob, *PyObject* \*callback)

回傳值：新的参照。属于稳定 ABI。返回对象 *ob* 的一个弱引用对象。该函数总是会返回一个新引用，但不保证创建一个新的对象；它有可能返回一个现有的引用对象。第二个形参 *callback* 可以是一个可调用对象，它会在 *ob* 被作为垃圾回收时接收通知；它应当接受一个单独形参，即弱引用对象本身。*callback* 也可以为 None 或 NULL。如果 *ob* 不是一个弱引用对象，或者如果 *callback* 不是可调用对象、None 或 NULL，则该函数将返回 NULL 并引发 *TypeError*。

*PyObject* \*PyWeakref\_NewProxy (*PyObject* \*ob, *PyObject* \*callback)

回傳值：新的参照。属于稳定 ABI。返回对象 *ob* 的一个弱引用代理对象。该函数将总是返回一个新的引用，但不保证创建一个新的对象；它有可能返回一个现有的代理对象。第二个形参 *callback* 可以是一个可调用对象，它会在 *ob* 被作为垃圾回收时接收通知；它应当接受一个单独形参，即弱引用对象本身。*callback* 也可以为 None 或 NULL。如果 *ob* 不是一个弱引用对象，或者如果 *callback* 不是可调用对象、None 或 NULL，则该函数将返回 NULL 并引发 *TypeError*。

*PyObject* \*PyWeakref\_GetObject (*PyObject* \*ref)

回傳值：借用参照。属于稳定 ABI。返回弱引用 *ref* 的被引用对象。如果被引用对象不再存在，则返回 *Py\_None*。

---

備註：该函数返回被引用对象的一个 *borrowed reference*。这意味着应该总是在该对象上调用 *Py\_INCREF()*，除非是当它在借入引用的最后一次被使用之前无法被销毁的时候。

---

*PyObject* \*PyWeakref\_GET\_OBJECT (*PyObject* \*ref)

回傳值：借用参照。类似于 *PyWeakref\_GetObject()*，但是不带错误检测。

void PyObject\_ClearWeakRefs (*PyObject* \*object)

属于稳定 ABI。此函数将被 *tp\_dealloc* 处理器调用以清空弱引用。

此函数将迭代 *object* 的弱引用并调用这些引用中可能存在的回调。它将在尝试了所有回调之后返回。

## 8.6.8 Capsule 对象

有关使用这些对象的更多信息请参阅 `using-capsules`。

在 3.1 版新加入。

type **PyCapsule**

这个 `PyObject` 的子类型代表一个隐藏的值，适用于需要将隐藏值（作为 `void*` 指针）通过 Python 代码传递到其他 C 代码的 C 扩展模块。它常常被用来让在一个模块中定义的 C 函数指针在其他模块中可用，这样就可以使用常规导入机制来访问在动态加载的模块中定义的 C API。

type **PyCapsule\_Destructor**

属于稳定 ABI。Capsule 的析构器回调的类型。定义如下：

```
typedef void (*PyCapsule_Destructor)(PyObject *);
```

参阅 `PyCapsule_New()` 来获取 `PyCapsule_Destructor` 返回值的语义。

int **PyCapsule\_CheckExact** (`PyObject *`p)

如果参数是一个 `PyCapsule` 则返回真值。此函数总是会成功执行。

`PyObject *`**PyCapsule\_New** (`void *`pointer, `const char *`name, `PyCapsule_Destructor` destructor)

回傳值：新的参照。属于稳定 ABI。创建一个封装了 `pointer` 的 `PyCapsule`。`pointer` 参考可以不为 NULL。

在失败时设置一个异常并返回 NULL。

字符串 `name` 可以是 NULL 或是一个指向有效的 C 字符串的指针。如果不为 NULL，则此字符串必须比 capsule 长（虽然也允许在 `destructor` 中释放它。）

如果 `destructor` 参数不为 NULL，则当它被销毁时将附带 capsule 作为参数来调用。

如果此 capsule 将被保存为一个模块的属性，则 `name` 应当被指定为 `modulename.attribute`。这将允许其他模块使用 `PyCapsule_Import()` 来导入此 capsule。

`void *`**PyCapsule\_GetPointer** (`PyObject *`capsule, `const char *`name)

属于稳定 ABI。提取保存在 capsule 中的 `pointer`。在失败时设置一个异常并返回 NULL。

`name` 参数必须与 capsule 中存储的名称完全一致。如果存储在 capsule 中的名称是 NULL，传入的 `name` 也必须是 NULL。Python 使用 C 函数 `strcmp()` 来比较 capsule 名称。

`PyCapsule_Destructor` **PyCapsule\_GetDestructor** (`PyObject *`capsule)

属于稳定 ABI。返回保存在 capsule 中的当前析构器。在失败时设置一个异常并返回 NULL。

capsule 具有 NULL 析构器是合法的。这会使得 NULL 返回码有些歧义；请使用 `PyCapsule_IsValid()` 或 `PyErr_Occurred()` 来消除歧义。

`void *`**PyCapsule\_GetContext** (`PyObject *`capsule)

属于稳定 ABI。返回保存在 capsule 中的当前上下文。在失败时设置一个异常并返回 NULL。

capsule 具有 NULL 上下文是全法的。这会使得 NULL 返回码有些歧义；请使用 `PyCapsule_IsValid()` 或 `PyErr_Occurred()` 来消除歧义。

`const char *`**PyCapsule\_GetName** (`PyObject *`capsule)

属于稳定 ABI。返回保存在 capsule 中的当前名称。在失败时设置一个异常并返回 NULL。

capsule 具有 NULL 名称是合法的。这会使得 NULL 返回码有些歧义；请使用 `PyCapsule_IsValid()` 或 `PyErr_Occurred()` 来消除歧义。

`void *`**PyCapsule\_Import** (`const char *`name, int no\_block)

属于稳定 ABI。从一个模块内的包装属性导入一个指向 C 对象的指针。`name` 形参应当指定该属性的完整名称，就像 `module.attribute` 这样。储存在包装中的 `name` 必须与此字符串完全匹配。

成功时返回 capsule 的内部指针。在失败时设置一个异常并返回 NULL。

在 3.3 版的變更: `no_block` 不再有任何影响。



**int PyCapsule\_IsValid** (*PyObject* \*capsule, const char \*name)

属于稳定 ABI. 确定 *capsule* 是否是一个有效的。有效的 *capsule* 必须不为 NULL, 传递 *PyCapsule\_CheckExact()*, 在其中存储一个不为 NULL 的指针, 并且其内部名称与 *name* 形参相匹配。(请参阅 *PyCapsule\_GetPointer()* 了解如何对 *capsule* 名称进行比较的有关信息。)

换句话说, 如果 *PyCapsule\_IsValid()* 返回真值, 则对任何访问器 (以 *PyCapsule\_Get* 开头的任何函数) 的调用都保证会成功。

如果对象有效并且匹配传入的名称则返回非零值。否则返回 0。此函数一定不会失败。

**int PyCapsule\_SetContext** (*PyObject* \*capsule, void \*context)

属于稳定 ABI. 将 *capsule* 内部的上下文指针设为 *context*。

成功时返回 0。失败时返回非零值并设置一个异常。

**int PyCapsule\_SetDestructor** (*PyObject* \*capsule, *PyCapsule\_Destructor* destructor)

属于稳定 ABI. 将 *capsule* 内部的析构器设为 *destructor*。

成功时返回 0。失败时返回非零值并设置一个异常。

**int PyCapsule\_SetName** (*PyObject* \*capsule, const char \*name)

属于稳定 ABI. 将 *capsule* 内部的名称设为 *name*。如果不为 NULL, 则名称的存在期必须比 *capsule* 更长。如果之前保存在 *capsule* 中的 *name* 不为 NULL, 则不会尝试释放它。

成功时返回 0。失败时返回非零值并设置一个异常。

**int PyCapsule\_SetPointer** (*PyObject* \*capsule, void \*pointer)

属于稳定 ABI. 将 *capsule* 内部的空指针设为 *pointer*。指针不可为 NULL。

成功时返回 0。失败时返回非零值并设置一个异常。

## 8.6.9 帧对象

**type PyFrameObject**

属于受限 API (作为不透明的结构体)。用于描述帧对象的对象 C 结构体。

此结构体中无公有成员。

在 3.11 版的變更: 此结构体的成员已从公有 C API 中移除。请参阅 What's New entry 了解详情。

可以使用函数 *PyEval\_GetFrame()* 与 *PyThreadState\_GetFrame()* 去获取一个帧对象。

可参考: *Reflection 1*

**PyTypeObject PyFrame\_Type**

帧对象的类型。它与 Python 层中的 *types.FrameType* 是同一对象。

在 3.11 版的變更: 在之前版本中, 此类型仅在包括 *<frameobject.h>* 之后可用。

**int PyFrame\_Check** (*PyObject* \*obj)

如果 *obj* 是一个帧对象则返回非零值。

在 3.11 版的變更: 在之前版本中, 只函数仅在包括 *<frameobject.h>* 之后可用。

**PyFrameObject \*PyFrame\_GetBack** (*PyFrameObject* \*frame)

获取 *frame* 为下一个外部帧。

返回一个 *strong reference*, 或者如果 *frame* 没有外部帧则返回 NULL。

在 3.9 版新加入。

**PyObject \*PyFrame\_GetBuiltins** (*PyFrameObject* \*frame)

获取 *frame* 的 *f\_builtins* 属性。

返回一个 *strong reference*。此结果不可为 NULL。

在 3.11 版新加入。



*PyCodeObject* \*PyFrame\_GetCode (*PyFrameObject* \*frame)

属于稳定 ABI 自 3.10 版起. 获取 *frame* 的代码。

返回一个 *strong reference*。

结果（帧代码）不可为 NULL。

在 3.9 版新加入。

*PyObject* \*PyFrame\_GetGenerator (*PyFrameObject* \*frame)

获取拥有该帧的生成器、协程或异步生成器，或者如果该帧不被某个生成器所拥有则为 NULL。不会引发异常，即使其返回值为 NULL。

返回一个 *strong reference*，或者 NULL。

在 3.11 版新加入。

*PyObject* \*PyFrame\_GetGlobals (*PyFrameObject* \*frame)

获取 *frame* 的 *f\_globals* 属性。

返回一个 *strong reference*。此结果不可为 NULL。

在 3.11 版新加入。

int PyFrame\_GetLasti (*PyFrameObject* \*frame)

获取 *frame* 的 *f\_lasti* 属性。

如果 *frame.f\_lasti* 为 None 则返回 -1。

在 3.11 版新加入。

*PyObject* \*PyFrame\_GetLocals (*PyFrameObject* \*frame)

获取 *frame* 的 *f\_locals* 属性 (dict)。

返回一个 *strong reference*。

在 3.11 版新加入。

int PyFrame\_GetLineNumber (*PyFrameObject* \*frame)

属于稳定 ABI 自 3.10 版起. 返回 *frame* 当前正在执行的行号。

## 8.6.10 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* 必須不為 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* 必須不為 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)。引數必須不為 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\_\_* 設為 *name* 和 *qualname*。此函式會取走一個對 *frame* 的參照。*frame* 引數必須不為 NULL。

### 8.6.11 Coroutine (協程) 物件

在 3.5 版新加入。

Coroutine 物件是那些以 `async` 關鍵字來宣告的函式所回傳的物件。

type **Py CoroObject**

用於 coroutine 物件的 C 結構。

*PyTypeObject* **Py Coro\_Type**

與 coroutine 物件對應的型物件。

int **Py Coro\_CheckExact** (*PyObject* \*ob)

如果 *ob* 的型是 *Py Coro\_Type* 則回傳真值；*ob* 必須不為 NULL。此函式總是會執行成功。

*PyObject* \***Py Coro\_New** (*PyFrameObject* \*frame, *PyObject* \*name, *PyObject* \*qualname)

回傳值：新的參照。基於 *frame* 物件來建立一個新的 coroutine 物件，其中 `__name__` 和 `__qualname__` 被設為 *name* 和 *qualname*。此函式會取得一個對 *frame* 的參照 (reference)。 *frame* 引數必須不為 NULL。

### 8.6.12 上下文变量对象

在 3.7 版新加入。

在 3.7.1 版的變更：

---

備註：在 Python 3.7.1 中，所有上下文变量 C API 的簽名被更改為使用 *PyObject* 指針而不是 *PyContext*，*PyContextVar* 以及 *PyContextToken*，例如：

```
// in 3.7.0:
PyContext *PyContext_New(void);

// in 3.7.1+:
PyObject *PyContext_New(void);
```

更多細節請見 [bpo-34762](#)。

---

本节深入介绍了 `contextvars` 模块的公用 C API。

type **PyContext**

用于表示 `contextvars.Context` 对象的 C 结构体。

type **PyContextVar**

用于表示 `contextvars.ContextVar` 对象的 C 结构体。

type **PyContextToken**

用于表示 `contextvars.Token` 对象的 C 结构体。

*PyTypeObject* **PyContext\_Type**

表示 `context` 类型的类型对象。

*PyTypeObject* **PyContextVar\_Type**

表示 `context variable` 类型的类型对象。

*PyTypeObject* **PyContextToken\_Type**

表示 `context variable token` 类型的类型对象。

类型检查宏：

int **PyContext\_CheckExact** (*PyObject* \*o)

如果 *o* 的类型为 *PyContext\_Type* 则返回真值。*o* 必须不为 NULL。此函数总是会成功执行。

**int PyContextVar\_CheckExact** (*PyObject* \*o)

如果 *o* 的类型为 *PyContextVar\_Type* 则返回真值。*o* 必须不为 NULL。此函数总是会成功执行。

**int PyContextToken\_CheckExact** (*PyObject* \*o)

如果 *o* 的类型为 *PyContextToken\_Type* 则返回真值。*o* 必须不为 NULL。此函数总是会成功执行。

上下文对象管理函数:

*PyObject* \***PyContext\_New** (void)

回傳值: 新的參照。创建一个新的空上下文对象。如果发生错误则返回 NULL。

*PyObject* \***PyContext\_Copy** (*PyObject* \*ctx)

回傳值: 新的參照。创建所传入的 *ctx* 上下文对象的浅拷贝。如果发生错误则返回 NULL。

*PyObject* \***PyContext\_CopyCurrent** (void)

回傳值: 新的參照。创建当前线程上下文的浅拷贝。如果发生错误则返回 NULL。

**int PyContext\_Enter** (*PyObject* \*ctx)

将 *ctx* 设为当前线程的当前上下文。成功时返回 0，出错时返回 -1。

**int PyContext\_Exit** (*PyObject* \*ctx)

取消激活 *ctx* 上下文并将之前的上下文恢复为当前线程的当前上下文。成功时返回 0，出错时返回 -1。

上下文变量函数:

*PyObject* \***PyContextVar\_New** (const char \*name, *PyObject* \*def)

回傳值: 新的參照。创建一个新的 ContextVar 对象。形参 *name* 用于自我检查和调试目的。形参 *def* 为上下文变量指定默认值，或为 NULL 表示无默认值。如果发生错误，这个函数会返回 NULL。

**int PyContextVar\_Get** (*PyObject* \*var, *PyObject* \*default\_value, *PyObject* \*\*value)

获取上下文变量的值。如果在查找过程中发生错误，返回 '-1'，如果没有发生错误，无论是否找到值，都返回 '0'，

如果找到上下文变量，*value* 将是指向它的指针。如果上下文变量没有找到，*value* 将指向:

- *default\_value*，如果非 "NULL"；
- *var* 的默认值，如果不是 NULL；
- NULL

除了返回 NULL，这个函数会返回一个新的引用。

*PyObject* \***PyContextVar\_Set** (*PyObject* \*var, *PyObject* \*value)

回傳值: 新的參照。在当前上下文中将 *var* 设为 *value*。返回针对此修改的新凭据对象，或者如果发生错误则返回 NULL。

**int PyContextVar\_Reset** (*PyObject* \*var, *PyObject* \*token)

将上下文变量 *var* 的状态重置为它在返回 *token* 的 *PyContextVar\_Set()* 被调用之前的状态。此函数成功时返回 0，出错时返回 -1。

### 8.6.13 DateTime 物件

`datetime` 模块提供了各种日期和时间对象。在使用这些函数之前，必须要在你的源代码中包含头文件 `datetime.h` (请注意此文件并未包括在 `Python.h` 中)，并且 `PyDateTime_IMPORT` 必须被发起调用，通常是作为模块初始化函数的一部分。这个宏会将指向特定 C 结构体的指针放入一个静态变量 `PyDateTimeAPI` 中，它将被下列的宏所使用。

**type PyDateTime\_Date**

*PyObject* 的这个子类型表示 Python 日期对象。

type **PyDateTime\_DateTime**

*PyObject* 的这个子类型表示 Python 日期时间对象。

type **PyDateTime\_Time**

*PyObject* 的这个子类型表示 Python 时间对象。

type **PyDateTime\_Delta**

*PyObject* 的这个子类型表示两个日期时间值之间的差值。

*PyTypeObject* **PyDateTime\_DateType**

这个 *PyTypeObject* 的实例代表 Python 日期类型；它与 Python 层面的 `datetime.date` 对象相同。

*PyTypeObject* **PyDateTime\_DateTimeType**

这个 *PyTypeObject* 的实例代表 Python 日期时间类型；它与 Python 层面的 `datetime.datetime` 对象相同。

*PyTypeObject* **PyDateTime\_TimeType**

这个 *PyTypeObject* 的实例代表 Python 时间类型；它与 Python 层面的 `datetime.time` 对象相同。

*PyTypeObject* **PyDateTime\_DeltaType**

这个 *PyTypeObject* 的实例是代表两个日期时间值之间差值的 Python 类型；它与 Python 层面的 `datetime.timedelta` 对象相同。

*PyTypeObject* **PyDateTime\_TZInfoType**

这个 *PyTypeObject* 的实例代表 Python 时区信息类型；它与 Python 层面的 `datetime.tzinfo` 对象相同。

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 版新加入。

型 檢查巨集：

int **PyDate\_Check** (*PyObject* \*ob)

如果 *ob* 为 *PyDateTime\_DateType* 类型或 *PyDateTime\_DateType* 的某个子类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyDate\_CheckExact** (*PyObject* \*ob)

如果 *ob* 的型 为 *PyDateTime\_DateType*，則回傳 true。*ob* 不得 为 NULL。這個函式一定會執行成功。

int **PyDateTime\_Check** (*PyObject* \*ob)

如果 *ob* 为 *PyDateTime\_DateTimeType* 类型或 *PyDateTime\_DateTimeType* 的某个子类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyDateTime\_CheckExact** (*PyObject* \*ob)

如果 *ob* 的型 为 *PyDateTime\_DateTimeType*，則回傳 true。*ob* 不得 为 NULL。這個函式一定會執行成功。

int **PyTime\_Check** (*PyObject* \*ob)

如果 *ob* 为 *PyDateTime\_TimeType* 类型或 *PyDateTime\_TimeType* 的某个子类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyTime\_CheckExact** (*PyObject* \*ob)

如果 *ob* 的型 为 *PyDateTime\_TimeType*，則回傳 true。*ob* 不得 为 NULL。這個函式一定會執行成功。

**int PyDelta\_Check** (*PyObject* \*ob)

如果 *ob* 为 *PyDateTime\_DeltaType* 类型或 *PyDateTime\_DeltaType* 的某个子类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

**int PyDelta\_CheckExact** (*PyObject* \*ob)

如果 *ob* 的型为 *PyDateTime\_DeltaType*，則回傳 true。*ob* 不得为 NULL。這個函式一定會執行成功。

**int PyTZInfo\_Check** (*PyObject* \*ob)

如果 *ob* 为 *PyDateTime\_TZInfoType* 类型或 *PyDateTime\_TZInfoType* 的某个子类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

**int PyTZInfo\_CheckExact** (*PyObject* \*ob)

如果 *ob* 的型为 *PyDateTime\_TZInfoType*，則回傳 true。*ob* 不得为 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 版新加入。

*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.time*。

*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 版新加入。

*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 版新加入。

*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 版新加入。

一些用来从日期对象中提取字段的宏。参数必须是 *PyDateTime\_Date* 包括其子类 (如 *PyDateTime\_DateTime*) 的实例。参数不能为 NULL，且不会检查类型：

**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。

一些用来从日期时间对象中提取字段的宏。参数必须是 *PyDateTime\_DateTime* 包括其子类的实例。参数不能为 NULL，并且不会检查类型：

**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 版新加入。

**PyObject \*PyDateTime\_DATE\_GET\_TZINFO** (*PyDateTime\_DateTime* \*o)

回傳 tzinfo (可能是 None)。

在 3.10 版新加入。

一些用来从时间对象中提取字段的宏。参数必须是 *PyDateTime\_Time* 包括其子类的实例。参数不能为 NULL，且不会检查类型：

**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 版新加入。

**PyObject \*PyDateTime\_TIME\_GET\_TZINFO** (*PyDateTime\_Time* \*o)

回傳 tzinfo (可能是 None)。

在 3.10 版新加入。

一些用来从时间差对象中提取字段的宏。参数必须是 *PyDateTime\_Delta* 包括其子类的实例。参数不能为 NULL，并且不会检查类型：

**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 版新加入。



`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 版新加入。

`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 版新加入。

為了方便模組實作 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。

## 8.6.14 型提示物件

提供了數個用於型提示的型。目前有兩種 -- `GenericAlias` 和 `Union`。只有 `GenericAlias` 有公開 (expose) 給 C。

`PyObject* Py_GenericAlias (PyObject *origin, PyObject *args)`

屬於穩定 ABI 自 3.9 版起。建立一個 `GenericAlias` 物件，等同於呼叫 Python 的 `types.GenericAlias` class。`origin` 和 `args` 引數分別設定了 `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為 1 的 tuple 且 `__args__` 會被設為 `(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[] = {
    // Other methods.
    ...
    {"__class_getitem__", Py_GenericAlias, METH_O|METH_CLASS, "See PEP 585"}
    ...
}
```

也參考：

數據模型方法 `__class_getitem__()`。

在 3.9 版新加入。

`PyTypeObject Py_GenericAliasType`

屬於穩定 ABI 自 3.9 版起。`Py_GenericAlias()` 所回傳該物件的 C 型。等價於 Python 中的 `types.GenericAlias`。

在 3.9 版新加入。



---

## 初始化，最终化和线程

---

请参阅Python 初始化配置。

### 9.1 在 Python 初始化之前

在一个植入了 Python 的应用程序中，`Py_Initialize()` 函数必须在任何其他 Python/C API 函数之前被调用；例外的只有个别函数和全局配置变量。

在初始化 Python 之前，可以安全地调用以下函数：

- 配置函数：

- `PyImport_AppendInittab()`
- `PyImport_ExtendInittab()`
- `PyInitFrozenExtensions()`
- `PyMem_SetAllocator()`
- `PyMem_SetupDebugHooks()`
- `PyObject_SetArenaAllocator()`
- `Py_SetPath()`
- `Py_SetProgramName()`
- `Py_SetPythonHome()`
- `Py_SetStandardStreamEncoding()`
- `PySys_AddWarnOption()`
- `PySys_AddXOption()`
- `PySys_ResetWarnOptions()`

- 信息函数：

- `Py_IsInitialized()`
- `PyMem_GetAllocator()`
- `PyObject_GetArenaAllocator()`

- `Py_GetBuildInfo()`
- `Py_GetCompiler()`
- `Py_GetCopyright()`
- `Py_GetPlatform()`
- `Py_GetVersion()`

- 工具

- `Py_DecodeLocale()`

- 内存分配器:

- `PyMem_RawMalloc()`
  - `PyMem_RawRealloc()`
  - `PyMem_RawCalloc()`
  - `PyMem_RawFree()`

---

備 註: 以下函数不应该在 `Py_Initialize()`: `Py_EncodeLocale()`, `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, `Py_GetProgramFullPath()`, `Py_GetPythonHome()`, `Py_GetProgramName()` 和 `PyEval_InitThreads()` 前调用。

---

## 9.2 全局配置变量

Python 有负责控制全局配置中不同特性和选项的变量。这些标志默认被 命令行选项。

当一个选项设置一个旗标时, 该旗标的值将是设置选项的次数。例如, `-b` 会将 `Py_BytesWarningFlag` 设为 1 而 `-bb` 会将 `Py_BytesWarningFlag` 设为 2。

### `int Py_BytesWarningFlag`

当将 `bytes` 或 `bytearray` 与 `str` 比较或者将 `bytes` 与 `int` 比较时发出警告。如果大于等于 2 则报错。

由 `-b` 选项设定。

### `int Py_DebugFlag`

开启解析器调试输出 (限专家使用, 依赖于编译选项)。

由 `-d` 选项与 `PYTHONDEBUG` 环境变数设定。

### `int Py_DontWriteBytecodeFlag`

如果设置为非零, Python 不会在导入源代码时尝试写入 `.pyc` 文件

由 `-B` 选项与 `PYTHONDONTWRITEBYTECODE` 环境变数设定。

### `int Py_FrozenFlag`

当在 `Py_GetPath()` 中计算模块搜索路径时屏蔽错误消息。

由 `_freeze_importlib` 和 `frozenmain` 程序使用的私有旗标。

### `int Py_HashRandomizationFlag`

如果环境变数 `PYTHONHASHSEED` 被设定一个非空字符串则设 1。

如果该旗标为非零值, 则读取 `PYTHONHASHSEED` 环境变量来初始化加密哈希种子。

### `int Py_IgnoreEnvironmentFlag`

忽略所有 `PYTHON*` 环境变量, 例如可能设置的 `PYTHONPATH` 和 `PYTHONHOME`。

由 `-E` 与 `-I` 选项设定。

**int Py\_InspectFlag**

当将脚本作为第一个参数传入或是使用了 `-c` 选项时，则会在执行该脚本或命令后进入交互模式，即使在 `sys.stdin` 并非一个终端时也是如此。

由 `-i` 選項與 `PYTHONINSPECT` 環境變數設定。

**int Py\_InteractiveFlag**

由 `-i` 選項設定。

**int Py\_IsolatedFlag**

以隔离模式运行 Python。在隔离模式下 `sys.path` 将不包含脚本的目录或用户的 `site-packages` 目录。

由 `-i` 選項設定。

在 3.4 版新加入。

**int Py\_LegacyWindowsFSEncodingFlag**

如果该旗标为非零值，则使用 `mbcs` 编码和“replace”错误处理器，而不是 UTF-8 编码和 `surrogatepass` 错误处理器作用 *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](#)。

適用：Windows。

**int Py\_LegacyWindowsStdioFlag**

如果该旗标为非零值，则会使用 `io.FileIO` 而不是 `io._WindowsConsoleIO` 作为 `sys` 标准流。

如果 `PYTHONLEGACYWINDOWSSTDIO` 环境变量被设为非空字符串则设为 1。

更多詳情請見 [PEP 528](#)。

適用：Windows。

**int Py\_NoSiteFlag**

禁用 `site` 的导入及其所附带的基于站点对 `sys.path` 的操作。如果 `site` 会在稍后被显式地导入也会禁用这些操作 (如果你希望触发它们则应调用 `site.main()`)。

由 `-S` 選項設定。

**int Py\_NoUserSiteDirectory**

不要将用户 `site-packages` 目录添加到 `sys.path`。

由 `-s` 選項、`-I` 選項與 `PYTHONNOUSERSITE` 環境變數設定。

**int Py\_OptimizeFlag**

由 `-O` 選項與 `PYTHONOPTIMIZE` 環境變數設定。

**int Py\_QuietFlag**

即使在交互模式下也不显示版权和版本信息。

由 `-q` 選項設定。

在 3.2 版新加入。

**int Py\_UnbufferedStdioFlag**

强制 `stdout` 和 `stderr` 流不带缓冲。

由 `-u` 選項與 `PYTHONUNBUFFERED` 環境變數設定。

**int Py\_VerboseFlag**

每次初始化模块时打印一条消息，显示加载模块的位置（文件名或内置模块）。如果大于或等于 2，则为搜索模块时检查的每个文件打印一条消息。此外还会在退出时提供模块清理信息。

由 `-v` 選項與 `PYTHONVERBOSE` 環境變數設定。

## 9.3 初始化和最终化解释器

**void Py\_Initialize()**

属于稳定 ABI。初始化 Python 解释器。在嵌入 Python 的应用程序中，它应当在使用任何其他 Python/C API 函数之前被调用；请参阅在 *Python 初始化之前* 了解少数的例外情况。

这将初始化已加载模块表 (sys.modules)，并创建基本模块 builtins、\_\_main\_\_ 和 sys。它还会初始化模块搜索路径 (sys.path)。它不会设置 sys.argv；如有需要请使用 *PySys\_SetArgvEx()*。当第二次调用时 (在未事先调用 *Py\_FinalizeEx()* 的情况下) 将不会执行任何操作。它没有返回值；如果初始化失败则会发生致命错误。

---

**備註：** 在 Windows 上，将控制台模式从 O\_TEXT 改为 O\_BINARY，这还将影响使用 C 运行时的非 Python 的控制台使用。

---

**void Py\_InitializeEx(int initsigs)**

属于稳定 ABI。如果 *initsigs* 为 1 则该函数的工作方式与 *Py\_Initialize()* 类似。如果 *initsigs* 为 0，它将跳过信号处理器的初始化注册，这在嵌入 Python 时可能会很有用处。

**int Py\_IsInitialized()**

属于稳定 ABI。如果 Python 解释器已初始化，则返回真值（非零）；否则返回假值（零）。在调用 *Py\_FinalizeEx()* 之后，此函数将返回假值直到 *Py\_Initialize()* 再次被调用。

**int Py\_FinalizeEx()**

属于稳定 ABI 自 3.6 版起。撤销 *Py\_Initialize()* 所做的所有初始化操作和后续对 Python/C API 函数的使用，并销毁自上次调用 *Py\_Initialize()* 以来创建但尚未销毁的所有子解释器（参见下文 *Py\_NewInterpreter()* 一节）。在理想情况下，这会释放 Python 解释器分配的所有内存。当第二次调用时（在未再次调用 *Py\_Initialize()* 的情况下），这将不执行任何操作。正常情况下返回值是 0。如果在最终化（刷新缓冲数据）过程中出现错误，则返回 -1。

提供此函数的原因有很多。嵌入应用程序可能希望重新启动 Python，而不必重新启动应用程序本身。从动态可加载库（或 DLL）加载 Python 解释器的应用程序可能希望在卸载 DLL 之前释放 Python 分配的所有内存。在搜索应用程序内存泄漏的过程中，开发人员可能希望在退出应用程序之前释放 Python 分配的所有内存。

**程序问题和注意事项：** 模块和模块中对象的销毁是按随机顺序进行的；这可能导致依赖于其他对象（甚至函数）或模块的析构器（即 *\_\_del\_\_()* 方法）出错。Python 所加载的动态加载扩展模块不会被卸载。Python 解释器所分配的少量内存可能不会被释放（如果发现内存泄漏，请报告问题）。对象间循环引用所占用的内存不会被释放。扩展模块所分配的某些内存可能不会被释放。如果某些扩展的初始化例程被调用多次它们可能无法正常工作；如果应用程序多次调用了 *Py\_Initialize()* 和 *Py\_FinalizeEx()* 就可能发生这种情况。

引發一個不附帶引數的稽核事件 *cpython.\_PySys\_ClearAuditHooks*。

在 3.6 版新加入。

**void Py\_Finalize()**

属于稳定 ABI。这是一个不考虑返回值的 *Py\_FinalizeEx()* 的向下兼容版本。



## 9.4 进程级参数

`int Py_SetStandardStreamEncoding(const char *encoding, const char *errors)`

此 API 被保留用于向下兼容：应当改为设置 `PyConfig.stdio_encoding` 和 `PyConfig.stdio_errors`，参见 *Python* 初始化配置。

如果要调用该函数，应当在 `Py_Initialize()` 之前调用。它指定了标准 IO 使用的编码格式和错误处理方式，其含义与 `str.encode()` 中的相同。

它覆盖了 `PYTHONIOENCODING` 的值，并允许嵌入代码以便在环境变量不起作用时控制 IO 编码格式。

`encoding` 和/或 `errors` 可以为 `NULL` 以使用 `PYTHONIOENCODING` 和/或默认值（取决于其他设置）。

请注意无论是否有此设置（或任何其他设置），`sys.stderr` 都会使用“backslashreplace”错误处理器。

如果调用了 `Py_FinalizeEx()`，则需要再次调用该函数以便影响对 `Py_Initialize()` 的后续调用。

成功时返回 0，出错时返回非零值（例如在解释器已被初始化后再调用）。

在 3.4 版新加入。

在 3.11 版之後被 用。

`void Py_SetProgramName(const wchar_t *name)`

属于稳定 ABI。此 API 被保留用于向下兼容：应当改为设置 `PyConfig.program_name`，参见 *Python* 初始化配置。

如果要调用该函数，应当在首次调用 `Py_Initialize()` 之前调用它。它将告诉解释器程序的 `main()` 函数的 `argv[0]` 参数的值（转换为宽字符）。`Py_GetPath()` 和下面的某些其他函数会使用它在相对于解释器的位置上查找可执行文件的 Python 运行时库。默认值是 'python'。参数应当指向静态存储中的一个以零值结束的宽字符串，其内容在程序执行期间不会发生改变。Python 解释器中的任何代码都不会改变该存储的内容。

使用 `Py_DecodeLocale()` 来解码字节串以得到一个 `wchar_t*` 字符串。

在 3.11 版之後被 用。

`wchar_t *Py_GetProgramName()`

属于稳定 ABI。返回用 `Py_SetProgramName()` 设置的程序名称，或默认的名称。返回的字符串指向静态存储；调用者不应修改其值。

此函数不应在 `Py_Initialize()` 之前被调用，否则将返回 `NULL`。

在 3.10 版的變更：现在如果它在 `Py_Initialize()` 之前被调用将返回 `NULL`。

`wchar_t *Py_GetPrefix()`

属于稳定 ABI。返回针对已安装的独立于平台文件的 `prefix`。这是通过基于使用 `Py_SetProgramName()` 设置的程序名称和某些环境变量所派生的一系列复杂规则获得的；举例来说，如果程序名称为 '/usr/local/bin/python'，则 `prefix` 为 '/usr/local'。返回的字符串将指向静态存储；调用方不应修改其值。这对应于最高层级 Makefile 中的 `prefix` 变量以及在编译时传给 `configure` 脚本的 `--prefix` 参数。该值将以 `sys.prefix` 的名称供 Python 代码使用。它仅适用于 Unix。另请参见下一个函数。

此函数不应在 `Py_Initialize()` 之前被调用，否则将返回 `NULL`。

在 3.10 版的變更：现在如果它在 `Py_Initialize()` 之前被调用将返回 `NULL`。

`wchar_t *Py_GetExecPrefix()`

属于稳定 ABI。返回针对已安装的依赖于平台文件的 `exec-prefix`。这是通过基于使用 `Py_SetProgramName()` 设置的程序名称和某些环境变量所派生的一系列复杂规则获得的；举例来说，如果程序名称为 '/usr/local/bin/python'，则 `exec-prefix` 为 '/usr/local'。返回的字符串将指向静态存储；调用方不应修改其值。这对应于最高层级 Makefile 中的 `exec_prefix`

变量以及在编译时传给 **configure** 脚本的 `--exec-prefix` 参数。该值将以 `sys.exec_prefix` 的名称供 Python 代码使用。它仅适用于 Unix。

背景：当依赖于平台的文件（如可执行文件和共享库）是安装于不同的目录树中的时候 `exec-prefix` 将会不同于 `prefix`。在典型的安装中，依赖于平台的文件可能安装于 `the /usr/local/plat` 子目录树而独立于平台的文件可能安装于 `/usr/local`。

总而言之，平台是一组硬件和软件资源的组合，例如所有运行 Solaris 2.x 操作系统的 Sparc 机器会被视为相同平台，但运行 Solaris 2.x 的 Intel 机器是另一种平台，而运行 Linux 的 Intel 机器又是另一种平台。相同操作系统的不同主要发布版通常也会构成不同的平台。非 Unix 操作系统的情况又有所不同；这类系统上的安装策略差别巨大因此 `prefix` 和 `exec-prefix` 是没有意义的，并将被设为空字符串。请注意已编译的 Python 字节码是独立于平台的（但并不独立于它们编译时所使用的 Python 版本!）

系统管理员知道如何配置 **mount** 或 **automount** 程序以在平台间共享 `/usr/local` 而让 `/usr/local/plat` 成为针对不同平台的不同文件系统。

此函数不应在 `Py_Initialize()` 之前被调用，否则将返回 NULL。

在 3.10 版的變更：现在如果它在 `Py_Initialize()` 之前被调用将返回 NULL。

`wchar_t *Py_GetProgramFullPath()`

属于稳定 ABI。返回 Python 可执行文件的完整程序名称；这是作为根据程序名称（由上述 `Py_SetProgramName()` 设置）派生默认模块搜索路径的附带影响计算得出的。返回的字符串将指向静态存储；调用方不应修改其值。该值将以 `sys.executable` 的名称供 Python 代码使用。

此函数不应在 `Py_Initialize()` 之前被调用，否则将返回 NULL。

在 3.10 版的變更：现在如果它在 `Py_Initialize()` 之前被调用将返回 NULL。

`wchar_t *Py_GetPath()`

属于稳定 ABI。返回默认模块搜索路径；这是根据程序名称（由上述 `Py_SetProgramName()` 设置）和某些环境变量计算得出的。返回的字符串由一系列由依赖于平台的分隔符分开的目录名称组成。分隔符在 Unix 和 macOS 上为 `':'` 而在 Windows 上为  `';'` 。返回的字符串将指向静态存储；调用方不应修改其值。列表 `sys.path` 将在解释器启动时使用该值来初始化；它可以在随后被修改（并且通常都会被修改）以变更加载模块的搜索路径。

此函数不应在 `Py_Initialize()` 之前被调用，否则将返回 NULL。

在 3.10 版的變更：现在如果它在 `Py_Initialize()` 之前被调用将返回 NULL。

`void Py_SetPath(const wchar_t*)`

属于稳定 ABI 自 3.7 版起。此 API 被保留用于向下兼容：应当改为采用设置 `PyConfig.module_search_paths` 和 `PyConfig.module_search_paths_set`，参见 *Python 初始化配置*。

设置默认的模块搜索路径。如果此函数在 `Py_Initialize()` 之前被调用，则 `Py_GetPath()` 将不会尝试计算默认的搜索路径而是改用已提供的路径。这适用于由一个完全知晓所有模块的位置的应用程序来嵌入 Python 的情况。路径组件应当由平台专属的分隔符来分隔，在 Unix 和 macOS 上是 `':'` 而在 Windows 上则是  `';'` 。

这也将导致 `sys.executable` 被设为程序的完整路径（参见 `Py_GetProgramFullPath()`）而 `sys.prefix` 和 `sys.exec_prefix` 变为空值。如果在调用 `Py_Initialize()` 之后有需要则应由调用方来修改它们。

使用 `Py_DecodeLocale()` 来解码字节串以得到一个 `wchar_t*` 字符串。

路径参数会在内部被复制，使调用方可以在调用结束后释放它。

在 3.8 版的變更：现在 `sys.executable` 将使用程序的完整路径，而不是程序文件名。

在 3.11 版之後被弃用。

`const char *Py_GetVersion()`

属于稳定 ABI。返回 Python 解释器的版本。这将为如下形式的字符串

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

第一个单词（到第一个空格符为止）是当前的 Python 版本；前面的字符是以点号分隔的主要和次要版本号。返回的字符串将指向静态存储；调用方不应修改其值。该值将以 `sys.version` 的名称供 Python 代码使用。

另请参阅 `Py_Version` 常量。

**const char \*Py\_GetPlatform()**

属于稳定 ABI。返回当前平台的平台标识符。在 Unix 上，这将以操作系统的“官方”名称为基础，转换为小写形式，再加上主版本号；例如，对于 Solaris 2.x，或称 SunOS 5.x，该值将为 'sunos5'。在 macOS 上，它将为 'darwin'。在 Windows 上它将为 'win'。返回的字符串指向静态存储；调用方不应修改其值。Python 代码可通过 `sys.platform` 获取该值。

**const char \*Py\_GetCopyright()**

属于稳定 ABI。返回当前 Python 版本的官方版权字符串，例如

```
'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'
```

返回的字符串指向静态存储；调用者不应修改其值。Python 代码可通过 `sys.copyright` 获取该值。

**const char \*Py\_GetCompiler()**

属于稳定 ABI。返回用于编译当前 Python 版本的编译器指令，为带方括号的形式，例如：

```
"[GCC 2.7.2.2]"
```

返回的字符串指向静态存储；调用者不应修改其值。Python 代码可以从变量 `sys.version` 中获取该值。

**const char \*Py\_BuildInfo()**

属于稳定 ABI。返回有关当前 Python 解释器实例的序列号和构建日期和时间的信息，例如：

```
"#67, Aug 1 1997, 22:34:28"
```

返回的字符串指向静态存储；调用者不应修改其值。Python 代码可以从变量 `sys.version` 中获取该值。

**void PySys\_SetArgvEx(int argc, wchar\_t \*\*argv, int updatepath)**

属于稳定 ABI。此 API 被保留用于向下兼容：应当改为设置 `PyConfig.argv`、`PyConfig.parse_argv` 和 `PyConfig.safe_path`，参见 [Python 初始化配置](#)。

根据 `argc` 和 `argv` 设置 `sys.argv`。这些形参与传给程序的 `main()` 函数的类似，区别在于第一项应当指向要执行的脚本文件而不是 Python 解释器对应的可执行文件。如果没有要运行的脚本，则 `argv` 中的第一项可以为空字符串。如果此函数无法初始化 `sys.argv`，则将使用 `Py_FatalError()` 发出严重情况信号。

如果 `updatepath` 为零，此函数将完成操作。如果 `updatepath` 为非零值，则此函数还将根据以下算法修改 `sys.path`：

- 如果在 `argv[0]` 中传入一个现有脚本，则脚本所在目录的绝对路径将被添加到 `sys.path` 的开头。
- 在其他情况下（也就是说，如果 `argc` 为 0 或 `argv[0]` 未指向现有文件名），则将在 `sys.path` 的开头添加一个空字符串，这等价于添加当前工作目录（"."）。

使用 `Py_DecodeLocale()` 来解码字节串以得到一个 `wchar_t*` 字符串。

另请参阅 [Python 初始化配置](#) 的 `PyConfig.orig_argv` 和 `PyConfig.argv` 成员。

**備註：**建议在出于执行单个脚本以外的目的嵌入 Python 解释器的应用程序传入 0 作为 `updatepath`，并在需要时更新 `sys.path` 本身。参见 [CVE-2008-5983](#)。

在 3.1.3 之前的版本中，你可以通过在调用 `PySys_SetArgv()` 之后手动弹出第一个 `sys.path` 元素，例如使用：

```
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```

在 3.1.3 版新加入。

在 3.11 版之後被 用。

void **PySys\_SetArgv** (int argc, wchar\_t \*\*argv)

属于稳定 ABI。此 API 仅为向下兼容而保留：应当改为设置 `PyConfig.argv` 并改用 `PyConfig.parse_argv`，参见 *Python 初始化配置*。

此函数相当于 `PySys_SetArgvEx()` 设置了 `updatepath` 为 1 除非 **python** 解释器启动时附带了 `-I`。

使用 `Py_DecodeLocale()` 来解码字节串以得到一个 `wchar_t*` 字符串。

另请参阅 *Python 初始化配置* 的 `PyConfig.orig_argv` 和 `PyConfig.argv` 成员。

在 3.4 版的變更: `updatepath` 值依赖于 `-I`。

在 3.11 版之後被 用。

void **Py\_SetPythonHome** (const wchar\_t \*home)

属于稳定 ABI。此 API 被保留用于向下兼容：应当改为设置 `PyConfig.home`，参见 *Python 初始化配置*。

设置默认的“home”目录，也就是标准 Python 库所在的位置。请参阅 `PYTHONHOME` 了解该参数字符串的含义。

此参数应当指向静态存储中一个以零值结束的字符串，其内容在程序执行期间将保持不变。Python 解释器中的代码绝不会修改此存储中的内容。

使用 `Py_DecodeLocale()` 来解码字节串以得到一个 `wchar_t*` 字符串。

在 3.11 版之後被 用。

wchar\_t \***Py\_GetPythonHome** ()

属于稳定 ABI。返回默认的“home”，就是由之前对 `Py_SetPythonHome()` 的调用所设置的值，或者在设置了 `PYTHONHOME` 环境变量的情况下该环境变量的值。

此函数不应在 `Py_Initialize()` 之前被调用，否则将返回 `NULL`。

在 3.10 版的變更: 现在如果它在 `Py_Initialize()` 之前被调用将返回 `NULL`。

## 9.5 线程状态和全局解释器锁

Python 解释器不是完全线程安全的。为了支持多线程的 Python 程序，设置了一个全局锁，称为 *global interpreter lock* 或 *GIL*，当前线程必须在持有它之后才能安全地访问 Python 对象。如果没有这个锁，即使最简单的操作也可能在多线程的程序中导致问题：例如，当两个线程同时增加相同对象的引用计数时，引用计数可能最终只增加了一次而不是两次。

因此，规则要求只有获得 *GIL* 的线程才能在 Python 对象上执行操作或调用 Python/C API 函数。为了模拟并发执行，解释器会定期尝试切换线程（参见 `sys.setswitchinterval()`）。锁也会在读写文件等可能造成阻塞的 I/O 操作时释放，以便其他 Python 线程可以同时运行。

Python 解释器会在一个名为 `PyThreadState` 的数据结构体中保存一些线程专属的记录信息。还有一个全局变量指向当前的 `PyThreadState`：它可以使用 `PyThreadState_Get()` 来获取。



### 9.5.1 从扩展代码中释放 GIL

大多数操作 *GIL* 的扩展代码具有以下简单结构：

```
Save the thread state in a local variable.
Release the global interpreter lock.
... Do some blocking I/O operation ...
Reacquire the global interpreter lock.
Restore the thread state from the local variable.
```

这是如此常用因此增加了一对宏来简化它：

```
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

`Py_BEGIN_ALLOW_THREADS` 宏将打开一个新块并声明一个隐藏的局部变量；`Py_END_ALLOW_THREADS` 宏将关闭这个块。

上面的代码块可扩展为下面的代码：

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

这些函数的工作原理如下：全局解释器锁被用来保护指向当前线程状态的指针。当释放锁并保存线程状态时，必须在锁被释放之前获取当前线程状态指针（因为另一个线程可以立即获取锁并将自己的线程状态存储到全局变量中）。相应地，当获取锁并恢复线程状态时，必须在存储线程状态指针之前先获取锁。

**備註：**调用系统 I/O 函数是释放 GIL 的最常见用例，但它在调用不需要访问 Python 对象的长期运行计算，比如针对内存缓冲区进行操作的压缩或加密函数之前也很有用。举例来说，在对数据执行压缩或哈希操作时标准 `zlib` 和 `hashlib` 模块就会释放 GIL。

### 9.5.2 非 Python 创建的线程

当使用专门的 Python API（如 `threading` 模块）创建线程时，会自动关联一个线程状态因而上面显示的代码是正确的。但是，如果线程是用 C 创建的（例如由具有自己的线程管理的第三方库创建），它们就不持有 GIL 也没有对应的线程状态结构体。

如果你需要从这些线程调用 Python 代码（这通常会上述第三方库所提供的回调 API 的一部分），你必须首先通过创建线程状态数据结构体向解释器注册这些线程，然后获取 GIL，最后存储它们的线程状态指针，这样你才能开始使用 Python/C API。完成以上步骤后，你应当重置线程状态指针，释放 GIL，最后释放线程状态数据结构体。

`PyGILState_Ensure()` 和 `PyGILState_Release()` 函数会自动完成上述的所有操作。从 C 线程调用到 Python 的典型方式如下：

```
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);
```

请注意 `PyGILState_*` 函数会假定只有一个全局解释器（由 `Py_Initialize()` 自动创建）。Python 支持创建额外的解释器（使用 `Py_NewInterpreter()` 创建），但不支持混合使用多个解释器和 `PyGILState_*` API。

### 9.5.3 有关 `fork()` 的注意事项

有关线程的另一个需要注意的重要问题是它们在面对 C `fork()` 调用时的行为。在大多数支持 `fork()` 的系统中，当一个进程执行 `fork` 之后将只有发出 `fork` 的线程存在。这对需要如何处理锁以及 CPython 的运行时内所有的存储状态都会有实质性的影响。

只保留“当前”线程这一事实意味着任何由其他线程所持有的锁永远不会被释放。Python 通过在 `fork` 之前获取内部使用的锁，并随后释放它们的方式为 `os.fork()` 解决了这个问题。此外，它还会重置子进程中的任何 lock-objects。在扩展或嵌入 Python 时，没有办法通知 Python 在 `fork` 之前或之后需要获取或重置的附加（非 Python）锁。需要使用 OS 工具例如 `pthread_atfork()` 来完成同样的事情。此外，在扩展或嵌入 Python 时，直接调用 `fork()` 而不是通过 `os.fork()`（并返回到或调用至 Python 中）调用可能会导致某个被 `fork` 之后失效的线程所持有的 Python 内部锁发生死锁。`PyOS_AfterFork_Child()` 会尝试重置必要的锁，但并不总是能够做到。

所有其他线程都将结束这一事实也意味着 CPython 的运行时状态必须妥善清理，`os.fork()` 就是这样做的。这意味着最终化归属于当前解释器的所有其他 `PyThreadState` 对象以及所有其他 `PyInterpreterState` 对象。由于这一点以及“main”解释器的特殊性质，`fork()` 应当只在该解释器的“main”线程中被调用，而 CPython 全局运行时最初就是在该线程中初始化的。只有当 `exec()` 将随后立即被调用的情况是唯一的例外。

### 9.5.4 高階 API

这些是在编写 C 扩展代码或在嵌入 Python 解释器时最常用的类型和函数：

type **PyInterpreterState**

属于受限 API（作为不透明的结构体）。该数据结构代表多个合作线程所共享的状态。属于同一解释器的线程将共享其模块管理以及其他一些内部条目。该结构体中不包含公有成员。

最初归属于不同解释器的线程不会共享任何东西，但进程状态如可用内存、打开的文件描述符等等除外。全局解释器锁也会被所有线程共享，无论它们归属于哪个解释器。

type **PyThreadState**

属于受限 API（作为不透明的结构体）。该数据结构代表单个线程的状态。唯一的公有数据成员为：

`PyInterpreterState *interp`

该线程的解释器状态。

void **PyEval\_InitThreads()**

属于稳定 ABI。不执行任何操作的已弃用函数。

在 Python 3.6 及更老的版本中，此函数会在 GIL 不存在时创建它。

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：该函数现在由 `Py_Initialize()` 调用，因此你无需再自行调用它。

在 3.2 版的變更：此函数已不再被允许在 `Py_Initialize()` 之前调用。

在 3.9 版之後被弃用。

int **PyEval\_ThreadsInitialized()**

属于稳定 ABI。如果 `PyEval_InitThreads()` 已经被调用则返回非零值。此函数可在不持有 GIL 的情况下被调用，因而可被用来避免在单线程运行时对加锁 API 的调用。

在 3.7 版的變更：现在 GIL 将由 `Py_Initialize()` 来初始化。

在 3.9 版之後被弃用。



***PyThreadState* \*PyEval\_SaveThread()**

属于稳定 ABI。释放全局解释器锁 (如果已创建) 并将线程状态重置为 NULL, 返回之前的线程状态 (不为 NULL)。如果锁已被创建, 则当前线程必须已获取到它。

**void PyEval\_RestoreThread(*PyThreadState* \*tstate)**

属于稳定 ABI。获取全局解释器锁 (如果已创建) 并将线程状态设为 *tstate*, 它必须不为 NULL。如果锁已被创建, 则当前线程必须尚未获取它, 否则将发生死锁。

---

**備註:** 当运行时正在最终化时从某个线程调用此函数将终结该线程, 即使线程不是由 Python 创建的。你可以在调用此函数之前使用 `_Py_IsFinalizing()` 或 `sys.is_finalizing()` 来检查解释器是否还处于最终化过程中以避免不必要的终结。

---

***PyThreadState* \*PyThreadState\_Get()**

属于稳定 ABI。返回当前线程状态。全局解释器锁必须被持有。在当前状态为 NULL 时, 这将发出一个致命错误 (这样调用方将无须检查是否为 NULL)。

***PyThreadState* \*PyThreadState\_Swap(*PyThreadState* \*tstate)**

属于稳定 ABI。交换当前线程状态与由参数 *tstate* (可能为 NULL) 给出的线程状态。全局解释器锁必须被持有且未被释放。

下列函数使用线程级本地存储, 并且不能兼容子解释器:

**PyGILState\_STATE PyGILState\_Ensure()**

属于稳定 ABI。确保当前线程已准备好调用 Python C API 而不管 Python 或全局解释器锁的当前状态如何。只要每次调用都与 `PyGILState_Release()` 的调用相匹配就可以通过线程调用此函数任意多次。一般来说, 只要线程状态恢复到 `Release()` 之前的状态就可以在 `PyGILState_Ensure()` 和 `PyGILState_Release()` 调用之间使用其他与线程相关的 API。例如, 可以正常使用 `Py_BEGIN_ALLOW_THREADS` 和 `Py_END_ALLOW_THREADS` 宏。

返回值是一个当 `PyGILState_Ensure()` 被调用时的线程状态的不透明“句柄”, 并且必须被传递给 `PyGILState_Release()` 以确保 Python 处于相同状态。虽然允许递归调用, 但这些句柄不能被共享——每次对 `PyGILState_Ensure()` 的单独调用都必须保存其对 `PyGILState_Release()` 的调用的句柄。

当该函数返回时, 当前线程将持有 GIL 并能够调用任意 Python 代码。执行失败将导致致命级错误。

---

**備註:** 当运行时正在最终化时从某个线程调用此函数将终结该线程, 即使线程不是由 Python 创建的。你可以在调用此函数之前使用 `_Py_IsFinalizing()` 或 `sys.is_finalizing()` 来检查解释器是否还处于最终化过程中以避免不必要的终结。

---

**void PyGILState\_Release(PyGILState\_STATE)**

属于稳定 ABI。释放之前获取的任何资源。在此调用之后, Python 的状态将与其在对相应 `PyGILState_Ensure()` 调用之前的一样 (但是通常此状态对调用方来说将是未知的, 对 GILState API 的使用也是如此)。

对 `PyGILState_Ensure()` 的每次调用都必须与在同一线程上对 `PyGILState_Release()` 的调用相匹配。

***PyThreadState* \*PyGILState\_GetThisThreadState()**

属于稳定 ABI。获取此线程的当前线程状态。如果当前线程上没有使用过 GILState API 则可以返回 NULL。请注意主线程总是会有这样一个线程状态, 即使没有在主线程上执行过自动线程状态调用。这主要是一个辅助/诊断函数。

**int PyGILState\_Check()**

如果当前线程持有 GIL 则返回 1 否则返回 0。此函数可以随时从任何线程调用。只有当它的 Python 线程状态已经初始化并且当前持有 GIL 时它才会返回 1。这主要是一个辅助/诊断函数。例如在回调上下文或内存分配函数中会很有用处, 当知道 GIL 被锁定时可以允许调用方执行敏感的操作或是在其他情况下做出不同的行为。

在 3.4 版新加入。

以下的宏被使用时通常不带末尾分号；请在 Python 源代码发布包中查看示例用法。

#### **Py\_BEGIN\_ALLOW\_THREADS**

属于稳定 ABI。此宏会扩展为 `{ PyThreadState *_save; _save = PyEval_SaveThread();`。请注意它包含一个开头花括号；它必须与后面的 `Py_END_ALLOW_THREADS` 宏匹配。有关此宏的进一步讨论请参阅上文。

#### **Py\_END\_ALLOW\_THREADS**

属于稳定 ABI。此宏扩展为 `PyEval_RestoreThread(_save); }`。注意它包含一个右花括号；它必须与之前的 `Py_BEGIN_ALLOW_THREADS` 宏匹配。请参阅上文以进一步讨论此宏。

#### **Py\_BLOCK\_THREADS**

属于稳定 ABI。这个宏扩展为 `PyEval_RestoreThread(_save);`；它等价于没有关闭花括号的 `Py_END_ALLOW_THREADS`。

#### **Py\_UNBLOCK\_THREADS**

属于稳定 ABI。这个宏扩展为 `_save = PyEval_SaveThread();`；它等价于没有开始花括号和变量声明的 `Py_BEGIN_ALLOW_THREADS`。

## 9.5.5 低階 API

下列所有函数都必须在 `Py_Initialize()` 之后被调用。

在 3.7 版的變更: `Py_Initialize()` 现在会初始化 *GIL*。

#### *PyInterpreterState* \***PyInterpreterState\_New**()

属于稳定 ABI。创建一个新的解释器状态对象。不需要持有全局解释器锁，但如果有必要序列化对此函数的调用则可能会持有。

引發一個不附帶引數的稽核事件 `cpython.PyInterpreterState_New`。

#### void **PyInterpreterState\_Clear** (*PyInterpreterState* \*interp)

属于稳定 ABI。重置解释器状态对象中的所有信息。必须持有全局解释器锁。

引發一個不附帶引數的稽核事件 `cpython.PyInterpreterState_Clear`。

#### void **PyInterpreterState\_Delete** (*PyInterpreterState* \*interp)

属于稳定 ABI。销毁解释器状态对象。不需要持有全局解释器锁。解释器状态必须使用之前对 `PyInterpreterState_Clear()` 的调用来重置。

#### *PyThreadState* \***PyThreadState\_New** (*PyInterpreterState* \*interp)

属于稳定 ABI。创建属于给定解释器对象的新线程状态对象。全局解释器锁不需要保持，但如果需要序列化对此函数的调用，则可以保持。

#### void **PyThreadState\_Clear** (*PyThreadState* \*tstate)

属于稳定 ABI。重置线程状态对象中的所有信息。必须持有全局解释器锁。

在 3.9 版的變更: 此函数现在会调用 `PyThreadState.on_delete` 回调。在之前版本中，此操作是发生在 `PyThreadState_Delete()` 中的。

#### void **PyThreadState\_Delete** (*PyThreadState* \*tstate)

属于稳定 ABI。销毁线程状态对象。不需要持有全局解释器锁。线程状态必须使用之前对 `PyThreadState_Clear()` 的调用来重置。

#### void **PyThreadState\_DeleteCurrent** (void)

销毁当前线程状态并释放全局解释器锁。与 `PyThreadState_Delete()` 类似，不需要持有全局解释器锁。线程状态必须已使用之前对 `PyThreadState_Clear()` 调用来重置。

#### *PyFrameObject* \***PyThreadState\_GetFrame** (*PyThreadState* \*tstate)

属于稳定 ABI 自 3.10 版起。获取 Python 线程状态 *tstate* 的当前帧。

返回一个 *strong reference*。如果没有当前执行的帧则返回 `NULL`。

也請見 `PyEval_GetFrame()`。

`tstate` 不可 `NULL`。

在 3.9 版新加入。

`uint64_t PyThreadState_GetID (PyThreadState *tstate)`

属于稳定 ABI 自 3.10 版起, 获取 Python 线程状态 `tstate` 的唯一线程状态标识符。

`tstate` 不可 `NULL`。

在 3.9 版新加入。

`PyInterpreterState *PyThreadState_GetInterpreter (PyThreadState *tstate)`

属于稳定 ABI 自 3.10 版起, 获取 Python 线程状态 `tstate` 对应的解释器。

`tstate` 不可 `NULL`。

在 3.9 版新加入。

`void PyThreadState_EnterTracing (PyThreadState *tstate)`

暂停 Python 线程状态 `tstate` 中的追踪和性能分析。

使用 `PyThreadState_LeaveTracing()` 函数来恢复它们。

在 3.11 版新加入。

`void PyThreadState_LeaveTracing (PyThreadState *tstate)`

恢复 Python 线程状态 `tstate` 中被 `PyThreadState_EnterTracing()` 函数暂停的追踪和性能分析。

另请参阅 `PyEval_SetTrace()` 和 `PyEval_SetProfile()` 函数。

在 3.11 版新加入。

`PyInterpreterState *PyInterpreterState_Get (void)`

属于稳定 ABI 自 3.9 版起, 获取当前解释器。

如果不存在当前 Python 线程状态或不存在当前解释器则将发出致命级错误信号。它无法返回 `NULL`。

调用时必须携带 GIL。

在 3.9 版新加入。

`int64_t PyInterpreterState_GetID (PyInterpreterState *interp)`

属于稳定 ABI 自 3.7 版起, 返回解释器的唯一 ID。如果执行过程中发生任何错误则将返回 -1 并设置错误。

调用时必须携带 GIL。

在 3.7 版新加入。

`PyObject *PyInterpreterState_GetDict (PyInterpreterState *interp)`

属于稳定 ABI 自 3.8 版起, 返回一个存储解释器专属数据的字典。如果此函数返回 `NULL` 则没有任何异常被引发并且调用方应当将解释器专属字典视为不可用。

这不是 `PyModule_GetState()` 的替代, 扩展仍应使用它来存储解释器专属的状态信息。

在 3.8 版新加入。

`typedef PyObject *(*_PyFrameEvalFunction)(PyThreadState *tstate, _PyInterpreterFrame *frame, int throwflag)`

帧评估函数的类型

`throwflag` 形参将由生成器的 `throw()` 方法来使用: 如为非零值, 则处理当前异常。

在 3.9 版的變更: 此函数现在可接受一个 `tstate` 形参。

在 3.11 版的變更: `frame` 形参由 `PyFrameObject*` 改为 `_PyInterpreterFrame*`。

`_PyFrameEvalFunction _PyInterpreterState_GetEvalFrameFunc (PyInterpreterState *interp)`

获取帧评估函数。

请参阅 [PEP 523](#) "Adding a frame evaluation API to CPython"。

在 3.9 版新加入。

`void _PyInterpreterState_SetEvalFrameFunc (PyInterpreterState *interp, _PyFrameEvalFunction eval_frame)`

设置帧评估函数。

请参阅 [PEP 523](#) "Adding a frame evaluation API to CPython"。

在 3.9 版新加入。

`PyObject *PyThreadState_GetDict ()`

返回值：借用参照。属于 [稳定 ABI](#)。返回一个扩展可以在其中存储线程专属状态信息的字典。每个扩展都应当使用一个独有的键用来在该字典中存储状态。在没有可用的当前线程状态时也可以调用此函数。如果此函数返回 NULL，则还没有任何异常被引发并且调用方应当假定没有可用的当前线程状态。

`int PyThreadState_SetAsyncExc (unsigned long id, PyObject *exc)`

属于 [稳定 ABI](#)。在一个线程中异步地引发异常。`id` 参数是目标线程的线程 id；`exc` 是要引发的异常对象。该函数不会窃取任何对 `exc` 的引用。为防止随意滥用，你必须编写你自己的 C 扩展来调用它。调用时必须持有 GIL。返回已修改的线程状态数量；该值通常为一，但如果未找到线程 id 则会返回 0。如果 `exc` 为 "NULL"，则会清除线程的待处理异常（如果存在）。这不会引发异常。

在 3.7 版的变更：`id` 形参的类型已从 long 变为 unsigned long。

`void PyEval_AcquireThread (PyThreadState *tstate)`

属于 [稳定 ABI](#)。获取全局解释器锁并将当前线程状态设为 `tstate`，它必须不为 NULL。锁必须在此之前已被创建。如果该线程已获取锁，则会发生死锁。

---

**備註：**当运行时正在最终化时从某个线程调用此函数将终结该线程，即使线程不是由 Python 创建的。你可以在调用此函数之前使用 `_Py_IsFinalizing()` 或 `sys.is_finalizing()` 来检查解释器是否还处于最终化过程中以避免不必要的终结。

---

在 3.8 版的变更：已被更新为与 `PyEval_RestoreThread()`、`Py_END_ALLOW_THREADS()` 和 `PyGILState_Ensure()` 保持一致，如果在解释器正在最终化时被调用则会终结当前线程。

`PyEval_RestoreThread()` 是一个始终可用的（即使线程尚未初始化）更高层级函数。

`void PyEval_ReleaseThread (PyThreadState *tstate)`

属于 [稳定 ABI](#)。将当前线程状态重置为 NULL 并释放全局解释器锁。在此之前锁必须已被创建并且必须由当前的线程所持有。`tstate` 参数必须不为 NULL，该参数仅被用于检查它是否代表当前线程状态 --- 如果不是，则会报告一个致命级错误。

`PyEval_SaveThread()` 是一个始终可用的（即使线程尚未初始化）更高层级函数。

`void PyEval_AcquireLock ()`

属于 [稳定 ABI](#)。获取全局解释器锁。锁必须在此之前已被创建。如果该线程已经拥有锁，则会出现死锁。

在 3.2 版之後被弃用：此函数不会更新当前线程状态。请改用 `PyEval_RestoreThread()` 或 `PyEval_AcquireThread()`。

---

**備註：**当运行时正在最终化时从某个线程调用此函数将终结该线程，即使线程不是由 Python 创建的。你可以在调用此函数之前使用 `_Py_IsFinalizing()` 或 `sys.is_finalizing()` 来检查解释器是否还处于最终化过程中以避免不必要的终结。

---

在 3.8 版的变更：已被更新为与 `PyEval_RestoreThread()`、`Py_END_ALLOW_THREADS()` 和 `PyGILState_Ensure()` 保持一致，如果在解释器正在最终化时被调用则会终结当前线程。



void **PyEval\_ReleaseLock** ()

属于稳定 ABI. 释放全局解释器锁。锁必须在此之前已被创建。

在 3.2 版之後被 用: 此函数不会更新当前线程状态。请改用 `PyEval_SaveThread()` 或 `PyEval_ReleaseThread()`。

## 9.6 子解释器支持

虽然在大多数用例中，你都只会嵌入一个单独的 Python 解释器，但某些场景需要你在同一个进程甚至同一个线程中创建多个独立的解释器。子解释器让你能够做到这一点。

“主”解释器是在运行时初始化时创建的第一个解释器。它通常是一个进程中唯一的 Python 解释器。与子解释器不同，主解释器具有唯一的进程全局责任比如信号处理等。它还负责在运行时初始化期间的执行并且通常还是运行时最终化期间的活动解释器。 `PyInterpreterState_Main()` 函数将返回一个指向其状态的指针。

你可以使用 `PyThreadState_Swap()` 函数在子解释器之间进行切换。你可以使用下列函数来创建和销毁它们：

`PyThreadState *Py_NewInterpreter` ()

属于稳定 ABI. 新建一个子解释器。这是一个 (几乎) 完全隔离的 Python 代码执行环境。特别需要注意，新的子解释器具有全部已导入模块的隔离的、独立的版本，包括基本模块 `builtins`，`__main__` 和 `sys` 等。已加载模块表 (`sys.modules`) 和模块搜索路径 (`sys.path`) 也是隔离的。新环境没有 `sys.argv` 变量。它具有新的标准 I/O 流文件对象 `sys.stdin`, `sys.stdout` 和 `sys.stderr` (不过这些对象都指向相同的底层文件描述符)。

The return value points to the first thread state created in the new sub-interpreter. This thread state is made in the current thread state. Note that no actual thread is created; see the discussion of thread states below. If creation of the new interpreter is unsuccessful, NULL is returned; no exception is set since the exception state is stored in the current thread state and there may not be a current thread state. (Like all other Python/C API functions, the global interpreter lock must be held before calling this function and is still held when it returns; however, unlike most other Python/C API functions, there needn't be a current thread state on entry.)

扩展模块将以如下方式在 (子) 解释器之间共享：

- 对于使用多阶段初始化的模块，例如 `PyModule_FromDefAndSpec()`，将为每个解释器创建并初始化一个单独的模块对象。只有 C 层级的静态和全局变量能在这些模块对象之间共享。
- 对于使用单阶段初始化的模块，例如 `PyModule_Create()`，当特定扩展被首次导入时，它将被正常初始化，并会保存其模块字典的一个 (浅) 拷贝。当同一扩展被另一个 (子) 解释器导入时，将初始化一个新模块并填充该拷贝的内容；扩展的 `init` 函数不会被调用。因此模块字典中的对象最终会被 (子) 解释器所共享，这可能会导致预期之外的行为 (参见下文的 *Bugs and caveats*)。

请注意这不同于在调用 `Py_FinalizeEx()` 和 `Py_Initialize()` 完全重新初始化解释器之后导入扩展时所发生的情况；对于那种情况，扩展的 `initmodule` 函数会被再次调用。与多阶段初始化一样，这意味着只有 C 层级的静态和全局变量能在这些模块之间共享。

void **Py\_EndInterpreter** (`PyThreadState *tstate`)

属于稳定 ABI. Destroy the (sub-)interpreter represented by the given thread state. The given thread state must be the current thread state. See the discussion of thread states below. When the call returns, the current thread state is NULL. All thread states associated with this interpreter are destroyed. (The global interpreter lock must be held before calling this function and is still held when it returns.) `Py_FinalizeEx()` will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

## 9.6.1 错误和警告

由于子解释器 (以及主解释器) 都是同一个进程的组成部分, 它们之间的隔离状态并非完美 --- 举例来说, 使用低层级的文件操作如 `os.close()` 时它们可能 (无意或恶意地) 影响它们各自打开的文件。由于 (子) 解释器之间共享扩展的方式, 某些扩展可能无法正常工作; 在使用单阶段初始化或者 (静态) 全局变量时尤其如此。在一个子解释器中创建的对象有可能被插入到另一个 (子) 解释器的命名空间中; 这种情况应当尽可能地避免。

应当特别注意避免在子解释器之间共享用户自定义的函数、方法、实例或类, 因为由这些对象执行的导入操作可能会影响错误的已加载模块的 (子) 解释器的字典。同样重要的一点是应当避免共享可被上述对象访问的对象。

还要注意的一点是将此功能与 `PyGILState_*` API 结合使用是很微妙的, 因为这些 API 会假定 Python 线程状态与操作系统级线程之间存在双向投影关系, 而子解释器的存在打破了这一假定。强烈建议你不要在一对互相匹配的 `PyGILState_Ensure()` 和 `PyGILState_Release()` 调用之间切换子解释器。此外, 使用这些 API 以允许从非 Python 创建的线程调用 Python 代码的扩展 (如 `ctypes`) 在使用子解释器时很可能会出现问題。

## 9.7 异步通知

提供了一种向主解释器线程发送异步通知的机制。这些通知将采用函数指针和空指针参数的形式。

`int Py_AddPendingCall (int (*func)(void*), void *arg)`

属于 **稳定 ABI**。将一个函数加入从主解释器线程调用的计划任务。成功时, 将返回 0 并将 *func* 加入要被主线程调用的等待队列。失败时, 将返回 -1 但不会设置任何异常。

当成功加入队列后, *func* 将最终附带参数 *arg* 被主解释器线程调用。对于正常运行的 Python 代码来说它将被异步地调用, 但要同时满足以下两个条件:

- 位于 *bytecode* 的边界上;
- 主线程持有 *global interpreter lock* (因此 *func* 可以使用完整的 C API)。

*func* 必须在成功时返回 0, 或在失败时返回 -1 并设置一个异常集合。*func* 不会被中断来递归地执行另一个异步通知, 但如果全局解释器锁被释放则它仍可被中断以切换线程。

此函数的运行不需要当前线程状态, 也不需要全局解释器锁。

要在子解释器中调用函数, 调用方必须持有 GIL。否则, 函数 *func* 可能会被安排给错误的解释器来调用。

**警告:** 这是一个低层级函数, 只在非常特殊的情况下有用。不能保证 *func* 会尽快被调用。如果主线程忙于执行某个系统调用, *func* 将不会在系统调用返回之前被调用。此函数通常 **不适合** 从任意 C 线程调用 Python 代码。作为替代, 请使用 *PyGILStateAPI*。

在 3.1 版新加入。

在 3.9 版的變更: 如果此函数在子解释器中被调用, 则函数 *func* 将被安排在子解释器中调用, 而不是在主解释器中调用。现在每个子解释器都有自己的计划调用列表。



## 9.8 分析和跟踪

Python 解释器为附加的性能分析和执行跟踪工具提供了一些低层级的支持。它们可被用于性能分析、调试和覆盖分析工具。

这个 C 接口允许性能分析或跟踪代码避免调用 Python 层级的可调用对象带来的开销，它能直接执行 C 函数调用。此工具的基本属性没有变化；这个接口允许针对每个线程安装跟踪函数，并且向跟踪函数报告的基本事件与之前版本中向 Python 层级跟踪函数报告的事件相同。

```
typedef int (*Py_tracefunc)(PyObject *obj, PyFrameObject *frame, int what, PyObject *arg)
```

使用 `PyEval_SetProfile()` 和 `PyEval_SetTrace()` 注册的跟踪函数的类型。第一个形参是作为 `obj` 传递给注册函数的对象，`frame` 是与事件相关的帧对象，`what` 是常量 `PyTrace_CALL`，`PyTrace_EXCEPTION`，`PyTrace_LINE`，`PyTrace_RETURN`，`PyTrace_C_CALL`，`PyTrace_C_EXCEPTION`，`PyTrace_C_RETURN` 或 `PyTrace_OPCODE` 中的一个，而 `arg` 将依赖于 `what` 的值：

<i>what</i> 的值	<i>arg</i> 的含义
<code>PyTrace_CALL</code>	总是 <code>Py_None</code> 。
<code>PyTrace_EXCEPTION</code>	<code>sys.exc_info()</code> 返回的异常信息。
<code>PyTrace_LINE</code>	总是 <code>Py_None</code> 。
<code>PyTrace_RETURN</code>	返回给调用方的值，或者如果是由异常导致的则返回 <code>NULL</code> 。
<code>PyTrace_C_CALL</code>	正在调用函数对象。
<code>PyTrace_C_EXCEPTION</code>	正在调用函数对象。
<code>PyTrace_C_RETURN</code>	正在调用函数对象。
<code>PyTrace_OPCODE</code>	总是 <code>Py_None</code> 。

### int `PyTrace_CALL`

当对一个函数或方法的新调用被报告，或是向一个生成器增加新条目时传给 `Py_tracefunc` 函数的 `what` 形参的值。请注意针对生成器函数的迭代器的创建情况不会被报告因为在相应的帧中没有向 Python 字节码转移控制权。

### int `PyTrace_EXCEPTION`

当一个异常被引发时传给 `Py_tracefunc` 函数的 `what` 形参的值。在处理完任何字节码之后将附带 `what` 的值调用回调函数，在此之后该异常将会被设置在正在执行的帧中。这样做的效果是当异常传播导致 Python 栈展开时，被调用的回调函数将随异常传播返回到每个帧。只有跟踪函数才会接收到这些事件；性能分析器并不需要它们。

### int `PyTrace_LINE`

当一个行编号事件被报告时传给 `Py_tracefunc` 函数 (但不会传给性能分析函数) 的 `what` 形参的值。它可以通过将 `f_trace_lines` 设为 0 在某个帧中被禁用。

### int `PyTrace_RETURN`

当一个调用即将返回时传给 `Py_tracefunc` 函数的 `what` 形参的值。

### int `PyTrace_C_CALL`

当一个 C 函数即将被调用时传给 `Py_tracefunc` 函数的 `what` 形参的值。

### int `PyTrace_C_EXCEPTION`

当一个 C 函数引发异常时传给 `Py_tracefunc` 函数的 `what` 形参的值。

### int `PyTrace_C_RETURN`

当一个 C 函数返回时传给 `Py_tracefunc` 函数的 `what` 形参的值。

### int `PyTrace_OPCODE`

当一个新操作码即将被执行时传给 `Py_tracefunc` 函数 (但不会传给性能分析函数) 的 `what` 形参的值。在默认情况下此事件不会被发送：它必须通过在某个帧上将 `f_trace_opcodes` 设为 1 来显式地请求。

void **PyEval\_SetProfile** (*Py\_tracefunc* func, *PyObject* \*obj)

将性能分析器函数设为 *func*。*obj* 形参将作为第一个形参传给该函数，它可以是任意 Python 对象或为 NULL。如果性能分析函数需要维护状态，则为每个线程的 *obj* 使用不同的值将提供一个方便而线程安全的存储位置。这个性能分析函数将针对除 *PyTrace\_LINE* *PyTrace\_OPCODE* 和 *PyTrace\_EXCEPTION* 以外的所有被监控事件进行调用。

另请参阅 `sys.setprofile()` 函数。

呼叫者必须持有 *GIL*。

void **PyEval\_SetTrace** (*Py\_tracefunc* func, *PyObject* \*obj)

将跟踪函数设为 *func*。这类似于 *PyEval\_SetProfile()*，区别在于跟踪函数会接收行编号事件和操作码级事件，但不会接收与被调用的 C 函数对象相关的任何事件。使用 *PyEval\_SetTrace()* 注册的任何跟踪函数将不会接收 *PyTrace\_C\_CALL*、*PyTrace\_C\_EXCEPTION* 或 *PyTrace\_C\_RETURN* 作为 *what* 形参的值。

也請見 `sys.settrace()` 函式。

呼叫者必须持有 *GIL*。

## 9.9 高级调试器支持

这些函数仅供高级调试工具使用。

*PyInterpreterState* \***PyInterpreterState\_Head** ()

将解释器状态对象返回到由所有此类对象组成的列表的开头。

*PyInterpreterState* \***PyInterpreterState\_Main** ()

返回主解释器状态对象。

*PyInterpreterState* \***PyInterpreterState\_Next** (*PyInterpreterState* \*interp)

从由解释器状态对象组成的列表中返回 *interp* 之后的下一项。

*PyThreadState* \***PyInterpreterState\_ThreadHead** (*PyInterpreterState* \*interp)

在由与解释器 *interp* 相关联的线程组成的列表中返回指向第一个 *PyThreadState* 对象的指针。

*PyThreadState* \***PyThreadState\_Next** (*PyThreadState* \*tstate)

从属于同一个 *PyInterpreterState* 对象的线程状态对象组成的列表中返回 *tstate* 之后的下一项。

## 9.10 线程本地存储支持

Python 解释器提供也对线程本地存储 (TLS) 的低层级支持，它对下层的原生 TLS 实现进行了包装以支持 Python 层级的线程本地存储 API (`threading.local`)。CPython 的 C 层级 API 与 `pthreads` 和 Windows 所提供的类似：使用一个线程键和函数来为每个线程关联一个 `void*` 值。

当调用这些函数时无须持有 *GIL*；它们会提供自己的锁机制。

请注意 `Python.h` 并不包括 TLS API 的声明，你需要包括 `pythread.h` 来使用线程本地存储。

---

**備註：** 这些 API 函数都不会为 `void*` 的值处理内存管理问题。你需要自己分配和释放它们。如果 `void*` 值碰巧为 *PyObject\**，这些函数也不会对它们执行引用计数操作。

---

### 9.10.1 线程专属存储 (TSS) API

引入 TSSAPI 是为了取代 CPython 解释器中现有 TLS API 的使用。该 API 使用一个新类型 `Py_tss_t` 而不是 `int` 来表示线程键。

在 3.7 版新加入。

也参考：

“A New C-API for Thread-Local Storage in CPython” (PEP 539)

type `Py_tss_t`

该数据结构表示线程键的状态，其定义可能依赖于下层的 TLS 实现，并且它有一个表示键初始化状态的内部字段。该结构体中不存在公有成员。

当未定义 `Py_LIMITED_API` 时，允许由 `Py_tss_NEEDS_INIT` 执行此类型的静态分配。

`Py_tss_NEEDS_INIT`

这个宏将扩展为 `Py_tss_t` 变量的初始化器。请注意这个宏不会用 `Py_LIMITED_API` 来定义。

#### 动态分配

`Py_tss_t` 的动态分配，在使用 `Py_LIMITED_API` 编译的扩展模块中是必须的，在这些模块由于此类型的实现在编译时是不透明的因此它不可能静态分配。

`Py_tss_t *PyThread_tss_alloc()`

属于稳定 ABI 自 3.7 版起。返回一个与使用 `Py_tss_NEEDS_INIT` 初始化的值的状态相同的值，或者当动态分配失败时则返回 `NULL`。

`void PyThread_tss_free(Py_tss_t *key)`

属于稳定 ABI 自 3.7 版起。在首次调用 `PyThread_tss_delete()` 以确保任何相关联的线程局部变量已被撤销赋值之后释放由 `PyThread_tss_alloc()` 所分配的给定的 `key`。如果 `key` 参数为 `NULL` 则这将无任何操作。

---

備註：被释放的 `key` 将变成一个悬空指针。你应当将 `key` 重置为 `NULL`。

---

#### 方法

这些函数的形参 `key` 不可为 `NULL`。并且，如果给定的 `Py_tss_t` 还未被 `PyThread_tss_create()` 初始化则 `PyThread_tss_set()` 和 `PyThread_tss_get()` 的行为将是未定义的。

`int PyThread_tss_is_created(Py_tss_t *key)`

属于稳定 ABI 自 3.7 版起。如果给定的 `Py_tss_t` 已通过 `has been initialized by PyThread_tss_create()` 被初始化则返回一个非零值。

`int PyThread_tss_create(Py_tss_t *key)`

属于稳定 ABI 自 3.7 版起。当成功初始化一个 TSS 键时将返回零值。如果 `key` 参数所指向的值未被 `Py_tss_NEEDS_INIT` 初始化则其行为是未定义的。此函数可在相同的键上重复调用 -- 在已初始化的键上调用它将不执行任何操作并立即成功返回。

`void PyThread_tss_delete(Py_tss_t *key)`

属于稳定 ABI 自 3.7 版起。销毁一个 TSS 键以便在所有线程中遗忘与该键相关联的值，并将该键的初始化状态改为未初始化的。已销毁的键可以通过 `PyThread_tss_create()` 再次被初始化。此函数可以在同一个键上重复调用 -- 但在一个已被销毁的键上调用将是无效的。

`int PyThread_tss_set(Py_tss_t *key, void *value)`

属于稳定 ABI 自 3.7 版起。返回零值来表示成功将一个 `void*` 值与当前线程中的 TSS 键相关联。每个线程都有一个从键到 `void*` 值的独立映射。

`void *PyThread_tss_get (Py_tss_t *key)`

属于稳定 ABI 自 3.7 版起. 返回当前线程中与一个 TSS 键相关联的 `void*` 值。如果当前线程中没有与该键相关联的值则返回 `NULL`。

## 9.10.2 线程本地存储 (TLS) API

在 3.7 版之後被 用: 此 API 已被线程专属存储 (TSS) API 所取代。

---

**備 用:** 这个 API 版本不支持原生 TLS 键采用无法被安全转换为 `int` 的的定义方式的平台。在这样的平台上, `PyThread_create_key()` 将立即返回一个失败状态, 并且其他 TLS 函数在这样的平台上也都无效。

---

由于上面提到的兼容性问题, 不应在新代码中使用此版本的 API。

`int PyThread_create_key ()`

属于稳定 ABI.

`void PyThread_delete_key (int key)`

属于稳定 ABI.

`int PyThread_set_key_value (int key, void *value)`

属于稳定 ABI.

`void *PyThread_get_key_value (int key)`

属于稳定 ABI.

`void PyThread_delete_key_value (int key)`

属于稳定 ABI.

`void PyThread_ReInitTLS ()`

属于稳定 ABI.

---

Python 初始化配置

---

在 3.8 版新加入。

Python 可以使用 `Py_InitializeFromConfig()` 和 `PyConfig` 结构体来初始化。它可以使用 `Py_PreInitialize()` 和 `PyPreConfig` 结构体来预初始化。

有两种配置方式：

- **Python 配置** 可被用于构建一个定制的 Python，其行为与常规 Python 类似。例如，环境变量和命令行参数可被用于配置 Python。
- **隔离配置** 可被用于将 Python 嵌入到应用程序。它将 Python 与系统隔离开来。例如，环境变量将被忽略，`LC_CTYPE` 语言区域设置保持不变并且不会注册任何信号处理器。

`Py_RunMain()` 函数可被用来编写定制的 Python 程序。

参见 *Initialization, Finalization, and Threads*。

也参考：

**PEP 587** “Python 初始化配置”。

## 10.1 范例

定制的 Python 的示例总是会以隔离模式运行：

```
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;
    }
}
```

(繼續下一頁)

```

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);
}

```

## 10.2 PyWideStringList

type **PyWideStringList**

由 `wchar_t*` 字符串组成的列表。

如果 `length` 为非零值，则 `items` 必须不为 `NULL` 并且所有字符串均必须不为 `NULL`。

方法

**PyStatus PyWideStringList\_Append** (*PyWideStringList* \*list, const `wchar_t` \*item)

将 `item` 添加到 `list`。

Python 必须被预初始化以便调用此函数。

**PyStatus PyWideStringList\_Insert** (*PyWideStringList* \*list, *Py\_ssize\_t* index, const `wchar_t` \*item)

将 `item` 插入到 `list` 的 `index` 位置上。

如果 `index` 大于等于 `list` 的长度，则将 `item` 添加到 `list`。

`index` 必须大于等于 0。

Python 必须被预初始化以便调用此函数。

结构体字段:

**Py\_ssize\_t length**

List 长度。

`wchar_t **items`

列表项目。



## 10.3 PyStatus

type **PyStatus**

存储初始函数状态：成功、错误或退出的结构体。

对于错误，它可以存储造成错误的 C 函数的名称。

结构体字段：

**int exitcode**

退出码。传给 `exit()` 的参数。

**const char \*err\_msg**

錯誤訊息。

**const char \*func**

造成错误的函数的名称，可以为 `NULL`。

创建状态的函数：

*PyStatus* **PyStatus\_Ok** (void)

完成。

*PyStatus* **PyStatus\_Error** (const char \*err\_msg)

带消息的初始化错误。

*err\_msg* 不可为 `NULL`。

*PyStatus* **PyStatus\_NoMemory** (void)

内存分配失败（内存不足）。

*PyStatus* **PyStatus\_Exit** (int exitcode)

以指定的退出代码退出 Python。

处理状态的函数：

**int PyStatus\_Exception** (*PyStatus* status)

状态为错误还是退出？如为真值，则异常必须被处理；例如通过调用 `Py_ExitStatusException()`。

**int PyStatus\_IsError** (*PyStatus* status)

结果错误吗？

**int PyStatus\_IsExit** (*PyStatus* status)

结果是否退出？

**void Py\_ExitStatusException** (*PyStatus* status)

如果 *status* 是一个退出码则调用 `exit(exitcode)`。如果 *status* 是一个错误码则打印错误消息并设置一个非零退出码再退出。必须在 `PyStatus_Exception(status)` 为非零值时才能被调用。

---

備註：在内部，Python 将使用设置 `PyStatus.func` 的宏，而创建状态的函数则会将 `func` 设为 `NULL`。

---

範例：

```
PyStatus alloc(void **ptr, size_t size)
{
    *ptr = PyMem_RawMalloc(size);
    if (*ptr == NULL) {
        return PyStatus_NoMemory();
    }
}
```

(繼續下一頁)

(繼續上一頁)

```

    return PyStatus_Ok();
}

int main(int argc, char **argv)
{
    void *ptr;
    PyStatus status = alloc(&ptr, 16);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
    PyMem_Free(ptr);
    return 0;
}

```

## 10.4 PyPreConfig

type **PyPreConfig**

用于预初始化 Python 的结构体。

用于初始化预先配置的函数:

void **PyPreConfig\_InitPythonConfig** (*PyPreConfig* \*preconfig)

通过 *Python* 配置 来初始化预先配置。

void **PyPreConfig\_InitIsolatedConfig** (*PyPreConfig* \*preconfig)

通过 *隔离配置* 来初始化预先配置。

结构体字段:

int **allocator**

Python 内存分配器名称:

- PYMEM\_ALLOCATOR\_NOT\_SET (0): 不改变内存分配器 (使用默认)。
- PYMEM\_ALLOCATOR\_DEFAULT (1): 默认内存分配器。
- PYMEM\_ALLOCATOR\_DEBUG (2): 默认内存分配器 附带调试钩子。
- PYMEM\_ALLOCATOR\_MALLOC (3): 使用 C 库的 malloc()。
- PYMEM\_ALLOCATOR\_MALLOC\_DEBUG (4): 强制使用 malloc() 附带调试钩子。
- PYMEM\_ALLOCATOR\_PYMALLOC (5): *Python pymalloc* 内存分配器。
- PYMEM\_ALLOCATOR\_PYMALLOC\_DEBUG (6): *Python pymalloc* 内存分配器 附带调试钩子。

如果 Python 是使用 `--without-pymalloc` 进行配置则 PYMEM\_ALLOCATOR\_PYMALLOC 和 PYMEM\_ALLOCATOR\_PYMALLOC\_DEBUG 将不被支持。

請見 *記憶體管理*。

預設: PYMEM\_ALLOCATOR\_NOT\_SET。

int **configure\_locale**

将 LC\_CTYPE 语言区域设为用户选择的语言区域。

如果等于 0, 则将 *coerce\_c\_locale* 和 *coerce\_c\_locale\_warn* 的成员设为 0。

請見 *locale encoding*。

默认值: 在 Python 配置中为 1, 在隔离配置中为 0。

**int coerce\_c\_locale**

如果等于 2，强制转换 C 语言区域。

如果等于 1，则读取 LC\_CTYPE 语言区域来确定其是否应当被强制转换。

請見 *locale encoding*。

默认值: 在 Python 配置中为 -1，在隔离配置中为 0。

**int coerce\_c\_locale\_warn**

如为非零值，则会在 C 语言区域被强制转换时发出警告。

默认值: 在 Python 配置中为 -1，在隔离配置中为 0。

**int dev\_mode**

Python 开发模式: 参见 *PyConfig.dev\_mode*。

默认值: 在 Python 模式中为 -1，在隔离模式中为 0。

**int isolated**

隔离模式: 参见 *PyConfig.isolated*。

默认值: 在 Python 模式中为 0，在隔离模式中为 1。

**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"。

初始化来自 PYTHONLEGACYWINDOWSFSENCODING 的环境变量值。

仅在 Windows 上可用。#ifdef MS\_WINDOWS 宏可被用于 Windows 专属的代码。

預設: 0。

**int parse\_argv**

如为非零值, *Py\_PreInitializeFromArgs()* 和 *Py\_PreInitializeFromBytesArgs()* 将以与常规 Python 解析命令行参数的相同方式解析其 argv 参数: 参见 命令行参数。

默认值: 在 Python 配置中为 1，在隔离配置中为 0。

**int use\_environment**

使用 环境变量? 参见 *PyConfig.use\_environment*。

默认值: 在 Python 配置中为 1 而在隔离配置中为 0。

**int utf8\_mode**

如为非零值，则启用 Python UTF-8 模式。

通过 -X utf8 命令行选项和 PYTHONUTF8 环境变量设为 0 或 1。

如果 LC\_CTYPE 语言区域为 C 或 POSIX 也会被设为 1。

默认值: 在 Python 配置中为 -1 而在隔离配置中为 0。

## 10.5 使用 PyPreConfig 预初始化 Python

Python 的预初始化:

- 设置 Python 内存分配器 (*PyPreConfig.alloc**locator*)
- 配置 LC\_CTYPE 语言区域 (*locale encoding*)
- 设置 Python UTF-8 模式 (*PyPreConfig.utf8\_mode*)

当前的预配置 (PyPreConfig 类型) 保存在 `_PyRuntime.preconfig` 中。

用于预初始化 Python 的函数:

**PyStatus Py\_PreInitialize** (const *PyPreConfig* \*preconfig)

根据 *preconfig* 预配置来预初始化 Python。

*preconfig* 不可为 NULL。

**PyStatus Py\_PreInitializeFromBytesArgs** (const *PyPreConfig* \*preconfig, int argc, char \*const \*argv)

根据 *preconfig* 预配置来预初始化 Python。

如果 *preconfig* 的 *parse\_argv* 为非零值则解析 *argv* 命令行参数 (字节串)。

*preconfig* 不可为 NULL。

**PyStatus Py\_PreInitializeFromArgs** (const *PyPreConfig* \*preconfig, int argc, wchar\_t \*const \*argv)

根据 *preconfig* 预配置来预初始化 Python。

如果 *preconfig* 的 *parse\_argv* 为非零值则解析 *argv* 命令行参数 (宽字符串)。

*preconfig* 不可为 NULL。

调用方要负责使用 *PyStatus\_Exception()* 和 *Py\_ExitStatusException()* 来处理异常 (错误或退出)。

对于 *Python* 配置 (*PyPreConfig\_InitPythonConfig()*)，如果 Python 是用命令行参数初始化的，那么在预初始化 Python 时也必须传递命令行参数，因为它们会对编码格式等预配置产生影响。例如，`-x utf8` 命令行选项将启用 Python UTF-8 模式。

*PyMem\_SetAllocator()* 可在 *Py\_PreInitialize()* 之后、*Py\_InitializeFromConfig()* 之前被调用以安装自定义的内存分配器。如果 *PyPreConfig.alloc**locator* 被设为 `PYMEM_ALLOCATOR_NOT_SET` 则可在 *Py\_PreInitialize()* 之前被调用。

像 *PyMem\_RawMalloc()* 这样的 Python 内存分配函数不能在 Python 预初始化之前使用，而直接调用 *malloc()* 和 *free()* 则始终会是安全的。*Py\_DecodeLocale()* 不能在 Python 预初始化之前被调用。

使用预初始化来启用 Python UTF-8 模式的例子:

```
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();
```

## 10.6 PyConfig

type **PyConfig**

包含了大部分用于配置 Python 的形参的结构体。

在完成后，必须使用 `PyConfig_Clear()` 函数来释放配置内存。

结构体方法：

void **PyConfig\_InitPythonConfig** (*PyConfig* \*config)

通过 *Python* 配置 来初始化配置。

void **PyConfig\_InitIsolatedConfig** (*PyConfig* \*config)

通过隔离配置 来初始化配置。

*PyStatus* **PyConfig\_SetString** (*PyConfig* \*config, wchar\_t \*const \*config\_str, const wchar\_t \*str)

将宽字符串 *str* 拷贝至 \*config\_str。

在必要时预初始化 *Python*。

*PyStatus* **PyConfig\_SetBytesString** (*PyConfig* \*config, wchar\_t \*const \*config\_str, const char \*str)

使用 `Py_DecodeLocale()` 对 *str* 进行解码并将结果设置到 \*config\_str。

在必要时预初始化 *Python*。

*PyStatus* **PyConfig\_SetArgv** (*PyConfig* \*config, int argc, wchar\_t \*const \*argv)

根据宽字符串列表 *argv* 设置命令行参数 (*config* 的 *argv* 成员)。

在必要时预初始化 *Python*。

*PyStatus* **PyConfig\_SetBytesArgv** (*PyConfig* \*config, int argc, char \*const \*argv)

根据字节串列表 *argv* 设置命令行参数 (*config* 的 *argv* 成员)。使用 `Py_DecodeLocale()` 对字节串进行解码。

在必要时预初始化 *Python*。

*PyStatus* **PyConfig\_SetWideStringList** (*PyConfig* \*config, *PyWideStringList* \*list, *Py\_ssize\_t* length, wchar\_t \*\*items)

将宽字符串列表 *list* 设置为 *length* 和 *items*。

在必要时预初始化 *Python*。

*PyStatus* **PyConfig\_Read** (*PyConfig* \*config)

读取所有 *Python* 配置。

已经初始化的字段会保持不变。

调用此函数时不再计算或修改用于路径配置的字段，如 *Python* 3.11 那样。

`PyConfig_Read()` 函数只解析 `PyConfig.argv` 参数一次：在参数解析完成后，`PyConfig.parse_argv` 将被设为 2。由于 *Python* 参数是从 `PyConfig.argv` 中剥离的，因此解析参数两次会将应用程序选项解析为 *Python* 选项。

在必要时预初始化 *Python*。

在 3.10 版的變更： `PyConfig.argv` 参数现在只会被解析一次，在参数解析完成后，`PyConfig.parse_argv` 将被设为 2，只有当 `PyConfig.parse_argv` 等于 1 时才会解析参数。

在 3.11 版的變更： `PyConfig_Read()` 不会再计算所有路径，因此在 *Python* 路径配置 下列出的字段可能不会更新直到 `Py_InitializeFromConfig()` 被调用。

void **PyConfig\_Clear** (*PyConfig* \*config)

释放配置内存

如有必要大多数 `PyConfig` 方法将会预初始化 *Python*。在这种情况下, *Python* 预初始化配置 (*PyPreConfig*) 将以 *PyConfig* 为基础。如果要调整与 *PyPreConfig* 相同的配置字段, 它们必须在调用 *PyConfig* 方法之前被设置:

- `PyConfig.dev_mode`
- `PyConfig.isolated`
- `PyConfig.parse_argv`
- `PyConfig.use_environment`

此外, 如果使用了 `PyConfig_SetArgv()` 或 `PyConfig_SetBytesArgv()`, 则必须在调用其他方法之前调用该方法, 因为预初始化配置取决于命令行参数 (如果 `parse_argv` 为非零值)。

这些方法的调用者要负责使用 `PyStatus_Exception()` 和 `Py_ExitStatusException()` 来处理异常 (错误或退出)。

结构体字段:

#### *PyWideStringList* **argv**

命令行参数: `sys.argv`。

将 `parse_argv` 设为 1 将以与普通 *Python* 解析 *Python* 命令行参数相同的方式解析 *argv* 再从 *argv* 中剥离 *Python* 参数。

如果 *argv* 为空, 则会添加一个空字符串以确保 `sys.argv` 始终存在并且永远不为空。

预设值: `NULL`。

另请参阅 `orig_argv` 成员。

#### **int safe\_path**

如果等于零, `Py_RunMain()` 会在启动时向 `sys.path` 开头添加一个可能不安全的路径:

- 如果 `argv[0]` 等于 `L"-m"` (`python -m module`), 则添加当前工作目录。
- 如果是运行脚本 (`python script.py`), 则添加脚本的目录。如果是符号链接, 则会解析符号链接。
- 在其他情况下 (`python -c code` 和 `python`), 将添加一个空字符串, 这表示当前工作目录。

通过 `-P` 命令行选项和 `PYTHONSAFEPATH` 环境变量设置为 1。

默认值: *Python* 配置中为 0, 隔离配置中为 1。

在 3.11 版新加入。

#### **wchar\_t \*base\_exec\_prefix**

`sys.base_exec_prefix`。

预设值: `NULL`。

*Python* 路径配置 的一部分。

#### **wchar\_t \*base\_executable**

*Python* 基础可执行文件: `sys._base_executable`。

由 `__PYENV_LAUNCHER__` 环境变量设置。

如为 `NULL` 则从 `PyConfig.executable` 设置。

预设值: `NULL`。

*Python* 路径配置 的一部分。



`wchar_t *base_prefix`

`sys.base_prefix`。

預設值: `NULL`。

*Python* 路径配置 的一部分。

`int buffered_stdio`

如果等于 0 且 `configure_c_stdio` 为非零值, 则禁用 C 数据流 `stdout` 和 `stderr` 的缓冲。

通过 `-u` 命令行选项和 `PYTHONUNBUFFERED` 环境变量设置为 0。

`stdin` 始终以缓冲模式打开。

預設值: 1。

`int bytes_warning`

如果等于 1, 则在将 `bytes` 或 `bytearray` 与 `str` 进行比较, 或将 `bytes` 与 `int` 进行比较时发出警告。

如果大于等于 2, 则在这些情况下引发 `BytesWarning` 异常。

由 `-b` 命令行选项执行递增。

預設: 0。

`int warn_default_encoding`

如为非零值, 则在 `io.TextIOWrapper` 使用默认编码格式时发出 `EncodingWarning` 警告。详情请参阅 `io-encoding-warning`。

預設: 0。

在 3.10 版新加入。

`int code_debug_ranges`

如果等于 0, 则禁用在代码对象中包括末尾行和列映射。并且禁用在特定错误位置打印回溯标记。

通过 `PYTHONNODEBUGRANGES` 环境变量和 `-X no_debug_ranges` 命令行选项设置为 0。

預設值: 1。

在 3.11 版新加入。

`wchar_t *check_hash_pycs_mode`

控制基于哈希值的 `.pyc` 文件的验证行为: `--check-hash-based-pycs` 命令行选项的值。

有效的值:

- `L"always"`: 无论 `'check_source'` 旗标的值是什么都会对源文件进行哈希验证。
- `L"never"`: 假定基于哈希值的 `pyc` 始终是有效的。
- `L"default"`: 基于哈希值的 `pyc` 中的 `'check_source'` 旗标确定是否验证无效。

預設: `L"default"`。

参见 [PEP 552](#) "Deterministic pycs"。

`int configure_c_stdio`

如为非零值, 则配置 C 标准流:

- 在 Windows 中, 在 `stdin`, `stdout` 和 `stderr` 上设置二进制模式 (`O_BINARY`)。
- 如果 `buffered_stdio` 等于零, 则禁用 `stdin`, `stdout` 和 `stderr` 流的缓冲。
- 如果 `interactive` 为非零值, 则启用 `stdin` 和 `stdout` 上的流缓冲 (Windows 中仅限 `stdout`)。

默认值: 在 Python 配置中为 1, 在隔离配置中为 0。

**int dev\_mode**

如果为非零值，则启用 Python 开发模式。

通过 `-X dev` 选项和 `PYTHONDEVMODE` 环境变量设置为 1。

默认值: 在 Python 模式中为 -1，在隔离模式中为 0。

**int dump\_refs**

转储 Python 引用？

如果为非零值，则转储所有在退出时仍存活的对象。

由 `PYTHONDUMPREFS` 环境变量设置为 1。

需要定义了 `Py_TRACE_REFS` 宏的特殊 Python 编译版：参见 `configure --with-trace-refs` 选项。

预设：0。

**wchar\_t \*exec\_prefix**

安装依赖于平台的 Python 文件的站点专属目录前缀: `sys.exec_prefix`。

预设值：NULL。

*Python 路径配置* 的一部分。

**wchar\_t \*executable**

Python 解释器可执行二进制文件的绝对路径: `sys.executable`。

预设值：NULL。

*Python 路径配置* 的一部分。

**int faulthandler**

启用 faulthandler？

如果为非零值，则在启动时调用 `faulthandler.enable()`。

通过 `-X faulthandler` 和 `PYTHONFAULTHANDLER` 环境变量设为 1。

默认值: 在 Python 模式中为 -1，在隔离模式中为 0。

**wchar\_t \*filesystem\_encoding**

文件系统编码格式: `sys.getfilesystemencoding()`。

在 macOS, Android 和 VxWorks 上：默认使用 "utf-8"。

在 Windows 上：默认使用 "utf-8"，或者如果 *PyPreConfig* 的 *legacy\_windows\_fs\_encoding* 为非零值则使用 "mbcs"。

在其他平台上的默认编码格式：

- 如果 *PyPreConfig.utf8\_mode* 为非零值则使用 "utf-8"。
- 如果 Python 检测到 `nl_langinfo(CODESET)` 声明为 ASCII 编码格式，而 `mbstowcs()` 是从其他的编码格式解码（通常为 Latin1）则使用 "ascii"。
- 如果 `nl_langinfo(CODESET)` 返回空字符串则使用 "utf-8"。
- 在其他情况下，使用 *locale encoding*: `nl_langinfo(CODESET)` 的结果。

在 Python 启动时，编码格式名称会规范化为 Python 编解码器名称。例如，"ANSI\_X3.4-1968" 将被替换为 "ascii"。

参见 *filesystem\_errors* 的成员。

**wchar\_t \*filesystem\_errors**

文件系统错误处理器: `sys.getfilesystemencodeerrors()`。

在 Windows 上: 默认使用 "surrogatepass", 或者如果 `PyPreConfig` 的 `legacy_windows_fs_encoding` 为非零值则使用 "replace"。

在其他平台上: 默认使用 "surrogateescape"。

支持的错误处理器:

- "strict"
- "surrogateescape"
- "surrogatepass" (仅支持 UTF-8 编码格式)

参见 `filesystem_encoding` 的成员。

**unsigned long hash\_seed****int use\_hash\_seed**

随机化的哈希函数种子。

如果 `use_hash_seed` 为零, 则在 Python 启动时随机选择一个种子, 并忽略 `hash_seed`。

由 `PYTHONHASHSEED` 环境变量设置。

默认的 `use_hash_seed` 值: 在 Python 模式下为 -1, 在隔离模式下为 0。

**wchar\_t \*home**

Python 主目录。

如果 `Py_SetPythonHome()` 已被调用, 则当其参数不为 NULL 时将使用它。

由 `PYTHONHOME` 环境变量设置。

预设值: NULL。

`Python` 路径配置 输入的一部分。

**int import\_time**

如为非零值, 则对导入时间执行性能分析。

通过 `-X importtime` 选项和 `PYTHONPROFILEIMPORTTIME` 环境变量设置为 1。

预设: 0。

**int inspect**

在执行脚本或命令之后进入交互模式。

如果大于 0, 则启用检查: 当脚本作为第一个参数传入或使用了 `-c` 选项时, 在执行脚本或命令后进入交互模式, 即使在 `sys.stdin` 看来并非一个终端时也是如此。

通过 `-i` 命令行选项执行递增。如果 `PYTHONINSPECT` 环境变量为非空值则设为 1。

预设: 0。

**int install\_signal\_handlers**

安装 Python 信号处理器?

默认值: 在 Python 模式下为 1, 在隔离模式下为 0。

**int interactive**

如果大于 0, 则启用交互模式 (REPL)。

由 `-i` 命令行选项执行递增。

预设: 0。

**int isolated**

如果大于 0，则启用隔离模式：

- 将 `safe_path` 设置为 1：不要在 Python 启动时将潜在的不安全路径向前追加到 `sys.path`。
- 将 `use_environment` 設定 0。
- 将 `user_site_directory` 设为 0：不要将用户级站点目录添加到 `sys.path`。
- Python REPL 将不导入 `readline` 也不在交互提示符中启用默认的 `readline` 配置。

通过 `-I` 命令行选项设置为 1。

默认值：在 Python 模式中为 0，在隔离模式中为 1。

也請見 `PyPreConfig.isolated`。

**int legacy\_windows\_stdio**

如为非零值，则使用 `io.FileIO` 代替 `io._WindowsConsoleIO` 作为 `sys.stdin`、`sys.stdout` 和 `sys.stderr`。

如果 `PYTHONLEGACYWINDOWSSTDIO` 环境变量被设为非空字符串则设为 1。

仅在 Windows 上可用。`#ifdef MS_WINDOWS` 宏可被用于 Windows 专属的代码。

預設：0。

另请参阅 [PEP 528](#) (将 Windows 控制台编码格式更改为 UTF-8)。

**int malloc\_stats**

如为非零值，则在退出时转储 *Python pymalloc 内存分配器* 的统计数据。

由 `PYTHONMALLOCSTATS` 环境变量设置为 1。

如果 Python 是使用 `--without-pymalloc` 选项进行配置则该选项将被忽略。

預設：0。

**wchar\_t \*platlibdir**

平台库目录名称: `sys.platlibdir`。

由 `PYTHONPLATLIBDIR` 环境变量设置。

默认值：由 `configure --with-platlibdir` 选项设置的 `PLATLIBDIR` 宏的值 (默认值: `"lib"`，在 Windows 上则为 `"DLLs"`)。

*Python 路径配置* 输入的一部分。

在 3.9 版新加入。

在 3.11 版的變更: 目前在 Windows 系统中该宏被用于定位标准库扩展模块，通常位于 `DLLs` 下。不过，出于兼容性考虑，请注意在任何非标准布局包括树内构建和虚拟环境中，该值都将被忽略。

**wchar\_t \*pythonpath\_env**

模块搜索路径 (`sys.path`) 为一个用 `DELIM(os.pathsep)` 分隔的字符串。

由 `PYTHONPATH` 环境变量设置。

預設值：NULL。

*Python 路径配置* 输入的一部分。

***PyWideStringList* module\_search\_paths**

**int module\_search\_paths\_set**

模块搜索路径: `sys.path`。

如果 `module_search_paths_set` 等于 0, `Py_InitializeFromConfig()` 将替代 `module_search_paths` 并将 `module_search_paths_set` 设为 1。

默认值: 空列表 (`module_search_paths`) 和 0 (`module_search_paths_set`)。

*Python 路径配置* 的一部分。

**int optimization\_level**

编译优化级别:

- 0: Peephole 优化器, 将 `__debug__` 设为 `True`。
- 1: 0 级, 移除断言, 将 `__debug__` 设为 `False`。
- 2: 1 级, 去除文档字符串。

通过 `-O` 命令行选项递增。设置为 `PYTHONOPTIMIZE` 环境变量值。

预设: 0。

**PyWideStringList orig\_argv**

传给 Python 可执行程序的原型命令行参数列表: `sys.orig_argv`。

如果 `orig_argv` 列表为空并且 `argv` 不是一个只包含空字符串的列表, `PyConfig_Read()` 将在修改 `argv` 之前把 `argv` 拷贝至 `orig_argv` (如果 `parse_argv` 不为空)。

另请参阅 `argv` 成员和 `Py_GetArgcArgv()` 函数。

默认值: 空列表。

在 3.10 版新加入。

**int parse\_argv**

解析命令行参数?

如果等于 1, 则以与常规 Python 解析 命令行参数相同的方式解析 `argv`, 并从 `argv` 中剥离 Python 参数。

`PyConfig_Read()` 函数只解析 `PyConfig.argv` 参数一次: 在参数解析完成后, `PyConfig.parse_argv` 将被设为 2。由于 Python 参数是从 `PyConfig.argv` 中剥离的, 因此解析参数两次会将应用程序选项解析为 Python 选项。

默认值: 在 Python 模式下为 1, 在隔离模式下为 0。

在 3.10 版的变更: 现在只有当 `PyConfig.parse_argv` 等于 1 时才会解析 `PyConfig.argv` 参数。

**int parser\_debug**

解析器调试模式。如果大于 0, 则打开解析器调试输出 (仅针对专家, 取决于编译选项)。

通过 `-d` 命令行选项递增。设置为 `PYTHONDEBUG` 环境变量值。

预设: 0。

**int pathconfig\_warnings**

如为非零值, 则允许计算路径配置以将警告记录到 `stderr` 中。如果等于 0, 则抑制这些警告。

默认值: 在 Python 模式下为 1, 在隔离模式下为 0。

*Python 路径配置* 输入的一部分。

在 3.11 版的变更: 现在也适用于 Windows。

**wchar\_t \*prefix**

安装依赖于平台的 Python 文件的站点专属目录前缀: `sys.prefix`。

預設值: `NULL`。

*Python 路径配置* 的一部分。

**wchar\_t \*program\_name**

用于初始化 *executable* 和在 Python 初始化期间早期错误消息中使用的程序名称。

- 如果 `Py_SetProgramName()` 已被调用, 将使用其参数。
- 在 macOS 上, 如果设置了 `PYTHONEXECUTABLE` 环境变量则会使用它。
- 如果定义了 `WITH_NEXT_FRAMEWORK` 宏, 当设置了 `__PYENVV_LAUNCHER__` 环境变量时将会使用它。
- 如果 *argv* 的 `argv[0]` 可用并且不为空值则会使用它。
- 否则, 在 Windows 上将使用 `L"python"`, 在其他平台上将使用 `L"python3"`。

預設值: `NULL`。

*Python 路径配置* 输入的一部分。

**wchar\_t \*pycache\_prefix**

缓存 `.pyc` 文件被写入到的目录: `sys.pycache_prefix`。

通过 `-X pycache_prefix=PATH` 命令行选项和 `PYTHONPYCACHEPREFIX` 环境变量设置。

如果为 `NULL`, 则 `sys.pycache_prefix` 将被设为 `None`。

預設值: `NULL`。

**int quiet**

安静模式。如果大于 0, 则在交互模式下启动 Python 时不显示版权和版本。

由 `-q` 命令行选项执行递增。

預設: 0。

**wchar\_t \*run\_command**

`-c` 命令行选项的值。

由 *Py\_RunMain()* 使用。

預設值: `NULL`。

**wchar\_t \*run\_filename**

通过命令行传入的文件名: 不包含 `-c` 或 `-m` 的附加命令行参数。它会被 *Py\_RunMain()* 函数使用。

例如, 对于命令行 `python3 script.py arg` 它将被设为 `script.py`。

也請見 *PyConfig.skip\_source\_first\_line* 選項。

預設值: `NULL`。

**wchar\_t \*run\_module**

`-m` 命令行选项的值。

由 *Py\_RunMain()* 使用。

預設值: `NULL`。

**int show\_ref\_count**

在退出时显示引用总数?

通过 `-X showrefcount` 命令行选项设置为 1。

需要 Python 调试编译版 (必须定义 `Py_REF_DEBUG` 宏)。



預設：0。

#### **int site\_import**

在启动时导入 site 模块？

如果等于零，则禁用模块站点的导入以及由此产生的与站点相关的 `sys.path` 操作。

如果以后显式地导入 site 模块也要禁用这些操作（如果你希望触发这些操作，请调用 `site.main()` 函数）。

通过 `-S` 命令行选项设置为 0。

`sys.flags.no_site` 会被设为 `site_import` 取反后的值。

預設值：1。

#### **int skip\_source\_first\_line**

如为非零值，则跳过 `PyConfig.run_filename` 源的第一行。

它将允许使用非 Unix 形式的 `#!cmd`。这是针对 DOS 专属的破解操作。

通过 `-x` 命令行选项设置为 1。

預設：0。

#### **wchar\_t \*stdio\_encoding**

#### **wchar\_t \*stdio\_errors**

`sys.stdin`、`sys.stdout` 和 `sys.stderr` 的编码格式和编码格式错误（但 `sys.stderr` 将始终使用 "backslashreplace" 错误处理器）。

如果 `Py_SetStandardStreamEncoding()` 已被调用，则当其 `error` 和 `errors` 参数不为 NULL 时将使用它们。

如果 `PYTHONIOENCODING` 环境变量非空则会使用它。

默认编码格式：

- 如果 `PyPreConfig.utf8_mode` 为非零值则使用 "UTF-8"。
- 在其他情况下，使用 *locale encoding*。

默认错误处理器：

- 在 Windows 上：使用 "surrogateescape"。
- 如果 `PyPreConfig.utf8_mode` 为非零值，或者如果 `LC_CTYPE` 语言区域为 "C" 或 "POSIX" 则使用 "surrogateescape"。
- 在其他情况下则使用 "strict"。

#### **int tracemalloc**

启用 tracemalloc？

如果为非零值，则在启动时调用 `tracemalloc.start()`。

通过 `-X tracemalloc=N` 命令行选项和 `PYTHONTRACEMALLOC` 环境变量设置。

默认值：在 Python 模式中为 -1，在隔离模式中为 0。

#### **int use\_environment**

使用 环境变量？

如果等于零，则忽略 环境变量。

由 `-E` 环境变量设置为 0。

默认值：在 Python 配置中为 1 而在隔离配置中为 0。

**int `user_site_directory`**

如果为非零值，则将用户站点目录添加到 `sys.path`。

通过 `-s` 和 `-I` 命令行选项设置为 0。

由 `PYTHONNOUSERSITE` 环境变量设置为 0。

默认值：在 Python 模式下为 1，在隔离模式下为 0。

**int `verbose`**

详细模式。如果大于 0，则每次导入模块时都会打印一条消息，显示加载模块的位置（文件名或内置模块）。

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` 命令行选项执行递增。

Set to the `PYTHONVERBOSE` environment variable value.

預設：0。

***PyWideStringList* `warnoptions`**

`warnings` 模块用于构建警告过滤器的选项，优先级从低到高: `sys.warnoptions`。

`warnings` 模块以相反的顺序添加 `sys.warnoptions`: 最后一个 *PyConfig.warnoptions* 条目将成为 `warnings.filters` 的第一个条目并将最先被检查（最高优先级）。

`-W` 命令行选项会将其值添加到 *warnoptions* 中，它可以被多次使用。

`PYTHONWARNINGS` 环境变量也可被用于添加警告选项。可以指定多个选项，并以逗号 (,) 分隔。

默认值：空列表。

**int `write_bytecode`**

如果等于 0，Python 将不会尝试在导入源模块时写入 `.pyc` 文件。

通过 `-B` 命令行选项和 `PYTHONDONTWRITEBYTECODE` 环境变量设置为 0。

`sys.dont_write_bytecode` 会被初始化为 *write\_bytecode* 取反后的值。

預設值：1。

***PyWideStringList* `xoptions`**

`-X` 命令行选项的值: `sys._xoptions`。

默认值：空列表。

如果 *parse\_argv* 为非零值，则 *argv* 参数将以与常规 Python 解析 命令行参数相同的方式被解析，并从 *argv* 中剥离 Python 参数。

*xoptions* 选项将会被解析以设置其他选项：参见 `-X` 命令行选项。

在 3.9 版的變更: `show_alloc_count` 字段已被移除。

## 10.7 使用 PyConfig 初始化

用于初始化 Python 的函数：

**PyStatus Py\_InitializeFromConfig** (const *PyConfig* \*config)

根据 *config* 配置来初始化 Python。

调用方要负责使用 *PyStatus\_Exception()* 和 *Py\_ExitStatusException()* 来处理异常（错误或退出）。

如果使用了 *PyImport\_FrozenModules()*、*PyImport\_AppendInittab()* 或 *PyImport\_ExtendInittab()*，则必须在 Python 预初始化之后、Python 初始化之前设置或调用它们。如果 Python 被多次初始化，则必须在每次初始化 Python 之前调用 *PyImport\_AppendInittab()* 或 *PyImport\_ExtendInittab()*。

当前的配置 (PyConfig 类型) 保存在 *PyInterpreterState.config* 中。

设置程序名称的示例：

```
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);
}
```

更完整的示例会修改默认配置，读取配置，然后覆盖某些参数。请注意自 3.11 版开始，许多参数在初始化之前不会被计算，因此无法从配置结构体中读取值。在调用初始化之前设置的任何值都将不会被初始化操作改变：

```
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);
```

(繼續下一頁)

(繼續上一頁)

```

    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Read all configuration at once */
    status = PyConfig_Read(&config);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Specify sys.path explicitly */
    /* If you want to modify the default set of paths, finish
       initialization first and then use PySys_GetObject("path") */
    config.module_search_paths_set = 1;
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/stdlib");
    if (PyStatus_Exception(status)) {
        goto done;
    }
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/more/modules");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Override executable computed by PyConfig_Read() */
    status = PyConfig_SetString(&config, &config.executable,
                               L"/path/to/my_executable");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    status = Py_InitializeFromConfig(&config);

done:
    PyConfig_Clear(&config);
    return status;
}

```

## 10.8 隔離配置

`PyPreConfig_InitIsolatedConfig()` 和 `PyConfig_InitIsolatedConfig()` 函数会创建一个配置来将 Python 与系统隔离开来。例如，将 Python 嵌入到某个应用程序。

该配置将忽略全局配置变量、环境变量、命令行参数 (`PyConfig.argv` 将不会被解析) 和用户站点目录。C 标准流 (例如 `stdout`) 和 `LC_CTYPE` 语言区域将保持不变。信号处理器将不会被安装。

该配置仍然会使用配置文件来确定未被指明的路径。请确保指定了 `PyConfig.home` 以避免计算默认的路径配置。

## 10.9 Python 配置

`PyPreConfig_InitPythonConfig()` 和 `PyConfig_InitPythonConfig()` 函数会创建一个配置来构建一个行为与常规 Python 相同的自定义 Python。

环境变量和命令行参数将被用于配置 Python，而全局配置变量将被忽略。

此函数将根据 `LC_CTYPE` 语言区域、`PYTHONUTF8` 和 `PYTHONCOERCECLOCALE` 环境变量启用 C 语言区域强制转换 (PEP 538) 和 Python UTF-8 模式 (PEP 540)。

## 10.10 Python 路径配置

`PyConfig` 包含多个用于路径配置的字段：

- 路径配置输入：
  - `PyConfig.home`
  - `PyConfig.platlibdir`
  - `PyConfig.pathconfig_warnings`
  - `PyConfig.program_name`
  - `PyConfig.pythonpath_env`
  - 当前工作目录：用于获取绝对路径
  - `PATH` 环境变量用于获取程序的完整路径 (来自 `PyConfig.program_name`)
  - `__PYENVV_LAUNCHER__` 環境變數
  - (仅限 Windows only) 注册表 `HKEY_CURRENT_USER` 和 `HKEY_LOCAL_MACHINE` 的“SoftwarePythonPythonCoreX.YPythonPath”项下的应用程序目录 (其中 X.Y 为 Python 版本)。
- 路径配置输出字段：
  - `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`

如果至少有一个“输出字段”未被设置，Python 就会计算路径配置来填充未设置的字段。如果 `module_search_paths_set` 等于 0，则 `module_search_paths` 将被覆盖并且 `module_search_paths_set` 将被设置为 1。

通过显式地设置上述所有路径配置输出字段可以完全忽略计算默认路径配置的函数。即使字符串不为空也会被视为已设置。如果 `module_search_paths_set` 被设为 1 则 `module_search_paths` 会被视为已设置。在这种情况下，`module_search_paths` 将不加修改地被使用。

将 `pathconfig_warnings` 设为 0 以便在计算路径配置时抑制警告 (仅限 Unix，Windows 不会记录任何警告)。

如果 `base_prefix` 或 `base_exec_prefix` 字段未设置，它们将分别从 `prefix` 和 `exec_prefix` 继承其值。

`Py_RunMain()` 和 `Py_Main()` 将修改 `sys.path`：

- 如果`run_filename`已设置并且是一个包含`__main__.py`脚本的目录，则会将`run_filename`添加到`sys.path`的开头。
- 如果`isolated`为零：
  - 如果设置了`run_module`，则将当前目录添加到`sys.path`的开头。如果无法读取当前目录则不执行任何操作。
  - 如果设置了`run_filename`，则将文件名的目录添加到`sys.path`的开头。
  - 在其他情况下，则将一个空字符串添加到`sys.path`的开头。

如果`site_import`为非零值，则`sys.path`可通过`site`模块修改。如果`user_site_directory`为非零值且用户的`site-package`目录存在，则`site`模块会将用户的`site-package`目录附加到`sys.path`。

路径配置会使用以下配置文件：

- `pyvenv.cfg`
- `._pth` 文件 (例如: `python._pth`)
- `pybuilddir.txt` (仅 Unix)

如果存在`._pth`文件：

- 將`isolated`設定 1。
- 將`use_environment`設定 0。
- 將`site_import`設定 0。
- 將`safe_path`設定 1。

`__PYENVENV_LAUNCHER__` 环境变量将被用于设置`PyConfig.base_executable`

## 10.11 Py\_RunMain()

`int Py_RunMain (void)`

执行在命令行或配置中指定的命令 (`PyConfig.run_command`)、脚本 (`PyConfig.run_filename`) 或模块 (`PyConfig.run_module`)。

在默认情况下如果使用了 `-i` 选项，则运行 REPL。

最后，终结化 Python 并返回一个可传递给 `exit()` 函数的退出状态。

请参阅 [Python 配置](#) 查看一个使用 `Py_RunMain()` 在隔离模式下始终运行自定义 Python 的示例。

## 10.12 Py\_GetArgcArgv()

`void Py_GetArgcArgv (int *argc, wchar_t ***argv)`

在 Python 修改原始命令行参数之前，获取这些参数。

另请参阅 `PyConfig.orig_argv` 成员。



## 10.13 多阶段初始化私有暂定 API

本节介绍的私有暂定 API 引入了多阶段初始化，它是 **PEP 432** 的核心特性：

- “核心” 初始化阶段，“最小化的基本 Python”：
  - 内置类型；
  - 内置异常；
  - 内置和已冻结模块；
  - `sys` 模块仅部分初始化（例如：`sys.path` 尚不存在）。
- ”主要” 初始化阶段，Python 被完全初始化：
  - 安装并配置 `importlib`；
  - 应用路径配置；
  - 安装信号处理器；
  - 完成 `sys` 模块初始化（例如：创建 `sys.stdout` 和 `sys.path`）；
  - 启用 `faulthandler` 和 `tracemalloc` 等可选功能；
  - 导入 `site` 模块；
  - 等等。

私有临时 API：

- `PyConfig._init_main`: 如果设为 0, `Py_InitializeFromConfig()` 将在“核心” 初始化阶段停止。
- `PyConfig._isolated_interpreter`: if non-zero, disallow threads, subprocesses and fork.

**`PyStatus_Py_InitializeMain`** (void)

进入“主要” 初始化阶段，完成 Python 初始化。

在“核心” 阶段不会导入任何模块，也不会配置 `importlib` 模块：路径配置 只会在“主要” 阶段期间应用。这可能允许在 Python 中定制 Python 以覆盖或微调路径配置，也可能会安装自定义的 `sys.meta_path` 导入器或导入钩子等等。

It may become possible to calculate the *Path Configuration* in Python, after the Core phase and before the Main phase, which is one of the **PEP 432** motivation.

“核心” 阶段并没有完整的定义：在这一阶段什么应该可用什么不应该可用都尚未被指明。该 API 被标记为私有和暂定的：也就是说该 API 可以随时被修改甚至被移除直到设计出适用的公共 API。

在“核心” 和“主要” 初始化阶段之间运行 Python 代码的示例：

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config._init_main = 0;

    /* ... customize 'config' configuration ... */

    status = Py_InitializeFromConfig(&config);
    PyConfig_Clear(&config);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
}
```

(繼續下一頁)

(繼續上一頁)

```
/* Use sys.stderr because sys.stdout is only created
   by _Py_InitializeMain() */
int res = PyRun_SimpleString(
    "import sys; "
    "print('Run Python code before _Py_InitializeMain', "
        "file=sys.stderr)");
if (res < 0) {
    exit(1);
}

/* ... put more configuration code here ... */

status = _Py_InitializeMain();
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}
}
```

## 11.1 總覽

在 Python 中，内存管理涉及到一个包含所有 Python 对象和数据结构的私有堆（heap）。这个私有堆的管理由内部的 Python 内存管理器（*Python memory manager*）保证。Python 内存管理器有不同的组件来处理各种动态存储管理方面的问题，如共享、分割、预分配或缓存。

在最底层，一个原始内存分配器通过与操作系统的内存管理器交互，确保私有堆中有足够的空间来存储所有与 Python 相关的数据。在原始内存分配器的基础上，几个对象特定的分配器在同一堆上运行，并根据每种对象类型的特点实现不同的内存管理策略。例如，整数对象在堆内的管理方式不同于字符串、元组或字典，因为整数需要不同的存储需求和速度与空间的权衡。因此，Python 内存管理器将一些工作分配给对象特定分配器，但确保后者在私有堆的范围内运行。

Python 堆内存的管理是由解释器来执行，用户对它没有控制权，即使他们经常操作指向堆内内存块的对象指针，理解这一点十分重要。Python 对象和其他内部缓冲区的堆空间分配是由 Python 内存管理器按需通过本文档中列出的 Python/C API 函数进行的。

为了避免内存破坏，扩展的作者永远不应该试图用 C 库函数导出的函数来对 Python 对象进行操作，这些函数包括：`malloc()`、`calloc()`、`realloc()` 和 `free()`。这将导致 C 分配器和 Python 内存管理器之间的混用，引发严重后果，这是由于它们实现了不同的算法，并在不同的堆上操作。但是，我们可以安全地使用 C 库分配器为单独的目的分配和释放内存块，如下例所示：

```
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;
```

在这个例子中，I/O 缓冲区的内存请求是由 C 库分配器处理的。Python 内存管理器只参与了分配作为结果返回的字节对象。

然而，在大多数情况下，都建议专门基于 Python 堆来分配内存，因为后者是由 Python 内存管理器控制的。例如，当解释器使用 C 编写的新对象类型进行扩展时必须这样做。使用 Python 堆的另一个理由是需要能通知 Python 内存管理器有关扩展模块的内存需求。即使所请求的内存全部只用于内部的、高度特定的目的，将所有的内存请求交给 Python 内存管理器能让解释器对其内存占用的整体情况有更准确的了

解。因此，在特定情况下，Python 内存管理器可能会触发或不触发适当的操作，如垃圾回收、内存压缩或其他的预防性操作。请注意通过使用前面例子所演示的 C 库分配器，为 I/O 缓冲区分配的内存将完全不受 Python 内存管理器的控制。

#### 也参考：

环境变量 `PYTHONMALLOC` 可被用来配置 Python 所使用的内存分配器。

环境变量 `PYTHONMALLOCSTATS` 可以用来在每次创建和关闭新的 `pymalloc` 对象区域时打印 `pymalloc` 内存分配器的统计数据。

## 11.2 分配器域

所有分配函数都属于三个不同的“分配器域”之一（见 `PyMemAllocatorDomain`）。这些域代表了不同的分配策略，并为不同目的进行了优化。每个域如何分配内存和每个域调用哪些内部函数的具体细节被认为是实现细节，但是出于调试目的，可以在[此处](#)找到一张简化的表格。没有硬性要求将属于给定域的分配函数返回的内存，仅用于该域提示的目的（虽然这是推荐的做法）。例如，你可以将 `PyMem_RawMalloc()` 返回的内存用于分配 Python 对象，或者将 `PyObject_Malloc()` 返回的内存用作缓冲区。

三个分配域分别是：

- 原始域：用于为通用内存缓冲区分配内存，分配 \* 必须 \* 转到系统分配器并且分配器可以在没有 `GIL` 的情况下运行。内存直接请求自系统。
- “Mem”域：用于为 Python 缓冲区和通用内存缓冲区分配内存，分配时必须持有 `GIL`。内存取自于 Python 私有堆。
- 对象域：用于分配属于 Python 对象的内存。内存取自于 Python 私有堆。

当释放属于给定域的分配函数先前分配的内存时，必须使用对应的释放函数。例如，`PyMem_Free()` 来释放 `PyMem_Malloc()` 分配的内存。

## 11.3 原始内存接口

以下函数集封装了系统分配器。这些函数是线程安全的，不需要持有全局解释器锁。

默认原始内存分配器使用以下函数：`malloc()`、`calloc()`、`realloc()` 和 `free()`；当请求零个字节时则调用 `malloc(1)`（或 `calloc(1, 1)`）。

在 3.4 版新加入。

`void *PyMem_RawMalloc (size_t n)`

分配  $n$  个字节并返回一个指向所分配内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyMem_RawMalloc(1)` 一样。但是内存不会以任何方式被初始化。

`void *PyMem_RawCalloc (size_t nelem, size_t elsize)`

分配  $nelem$  个元素，每个元素的大小为  $elsize$  个字节，并返回指向所分配的内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。内存会被初始化为零。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyMem_RawCalloc(1, 1)` 一样。

在 3.5 版新加入。

`void *PyMem_RawRealloc (void *p, size_t n)`

将  $p$  指向的内存块大小调整为  $n$  字节。以新旧内存块大小中的最小值为准，其中内容保持不变，

如果  $p$  是 `NULL`，则相当于调用 `PyMem_RawMalloc(n)`；如果  $n$  等于 0，则内存块大小会被调整，但不会被释放，返回非 `NULL` 指针。

除非  $p$  是 `NULL`，否则它必须是之前调用 `PyMem_RawMalloc()`、`PyMem_RawRealloc()` 或 `PyMem_RawCalloc()` 所返回的。

如果请求失败，`PyMem_RawRealloc()` 返回 `NULL`， $p$  仍然是指向先前内存区域的有效指针。

**void `PyMem_RawFree` (void \*p)**

释放  $p$  指向的内存块。 $p$  必须是之前调用 `PyMem_RawMalloc()`、`PyMem_RawRealloc()` 或 `PyMem_RawCalloc()` 所返回的指针。否则，或在 `PyMem_RawFree(p)` 之前已经调用过的情况下，未定义的行为会发生。

如果  $p$  是 `NULL`，那么什么操作也不会进行。

## 11.4 内存接口

以下函数集，仿照 ANSI C 标准，并指定了请求零字节时的行为，可用于从 Python 堆分配和释放内存。

默认内存分配器使用了 `pymalloc` 内存分配器。

**警告：** 在使用这些函数时，必须持有全局解释器锁 (*GIL*)。

在 3.6 版的變更：现在默认的分配器是 `pymalloc` 而非系统的 `malloc()`。

**void \*`PyMem_Malloc` (size\_t n)**

属于稳定 ABI。分配  $n$  个字节并返回一个指向所分配内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyMem_Malloc(1)` 一样。但是内存不会以任何方式被初始化。

**void \*`PyMem_Calloc` (size\_t nelem, size\_t elsize)**

属于稳定 ABI 自 3.7 版起。分配  $nelem$  个元素，每个元素的大小为  $elsize$  个字节，并返回指向所分配的内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。内存会被初始化为零。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyMem_Calloc(1, 1)` 一样。

在 3.5 版新加入。

**void \*`PyMem_Realloc` (void \*p, size\_t n)**

属于稳定 ABI。将  $p$  指向的内存块大小调整为  $n$  字节。以新旧内存块大小中的最小值为准，其中内容保持不变，

如果  $p$  是 `NULL`，则相当于调用 `PyMem_Malloc(n)`；如果  $n$  等于 0，则内存块大小会被调整，但不会被释放，返回非 `NULL` 指针。

除非  $p$  是 `NULL`，否则它必须是之前调用 `PyMem_Malloc()`、`PyMem_Realloc()` 或 `PyMem_Calloc()` 所返回的。

如果请求失败，`PyMem_Realloc()` 返回 `NULL`， $p$  仍然是指向先前内存区域的有效指针。

**void `PyMem_Free` (void \*p)**

属于稳定 ABI。释放  $p$  指向的内存块。 $p$  必须是之前调用 `PyMem_Malloc()`、`PyMem_Realloc()` 或 `PyMem_Calloc()` 所返回的指针。否则，或在 `PyMem_Free(p)` 之前已经调用过的情况下，未定义的行为会发生。

如果  $p$  是 `NULL`，那么什么操作也不会进行。

以下面向类型的宏为方便而提供。注意 *TYPE* 可以指任何 C 类型。

**`PyMem_New` (TYPE, n)**

与 `PyMem_Malloc()` 相同，但会分配  $(n * \text{sizeof}(\text{TYPE}))$  字节的内存。返回一个转换为 `TYPE*` 的指针。内存不会以任何方式被初始化。

**PyMem\_Resize** (p, TYPE, n)

与 `PyMem_Realloc()` 类似，但内存块的大小被调整为  $(n * \text{sizeof}(\text{TYPE}))$  个字节。返回一个转换为 `TYPE*` 的指针。在返回时，`p` 将是一个指向新内存区域的指针，或者如果执行失败则为 `NULL`。

这是一个 C 预处理宏，`p` 总是被重新赋值。请保存 `p` 的原始值，以避免在处理错误时丢失内存。

void **PyMem\_Del** (void \*p)

和 `PyMem_Free()` 相同。

此外，我们还提供了以下宏集用于直接调用 Python 内存分配器，而不涉及上面列出的 C API 函数。但是请注意，使用它们并不能保证跨 Python 版本的二进制兼容性，因此在扩展模块被弃用。

- `PyMem_MALLOC(size)`
- `PyMem_NEW(type, size)`
- `PyMem_REALLOC(ptr, size)`
- `PyMem_RESIZE(ptr, type, size)`
- `PyMem_FREE(ptr)`
- `PyMem_DEL(ptr)`

## 11.5 对象分配器

以下函数集，仿照 ANSI C 标准，并指定了请求零字节时的行为，可用于从 Python 堆分配和释放内存。

**備註：**当通过自定义内存分配器部分描述的方法拦截该域中的分配函数时，无法保证这些分配器返回的内存可以被成功地转换成 Python 对象。

默认对象分配器使用 `pymalloc` 内存分配器。

**警告：**在使用这些函数时，必须持有全局解释器锁（*GIL*）。

void \***PyObject\_Malloc** (size\_t n)

属于稳定 ABI。分配  $n$  个字节并返回一个指向所分配内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyObject_Malloc(1)` 一样。但是内存不会以任何方式被初始化。

void \***PyObject\_Calloc** (size\_t nelem, size\_t elsize)

属于稳定 ABI 自 3.7 版起。分配  $nelem$  个元素，每个元素的大小为  $elsize$  个字节，并返回指向所分配的内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。内存会被初始化为零。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyObject_Calloc(1, 1)` 一样。

在 3.5 版新加入。

void \***PyObject\_Realloc** (void \*p, size\_t n)

属于稳定 ABI。将 `p` 指向的内存块大小调整为  $n$  字节。以新旧内存块大小中的最小值为准，其中内容保持不变。

如果 `*p` 是 `NULL`，则相当于调用 `PyObject_Malloc(n)`；如果  $n$  等于 0，则内存块大小会被调整，但不会被释放，返回非 `NULL` 指针。

除非 `p` 是 `NULL`，否则它必须是之前调用 `PyObject_Malloc()`、`PyObject_Realloc()` 或 `PyObject_Calloc()` 所返回的。



如果请求失败, `PyObject_Realloc()` 返回 `NULL`, `p` 仍然是指向先前内存区域的有效指针。

void **PyObject\_Free**(void \*p)

属于稳定 ABI. 释放 `p` 指向的内存块。`p` 必须是之前调用 `PyObject_Malloc()`、`PyObject_Realloc()` 或 `PyObject_Calloc()` 所返回的指针。否则, 或在 `PyObject_Free(p)` 之前已经调用过的情况下, 未定义行为会发生。

如果 `p` 是 `NULL`, 那么什么操作也不会进行。

## 11.6 默认内存分配器

默认内存分配器:

配置	名称	PyMem_RawMallc	PyMem_Malloc	PyOb- ject_Malloc
发布版本	"pymalloc"	malloc	pymalloc	pymalloc
调试构建	"pymalloc_debug"	malloc + debug	pymalloc + de- bug	pymalloc + de- bug
没有 pymalloc 的发布 版本	"malloc"	malloc	malloc	malloc
没有 pymalloc 的调试 构建	"malloc_debug"	malloc + debug	malloc + debug	malloc + debug

说明:

- 名称: `PYTHONMALLOC` 环境变量的值。
- `malloc`: 来自 C 标准库的系统分配器, C 函数: `malloc()`、`calloc()`、`realloc()` 和 `free()`。
- `pymalloc`: *pymalloc* 内存分配器。
- "+ debug": 附带 *Python* 内存分配器的调试钩子。
- “调试构建”: 调试模式下的 Python 构建。

## 11.7 自定义内存分配器

在 3.4 版新加入。

type **PyMemAllocatorEx**

用于描述内存块分配器的结构体。该结构体下列字段:

域	含意
void *ctx	作为第一个参数传入的用户上下文
void* malloc(void *ctx, size_t size)	分配一个内存块
void* calloc(void *ctx, size_t nelem, size_t elsize)	分配一个初始化为 0 的内存块
void* realloc(void *ctx, void *ptr, size_t new_size)	分配一个内存块或调整其大小
void free(void *ctx, void *ptr)	释放一个内存块

在 3.5 版的變更: `PyMemAllocator` 结构被重命名为 *PyMemAllocatorEx* 并新增了一个 `calloc` 字段。

type **PyMemAllocatorDomain**

用来识别分配器域的枚举类。域有：

**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)

获取指定域的内存块分配器。

void **PyMem\_SetAllocator** (*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* \*allocator)

设置指定域的内存块分配器。

当请求零字节时，新的分配器必须返回一个独特的非 NULL 指针。

对于 *PYMEM\_DOMAIN\_RAW* 域，分配器必须是线程安全的：当分配器被调用时将不持有 *GIL*。

如果新的分配器不是钩子（不调用之前的分配器），必须调用 *PyMem\_SetupDebugHooks()* 函数在新分配器上重新安装调试钩子。

另请参阅 *PyPreConfig.allocator* 和 *Preinitialize Python with PyPreConfig*。

**警告：** *PyMem\_SetAllocator()* 没有以下合约：

- 可以在 *Py\_PreInitialize()* 之后 *Py\_InitializeFromConfig()* 之前调用它来安装自定义的内存分配器。对于所安装的分配器除了域的规定以外没有任何其他限制（例如 Raw Domain 允许分配器在不持有 GIL 的情况下被调用）。请参阅有关分配器域的章节了解详情。
- 如果在 Python 已完成初始化之后（即 *Py\_InitializeFromConfig()* 被调用之后）被调用则自定义分配器 **must** 必须包装现有的分配器。将现有分配器替换为任意的其他分配器是 **不受支持的**。

void **PyMem\_SetupDebugHooks** (void)

设置 *Python* 内存分配器的调试钩子 以检测内存错误。

## 11.8 Python 内存分配器的调试钩子

当 Python 在调试模式下构建, `PyMem_SetupDebugHooks()` 函数在 Python 预初始化时被调用, 以在 Python 内存分配器上设置调试钩子以检测内存错误。

PYTHONMALLOC 环境变量可被用于在以发行模式下编译的 Python 上安装调试钩子 (例如: PYTHONMALLOC=debug)。

`PyMem_SetupDebugHooks()` 函数可被用于在调用了 `PyMem_SetAllocator()` 之后设置调试钩子。

这些调试钩子用特殊的、可辨认的位模式填充动态分配的内存块。新分配的内存用字节 0xCD (PYMEM\_CLEANBYTE) 填充, 释放的内存用字节 0xDD (PYMEM\_DEADBYTE) 填充。内存块被填充了字节 0xFD (PYMEM\_FORBIDDENBYTE) 的“禁止字节”包围。这些字节串不太可能是合法的地址、浮点数或 ASCII 字符串

运行时检查:

- 检测对 API 的违反。例如: 检测对 `PyMem_Malloc()` 分配的内存块调用 `PyObject_Free()`。
- 检测缓冲区起始位置前的写入 (缓冲区下溢)。
- 检测缓冲区终止位置后的写入 (缓冲区溢出)。
- 检测当调用 `PYMEM_DOMAIN_OBJ` (如: `PyObject_Malloc()`) 和 `PYMEM_DOMAIN_MEM` (如: `PyMem_Malloc()`) 域的分配器函数时是否持有 GIL。

在出错时, 调试钩子使用 `tracemalloc` 模块来回溯内存块被分配的位置。只有当 `tracemalloc` 正在追踪 Python 内存分配, 并且内存块被追踪时, 才会显示回溯。

让  $S = \text{sizeof}(\text{size\_t})$ 。将  $2 \times S$  个字节添加到每个被请求的  $N$  字节数据块的两端。内存的布局像是这样, 其中  $p$  代表由类似 `malloc` 或类似 `realloc` 的函数所返回的地址 ( $p[i:j]$  表示从  $*(p+i)$  左侧开始到  $*(p+j)$  左侧止的字节数据切片; 请注意对负索引号的处理与 Python 切片是不同的):

**`p[-2*S:-S]`**

最初所要求的字节数。这是一个 `size_t`, 为大端序 (易于在内存转储中读取)。

**`p[-S]`**

API 标识符 (ASCII 字符):

- 'r' 表示 `PYMEM_DOMAIN_RAW`。
- 'm' 表示 `PYMEM_DOMAIN_MEM`。
- 'o' 表示 `PYMEM_DOMAIN_OBJ`。

**`p[-S+1:0]`**

PYMEM\_FORBIDDENBYTE 的副本。用于捕获下层的写入和读取。

**`p[0:N]`**

所请求的内存, 用 PYMEM\_CLEANBYTE 的副本填充, 用于捕获对未初始化内存的引用。当调用 `realloc` 之类的函数来请求更大的内存块时, 额外新增的字节也会用 PYMEM\_CLEANBYTE 来填充。当调用 `free` 之类的函数时, 这些字节会用 PYMEM\_DEADBYTE 来重写, 以捕获对已释放内存的引用。当调用 `realloc` 之类的函数来请求更小的内存块时, 多余的旧字节也会用 PYMEM\_DEADBYTE 来填充。

**`p[N:N+S]`**

PYMEM\_FORBIDDENBYTE 的副本。用于捕获超限的写入和读取。

**`p[N+S:N+2*S]`**

仅当定义了 PYMEM\_DEBUG\_SERIALNO 宏时会被使用 (默认情况下将不定义)。

一个序列号, 每次调用 `malloc` 或 `realloc` 之类的函数时都会递增 1。大端序的 `size_t`。如果之后检测到了“被破坏的内存”, 此序列号提供了一个很好的手段用来在下次运行时设置中断点, 以捕获该内存块被破坏的瞬间。obmalloc.c 中的静态函数 `bumpserialno()` 是唯一会递增序列号的函数, 它的存在让你可以轻松地设置这样的中断点。

一个 `realloc` 之类或 `free` 之类的函数会先检查两端的 `PYMEM_FORBIDDENBYTE` 字节是否完好。如果它们被改变了，则会将诊断输出写入到 `stderr`，并且程序将通过 `Py_FatalError()` 中止。另一种主要的失败模式是当程序读到某种特殊的比特模式并试图将其用作地址时触发内存错误。如果你随即进入调试器并查看该对象，你很可能会看到它已完全被填充为 `PYMEM_DEADBYTE` (意味着已释放的内存被使用) 或 `PYMEM_CLEANBYTE` (意味着未初始货摊内存被使用)。

在 3.6 版的變更: `PyMem_SetupDebugHooks()` 函数现在也能在使用发布模式编译的 Python 上工作。当发生错误时，调试钩子现在会使用 `tracemalloc` 来获取已分配内存块的回溯信息。调试钩子现在还会在 `PYMEM_DOMAIN_OBJ` 和 `PYMEM_DOMAIN_MEM` 作用域的函数被调用时检查是否持有 GIL。

在 3.8 版的變更: 字节模式 `0xCB` (`PYMEM_CLEANBYTE`)、`0xDB` (`PYMEM_DEADBYTE`) 和 `0xFB` (`PYMEM_FORBIDDENBYTE`) 已被 `0xCD`、`0xDD` 和 `0xFD` 替代以使用与 Windows CRT 调试 `malloc()` 和 `free()` 相同的值。

## 11.9 pymalloc 分配器

Python 有一个针对短生命周期的小对象 (小于或等于 512 字节) 进行了优化的 `pymalloc` 分配器。它使用名为 “arena” 的内存映射，在 32 位平台上的固定大小为 256 KiB，在 64 位平台上的固定大小为 1 MiB。对于大于 512 字节的分配，它会回退为 `PyMem_RawMalloc()` 和 `PyMem_RawRealloc()`。

`pymalloc` 是 `PYMEM_DOMAIN_MEM` (例如: `PyMem_Malloc()`) 和 `PYMEM_DOMAIN_OBJ` (例如: `PyObject_Malloc()`) 域的默认分配器。

arena 分配器使用以下函数：

- Windows 上的 `VirtualAlloc()` 和 `VirtualFree()`，
- `mmap()` 和 `munmap()`，如果可用的话，
- 否则，`malloc()` 和 `free()`。

如果 Python 配置了 `--without-pymalloc` 选项，那么此分配器将被禁用。也可以在运行时使用 `PYTHONMALLOC`` (例如: ``PYTHONMALLOC=malloc``) 环境变量来禁用它。

### 11.9.1 自定义 pymalloc Arena 分配器

在 3.4 版新加入。

type **PyObjectArenaAllocator**

用来描述一个 arena 分配器的结构体。这个结构体有三个字段：

域	含意
<code>void *ctx</code>	作为第一个参数传入的用户上下文
<code>void* alloc(void *ctx, size_t size)</code>	分配一块 <code>size</code> 字节的区域
<code>void free(void *ctx, void *ptr, size_t size)</code>	释放一块区域

void **PyObject\_GetArenaAllocator** (*PyObjectArenaAllocator* \*allocator)

获取 arena 分配器

void **PyObject\_SetArenaAllocator** (*PyObjectArenaAllocator* \*allocator)

设置 arena 分配器

## 11.10 tracemalloc C API

在 3.7 版新加入。

`int PyTraceMalloc_Track (unsigned int domain, uintptr_t ptr, size_t size)`

在 `tracemalloc` 模块中跟踪一个已分配的内存块。

成功时返回 0，出错时返回 -1 (无法分配内存来保存跟踪信息)。如果禁用了 `tracemalloc` 则返回 -2。

如果内存块已被跟踪，则更新现有跟踪信息。

`int PyTraceMalloc_Untrack (unsigned int domain, uintptr_t ptr)`

在 `tracemalloc` 模块中取消跟踪一个已分配的内存块。如果内存块未被跟踪则不执行任何操作。

如果 `tracemalloc` 被禁用则返回 -2，否则返回 0。

## 11.11 范例

以下是来自總覽 小节的示例，经过重写以使 I/O 缓冲区是通过使用第一个函数集从 Python 堆中分配的：

```
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;
```

使用面向类型函数集의 相同代码：

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

请注意在以上两个示例中，缓冲区总是通过归属于相同集的函数来操纵的。事实上，对于一个给定的内存块必须使用相同的内存 API 族，以便使得混合不同分配器的风险减至最低。以下代码序列包含两处错误，其中一个被标记为 *fatal* 因为它混合了两种在不同堆上操作的不同分配器。

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3); /* Wrong -- should be PyMem_Free() */
free(buf2);      /* Right -- allocated via malloc() */
free(buf1);      /* Fatal -- should be PyMem_Del() */
```

除了用于处理来自 Python 堆的原始内存块的函数，Python 中的对象还通过 `PyObject_New`、`PyObject_NewVar` 和 `PyObject_Del()` 进行分配和释放。

这些将在有关如何在 C 中定义和实现新对象类型的下一章中讲解。





本章描述了定义新对象类型时所使用的函数、类型和宏。

## 12.1 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。属于稳定 ABI。用它的型別和初始參照來初始化新分配物件 *op*。已初始化的物件會被回傳。如果 *type* 表示了該物件參與圈垃圾檢查器，則將其新增到檢查器的觀察物件集合中。物件的其他欄位不受影響。

*PyVarObject* \***PyObject\_InitVar** (*PyVarObject* \*op, *PyTypeObject* \*type, *Py\_ssize\_t* size)

回傳值：借用參照。属于稳定 ABI。它會做到 *PyObject\_Init()* 的所有功能，並且會初始化一個大小可變物件的長度資訊。

**PyObject\_New** (TYPE, typeobj)

使用 C 结构类型 *TYPE* 和 Python 类型对象 *typeobj* (*PyTypeObject*\*) 分配一个新的 Python 对象。f 未在该 Python 对象标头中定义的字段不会被初始化。调用方将拥有对该对象的唯一引用（即引用计数将为 1）。内存分配的大小由类型对象的 *tp\_basicsize* 字段决定。

**PyObject\_NewVar** (TYPE, typeobj, size)

使用 C 结构类型 *TYPE* 和 Python 类型对象 *typeobj* (*PyTypeObject*\*) 分配一个新的 Python 对象。未在该 Python 对象标头中定义的字段不会被初始化。被分配的内存允许 *TYPE* 结构加 *typeobj* 的 *tp\_itemsize* 字段所给出的 *size* (*Py\_ssize\_t*) 个字段。这对于实现像元组这样能够在构造时确定其大小的对象来说很有用。将字段数组嵌入到相同的内在分配中可减少内存分配的次数，这提高了内存管理效率。

void **PyObject\_Del** (void \*op)

释放使用 *PyObject\_New* 或 *PyObject\_NewVar* 分配给一个对象的内存。这通常由在对象的类型中指定的 *tp\_dealloc* 处理器来调用。在此调用之后该对象中的字段不应再被访问因为原来的内存已不再是一个有效的 Python 对象。

**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：

**PyModule\_Create()**

分配記憶體和建立擴充模組。

## 12.2 通用物件結構

大量的結構體被用於定義 Python 的對象類型。這一節描述了這些的結構體和它們的使用方法。

### 12.2.1 基本的對象類型和宏

所有的 Python 對象都在對象的內存表示的開始部分共享少量的字段。這些字段用 `PyObject` 或 `PyVarObject` 類型來表示，這些類型又由一些宏定義，這些宏也直接間接地用於所有其他 Python 對象的定義。

**type PyObject**

屬於受限 ABI。（僅特定成員屬於穩定 ABI。）所有對象類型都是此類型的擴展。這是一個包含了 Python 將對象的指針當作對象來處理所需的資訊的類型。在一個普通的“發行”編譯版中，它只包含對象的引用計數和指向對應類型對象的指針。沒有什麼對象被實際聲明為 `PyObject`，但每個指向 Python 對象的指針都可以被轉換為 `PyObject*`。對成員的訪問必須通過使用 `Py_REFCNT` 和 `Py_TYPE` 宏來完成。

**type PyVarObject**

屬於受限 ABI。（僅特定成員屬於穩定 ABI。）這是一個添加了 `ob_size` 字段的 `PyObject` 擴展。它僅用於具有某些長度標記的對象。此類型並不經常在 Python/C API 中出現。對成員的訪問必須通過使用 `Py_REFCNT`、`Py_TYPE` 和 `Py_SIZE` 宏來完成。

**PyObject\_HEAD**

這是一個在聲明代表無可變長度對象的新類型時所使用的宏。`PyObject_HEAD` 宏被擴展為：

```
PyObject ob_base;
```

參見上面 `PyObject` 的文檔。

**PyObject\_VAR\_HEAD**

這是一個在聲明代表每個實例具有可變長度的對象時所使用的宏。`PyObject_VAR_HEAD` 宏被擴展為：

```
PyVarObject ob_base;
```

參見上面 `PyVarObject` 的文檔。

**int Py\_Is (PyObject \*x, PyObject \*y)**

屬於穩定 ABI 自 3.10 版起。測試 `x` 是否為 `y` 對象，與 Python 中的 `x is y` 相同。

在 3.10 版新加入。

**int Py\_IsNone (PyObject \*x)**

屬於穩定 ABI 自 3.10 版起。測試一個對象是否為 None 單例，與 Python 中的 `x is None` 相同。

在 3.10 版新加入。

**int Py\_IsTrue (PyObject \*x)**

屬於穩定 ABI 自 3.10 版起。測試一個對象是否為 True 單例，與 Python 中的 `x is True` 相同。

在 3.10 版新加入。

`int Py_IsFalse (PyObject *x)`

属于稳定 ABI 自 3.10 版起. 测试一个对象是否为 False 单例, 与 Python 中的 `x is False` 相同。  
在 3.10 版新加入。

`PyTypeObject *Py_TYPE (PyObject *o)`

获取 Python 对象 *o* 的类型。

返回一个 *borrowed reference*。

使用 `Py_SET_TYPE()` 函数来设置一个对象类型。

在 3.11 版的變更: `Py_TYPE()` 被改为一个内联的静态函数。形参类型不再是 `const PyObject*`。

`int Py_IS_TYPE (PyObject *o, PyTypeObject *type)`

如果对象 *o* 的类型为 *type* 则返回非零值。否则返回零。等价于: `Py_TYPE(o) == type`。

在 3.9 版新加入。

`void Py_SET_TYPE (PyObject *o, PyTypeObject *type)`

将对象 *o* 的类型设为 *type*。

在 3.9 版新加入。

`Py_ssize_t Py_REFCNT (PyObject *o)`

获取 Python 对象 *o* 的引用计数。

使用 `Py_SET_REFCNT()` 函数来设置一个对象引用计数。

在 3.11 版的變更: 形参类型不再是 `const PyObject*`。

在 3.10 版的變更: `Py_REFCNT()` 被改为内联的静态函数。

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

将对象 *o* 的引用计数器设为 *refcnt*。

在 3.9 版新加入。

`Py_ssize_t Py_SIZE (PyVarObject *o)`

获取 Python 对象 *o* 的大小。

使用 `Py_SET_SIZE()` 函数来设置一个对象大小。

在 3.11 版的變更: `Py_SIZE()` 被改为一个内联静态函数。形参类型不再是 `const PyVarObject*`。

`void Py_SET_SIZE (PyVarObject *o, Py_ssize_t size)`

将对象 *o* 的大小设为 *size*。

在 3.9 版新加入。

`PyObject_HEAD_INIT (type)`

这是一个为新的 `PyObject` 类型扩展初始化值的宏。该宏扩展为:

```
_PyObject_EXTRA_INIT
1, type,
```

`PyVarObject_HEAD_INIT (type, size)`

这是一个为新的 `PyVarObject` 类型扩展初始化值的宏, 包括 `ob_size` 字段。该宏会扩展为:

```
_PyObject_EXTRA_INIT
1, type, size,
```

## 12.2.2 实现函数和方法

### type `PyCFunction`

属于稳定 ABI. 用于在 C 中实现大多数 Python 可调用对象的函数类型。该类型的函数接受两个 `PyObject*` 形参并返回一个这样的值。如果返回值为 `NULL`, 则将设置一个异常。如果不为 `NULL`, 则返回值将被解读为 Python 中暴露的函数的返回值。此函数必须返回一个新的引用。

函数的签名为:

```
PyObject *PyCFunction(PyObject *self,
                      PyObject *args);
```

### type `PyCFunctionWithKeywords`

属于稳定 ABI. 用于在 C 中实现具有 `METH_VARARGS | METH_KEYWORDS` 签名的 Python 可调用对象的函数类型。函数的签名为:

```
PyObject *PyCFunctionWithKeywords(PyObject *self,
                                   PyObject *args,
                                   PyObject *kwargs);
```

### type `_PyCFunctionFast`

用于在 C 中实现具有 `METH_FASTCALL` 签名的 Python 可调用对象的函数类型。函数的签名为:

```
PyObject *_PyCFunctionFast(PyObject *self,
                           PyObject *const *args,
                           Py_ssize_t nargs);
```

### type `_PyCFunctionFastWithKeywords`

用于在 C 中实现具有 `METH_FASTCALL | METH_KEYWORDS` 签名的 Python 可调用对象的函数类型。函数的签名为:

```
PyObject *_PyCFunctionFastWithKeywords(PyObject *self,
                                        PyObject *const *args,
                                        Py_ssize_t nargs,
                                        PyObject *kwnames);
```

### type `PyMethod`

用于在 C 中实现具有 `METH_METHOD | METH_FASTCALL | METH_KEYWORDS` 签名的 Python 可调用对象的函数类型。函数的签名为:

```
PyObject *PyMethod(PyObject *self,
                   PyTypeObject *defining_class,
                   PyObject *const *args,
                   Py_ssize_t nargs,
                   PyObject *kwnames)
```

在 3.9 版新加入。

### type `PyMethodDef`

属于稳定 ABI (包括所有成员). 用于描述一个扩展类型的方法的结构体。该结构体有四个字段:

`const char *m1_name`

方法的名称。

`PyCFunction m1_meth`

指向 C 语言实现的指针。

`int m1_flags`

指明调用应当如何构建的旗标位。

```
const char *ml_doc
```

指向文档字符串的内容。

`ml_meth` 是一个 C 函数指针。该函数可以为不同类型，但它们将总是返回 `PyObject*`。如果该函数不属于 `PyCFunction`，则编译器将要求在方法表中进行转换。尽管 `PyCFunction` 将第一个参数定义为 `PyObject*`，但该方法的实现使用 `self` 对象的特定 C 类型也很常见。

`ml_flags` 字段是可以包含以下旗标的位字段。每个旗标表示一个调用惯例或绑定惯例。

调用惯例有如下这些：

#### **METH\_VARARGS**

这是典型的调用惯例，其中方法的类型为 `PyCFunction`。该函数接受两个 `PyObject*` 值。第一个是用于方法的 `self` 对象；对于模块函数，它将为模块对象。第二个形参（常被命名为 `args`）是一个代表所有参数的元组对象。该形参通常是使用 `PyArg_ParseTuple()` 或 `PyArg_UnpackTuple()` 来处理的。

#### **METH\_KEYWORDS**

只能用于同其他旗标形成特定的组合：`METH_VARARGS | METH_KEYWORDS`，`METH_FASTCALL | METH_KEYWORDS` 和 `METH_METHOD | METH_FASTCALL | METH_KEYWORDS`。

#### **METH\_VARARGS | METH\_KEYWORDS**

带有这些旗标的方法必须为 `PyCFunctionWithKeywords` 类型。该函数接受三个形参：`self`，`args`，`kwargs` 其中 `kwargs` 是一个包含所有关键字参数的字典或者如果没有关键字参数则可以为 `NULL`。这些形参通常是使用 `PyArg_ParseTupleAndKeywords()` 来处理的。

#### **METH\_FASTCALL**

快速调用惯例仅支持位置参数。这些方法的类型为 `_PyCFunctionFast`。第一个形参为 `self`，第二个形参是由表示参数的 `PyObject*` 值组成的数组而第三个形参是参数的数量（数组的长度）。

在 3.7 版新加入。

在 3.10 版的變更：`METH_FASTCALL` 现在是稳定 ABI 的一部分。

#### **METH\_FASTCALL | METH\_KEYWORDS**

`METH_FASTCALL` 的扩展也支持关键字参数，它使用类型为 `_PyCFunctionFastWithKeywords` 的方法。关键字参数的传递方式与 `vectorcall` 协议中的相同：还存在额外的第四个 `PyObject*` 参数，它是一个代表关键字参数名称（它将保证为字符串）的元组，或者如果没有关键字则可以为 `NULL`。关键字参数的值存放在 `args` 数组中，在位置参数之后。

在 3.7 版新加入。

#### **METH\_METHOD**

只能与其他旗标组合使用：`METH_METHOD | METH_FASTCALL | METH_KEYWORDS`。

#### **METH\_METHOD | METH\_FASTCALL | METH\_KEYWORDS**

`METH_FASTCALL | METH_KEYWORDS` 的扩展支持定义式类，也就是包含相应方法的类。定义式类可以是 `Py_TYPE(self)` 的超类。

该方法必须为 `PyCMethod` 类型，与在 `self` 之后添加了 `defining_class` 参数的 `METH_FASTCALL | METH_KEYWORDS` 一样。

在 3.9 版新加入。

#### **METH\_NOARGS**

如果通过 `METH_NOARGS` 旗标列出了参数则没有形参的方法无需检查是否给出了参数。它们必须为 `PyCFunction` 类型。第一个形参通常被命名为 `self` 并将持有对模块或对象实例的引用。在所有情况下第二个形参都将是 `NULL`。

该函数必须有 2 个形参。由于第二个形参不会被使用，`Py_UNUSED` 可以被用来防止编译器警告。

#### **METH\_O**

具有一个单独对象参数的方法可使用 `METH_O` 旗标列出，而不必发起调用 `PyArg_ParseTuple()` 并附带 "O" 参数。它们的类型为 `PyCFunction`，带有 `self` 形参，以及代表该单独参数的 `PyObject*` 形参。

这两个常量不是被用来指明调用惯例而是在配合类方法使用时指明绑定。它们不会被用于在模块上定义的函数。对于任何给定方法这些旗标最多只会设置其中一个。

#### METH\_CLASS

该方法将接受类型对象而不是类型的实例作为第一个形参。它会被用于创建 类方法，类似于使用 `classmethod()` 内置函数所创建的结果。

#### METH\_STATIC

该方法将接受 `NULL` 而不是类型的实例作为第一个形参。它会被用于创建 静态方法，类似于使用 `staticmethod()` 内置函数所创建的结果。

另一个常量控制方法是否将被载入来替代具有相同方法名的另一个定义。

#### METH\_COEXIST

该方法将被加载以替代现有的定义。如果没有 `METH_COEXIST`，默认将跳过重复的定义。由于槽位包装器会在方法表之前被加载，例如当存在 `sq_contains` 槽位时，将会生成一个名为 `__contains__()` 的已包装方法并阻止加载同名的相应 `PyCFunction`。如果定义了此旗标，`PyCFunction` 将被加载以替代此包装器对象并与槽位共存。因为对 `PyCFunction` 的调用相比对包装器对象调用更为优化所以这是很有帮助的。

*PyObject* \***PyMethod\_New** (*PyMethodDef* \*ml, *PyObject* \*self, *PyObject* \*module, *PyTypeObject* \*cls)

回傳值：新的參照。属于稳定 ABI 自 3.9 版起，将 `ml` 转为一个 Python *callable* 对象。调用方必须确保 `ml` 的生命期长于 *callable*。通常，`ml` 会被定义为一个静态变量。

*self* 形参将在发起调用时作为 `ml->ml_meth` 中 C 函数的 *self* 参数传入。*self* 可以为 `NULL`。

*callable* 对象的 `__module__` 属性可以根据给定的 *module* 参数来设置。*module* 应为一个 Python 字符串，它将被用作函数定义所在的模块名称。如果不可用，它将被设为 `None` 或 `NULL`。

也参考：

`function.__module__`

*cls* 形参将被作为 C 函数的 *defining\_class* 参数传入。如果在 `ml->ml_flags` 上设置了 `METH_METHOD` 则必须设置该形参。

在 3.9 版新加入。

*PyObject* \***PyCFunction\_NewEx** (*PyMethodDef* \*ml, *PyObject* \*self, *PyObject* \*module)

回傳值：新的參照。属于稳定 ABI。等价于 `PyMethod_New(ml, self, module, NULL)`。

*PyObject* \***PyCFunction\_New** (*PyMethodDef* \*ml, *PyObject* \*self)

回傳值：新的參照。属于稳定 ABI 自 3.4 版起。等价于 `PyMethod_New(ml, self, NULL, NULL)`。

## 12.2.3 访问扩展类型的属性

type **PyMemberDef**

属于稳定 ABI（包括所有成员）。描述与某个 C 结构体成员相对应的类型的属性的结构体。它的字段有：

域	C Type	含意
<code>name</code>	<code>const char *</code>	成员名称
<code>type</code>	<code>int</code>	C 结构体中成员的类型
<code>offset</code>	<code>Py_ssize_t</code>	成员在类型的对象结构体中所在位置的以字节表示的偏移量
<code>flags</code>	<code>int</code>	指明字段是否应为只读或可写的旗标位
<code>doc</code>	<code>const char *</code>	指向文档字符串的内容

`type` 可以是与各种 C 类型相对应的许多 `T_` 宏中的一个。当在 Python 中访问该成员时，它将被转换为等价的 Python 类型。



宏名称	C 类型
T_SHORT	short
T_INT	int
T_LONG	long
T_FLOAT	float
T_DOUBLE	double
T_STRING	const char *
T_OBJECT	PyObject *
T_OBJECT_EX	PyObject *
T_CHAR	char
T_BYTE	char
T_UBYTE	unsigned char
T_UINT	unsigned int
T_USHORT	unsigned short
T_ULONG	unsigned long
T_BOOL	char
T_LONGLONG	long long
T_ULONGLONG	unsigned long long
T_PYSSIZET	Py_ssize_t

T\_OBJECT 和 T\_OBJECT\_EX 的区别在于 T\_OBJECT 返回 None 表示其成员为 NULL 并且 T\_OBJECT\_EX 引发了 AttributeError。请尝试使用 T\_OBJECT\_EX 取代 T\_OBJECT 因为 T\_OBJECT\_EX 处理在属性上使用 del 语句比 T\_OBJECT 更正确。

flags 可以为 0 表示读写访问或 READONLY 表示只读访问。使用 T\_STRING 作为 type 表示 READONLY。T\_STRING 数据将被解读为 UTF-8 编码格式。只有 T\_OBJECT 和 T\_OBJECT\_EX 成员可以被删除。(它们会被设为 NULL)。

堆分配类型 (使用 `PyType_FromSpec()` 或类似函数创建), PyMemberDef 可以包含特殊成员 `__dictoffset__`, `__weaklistoffset__` 和 `__vectorcalloffset__` 的定义, 对应类型对象中的 `tp_dictoffset`, `tp_weaklistoffset` 和 `tp_vectorcall_offset`。它们必须使用 T\_PYSSIZET 和 READONLY 来定义, 例如:

```
static PyMemberDef spam_type_members[] = {
    {"__dictoffset__", T_PYSSIZET, offsetof(Spam_object, dict), READONLY},
    {NULL} /* Sentinel */
};
```

`PyObject*PyMember_GetOne` (const char \*obj\_addr, struct `PyMemberDef` \*m)

获取属于地址 `obj_addr` 上的对象的某个属性。该属性是以 `PyMemberDef m` 来描述的。出错时返回 NULL。

int `PyMember_SetOne` (char \*obj\_addr, struct `PyMemberDef` \*m, `PyObject` \*o)

将属于位于地址 `obj_addr` 的对象的属性设置到对象 `o`。要设置的属性由 `PyMemberDef m` 描述。成功时返回 0 而失败时返回负值。

type `PyGetSetDef`

属于稳定 ABI (包括所有成员)。用于定义针对某个类型的特征属性式的访问的结构体。另请参阅 `PyTypeObject.tp_getset` 槽位的描述。

域	C Type	含意
name	const char *	属性名称
get	getter	用于获取属性的 C 函数
set	setter	用于设置或删除属性的可选 C 函数, 如果省略则属性将为只读
doc	const char *	可选的文档字符串
closure	void *	optional user data pointer, providing additional data for getter and setter

The `get` function takes one `PyObject*` parameter (the instance) and a user data pointer (the associated closure):

```
typedef PyObject *(*getter)(PyObject *, void *);
```

它应当在成功时返回一个新的引用或在失败时返回 `NULL` 并设置异常。

`set` functions take two `PyObject*` parameters (the instance and the value to be set) and a user data pointer (the associated closure):

```
typedef int (*setter)(PyObject *, PyObject *, void *);
```

对于属性要被删除的情况第二个形参应为 `NULL`。成功时应返回 `0` 或在失败时返回 `-1` 并设置异常。

12.3 型物件

Python 对象系统中最重要的一個结构体也许是定义新类型的结构体: `PyTypeObject` 结构体。类型对象可以使用任何 `PyObject_*` 或 `PyType_*` 函数来处理, 但并未提供大多数 Python 应用程序会感兴趣的东西。这些对象是对象行为的基础, 所以它们对解释器本身及任何实现新类型的扩展模块都非常重要。

与大多数标准类型相比, 类型对象相当大。这么大的原因是每个类型对象存储了大量的值, 大部分是 C 函数指针, 每个指针实现了类型功能的一小部分。本节将详细描述类型对象的字段。这些字段将按照它们在结构中出现的顺序进行描述。

除了下面的快速参考, 範例 小节提供了快速了解 `PyTypeObject` 的含义和用法的例子。

12.3.1 快速参考

”tp 槽位”

PyTypeObject 槽位 <small>Page 227, 1</small>	类型	特殊方法/属性	信息 <small>Page 227, 2</small>			
			C	T	D	I
<R> <code>tp_name</code>	<code>const char *</code>	<code>__name__</code>	X	X		
<code>tp_basicsize</code>	<code>Py_ssize_t</code>		X	X		X
<code>tp_itemsize</code>	<code>Py_ssize_t</code>			X		X
<code>tp_dealloc</code>	<code>destructor</code>		X	X		X
<code>tp_vectorcall_offset</code>	<code>Py_ssize_t</code>			X		X
<code>(tp_getattr)</code>	<code>getattrfunc</code>	<code>__getattribute__, __getattr__</code>				G
<code>(tp_setattr)</code>	<code>setattrfunc</code>	<code>__setattr__, __delattr__</code>				G
<code>tp_as_async</code>	<code>PyAsyncMethods *</code>	子槽位				%
<code>tp_repr</code>	<code>reprfunc</code>	<code>__repr__</code>	X	X		X
<code>tp_as_number</code>	<code>PyNumberMethods *</code>	子槽位				%
<code>tp_as_sequence</code>	<code>PySequenceMethods *</code>	子槽位				%
<code>tp_as_mapping</code>	<code>PyMappingMethods *</code>	子槽位				%
<code>tp_hash</code>	<code>hashfunc</code>	<code>__hash__</code>	X			G
<code>tp_call</code>	<code>ternaryfunc</code>	<code>__call__</code>		X		X
<code>tp_str</code>	<code>reprfunc</code>	<code>__str__</code>	X			X
<code>tp_getattro</code>	<code>getattrofunc</code>	<code>__getattribute__, __getattr__</code>	X	X		G
<code>tp_setattro</code>	<code>setattrofunc</code>	<code>__setattr__, __delattr__</code>	X	X		G
<code>tp_as_buffer</code>	<code>PyBufferProcs *</code>					%
<code>tp_flags</code>	<code>unsigned long</code>		X	X		?
<code>tp_doc</code>	<code>const char *</code>	<code>__doc__</code>	X	X		
<code>tp_traverse</code>	<code>traverseproc</code>			X		G
<code>tp_clear</code>	<code>inquiry</code>			X		G

繼續下一頁

表格 1 – 繼續上一頁

PyTypeObject 槽位 <sup>Page 227, 1</sup>	类型	特殊方法/属性	信息 <sup>2</sup>				
			C	T	D	I	
<code>tp_richcompare</code>	<code>richcmpfunc</code>	<code>__lt__</code> , <code>__le__</code> , <code>__eq__</code> , <code>__ne__</code> , <code>__gt__</code> , <code>__ge__</code>	X				G
<code>tp_weaklistoffset</code>	<code>Py_ssize_t</code>			X			?
<code>tp_iter</code>	<code>getiterfunc</code>	<code>__iter__</code>					X
<code>tp_iternext</code>	<code>iternextfunc</code>	<code>__next__</code>					X
<code>tp_methods</code>	<code>PyMethodDef []</code>		X	X			
<code>tp_members</code>	<code>PyMemberDef []</code>			X			
<code>tp_getset</code>	<code>PyGetSetDef []</code>		X	X			
<code>tp_base</code>	<code>PyTypeObject *</code>	<code>__base__</code>				X	
<code>tp_dict</code>	<code>PyObject *</code>	<code>__dict__</code>				?	
<code>tp_descr_get</code>	<code>descrgetfunc</code>	<code>__get__</code>					X
<code>tp_descr_set</code>	<code>descrsetfunc</code>	<code>__set__</code> , <code>__delete__</code>					X
<code>tp_dictoffset</code>	<code>Py_ssize_t</code>			X			?
<code>tp_init</code>	<code>initproc</code>	<code>__init__</code>	X	X			X
<code>tp_alloc</code>	<code>allocfunc</code>		X		?	?	
<code>tp_new</code>	<code>newfunc</code>	<code>__new__</code>	X	X	?	?	
<code>tp_free</code>	<code>freefunc</code>		X	X	?	?	
<code>tp_is_gc</code>	<code>inquiry</code>			X			X
<code>&lt;tp_bases&gt;</code>	<code>PyObject *</code>	<code>__bases__</code>				~	
<code>&lt;tp_mro&gt;</code>	<code>PyObject *</code>	<code>__mro__</code>				~	
<code>[tp_cache]</code>	<code>PyObject *</code>						
<code>[tp_subclasses]</code>	<code>PyObject *</code>	<code>__subclasses__</code>					
<code>[tp_weaklist]</code>	<code>PyObject *</code>						
<code>(tp_del)</code>	<code>destructor</code>						
<code>[tp_version_tag]</code>	<code>unsigned int</code>						
<code>tp_finalize</code>	<code>destructor</code>	<code>__del__</code>					X
<code>tp_vectorcall</code>	<code>vectorcallfunc</code>						

<sup>1</sup> ()：括号中的插槽名称表示（实际上）已弃用。

<>：尖括号内的名称在初始时应设为 NULL 并被视为是只读的。

[]：方括号内的名称仅供内部使用。

<R>（作为前缀）表示字段是必需的（不能是 NULL）。

<sup>2</sup> 列：

"O"：在 `PyBaseObject_Type` 上设置

"T"：在 `PyType_Type` 上设置

"D"：默认设置（如果方法槽被设置为 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"：继承

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

注意，有些方法槽是通过普通属性查找链有效继承的。

## 子槽位

槽位	类型	特殊方法
<code>am_await</code>	<code>unaryfunc</code>	<code>__await__</code>
<code>am_aiter</code>	<code>unaryfunc</code>	<code>__aiter__</code>
<code>am_anext</code>	<code>unaryfunc</code>	<code>__anext__</code>
<code>am_send</code>	<code>sendfunc</code>	
<code>nb_add</code>	<code>binaryfunc</code>	<code>__add__</code> <code>__radd__</code>
<code>nb_inplace_add</code>	<code>binaryfunc</code>	<code>__iadd__</code>
<code>nb_subtract</code>	<code>binaryfunc</code>	<code>__sub__</code> <code>__rsub__</code>
<code>nb_inplace_subtract</code>	<code>binaryfunc</code>	<code>__isub__</code>
<code>nb_multiply</code>	<code>binaryfunc</code>	<code>__mul__</code> <code>__rmul__</code>
<code>nb_inplace_multiply</code>	<code>binaryfunc</code>	<code>__imul__</code>
<code>nb_remainder</code>	<code>binaryfunc</code>	<code>__mod__</code> <code>__rmod__</code>
<code>nb_inplace_remainder</code>	<code>binaryfunc</code>	<code>__imod__</code>
<code>nb_divmod</code>	<code>binaryfunc</code>	<code>__divmod__</code> <code>__rdivmod__</code>
<code>nb_power</code>	<code>ternaryfunc</code>	<code>__pow__</code> <code>__rpow__</code>
<code>nb_inplace_power</code>	<code>ternaryfunc</code>	<code>__ipow__</code>
<code>nb_negative</code>	<code>unaryfunc</code>	<code>__neg__</code>
<code>nb_positive</code>	<code>unaryfunc</code>	<code>__pos__</code>
<code>nb_absolute</code>	<code>unaryfunc</code>	<code>__abs__</code>
<code>nb_bool</code>	<code>inquiry</code>	<code>__bool__</code>
<code>nb_invert</code>	<code>unaryfunc</code>	<code>__invert__</code>
<code>nb_lshift</code>	<code>binaryfunc</code>	<code>__lshift__</code> <code>__rlshift__</code>
<code>nb_inplace_lshift</code>	<code>binaryfunc</code>	<code>__ilshift__</code>
<code>nb_rshift</code>	<code>binaryfunc</code>	<code>__rshift__</code> <code>__rrshift__</code>
<code>nb_inplace_rshift</code>	<code>binaryfunc</code>	<code>__irshift__</code>
<code>nb_and</code>	<code>binaryfunc</code>	<code>__and__</code> <code>__rand__</code>
<code>nb_inplace_and</code>	<code>binaryfunc</code>	<code>__iand__</code>
<code>nb_xor</code>	<code>binaryfunc</code>	<code>__xor__</code> <code>__rxor__</code>
<code>nb_inplace_xor</code>	<code>binaryfunc</code>	<code>__ixor__</code>
<code>nb_or</code>	<code>binaryfunc</code>	<code>__or__</code> <code>__ror__</code>
<code>nb_inplace_or</code>	<code>binaryfunc</code>	<code>__ior__</code>
<code>nb_int</code>	<code>unaryfunc</code>	<code>__int__</code>
<code>nb_reserved</code>	<code>void *</code>	
<code>nb_float</code>	<code>unaryfunc</code>	<code>__float__</code>
<code>nb_floor_divide</code>	<code>binaryfunc</code>	<code>__floordiv__</code>
<code>nb_inplace_floor_divide</code>	<code>binaryfunc</code>	<code>__ifloordiv__</code>
<code>nb_true_divide</code>	<code>binaryfunc</code>	<code>__truediv__</code>
<code>nb_inplace_true_divide</code>	<code>binaryfunc</code>	<code>__itruediv__</code>
<code>nb_index</code>	<code>unaryfunc</code>	<code>__index__</code>
<code>nb_matrix_multiply</code>	<code>binaryfunc</code>	<code>__matmul__</code> <code>__rmatmul__</code>
<code>nb_inplace_matrix_multiply</code>	<code>binaryfunc</code>	<code>__imatmul__</code>
<code>mp_length</code>	<code>lenfunc</code>	<code>__len__</code>
<code>mp_subscript</code>	<code>binaryfunc</code>	<code>__getitem__</code>
<code>mp_ass_subscript</code>	<code>objobjargproc</code>	<code>__setitem__</code> , <code>__delitem__</code>
<code>sq_length</code>	<code>lenfunc</code>	<code>__len__</code>
<code>sq_concat</code>	<code>binaryfunc</code>	<code>__add__</code>
<code>sq_repeat</code>	<code>ssizeargfunc</code>	<code>__mul__</code>

繼續下一頁

表格 2 - 繼續上一頁

槽位	类型	特殊方法
<code>sq_item</code>	<code>ssizeargfunc</code>	<code>__getitem__</code>
<code>sq_ass_item</code>	<code>ssizeobjargproc</code>	<code>__setitem__</code> <code>__delitem__</code>
<code>sq_contains</code>	<code>objobjproc</code>	<code>__contains__</code>
<code>sq_inplace_concat</code>	<code>binaryfunc</code>	<code>__iadd__</code>
<code>sq_inplace_repeat</code>	<code>ssizeargfunc</code>	<code>__imul__</code>
<code>bf_getbuffer</code>	<code>getbufferproc()</code>	
<code>bf_releasebuffer</code>	<code>releasebufferproc()</code>	

槽位 typedef

typedef	参数类型	返回类型
<i>allocfunc</i>	<i>PyTypeObject</i> * <i>Py_ssize_t</i>	<i>PyObject</i> *
<i>destructor</i>	<i>PyObject</i> *	void
<i>freefunc</i>	void *	void
<i>traverseproc</i>	<i>PyObject</i> * <i>visitproc</i> void *	int
<i>newfunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
<i>initproc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	int
<i>reprfunc</i>	<i>PyObject</i> *	<i>PyObject</i> *
<i>getattrfunc</i>	<i>PyObject</i> * const char *	<i>PyObject</i> *
<i>setattrfunc</i>	<i>PyObject</i> * const char * <i>PyObject</i> *	int
<i>getattrofunc</i>	<i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
<i>setattrofunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	int
<i>descrgetfunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
<i>descrsetfunc</i>	<i>PyObject</i> *	int
230	<i>PyObject</i> * <i>PyObject</i> *	Chapter 12. 对象实现支持
<i>hashfunc</i>	<i>PyObject</i> *	Py_hash_t
<i>richcmpfunc</i>		<i>PyObject</i> *



更多細節請見下方的槽位类型 *typedef*。

### 12.3.2 PyObject 定义

*PyObject* 的结构定义可以在 `Include/object.h` 中找到。为了方便参考，此处复述了其中的定义：

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrofunc tp_getattr;
    setattrofunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                   or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

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

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* 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)

```

/* Attribute descriptor and subclassing stuff */
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
// Strong reference on a heap type, borrowed reference on a static type
struct _typeobject *tp_base;
PyObject *tp_dict;
descrgetfunc tp_descr_get;
descrsetfunc tp_descr_set;
Py_ssize_t tp_dictoffset;
initproc tp_init;
allocfunc tp_alloc;
newfunc tp_new;
freefunc tp_free; /* Low-level free-memory routine */
inquiry tp_is_gc; /* For PyObject_IS_GC */
PyObject *tp_bases;
PyObject *tp_mro; /* method resolution order */
PyObject *tp_cache;
PyObject *tp_subclasses;
PyObject *tp_weaklist;
destructor tp_del;

/* Type attribute cache version tag. Added in version 2.6 */
unsigned int tp_version_tag;

destructor tp_finalize;
vectorcallfunc tp_vectorcall;
} PyTypeObject;

```

### 12.3.3 PyObject 槽位

类型对象结构体扩展了 `PyVarObject` 结构体。`ob_size` 字段用于动态类型（由 `type_new()` 创建，通常由 `class` 语句调用）。请注意 `PyType_Type`（元类型）会初始化 `tp_itemsize`，这意味着它的实例（即类型对象）必须具有 `ob_size` 字段。

#### `Py_ssize_t PyObject.ob_refcnt`

属于稳定 ABI。这是类型对象的引用计数，由 `PyObject_HEAD_INIT` 宏初始化为 1。请注意对于静态分配的类型对象，类型的实例（其 `ob_type` 指向该类型的对象）不会被计入引用。但对于动态分配的类型对象，实例会被计入引用。

**继承：**

子类型不继承此字段。

#### `PyTypeObject *PyObject.ob_type`

属于稳定 ABI。这是类型的类型，换句话说就是元类型，它由宏 `PyObject_HEAD_INIT` 的参数来做初始化，它的值一般情况下是 `&PyType_Type`。可是为了使动态可载入扩展模块至少在 Windows 上可用，编译器会报错这是一个不可用的初始化。因此按照惯例传递 `NULL` 给宏 `PyObject_HEAD_INIT` 并且在模块的初始化函数开始时候其他任何操作之前初始化这个字段。典型做法是这样的：

```
Foo_Type.ob_type = &PyType_Type;
```

这应当在创建类型的任何实例之前完成。`PyType_Ready()` 会检查 `ob_type` 是否为 `NULL`，如果是，则将其初始化为基类的 `ob_type` 字段。如果该字段为非零值则 `PyType_Ready()` 将不会更改它。

**继承：**

此字段会被子类型继承。

`PyObject *PyObject._ob_next`

`PyObject *PyObject._ob_prev`

这些字段仅在定义了宏 `Py_TRACE_REFS` 时存在（参阅 `configure --with-trace-refs option`）。

由 `PyObject_HEAD_INIT` 宏负责将它们初始化为 `NULL`。对于静态分配的对象，这两个字段始终为 `NULL`。对于动态分配的对象，这两个字段用于将对象链接到堆上所有活动对象的双向链表中。

它们可用于各种调试目的。目前唯一的用途是 `sys.getobjects()` 函数，在设置了环境变量 `PYTHONDUMPREFS` 时，打印运行结束时仍然活跃的对象。

**继承：**

这些字段不会被子类型继承。

### 12.3.4 PyVarObject 槽位

`Py_ssize_t PyVarObject.ob_size`

属于稳定 ABI。对于静态分配的内存对象，它应该初始化为 0。对于动态分配的类型对象，该字段具有特殊的内部含义。

**继承：**

子类型不继承此字段。

### 12.3.5 PyTypeObject 槽

每个槽位都有一个小节来描述继承关系。如果 `PyType_Ready()` 可以在字段被设为 `NULL` 时设置一个值那么还会有一个“默认”小节。（请注意在 `PyBaseObject_Type` 和 `PyType_Type` 上设置的许多字段实际上就是默认值。）

`const char *PyTypeObject.tp_name`

指向包含类型名称的以 `NUL` 结尾的字符串的指针。对于可作为模块全局访问的类型，该字符串应为模块全名，后面跟一个点号，然后再加类型名称；对于内置类型，它应当只是类型名称。如果模块是包的子模块，则包的全名将是模块的全名的一部分。例如，在包 `P` 的子包 `Q` 中的模块 `M` 中定义的名称为 `T` 的类型应当具有 `tp_name` 初始化器 `"P.Q.M.T"`。

对于动态分配的类型对象，这应为类型名称，而模块名称将作为 `'__module__'` 键的值显式地保存在类型字典中。

对于静态分配的类型对象，`tp_name` 字段应当包含一个点号。最后一个点号之前的所有内容都可作为 `__module__` 属性访问，而最后一个点号之后的所有内容都可作为 `__name__` 属性访问。

如果不存在点号，则整个 `tp_name` 字段将作为 `__name__` 属性访问，而 `__module__` 属性则将是未定义的（除非在字典中显式地设置，如上文所述）。这意味着你的类型将无法执行 `pickle`。此外，用 `pydoc` 创建的模块文档中也不会列出该类型。

该字段不可为 `NULL`。它是 `PyTypeObject()` 中唯一的必填字段（除了潜在的 `tp_itemsize` 以外）。

**继承：**

子类型不继承此字段。

`Py_ssize_t PyTypeObject.tp_basicsize`

`Py_ssize_t PyTypeObject.tp_itemsize`

通过这些字段可以计算出该类型实例以字节为单位的大小。

存在两种类型：具有固定长度实例的类型其 `tp_itemsize` 字段为零；具有可变长度实例的类型其 `tp_itemsize` 字段不为零。对于具有固定长度实例的类型，所有实例的大小都相同，具体大小由 `tp_basicsize` 给出。

对于具有可变长度实例的类型，实例必须有一个 `ob_size` 字段，实例大小为 `tp_basicsize` 加上 `N` 乘以 `tp_itemsize`，其中 `N` 是对象的“长度”。`N` 的值通常存储在实例的 `ob_size` 字段中。但也有例外：举例来说，整数类型使用负的 `ob_size` 来表示负数，`N` 在这里就是 `abs(ob_size)`。此外，在实例布局中存在 `ob_size` 字段并不意味着实例结构是可变长度的（例如，列表类型的结构体有固定长度的实例，但这些实例却包含一个有意义的 `ob_size` 字段）。

基本大小包括由宏 `PyObject_HEAD` 或 `PyObject_VAR_HEAD`（以用于声明实例结构的宏为准）声明的实例中的字段，如果存在 `_ob_prev` 和 `_ob_next` 字段则将相应地包括这些字段。这意味着为 `tp_basicsize` 获取初始化器的唯一正确方式是在用于声明实例布局的结构上使用 `sizeof` 操作符。基本大小不包括 GC 标头的大小。

关于对齐的说明：如果变量条目需要特定的对齐，则应通过 `tp_basicsize` 的值来处理。例如：假设某个类型实现了一个 `double` 数组。`tp_itemsize` 就是 `sizeof(double)`。程序员有责任确保 `tp_basicsize` 是 `sizeof(double)` 的倍数（假设这是 `double` 的对齐要求）。

对于任何具有可变长度实例的类型，该字段不可为 `NULL`。

#### 继承：

这些字段将由子类分别继承。如果基本类型有一个非零的 `tp_itemsize`，那么在子类型中将 `tp_itemsize` 设置为不同的非零值通常是不安全的（不过这取决于该基本类型的具体实现）。

#### destructor `PyTypeObject.tp_dealloc`

指向实例析构函数的指针。除非保证类型的实例永远不会被释放（就像单例对象 `None` 和 `Ellipsis` 那样），否则必须定义这个函数。函数声明如下：

```
void tp_dealloc(PyObject *self);
```

当新引用计数为零时，`Py_DECREF()` 和 `Py_XDECREF()` 宏会调用析构函数。此时，实例仍然存在，但已没有了对它的引用。析构函数应当释放该实例拥有的所有引用，释放实例拥有的所有内存缓冲区（使用与分配缓冲区时所用分配函数相对应的释放函数），并调用类型的 `tp_free` 函数。如果该类型不可子类型化（未设置 `Py_TPFLAGS_BASETYPE` 旗标位），则允许直接调用对象的释放器而不必通过 `tp_free`。对象的释放器应为分配实例时所使用的释放器；如果实例是使用 `PyObject_New` 或 `PyObject_NewVar` 分配的，则释放器通常为 `PyObject_Del()`；如果实例是使用 `PyObject_GC_New` 或 `PyObject_GC_NewVar` 分配的，则释放器通常为 `PyObject_GC_Del()`。

如果该类型支持垃圾回收（设置了 `Py_TPFLAGS_HAVE_GC` 旗标位），则析构器应在清除任何成员字段之前调用 `PyObject_GC_UnTrack()`。

```
static void foo_dealloc(foo_object *self) {
    PyObject_GC_UnTrack(self);
    Py_CLEAR(self->ref);
    Py_TYPE(self)->tp_free((PyObject *)self);
}
```

最后，如果该类型是堆分配的（`Py_TPFLAGS_HEAPTYPE`），则在调用类型释放器后，释放器应释放对其类型对象的所有引用（通过 `Py_DECREF()`）。为了避免悬空指针，建议的实现方式如下：

```
static void foo_dealloc(foo_object *self) {
    PyTypeObject *tp = Py_TYPE(self);
    // free references and buffers here
    tp->tp_free(self);
    Py_DECREF(tp);
}
```

#### 继承：

此字段会被子类型继承。

#### `Py_ssize_t PyTypeObject.tp_vectorcall_offset`

一个相对使用 `vectorcall` 协议实现调用对象的实例级函数的可选的偏移量，这是一种比简单的 `tp_call` 更有效的替代选择。

该字段仅在设置了 `Py_TPFLAGS_HAVE_VECTORCALL` 旗标时使用。在此情况下，它必须为一个包含 `vectorcallfunc` 指针实例中的偏移量的正整数。

`vectorcallfunc` 指针可能为 `NULL`，在这种情况下实例的行为就像 `Py_TPFLAGS_HAVE_VECTORCALL` 没有被设置一样：调用实例操作会回退至 `tp_call`。

任何设置了 `Py_TPFLAGS_HAVE_VECTORCALL` 的类也必须设置 `tp_call` 并确保其行为与 `vectorcallfunc` 函数一致。这可以通过将 `tp_call` 设为 `PyVectorcall_Call()` 来实现。

**警告：** It is 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.

在 3.8 版的變更：在 3.8 版之前，这个槽位被命名为 `tp_print`。在 Python 2.x 中，它被用于打印到文件。在 Python 3.0 至 3.7 中，它没有被使用。

#### 继承：

This field is always inherited. However, the `Py_TPFLAGS_HAVE_VECTORCALL` flag is not always inherited. If it's not, then the subclass won't use `vectorcall`, except when `PyVectorcall_Call()` is explicitly called. This is in particular the case for types without the `Py_TPFLAGS_IMMUTABLETYPE` flag set (including subclasses defined in Python).

#### `getattrfunc PyObject.tp_getattr`

一个指向获取属性字符串函数的可选指针。

该字段已弃用。当它被定义时，应该和 `tp_getattro` 指向同一个函数，但接受一个 C 字符串参数表示属性名，而不是 Python 字符串对象。

#### 继承：

分组： `tp_getattr`, `tp_getattro`

该字段会被子类型和 `tp_getattro` 所继承：当子类型的 `tp_getattr` 和 `tp_getattro` 均为 `NULL` 时该子类型将从它的基类型同时继承 `tp_getattr` 和 `tp_getattro`。

#### `setattrfunc PyObject.tp_setattr`

一个指向函数以便设置和删除属性的可选指针。

该字段已弃用。当它被定义时，应该和 `tp_setattro` 指向同一个函数，但接受一个 C 字符串参数表示属性名，而不是 Python 字符串对象。

#### 继承：

分组： `tp_setattr`, `tp_setattro`

该字段会被子类型和 `tp_setattro` 所继承：当子类型的 `tp_setattr` 和 `tp_setattro` 均为 `NULL` 时该子类型将同时从它的基类型继承 `tp_setattr` 和 `tp_setattro`。

#### `PyAsyncMethods *PyObject.tp_as_async`

指向一个包含仅与在 C 层级上实现 *awaitable* 和 *asynchronous iterator* 协议的对象相关联的字段的附加结构体。请参阅 [异步对象结构体](#) 了解详情。

在 3.5 版新加入：在之前被称为 `tp_compare` 和 `tp_reserved`。

#### 继承：

`tp_as_async` 字段不会被继承，但所包含的字段会被单独继承。

#### `reprfunc PyObject.tp_repr`

一个实现了内置函数 `repr()` 的函数的可选指针。

该签名与 `PyObject_Repr()` 的相同：

```
PyObject *tp_repr(PyObject *self);
```

该函数必须返回一个字符串或 Unicode 对象。在理想情况下，该函数应当返回一个字符串，当将其传给 `eval()` 时，只要有合适的环境，就会返回一个具有相同值的对象。如果这不可行，则它应当返回一个以 `'<'` 开头并以 `'>'` 结尾的可被用来推断出对象的类型和值的字符串。

#### 继承:

此字段会被子类型继承。

#### 预设:

如果未设置该字段，则返回 `<%s object at %p>` 形式的字符串，其中 `%s` 将替换为类型名称，`%p` 将替换为对象的内存地址。

#### *PyNumberMethods* \**PyTypeObject*.**tp\_as\_number**

指向一个附加结构体的指针，其中包含只与执行数字协议的对象相关的字段。这些字段的文档参见数字对象结构体。

#### 继承:

`tp_as_number` 字段不会被继承，但所包含的字段会被单独继承。

#### *PySequenceMethods* \**PyTypeObject*.**tp\_as\_sequence**

指向一个附加结构体的指针，其中包含只与执行序列协议的对象相关的字段。这些字段的文档见序列对象结构体。

#### 继承:

`tp_as_sequence` 字段不会被继承，但所包含的字段会被单独继承。

#### *PyMappingMethods* \**PyTypeObject*.**tp\_as\_mapping**

指向一个附加结构体的指针，其中包含只与执行映射协议的对象相关的字段。这些字段的文档见映射对象结构体。

#### 继承:

`tp_as_mapping` 字段不会继承，但所包含的字段会被单独继承。

#### *hashfunc* *PyTypeObject*.**tp\_hash**

一个指向实现了内置函数 `hash()` 的函数的可选指针。

其签名与 `PyObject_Hash()` 的相同:

```
Py_hash_t tp_hash(PyObject *);
```

-1 不应作为正常返回值被返回；当计算哈希值过程中发生错误时，函数应设置一个异常并返回 -1。

当该字段（和 `tp_richcompare`）都未设置，尝试对该对象取哈希会引发 `TypeError`。这与将其设为 `PyObject_HashNotImplemented()` 相同。

此字段可被显式设为 `PyObject_HashNotImplemented()` 以阻止从父类型继承哈希方法。在 Python 层面这被解释为 `__hash__ = None` 的等价物，使得 `isinstance(o, collections.Hashable)` 正确返回 `False`。请注意反过来也是如此：在 Python 层面设置一个类的 `__hash__ = None` 会使得 `tp_hash` 槽位被设置为 `PyObject_HashNotImplemented()`。

#### 继承:

分组: `tp_hash`, `tp_richcompare`

该字段会被子类型同 `tp_richcompare` 一起继承：当子类型的 `tp_richcompare` 和 `tp_hash` 均为 NULL 时子类型将同时继承 `tp_richcompare` 和 `tp_hash`。

#### *ternaryfunc* *PyTypeObject*.**tp\_call**

一个可选的实现对象调用的指向函数的指针。如果对象不是可调用对象则该值应为 NULL。其签名与 `PyObject_Call()` 的相同:

```
PyObject *tp_call(PyObject *self, PyObject *args, PyObject *kwargs);
```



**继承:**

此字段会被子类型继承。

*reprfunc* `PyTypeObject.tp_str`

一个可选的实现内置 `str()` 操作的函数的指针。(请注意 `str` 现在是一个类型, `str()` 是调用该类型的构造器。该构造器将调用 `PyObject_Str()` 执行实际操作, 而 `PyObject_Str()` 将调用该处理器。)

其签名与 `PyObject_Str()` 的相同:

```
PyObject *tp_str(PyObject *self);
```

该函数必须返回一个字符串或 Unicode 对象。它应当是一个“友好”的对象字符串表示形式, 因为这就是要在 `print()` 函数中与其他内容一起使用的表示形式。

**继承:**

此字段会被子类型继承。

**预设:**

当未设置该字段时, 将调用 `PyObject_Repr()` 来返回一个字符串表示形式。

*getattrfunc* `PyTypeObject.tp_getattro`

一个指向获取属性字符串函数的可选指针。

其签名与 `PyObject_GetAttr()` 的相同:

```
PyObject *tp_getattro(PyObject *self, PyObject *attr);
```

可以方便地将该字段设为 `PyObject_GenericGetAttr()`, 它实现了查找对象属性的通常方式。

**继承:**

分组: `tp_getattr`, `tp_getattro`

该字段会被子类型同 `tp_getattr` 一起继承: 当子类型的 `tp_getattr` 和 `tp_getattro` 均为 NULL 时子类型将同时继承 `tp_getattr` 和 `tp_getattro`。

**预设:**

`PyBaseObject_Type` 使用 `PyObject_GenericGetAttr()`。

*setattrfunc* `PyTypeObject.tp_setattro`

一个指向函数以便设置和删除属性的可选指针。

其签名与 `PyObject_SetAttr()` 的相同:

```
int tp_setattro(PyObject *self, PyObject *attr, PyObject *value);
```

此外, 还必须支持将 `value` 设为 NULL 来删除属性。通常可以方便地将该字段设为 `PyObject_GenericSetAttr()`, 它实现了设备对象属性的通常方式。

**继承:**

分组: `tp_setattr`, `tp_setattro`

该字段会被子类型同 `tp_setattr` 一起继承: 当子类型的 `tp_setattr` 和 `tp_setattro` 均为 NULL 时子类型将同时继承 `tp_setattr` 和 `tp_setattro`。

**预设:**

`PyBaseObject_Type` 使用 `PyObject_GenericSetAttr()`。

*PyBufferProcs* \**PyTypeObject*.**tp\_as\_buffer**

指向一个包含只与实现缓冲区接口的对象相关的字段的附加结构体的指针。这些字段的文档参见缓冲区对象结构体。

**继承:**

*tp\_as\_buffer* 字段不会被继承，但所包含的字段会被单独继承。

unsigned long *PyTypeObject*.**tp\_flags**

该字段是针对多个旗标的位掩码。某些旗标指明用于特定场景的变化语义；另一些旗标则用于指明类型对象（或通过*tp\_as\_number*, *tp\_as\_sequence*, *tp\_as\_mapping* 和 *tp\_as\_buffer* 引用的扩展结构体）中的特定字段，它们在历史上并不总是有效；如果这样的旗标位是清晰的，则它所保护的类型字段必须不可被访问并且必须被视为具有零或 NULL 值。

**继承:**

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*。

**位掩码:**

目前定义了以下位掩码；可以使用 | 运算符对它们进行 OR 运算以形成 *tp\_flags* 字段的值。宏 *PyType\_HasFeature()* 接受一个类型和一个旗标值 *tp* 和 *f*，并检查 *tp->tp\_flags & f* 是否为非零值。

**Py\_TPFLAGS\_HEAPTYPE**

当类型对象本身在堆上被分配时会设置这个旗标位，例如，使用 *PyType\_FromSpec()* 动态创建的类型。在此情况下，其实例的 *ob\_type* 字段会被视为指向该类型的引用，而类型对象将在一个新实例被创建时执行 INCREf，并在实例被销毁时执行 DECREf（这不会应用于子类型的实例；只有实例的 *ob\_type* 所引用的类型会执行 INCREf 或 DECREf）。

**继承:**

???

**Py\_TPFLAGS\_BASETYPE**

当此类型可被用作另一个类型的基类型时该比特位将被设置。如果该比特位被清除，则此类型将无法被子类型化（类似于 Java 中的“final”类）。

**继承:**

???

**Py\_TPFLAGS\_READY**

当此类型对象通过 *PyType\_Ready()* 被完全实例化时该比特位将被设置。

**继承:**

???

**Py\_TPFLAGS\_READYING**

当 *PyType\_Ready()* 处在初始化此类型对象过程中时该比特位将被设置。

**继承:**

???

**Py\_TPFLAGS\_HAVE\_GC**

当对象支持垃圾回收时会设置这个旗标位。如果设置了这个位，则实例必须使用 `PyObject_GC_New` 来创建并使用 `PyObject_GC_Del()` 来销毁。更多信息参见使对象类型支持循环垃圾回收。这个位还会假定类型对象中存在 GC 相关字段 `tp_traverse` 和 `tp_clear`。

**继承：**

分组: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

`Py_TPFLAGS_HAVE_GC` 旗标位会与 `tp_traverse` 和 `tp_clear` 字段一起被继承，也就是说，如果 `Py_TPFLAGS_HAVE_GC` 旗标位在子类型中被清空并且子类型中的 `tp_traverse` 和 `tp_clear` 字段存在并具有 NULL 值的话。

**Py\_TPFLAGS\_DEFAULT**

这是一个从属于类型对象及其扩展结构体的存在的所有位的位掩码。目前，它包括以下的位：`Py_TPFLAGS_HAVE_STACKLESS_EXTENSION`。

**继承：**

???

**Py\_TPFLAGS\_METHOD\_DESCRIPTOR**

这个位指明对象的行为类似于未绑定方法。

如果为 `type(meth)` 设置了该旗标，那么：

- `meth.__get__(obj, cls)(*args, **kwargs)` (其中 `obj` 不为 `None`) 必须等价于 `meth(obj, *args, **kwargs)`。
- `meth.__get__(None, cls)(*args, **kwargs)` 必须等价于 `meth(*args, **kwargs)`。

此旗标为 `obj.meth()` 这样的典型方法调用启用优化：它将避免为 `obj.meth` 创建临时的“绑定方法”对象。

在 3.8 版新加入。

**继承：**

此旗标绝不会被没有设置 `Py_TPFLAGS_IMMUTABLETYPE` 旗标的类型所继承。对于扩展类型，当 `tp_descr_get` 被继承时它也会被继承。

**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**

这些旗标被 `PyLong_Check()` 等函数用来快速确定一个类型是否为内置类型的子类；这样的专用检测比泛用检测如 `PyObject_IsInstance()` 要更快速。继承自内置类型的自定义类型应当正确地设置其 `tp_flags`，否则与这样的类型进行交互的代码将因所使用的检测种类而出现不同的行为。

**Py\_TPFLAGS\_HAVE\_FINALIZE**

当类型结构体中存在 `tp_finalize` 槽位时会设置这个比特位。

在 3.4 版新加入。

在 3.8 版之後被 用：此旗标已不再是必要的，因为解释器会假定类型结构体中总是存在 `tp_finalize` 槽位。

**Py\_TPFLAGS\_HAVE\_VECTORCALL**

当类实现了 `vectorcall` 协议 时会设置这个比特位。请参阅 `tp_vectorcall_offset` 了解详情。

**继承：**

This bit is inherited for types with the `Py_TPFLAGS_IMMUTABLETYPE` flag set, if `tp_call` is also inherited.

在 3.9 版新加入。

**Py\_TPFLAGS\_IMMUTABLETYPE**

不可变的类型对象会设置这个比特位：类型属性无法被设置或删除。

`PyType_Ready()` 会自动对 静态类型 应用这个旗标。

**继承：**

这个旗标不会被继承。

在 3.10 版新加入。

**Py\_TPFLAGS\_DISALLOW\_INSTANTIATION**

不允许创建此类型的实例：将 `tp_new` 设为 NULL 并且不会在类型字符中创建 `__new__` 键。

这个旗标必须在创建该类型之前设置，而不是在之后。例如，它必须在该类型调用 `PyType_Ready()` 之前被设置。

如果 `tp_base` 为 NULL 或者 `&PyBaseObject_Type` 和 `tp_new` 为 NULL 则该旗标会在 静态类型 上自动设置。

**继承：**

这个旗标不会被继承。但是，子类将不能被实例化，除非它们提供了不为 NULL 的 `tp_new` (这只能通过 C API 实现)。

---

**備：** 要禁止直接实例化一个类但允许实例化其子类 (例如对于 *abstract base class*)，请勿使用此旗标。替代的做法是，让 `tp_new` 只对子类可用。

---

在 3.10 版新加入。

**Py\_TPFLAGS\_MAPPING**

这个比特位指明该类的实例可以在被用作 `match` 代码块的目标时匹配映射模式。它会在注册或子类化 `collections.abc.Mapping` 时自动设置，并在注册 `collections.abc.Sequence` 时取消设置。

---

**備：** `Py_TPFLAGS_MAPPING` 和 `Py_TPFLAGS_SEQUENCE` 是互斥的；同时启用两个旗标将导致报错。

---

**继承：**

这个旗标将被尚未设置 `Py_TPFLAGS_SEQUENCE` 的类型所继承。

**也参考：**

**PEP 634** —— 结构化模式匹配：规范

在 3.10 版新加入。

**Py\_TPFLAGS\_SEQUENCE**

这个比特位指明该类的实例可以在被用作 `match` 代码块的目标时匹配序列模式。它会在注册或子类化 `collections.abc.Sequence` 时自动设置，并在注册 `collections.abc.Mapping` 时取消设置。

**備註：** `Py_TPFLAGS_MAPPING` 和 `Py_TPFLAGS_SEQUENCE` 是互斥的；同时启用两个旗标将导致报错。

**继承：**

这个旗标将被尚未设置 `Py_TPFLAGS_MAPPING` 的类型所继承。

**也参考：**

**PEP 634** —— 结构化模式匹配：规范

在 3.10 版新加入。

`const char *PyTypeObject.tp_doc`

一个可选的指向给出该类型对象的文档字符串的以 NUL 结束的 C 字符串的指针。该指针被暴露为类型和类型实例上的 `__doc__` 属性。

**继承：**

这个字段 不会被子类型继承。

`traverseproc PyTypeObject.tp_traverse`

一个可选的指向针对垃圾回收器的遍历函数的指针。该指针仅会在设置了 `Py_TPFLAGS_HAVE_GC` 旗标位时被使用。函数签名为：

```
int tp_traverse(PyObject *self, visitproc visit, void *arg);
```

有关 Python 垃圾回收方案的更多信息可在[使对象类型支持循环垃圾回收](#)一节中查看。

`tp_traverse` 指针被垃圾回收器用来检测循环引用。`tp_traverse` 函数的典型实现会在实例的每个属于该实例所拥有的 Python 对象的成员上简单地调用 `Py_VISIT()`。例如，以下是来自 `_thread` 扩展模块的函数 `local_traverse()`：

```
static int
local_traverse(localobject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

请注意 `Py_VISIT()` 仅能在可以参加循环引用的成员上被调用。虽然还存在一个 `self->key` 成员，但它只能为 NULL 或 Python 字符串因而不能成为循环引用的一部分。

在另一方面，即使你知道某个成员永远不会成为循环引用的一部分，作为调试的辅助你仍然可能想要访问它因此 `gc` 模块的 `get_referents()` 函数将会包括它。

**警告：** 当实现 `tp_traverse` 时，只有实例所拥有的成员（就是有指向它们的强引用）才必须被访问。举例来说，如果一个对象通过 `tp_weaklist` 槽位支持弱引用，那么支持链表（`tp_weaklist` 所指向的对象）的指针就 **不能被访问** 因为实例并不直接拥有指向自身的弱引用（弱引用列表被用来支持弱引用机制，但实例没有指向其中的元素的强引用，因为即使实例还存在它们也允许被删除）。

请注意 `Py_VISIT()` 要求传给 `local_traverse()` 的 `visit` 和 `arg` 形参具有指定的名称；不要随意命名它们。

堆分配类型的实例会持有一个指向其类型的引用。因此它们的遍历函数必须要么访问 `Py_TYPE(self)`，要么通过调用其他堆分配类型（例如一个堆分配超类）的 `tp_traverse` 将此任务委托出去。如果没有这样做，类型对象可能不会被垃圾回收。

在 3.9 版的變更: 堆分配类型应当访问 `tp_traverse` 中的 `Py_TYPE(self)`。在较早的 Python 版本中，由于 [bug 40217](#)，这样做可能会导致在超类中发生崩溃。

**继承:**

分组: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

该字段会与 `tp_clear` 和 `Py_TPFLAGS_HAVE_GC` 旗标位一起被子类型所继承：如果旗标位，`tp_traverse` 和 `tp_clear` 在子类型中均为零则它们都将从基类型继承。

*inquiry* `PyTypeObject.tp_clear`

一个可选的指向针对垃圾回收器的清理函数的指针。该指针仅会在设置了 `Py_TPFLAGS_HAVE_GC` 旗标位时被使用。函数签名为:

```
int tp_clear(PyObject *);
```

`tp_clear` 成员函数被用来打破垃圾回收器在循环垃圾中检测到的循环引用。总的来说，系统中的所有 `tp_clear` 函数必须合到一起以打破所有引用循环。这是个微妙的问题，并且如有任何疑问都需要提供 `tp_clear` 函数。例如，元组类型不会实现 `tp_clear` 函数，因为有可能证明完全用元组是不会构成循环引用的。因此其他类型的 `tp_clear` 函数必须足以打破任何包含元组的循环。这不是立即能明确的，并且很少会有避免实现 `tp_clear` 的适当理由。

`tp_clear` 的实现应当丢弃实例指向其成员的可能为 Python 对象的引用，并将指向这些成员的指针设为 NULL，如下面的例子所示:

```
static int
local_clear(localobject *self)
{
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

应当使用 `Py_CLEAR()` 宏，因为清除引用是很微妙的：指向被包含对象的引用必须在指向被包含对象的指针被设为 NULL 之后才能被释放（通过 `Py_DECREF()`）。这是因为释放引用可能会导致被包含的对象变成垃圾，触发一连串的回收活动，其中可能包括发起调用任意 Python 代码（由于关联到被包含对象的终结器或弱引用回调）。如果这样的代码有可能再次引用 `self`，那么这时指向被包含对象的指针为 NULL 就是非常重要的，这样 `self` 就知道被包含对象不可再被使用。`Py_CLEAR()` 宏将以安全的顺序执行此操作。

请注意 `tp_clear` 并非总是在实例被取消分配之前被调用。例如，当引用计数足以确定对象不再被使用时，就不会涉及循环垃圾回收器而是直接调用 `tp_dealloc`。

因为 `tp_clear` 函数的目的是打破循环引用，所以不需要清除所包含的对象如 Python 字符串或 Python 整数，它们无法参与循环引用。另一方面，清除所包含的全部 Python 对象，并编写类型的 `tp_dealloc` 函数来发起调用 `tp_clear` 也很方便。

有关 Python 垃圾回收方案的更多信息可在[使对象类型支持循环垃圾回收](#)一节中查看。

**继承:**

分组: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

该字段会与 `tp_traverse` 和 `Py_TPFLAGS_HAVE_GC` 旗标位一起被子类型所继承：如果旗标位，`tp_traverse` 和 `tp_clear` 在子类型中均为零则它们都将从基类型继承。

*richcmpfunc* `PyTypeObject.tp_richcompare`

一个可选的指向富比较函数的指针，函数的签名为:



```
PyObject *tp_richcompare(PyObject *self, PyObject *other, int op);
```

第一个形参将保证为 `PyTypeObject` 所定义的类型实例。  
该函数应当返回比较的结果 (通常为 `Py_True` 或 `Py_False`)。如果未定义比较运算，它必须返回 `Py_NotImplemented`，如果发生了其他错误则它必须返回 `NULL` 并设置一个异常条件。  
以下常量被定义用作 `tp_richcompare` 和 `PyObject_RichCompare()` 的第三个参数：

常数	对照
<code>Py_LT</code>	<code>&lt;</code>
<code>Py_LE</code>	<code>&lt;=</code>
<code>Py_EQ</code>	<code>==</code>
<code>Py_NE</code>	<code>!=</code>
<code>Py_GT</code>	<code>&gt;</code>
<code>Py_GE</code>	<code>&gt;=</code>

定义以下宏是为了简化编写丰富的比较函数：  
**Py\_RETURN\_RICHCOMPARE**(VAL\_A, VAL\_B, op)  
从该函数返回 `Py_True` 或 `Py_False`，这取决于比较的结果。`VAL_A` 和 `VAL_B` 必须是可通过 C 比较运算符进行排序的（例如，它们可以为 C 整数或浮点数）。第三个参数指明所请求的运算，与 `PyObject_RichCompare()` 的参数一样。  
返回值是一个新的 *strong reference*。  
发生错误时，将设置异常并从该函数返回 `NULL`。  
在 3.7 版新加入。

**继承：**  
分组: `tp_hash`, `tp_richcompare`  
该字段会被子类型同 `tp_hash` 一起继承：当子类型的 `tp_richcompare` 和 `tp_hash` 均为 `NULL` 时子类型将同时继承 `tp_richcompare` 和 `tp_hash`。

**預設：**  
`PyBaseObject_Type` 提供了一个 `tp_richcompare` 的实现，它可以被继承。但是，如果只定义了 `tp_hash`，则不会使用被继承的函数并且该类型的实例将无法参加任何比较。

**`Py_ssize_t PyTypeObject.tp_weaklistoffset`**  
如果此类型的实例是可被弱引用的，则该字段将大于零并包含在弱引用列表头的实例结构体中的偏移量（忽略 GC 头，如果存在的话）；该偏移量将被 `PyObject_ClearWeakRefs()` 和 `PyWeakref_*` 函数使用。实例结构体需要包括一个 `PyObject*` 类型的字段并初始化为 `NULL`。  
不要将该字段与 `tp_weaklist` 混淆；后者是指向类型对象本身的弱引用的列表头。

**继承：**

该字段会被子类型继承，但注意参阅下面列出的规则。子类型可以覆盖此偏移量；这意味着子类型将使用不同于基类型的弱引用列表。由于列表头总是通过 `tp_weaklistoffset` 找到的，所以这应该不成问题。

When a type defined by a class statement has no `__slots__` declaration, and none of its base types are weakly referenceable, the type is made weakly referenceable by adding a weak reference list head slot to the instance layout and setting the `tp_weaklistoffset` of that slot's offset.

When a type's `__slots__` declaration contains a slot named `__weakref__`, that slot becomes the weak reference list head for instances of the type, and the slot's offset is stored in the type's `tp_weaklistoffset`.

When a type's `__slots__` declaration does not contain a slot named `__weakref__`, the type inherits its `tp_weaklistoffset` from its base type.

#### *getterfunc* `PyTypeObject.tp_iter`

一个可选的指向函数的指针，该函数返回对象的 *iterator*。它的存在通常表明该类型的实例为 *iterable*（尽管序列在没有此函数的情况下也可能为可迭代对象）。

此函数的签名与 `PyObject_GetIter()` 的相同：

```
PyObject *tp_iter(PyObject *self);
```

#### 继承：

此字段会被子类型继承。

#### *iternextfunc* `PyTypeObject.tp_iternext`

一个可选的指向函数的指针，该函数返回 *iterator* 中的下一项。其签名为：

```
PyObject *tp_iternext(PyObject *self);
```

当该迭代器被耗尽时，它必须返回 `NULL`；`StopIteration` 异常可能会设置也可能不设置。当发生另一个错误时，它也必须返回 `NULL`。它的存在表明该类型的实际是迭代器。

迭代器类型也应当定义 `tp_iter` 函数，并且该函数应当返回迭代器实例本身（而不是新的迭代器实例）。

此函数的签名与 `PyIter_Next()` 的相同。

#### 继承：

此字段会被子类型继承。

#### `struct PyMethodDef *PyTypeObject.tp_methods`

一个可选的指向 `PyMethodDef` 结构体的以 `NULL` 结束的静态数组的指针，它声明了此类型的常规方法。

对于该数组中的每一项，都会向类型的字典（参见下面的 `tp_dict`）添加一个包含方法描述器的条目。

#### 继承：

该字段不会被子类型所继承（方法是通过不同的机制来继承的）。

#### `struct PyMemberDef *PyTypeObject.tp_members`

一个可选的指向 `PyMemberDef` 结构体的以 `NULL` 结束的静态数组的指针，它声明了此类型的常规数据成员（字段或槽位）。

对于该数组中的每一项，都会向类型的字典（参见下面的 `tp_dict`）添加一个包含方法描述器的条目。

#### 继承：

该字段不会被子类型所继承（成员是通过不同的机制来继承的）。

`struct PyGetSetDef *PyTypeObject.tp_getset`

一个可选的指向 `PyGetSetDef` 结构体的以 `NULL` 结束的静态数组的指针，它声明了此类型的实例中的被计算属性。

对于该数组中的每一项，都会向类型的字典 (参见下面的 `tp_dict`) 添加一个包含读写描述器的条目。

**继承：**

该字段不会被子类型所继承（被计算属性是通过不同的机制来继承的）。

`PyTypeObject *PyTypeObject.tp_base`

一个可选的指向类型特征属性所继承的基类型的指针。在这个层级上，只支持单继承；多重继承需要通过调用元类型动态地创建类型对象。

---

**備註：** 槽位初始化需要遵循初始化全局变量的规则。C99 要求初始化器为“地址常量”。隐式转换为指针的函数指示器如 `PyType_GenericNew()` 都是有效的 C99 地址常量。

但是，生成地址常量并不需要应用于非静态变量如 `PyBaseObject_Type` 的单目运算符 `&`。编译器可能支持该运算符（如 `gcc`），但 `MSVC` 则不支持。这两种编译器在这一特定行为上都是严格符合标准的。

因此，应当在扩展模块的初始化函数中设置 `tp_base`。

---

**继承：**

该字段不会被子类型继承（显然）。

**預設：**

该字段默认为 `&PyBaseObject_Type` (对 Python 程序员来说即 `object` 类型)。

`PyObject *PyTypeObject.tp_dict`

类型的字典将由 `PyType_Ready()` 存储到这里。

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__()`).

**继承：**

该字段不会被子类型所继承（但在这里定义的属性是通过不同的机制来继承的）。

**預設：**

如果该字段为 `NULL`，`PyType_Ready()` 将为它分配一个新字典。

**警告：** 通过字典 C-API 使用 `PyDict_SetItem()` 或修改 `tp_dict` 是不安全的。

`descrgetfunc PyTypeObject.tp_descr_get`

一个可选的指向“描述器获取”函数的指针。

函数的签名为：

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *type);
```

**继承：**

此字段会被子类型继承。

*descrsetfunc* `PyTypeObject.tp_descr_set`

一个指向用于设置和删除描述器值的函数的选项指针。

函数的签名为:

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

将 *value* 参数设为 NULL 以删除该值。

**继承:**

此字段会被子类型继承。

*Py\_ssize\_t* `PyTypeObject.tp_dictoffset`

如果该类型的实例具有一个包含实例变量的字典, 则此字段将为非零值并包含该实例变量字典的类型的实例的偏移量; 该偏移量将由 `PyObject_GenericGetAttr()` 使用。

不要将该字段与 `tp_dict` 混淆; 后者是由类型对象本身的属性组成的字典。

If the value of this field is greater than zero, it specifies the offset from the start of the instance structure. If the value is less than zero, it specifies the offset from the *end* of the instance structure. A negative offset is more expensive to use, and should only be used when the instance structure contains a variable-length part. This is used for example to add an instance variable dictionary to subtypes of `str` or `tuple`. Note that the `tp_basicsize` field should account for the dictionary added to the end in that case, even though the dictionary is not included in the basic object layout. On a system with a pointer size of 4 bytes, `tp_dictoffset` should be set to `-4` to indicate that the dictionary is at the very end of the structure.

`tp_dictoffset` 应当被视为是只读的。用于获取指向字典调用 `PyObject_GenericGetDict()` 的指针。调用 `PyObject_GenericGetDict()` 可能需要为字典分配内存, 因此在访问对象上的属性时调用 `PyObject_GetAttr()` 可能会更有效率。

**继承:**

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype instances store the dictionary at a difference offset than the base type. Since the dictionary is always found via `tp_dictoffset`, this should not be a problem.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types has an instance variable dictionary, a dictionary slot is added to the instance layout and the `tp_dictoffset` is set to that slot's offset.

When a type defined by a class statement has a `__slots__` declaration, the type inherits its `tp_dictoffset` from its base type.

(Adding a slot named `__dict__` to the `__slots__` declaration does not have the expected effect, it just causes confusion. Maybe this should be added as a feature just like `__weakref__` though.)

**預設:**

这个槽位没有默认值。对于静态类型, 如果该字段为 NULL 则不会为实例创建 `__dict__`。

*initproc* `PyTypeObject.tp_init`

一个可选的指向实例初始化函数的指针。

此函数对应于类的 `__init__()` 方法。和 `__init__()` 一样, 创建实例时不调用 `__init__()` 是有可能的, 并且通过再次调用实例的 `__init__()` 方法将其重新初始化也是有可能的。

函数的签名为:

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwds);
```

*self* 参数是将要初始化的实例; *args* 和 *kwds* 参数代表调用 `__init__()` 时传入的位置和关键字参数。

`tp_init` 函数如果不为 NULL, 将在通过调用类型正常创建其实例时被调用, 即在类型的 `tp_new` 函数返回一个该类型的实例时。如果 `tp_new` 函数返回了一个不是原始类型的子类型的其他类型的

实例，则 `tp_init` 函数不会被调用；如果 `tp_new` 返回了一个原始类型的子类型的实例，则该子类型的 `tp_init` 将被调用。

成功时返回 0，发生错误时则返回 -1 并在错误上设置一个异常。and sets an exception on error.

#### 继承：

此字段会被子类型继承。

#### 預設：

对于静态类型来说该字段没有默认值。

#### *allocfunc* `PyTypeObject.tp_alloc`

指向一个实例分配函数的可选指针。

函数的签名为：

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems);
```

#### 继承：

该字段会被静态子类型继承，但不会被动态子类型（通过 `class` 语句创建的子类型）继承。

#### 預設：

对于动态子类型，该字段总是会被设为 `PyType_GenericAlloc()`，以强制应用标准的堆分配策略。

对于静态子类型，`PyBaseObject_Type` 将使用 `PyType_GenericAlloc()`。这是适用于所有静态定义类型的推荐值。

#### *newfunc* `PyTypeObject.tp_new`

一个可选的指向实例创建函数的指针。

函数的签名为：

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwargs);
```

`subtype` 参数是被创建的对象类型；`args` 和 `kwargs` 参数表示调用类型时传入的位置和关键字参数。请注意 `subtype` 不是必须与被调用的 `tp_new` 函数所属的类型相同；它可以是该类型的子类型（但不能是完全无关的类型）。

`tp_new` 函数应当调用 `subtype->tp_alloc(subtype, nitems)` 来为对象分配空间，然后只执行绝对有必要的进一步初始化操作。可以安全地忽略或重复的初始化操作应当放在 `tp_init` 处理器中。一个关键的规则是对于不可变类型来说，所有初始化操作都应当在 `tp_new` 中发生，而对于可变类型，大部分初始化操作都应当推迟到 `tp_init` 再执行。

设置 `Py_TPFLAGS_DISALLOW_INSTANTIATION` 旗标以禁止在 Python 中创建该类型的实例。

#### 继承：

该字段会被子类型所继承，例外情况是它不会被 `tp_base` 为 `NULL` 或 `&PyBaseObject_Type` 的静态类型所继承。

#### 預設：

对于静态类型该字段没有默认值。这意味着如果槽位被定义为 `NULL`，则无法调用此类型来创建新的实例；应当存在其他办法来创建实例，例如工厂函数等。

#### *freefunc* `PyTypeObject.tp_free`

一个可选的指向实例释放函数的指针。函数的签名为：

```
void tp_free(void *self);
```

一个兼容该签名的初始化器是 `PyObject_Free()`。

#### 继承：

该字段会被静态子类型继承，但不会被动态子类型（通过 `class` 语句创建的子类型）继承

#### 預設:

在动态子类型中，该字段会被设为一个适合与 `PyType_GenericAlloc()` 以及 `Py_TPFLAGS_HAVE_GC` 旗标位的值相匹配的释放器。

对于静态子类型，`PyBaseObject_Type` 将使用 `PyObject_Del()`。

#### *inquiry* `PyTypeObject.tp_is_gc`

可选的指向垃圾回收器所调用的函数的指针。

垃圾回收器需要知道某个特定的对象是否可以被回收。在一般情况下，垃圾回收器只需要检查这个对象类型的 `tp_flags` 字段、以及 `Py_TPFLAGS_HAVE_GC` 标识位即可做出判断；但是有一些类型同时混合包含了静态和动态分配的实例，其中静态分配的实例不应该也无法被回收。本函数为后者情况而设计：对于可被垃圾回收的实例，本函数应当返回 1；对于不可被垃圾回收的实例，本函数应当返回 0。函数的签名为：

```
int tp_is_gc(PyObject *self);
```

（此对象的唯一样例是类型本身。元类型 `PyType_Type` 定义了该函数来区分静态和动态分配的类型。）

#### 继承:

此字段会被子类型继承。

#### 預設:

此槽位没有默认值。如果该字段为 `NULL`，则将使用 `Py_TPFLAGS_HAVE_GC` 作为相同功能的替代。

#### *PyObject\** `PyTypeObject.tp_bases`

基类型的元组。

此字段应当被设为 `NULL` 并被视为只读。Python 将在类型初始化时填充它。

对于动态创建的类，可以使用 `Py_tp_bases` 槽位来代替 `PyType_FromSpecWithBases()` 的 `bases` 参数。推荐使用参数形式。

**警告：** 多重继承不适合静态定义的类型。如果你将 `tp_bases` 设为一个元组，Python 将不会引发错误，但某些槽位将只从第一个基类型继承。

#### 继承:

这个字段不会被继承。

#### *PyObject\** `PyTypeObject.tp_mro`

包含基类型的扩展集的元组，以类型本身开始并以 `object` 作为结束，使用方法解析顺序。

此字段应当被设为 `NULL` 并被视为只读。Python 将在类型初始化时填充它。

#### 继承:

这个字段不会被继承；它是通过 `PyType_Ready()` 计算得到的。

#### *PyObject\** `PyTypeObject.tp_cache`

尚未使用。仅供内部使用。

#### 继承:

这个字段不会被继承。

#### *PyObject\** `PyTypeObject.tp_subclasses`

由对子类的弱引用组成的列表。仅供内部使用。

#### 继承:

这个字段不会被继承。



*PyObject* \*PyTypeObject.**tp\_weaklist**

弱引用列表头，用于指向该类型对象的弱引用。不会被继承。仅限内部使用。

**继承：**

这个字段不会被继承。

*destructor* PyTypeObject.**tp\_del**

该字段已被弃用。请改用 *tp\_finalize*。

unsigned int PyTypeObject.**tp\_version\_tag**

用于索引至方法缓存。仅限内部使用。

**继承：**

这个字段不会被继承。

*destructor* PyTypeObject.**tp\_finalize**

一个可选的指向实例最终化函数的指针。函数的签名为：

```
void tp_finalize(PyObject *self);
```

如果设置了 *tp\_finalize*，解释器将在最终化特定实例时调用它一次。它将由垃圾回收器调用（如果实例是单独循环引用的一部分）或是在对象被释放之前被调用。不论是哪种方式，它都肯定会在尝试打破循环引用之前被调用，以确保它所操作的对象处于正常状态。

*tp\_finalize* 不应改变当前异常状态；因此，编写非关键终结器的推荐做法如下：

```
static void
local_finalize(PyObject *self)
{
    PyObject *error_type, *error_value, *error_traceback;

    /* Save the current exception, if any. */
    PyErr_Fetch(&error_type, &error_value, &error_traceback);

    /* ... */

    /* Restore the saved exception. */
    PyErr_Restore(error_type, error_value, error_traceback);
}
```

另外还需要注意，在应用垃圾回收机制的 Python 中，*tp\_dealloc* 可以从任意 Python 线程被调用，而不仅是创建该对象的线程（如果对象成为引用计数循环的一部分，则该循环可能会被任何线程上的垃圾回收操作所回收）。这对 Python API 调用来说不是问题，因为 *tp\_dealloc* 调用所在的线程将持有全局解释器锁（GIL）。但是，如果被销毁的对象又销毁了来自其他 C 或 C++ 库的对象，则应当小心确保在调用 *tp\_dealloc* 的线程上销毁这些对象不会破坏这些库的任何资源。

**继承：**

此字段会被子类型继承。

在 3.4 版新加入。

在 3.8 版的变更：在 3.8 版之前必须设置 *Py\_TPFLAGS\_HAVE\_FINALIZE* 旗标才能让该字段被使用。现在已不再需要这样做。

**也参考：**

”安全的对象最终化” (PEP 442)

*vectorcallfunc* PyTypeObject.**tp\_vectorcall**

用于此类型对象的调用的 *vectorcall* 函数。换句话说，它是被用来实现 *type.\_\_call\_\_* 的 *vectorcall*。如果 *tp\_vectorcall* 为 NULL，默认调用实现将使用 *\_\_new\_\_()* 并且 *\_\_init\_\_()* 将被使用。

**继承：**

这个字段不会被继承。

在 3.9 版新加入：（这个字段从 3.8 起即存在，但是从 3.9 开始投入使用）

### 12.3.6 静态类型

在传统上，在 C 代码中定义的类型都是静态的，也就是说，`PyTypeObject` 结构体在代码中直接定义并使用 `PyType_Ready()` 来初始化。

这就导致了与在 Python 中定义的类型相关联的类型限制：

- 静态类型只能拥有一个基类；换句话说，他们不能使用多重继承。
- 静态类型对象（但并非它们的实例）是不可变对象。不可能在 Python 中添加或修改类型对象的属性。
- 静态类型对象是跨子解释器共享的，因此它们不应包括任何子解释器专属的状态。

此外，由于 `PyTypeObject` 只是作为不透明结构的受限 API 的一部分，因此任何使用静态类型的扩展模块都必须针对特定的 Python 次版本进行编译。

### 12.3.7 堆类型

一种静态类型的替代物是堆分配类型，或者简称堆类型，它与使用 Python 的 `class` 语句创建的类紧密对应。堆类型设置了 `Py_TPFLAGS_HEAPTYPE` 旗标。

This is done by filling a `PyType_Spec` structure and calling `PyType_FromSpec()`, `PyType_FromSpecWithBases()`, or `PyType_FromModuleAndSpec()`.

## 12.4 数字对象结构体

type **PyNumberMethods**

该结构体持有指向被对象用来实现数字协议的函数的指针。每个函数都被数字协议一节中记录的对应名称的函数所使用。

结构体定义如下：

```
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;
```

(繼續下一頁)

(繼續上一頁)

```

    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;

```

**備註：** 双目和三目函数必须检查其所有操作数的类型，并实现必要的转换（至少有一个操作数是所定义类型的实例）。如果没有为所给出的操作数定义操作，则双目和三目函数必须返回 `Py_NotImplemented`，如果发生了其他错误则它们必须返回 `NULL` 并设置一个异常。

**備註：** `nb_reserved` 字段应当始终为 `NULL`。在之前版本中其名称为 `nb_long`，并在 Python 3.0.1 中改名。

*binaryfunc* `PyNumberMethods.nb_add`

*binaryfunc* `PyNumberMethods.nb_subtract`

*binaryfunc* `PyNumberMethods.nb_multiply`

*binaryfunc* `PyNumberMethods.nb_remainder`

*binaryfunc* `PyNumberMethods.nb_divmod`

*ternaryfunc* `PyNumberMethods.nb_power`

*unaryfunc* `PyNumberMethods.nb_negative`

*unaryfunc* `PyNumberMethods.nb_positive`

*unaryfunc* `PyNumberMethods.nb_absolute`

*inquiry* `PyNumberMethods.nb_bool`

*unaryfunc* `PyNumberMethods.nb_invert`

*binaryfunc* `PyNumberMethods.nb_lshift`

*binaryfunc* `PyNumberMethods.nb_rshift`

*binaryfunc* `PyNumberMethods.nb_and`

*binaryfunc* `PyNumberMethods.nb_xor`

*binaryfunc* `PyNumberMethods.nb_or`

*unaryfunc* *PyNumberMethods.nb\_int*

*void* \**PyNumberMethods.nb\_reserved*

*unaryfunc* *PyNumberMethods.nb\_float*

*binaryfunc* *PyNumberMethods.nb\_inplace\_add*

*binaryfunc* *PyNumberMethods.nb\_inplace\_subtract*

*binaryfunc* *PyNumberMethods.nb\_inplace\_multiply*

*binaryfunc* *PyNumberMethods.nb\_inplace\_remainder*

*ternaryfunc* *PyNumberMethods.nb\_inplace\_power*

*binaryfunc* *PyNumberMethods.nb\_inplace\_lshift*

*binaryfunc* *PyNumberMethods.nb\_inplace\_rshift*

*binaryfunc* *PyNumberMethods.nb\_inplace\_and*

*binaryfunc* *PyNumberMethods.nb\_inplace\_xor*

*binaryfunc* *PyNumberMethods.nb\_inplace\_or*

*binaryfunc* *PyNumberMethods.nb\_floor\_divide*

*binaryfunc* *PyNumberMethods.nb\_true\_divide*

*binaryfunc* *PyNumberMethods.nb\_inplace\_floor\_divide*

*binaryfunc* *PyNumberMethods.nb\_inplace\_true\_divide*

*unaryfunc* *PyNumberMethods.nb\_index*

*binaryfunc* *PyNumberMethods.nb\_matrix\_multiply*

*binaryfunc* *PyNumberMethods.nb\_inplace\_matrix\_multiply*

## 12.5 映射对象结构体

type **PyMappingMethods**

该结构体持有指向对象用于实现映射协议的函数的指针。它有三个成员：

*lenfunc* *PyMappingMethods.mp\_length*

该函数将被 *PyMapping\_Size()* 和 *PyObject\_Size()* 使用，并具有相同的签名。如果对象没有定义长度则此槽位可被设为 NULL。

*binaryfunc* *PyMappingMethods.mp\_subscript*

该函数将被 *PyObject\_GetItem()* 和 *PySequence\_GetSlice()* 使用，并具有与 *PyObject\_GetItem()* 相同的签名。此槽位必须被填充以便 *PyMapping\_Check()* 函数返回 1，否则它可以为 NULL。

*objobjargproc* *PyMappingMethods.mp\_ass\_subscript*

该函数将被 *PyObject\_SetItem()*、*PyObject\_DelItem()*、*PySequence\_SetSlice()* 和 *PySequence\_DelSlice()* 使用。它具有与 *PyObject\_SetItem()* 相同的签名，但 *v* 也可以被设为 NULL 以删除一个条目。如果此槽位为 NULL，则对象将不支持条目赋值和删除。

## 12.6 序列对象结构体

type **PySequenceMethods**

该结构体持有指向对象用于实现序列协议的函数的指针。

*lenfunc* **PySequenceMethods.sq\_length**

此函数被 `PySequence_Size()` 和 `PyObject_Size()` 所使用, 并具有与它们相同的签名。它还被用于通过 `sq_item` 和 `sq_ass_item` 槽位来处理负索引号。

*binaryfunc* **PySequenceMethods.sq\_concat**

此函数被 `PySequence_Concat()` 所使用并具有相同的签名。在尝试通过 `nb_add` 槽位执行数值相加之后它还会被用于 `+` 运算符。

*ssizeargfunc* **PySequenceMethods.sq\_repeat**

此函数被 `PySequence_Repeat()` 所使用并具有相同的签名。在尝试通过 `nb_multiply` 槽位执行数值相乘之后它还会被用于 `*` 运算符。

*ssizeargfunc* **PySequenceMethods.sq\_item**

此函数被 `PySequence_GetItem()` 所使用并具有相同的签名。在尝试通过 `mp_subscript` 槽位执行下标操作之后它还会被用于 `PyObject_GetItem()`。该槽位必须被填充以便 `PySequence_Check()` 函数返回 1, 否则它可以为 NULL。

负索引号是按如下方式处理的: 如果 `sq_length` 槽位已被填充, 它将被调用并使用序列长度来计算出正索引号并传给 `sq_item`。如果 `sq_length` 为 NULL, 索引号将原样传给此函数。

*ssizeobjargproc* **PySequenceMethods.sq\_ass\_item**

此函数被 `PySequence_SetItem()` 所使用并具有相同的签名。在尝试通过 `mp_ass_subscript` 槽位执行条目赋值和删除操作之后它还会被用于 `PyObject_SetItem()` 和 `PyObject_DelItem()`。如果对象不支持条目和删除则该槽位可以保持为 NULL。

*objobjproc* **PySequenceMethods.sq\_contains**

该函数可供 `PySequence_Contains()` 使用并具有相同的签名。此槽位可以保持为 NULL, 在此情况下 `PySequence_Contains()` 只需遍历该序列直到找到一个匹配。

*binaryfunc* **PySequenceMethods.sq\_inplace\_concat**

此函数被 `PySequence_InPlaceConcat()` 所使用并具有相同的签名。它应当修改它的第一个操作数, 并将其返回。该槽位可以保持为 NULL, 在此情况下 `PySequence_InPlaceConcat()` 将回退到 `PySequence_Concat()`。在尝试通过 `nb_inplace_add` 槽位执行数字原地相加之后它还会被用于增强赋值运算符 `+=`。

*ssizeargfunc* **PySequenceMethods.sq\_inplace\_repeat**

此函数被 `PySequence_InPlaceRepeat()` 所使用并具有相同的签名。它应当修改它的第一个操作数, 并将其返回。该槽位可以保持为 NULL, 在此情况下 `PySequence_InPlaceRepeat()` 将回退到 `PySequence_Repeat()`。在尝试通过 `nb_inplace_multiply` 槽位执行数字原地相乘之后它还会被用于增强赋值运算符 `*=`。

## 12.7 缓冲区对象结构体

type **PyBufferProcs**

此结构体持有指向缓冲区协议所需要的函数的指针。该协议定义了导出方对象要如何向消费方对象暴露其内部数据。

*getbufferproc* **PyBufferProcs.bf\_getbuffer**

此函数的签名为:

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

处理发给 *exporter* 的请求来填充 *flags* 所指定的 *view*。除第 (3) 点外，此函数的实现必须执行以下步骤：

- (1) 检查请求是否能被满足。如果不能，则会引发 `BufferError`，将 `view->obj` 设为 `NULL` 并返回 `-1`。
- (2) 填充请求的字段。
- (3) 递增用于保存导出次数的内部计数器。
- (4) 将 `view->obj` 设为 *exporter* 并递增 `view->obj`。
- (5) 返回 `0`。

如果 *exporter* 是缓冲区提供方的链式或树型结构的一部分，则可以使用两种主要方案：

- 重导出：树型结构的每个成员作为导出对象并将 `view->obj` 设为对其自身的新引用。
- 重定向：缓冲区请求将被重定向到树型结构的根对象。在此，`view->obj` 将为对根对象的新引用。

*view* 中每个字段的描述参见缓冲区结构体一节，导出方对于特定请求应当如何反应参见缓冲区请求类型一节。

所有在 `Py_buffer` 结构体中被指向的内存都属于导出方并必须保持有效直到不再有任何消费方。*format*, *shape*, *strides*, *suboffsets* 和 *internal* 对于消费方来说是只读的。

`PyBuffer_FillInfo()` 提供了一种暴露简单字节缓冲区同时正确处理地所有请求类型的简便方式。

`PyObject_GetBuffer()` 是针对包装此函数的消费方的接口。

`releasebufferproc PyBufferProcs.bf_releasebuffer`

此函数的签名为：

```
void (PyObject *exporter, Py_buffer *view);
```

处理释放缓冲区资源的请求。如果不需要释放任何资源，则 `PyBufferProcs.bf_releasebuffer` 可以为 `NULL`。在其他情况下，此函数的标准实现将执行以下的可选步骤：

- (1) 递减用于保存导出次数的内部计数器。
- (2) 如果计数器为 `0`，则释放所有关联到 *view* 的内存。

导出方必须使用 *internal* 字段来记录缓冲区专属的资源。该字段将确保恒定，而消费方则可能将原始缓冲区作为 *view* 参数传入。

此函数不可递减 `view->obj`，因为这是在 `PyBuffer_Release()` 中自动完成的（此方案适用于打破循环引用）。

`PyBuffer_Release()` 是针对包装此函数的消费方的接口。

## 12.8 异步对象结构体

在 3.5 版新加入。

type `PyAsyncMethods`

此结构体将持有指向需要用来实现 *awaitable* 和 *asynchronous iterator* 对象的函数的指针。

结构体定义如下：

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
```

(繼續下一頁)



(繼續上一頁)

```
sendfunc am_send;
} PyAsyncMethods;
```

**unaryfunc PyAsyncMethods.am\_await**

此函数的签名为:

```
PyObject *am_await(PyObject *self);
```

返回的对象必须为 *iterator*，即对其执行 `PyIter_Check()` 必须返回 1。

如果一个对象不是 *awaitable* 则此槽位可被设为 NULL。

**unaryfunc PyAsyncMethods.am\_aiter**

此函数的签名为:

```
PyObject *am_aiter(PyObject *self);
```

必须返回一个 *asynchronous iterator* 对象。请参阅 `__anext__()` 了解详情。

如果一个对象没有实现异步迭代协议则此槽位可被设为 NULL。

**unaryfunc PyAsyncMethods.am\_anext**

此函数的签名为:

```
PyObject *am_anext(PyObject *self);
```

必须返回一个 *awaitable* 对象。请参阅 `__anext__()` 了解详情。此槽位可被设为 NULL。

**sendfunc PyAsyncMethods.am\_send**

此函数的签名为:

```
PySendResult am_send(PyObject *self, PyObject *arg, PyObject **result);
```

请参阅 `PyIter_Send()` 了解详情。此槽位可被设为 NULL。

在 3.10 版新加入。

## 12.9 槽位类型 typedef

**typedef PyObject \*(\**allocfunc*)(PyTypeObject \*cls, Py\_ssize\_t nitems)**

属于稳定 ABI。此函数的设计目标是将内存分配与内存初始化进行分离。它应当返回一个指向足够容纳实例长度，适当对齐，并初始化为零的内存块的指针，但将 `ob_refcnt` 设为 1 并将 `ob_type` 设为 `type` 参数。如果类型的 `tp_itemsize` 为非零值，则对象的 `ob_size` 字段应当被初始化为 `nitems` 而分配内存块的长度应为 `tp_basicsize + nitems*tp_itemsize`，并舍入到 `sizeof(void*)` 的倍数；在其他情况下，`nitems` 将不会被使用而内存块的长度应为 `tp_basicsize`。

此函数不应执行任何其他实例初始化操作，即使是分配额外内存也不应执行；那应当由 `tp_new` 来完成。

**typedef void (\**destructor*)(PyObject\*)**

属于稳定 ABI。

**typedef void (\**freefunc*)(void\*)**

請見 `tp_free`。

**typedef PyObject \*(\**newfunc*)(PyObject\*, PyObject\*, PyObject\*)**

属于稳定 ABI。請見 `tp_new`。

```
typedef int (*initproc)(PyObject*, PyObject*, PyObject*)
```

属于稳定 ABI. 請見 `tp_init`。

```
typedef PyObject* (*reprfunc)(PyObject*)
```

属于稳定 ABI. 請見 `tp_repr`。

```
typedef PyObject* (*getattrfunc)(PyObject* self, char* attr)
```

属于稳定 ABI. 返回对象的指定属性的值。

```
typedef int (*setattrfunc)(PyObject* self, char* attr, PyObject* value)
```

属于稳定 ABI. 为对象设置指定属性的值。将 `value` 参数设为 `NULL` 将删除该属性。

```
typedef PyObject* (*getattrofunc)(PyObject* self, PyObject* attr)
```

属于稳定 ABI. 返回对象的指定属性的值。

請見 `tp_getattro`。

```
typedef int (*setattrofunc)(PyObject* self, PyObject* attr, PyObject* value)
```

属于稳定 ABI. 为对象设置指定属性的值。将 `value` 参数设为 `NULL` 将删除该属性。

請見 `tp_setattro`。

```
typedef PyObject* (*descrgetfunc)(PyObject*, PyObject*, PyObject*)
```

属于稳定 ABI. 請見 `tp_descr_get`。

```
typedef int (*descrsetfunc)(PyObject*, PyObject*, PyObject*)
```

属于稳定 ABI. 請見 `tp_descr_set`。

```
typedef Py_hash_t (*hashfunc)(PyObject*)
```

属于稳定 ABI. 請見 `tp_hash`。

```
typedef PyObject* (*richcmpfunc)(PyObject*, PyObject*, int)
```

属于稳定 ABI. 請見 `tp_richcompare`。

```
typedef PyObject* (*getiterfunc)(PyObject*)
```

属于稳定 ABI. 請見 `tp_iter`。

```
typedef PyObject* (*iternextfunc)(PyObject*)
```

属于稳定 ABI. 請見 `tp_iternext`。

```
typedef Py_ssize_t (*lenfunc)(PyObject*)
```

属于稳定 ABI.

```
typedef int (*getbufferproc)(PyObject*, Py_buffer*, int)
```

```
typedef void (*releasebufferproc)(PyObject*, Py_buffer*)
```

```
typedef PyObject* (*unaryfunc)(PyObject*)
```

属于稳定 ABI.

```
typedef PyObject* (*binaryfunc)(PyObject*, PyObject*)
```

属于稳定 ABI.

```
typedef PySendResult (*sendfunc)(PyObject*, PyObject*, PyObject**)
```

請見 `am_send`。

```
typedef PyObject* (*ternaryfunc)(PyObject*, PyObject*, PyObject*)
```

属于稳定 ABI.

```
typedef PyObject* (*ssizeargfunc)(PyObject*, Py_ssize_t)
```

属于稳定 ABI.

```
typedef int (*ssizeobjargproc)(PyObject*, Py_ssize_t, PyObject*)
```

属于稳定 ABI.

```
typedef int (*objobjproc)(PyObject*, PyObject*)
```

属于稳定 ABI.

```
typedef int (*objobjargproc)(PyObject*, PyObject*, PyObject*)
```

属于稳定 ABI.

## 12.10 范例

下面是一些 Python 类型定义的简单示例。其中包括你可能会遇到的通常用法。有些演示了令人困惑的边际情况。要获取更多示例、实践信息以及教程，请参阅 [defining-new-types](#) 和 [new-types-topics](#)。

一个基本的静态类型：

```
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,
};
```

你可能还会看到带有更繁琐的初始化器的较旧代码（特别是在 CPython 代码库中）：

```
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 */

```

(繼續下一頁)

(繼續上一頁)

```

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 */
};

```

一个支持弱引用、实例字典和哈希运算的类型:

```

typedef struct {
    PyObject_HEAD
    const char *data;
    PyObject *inst_dict;
    PyObject *weakreflist;
} 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_weaklistoffset = offsetof(MyObject, weakreflist),
    .tp_dictoffset = offsetof(MyObject, inst_dict),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
    .tp_new = myobj_new,
    .tp_traverse = (traverseproc)myobj_traverse,
    .tp_clear = (inquiry)myobj_clear,
    .tp_alloc = PyType_GenericNew,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
    .tp_hash = (hashfunc)myobj_hash,
    .tp_richcompare = PyBaseObject_Type.tp_richcompare,
};

```

一个不能被子类化且不能被调用以使用 `Py_TPFLAGS_DISALLOW_INSTANTIATION` 旗标创建实例（例如使用单独的工厂函数）的 `str` 子类:

```

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,
};

```

最简单的固定长度实例静态类型:

```

typedef struct {
    PyObject_HEAD
} MyObject;

```

(繼續下一頁)

(繼續上一頁)

```
static PyObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
};
```

最简单的具有可变长度实例的静态类型:

```
typedef struct {
    PyObject_VAR_HEAD
    const char *data[1];
} MyObject;

static PyObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject) - sizeof(char *),
    .tp_itemsize = sizeof(char *),
};
```

## 12.11 使对象类型支持循环垃圾回收

Python 对循环引用的垃圾检测与回收需要“容器”对象类型的支持，此类型的容器对象中可能包含其它容器对象。不保存其它对象的引用的类型，或者只保存原子类型（如数字或字符串）的引用的类型，不需要显式提供垃圾回收的支持。

要创建一个容器类，类型对象的 `tp_flags` 字段必须包括 `Py_TPFLAGS_HAVE_GC` 并提供一个 `tp_traverse` 处理器的实现。如果该类型的实例是可变的，则还必须提供 `tp_clear` 的实现。

### `Py_TPFLAGS_HAVE_GC`

设置了此标志位的类型的对象必须符合此处记录的规则。为方便起见，下文把这些对象称为容器对象。

容器类型的构造函数必须符合两个规则：

1. 该对象的内在必须使用 `PyObject_GC_New` 或 `PyObject_GC_NewVar` 来分配。
2. 初始化了所有可能包含其他容器的引用的字段后，它必须调用 `PyObject_GC_Track()`。

同样的，对象的释放器必须符合两个类似的规则：

1. 在引用其它容器的字段失效前，必须调用 `PyObject_GC_UnTrack()`。
2. 必须使用 `PyObject_GC_Del()` 释放对象的内存。

**警告：** 如果一个类型添加了 `Py_TPFLAGS_HAVE_GC`，则它必须实现至少一个 `tp_traverse` 句柄或显式地使用来自其一个或多个子类的句柄。

当调用 `PyType_Ready()` 或者某些间接调用该函数的 API 如 `PyType_FromSpecWithBases()` 或 `PyType_FromSpec()` 时解释器将自动填充 `tp_flags`, `tp_traverse` 和 `tp_clear` 字段，如果该类型是继承自实现了垃圾回收器协议的类并且该子类没有包括 `Py_TPFLAGS_HAVE_GC` 旗标的话。

**`PyObject_GC_New`** (TYPE, typeobj)

类似于 `PyObject_New` 但专用于设置了 `Py_TPFLAGS_HAVE_GC` 旗标的容器对象。

**`PyObject_GC_NewVar`** (TYPE, typeobj, size)

与 `PyObject_NewVar` 类似但专用于设置了 `Py_TPFLAGS_HAVE_GC` 旗标的容器对象。

**PyObject\_GC\_Resize** (TYPE, op, newsize)

重新调整 *PyObject\_NewVar* 所分配对象的大小。返回调整大小后的类型为 TYPE\* 的对象（指向任意 C 类型）或在失败时返回 NULL。

*op* 必须为 *PyVarObject\** 类型并且不能已被回收器所追踪。*newsize* 必须为 *Py\_ssize\_t* 类型。

**void PyObject\_GC\_Track** (*PyObject* \*op)

属于稳定 ABI。把对象 *op* 加入到垃圾回收器跟踪的容器对象中。对象在被回收器跟踪时必须保持有效的，因为回收器可能在任何时候开始运行。在 *tp\_traverse* 处理前的所有字段变为有效后，必须调用此函数，通常在靠近构造函数末尾的位置。

**int PyObject\_IS\_GC** (*PyObject* \*obj)

如果对象实现了垃圾回收器协议则返回非零值，否则返回 0。

如果此函数返回 0 则对象无法被垃圾回收器追踪。

**int PyObject\_GC\_IsTracked** (*PyObject* \*op)

属于稳定 ABI 自 3.9 版起。如果 *op* 对象的类型实现了 GC 协议且 *op* 目前正被垃圾回收器追踪则返回 1，否则返回 0。

这类似于 Python 函数 `gc.is_tracked()`。

在 3.9 版新加入。

**int PyObject\_GC\_IsFinalized** (*PyObject* \*op)

属于稳定 ABI 自 3.9 版起。如果 *op* 对象的类型实现了 GC 协议且 *op* 已经被垃圾回收器终结则返回 1，否则返回 0。

这类似于 Python 函数 `gc.is_finalized()`。

在 3.9 版新加入。

**void PyObject\_GC\_Del** (void \*op)

属于稳定 ABI。使用 *PyObject\_GC\_New* 或 *PyObject\_GC\_NewVar* 释放分配给对象的内存。

**void PyObject\_GC\_UnTrack** (void \*op)

属于稳定 ABI。从回收器跟踪的容器对象集合中移除 *op* 对象。请注意可以在此对象上再次调用 *PyObject\_GC\_Track()* 以将其加回到被跟踪对象集合。释放器 (*tp\_dealloc* 句柄) 应当在 *tp\_traverse* 句柄所使用的任何字段失效之前为对象调用此函数。

在 3.8 版的變更: `_PyObject_GC_TRACK()` 和 `_PyObject_GC_UNTRACK()` 宏已从公有 C API 中删除。

*tp\_traverse* 处理接收以下类型的函数形参。

**typedef int (\*visitproc)** (*PyObject* \*object, void \*arg)

属于稳定 ABI。传给 *tp\_traverse* 处理的访问函数的类型。*object* 是容器中需要被遍历的一个对象，第三个形参对应于 *tp\_traverse* 处理的 *arg*。Python 核心使用多个访问者函数实现循环引用的垃圾检测，不需要用户自行实现访问者函数。

*tp\_traverse* 处理必须是以下类型：

**typedef int (\*traverseproc)** (*PyObject* \*self, *visitproc* visit, void \*arg)

属于稳定 ABI。用于容器对象的遍历函数。它的实现必须对 *self* 所直接包含的每个对象调用 *visit* 函数，*visit* 的形参为所包含对象和传给处理程序的 *arg* 值。*visit* 函数调用不可附带 NULL 对象作为参数。如果 *visit* 返回非零值，则该值应当被立即返回。

为了简化 *tp\_traverse* 处理的实现，Python 提供了一个 `Py_VISIT()` 宏。若要使用这个宏，必须把 *tp\_traverse* 的参数命名为 *visit* 和 *arg*。

**void Py\_VISIT** (*PyObject* \*o)

如果 *o* 不为 NULL，则调用 *visit* 回调函数，附带参数 *o* 和 *arg*。如果 *visit* 返回一个非零值，则返回该值。使用此宏之后，*tp\_traverse* 处理程序的形式如下：



```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

`tp_clear` 处理程序必须为 `inquiry` 类型，如果对象不可变则为 `NULL`。

```
typedef int (*inquiry)(PyObject *self)
```

属于稳定 ABI。丢弃产生循环引用的引用。不可变对象不需要声明此方法，因为他们不可能直接产生循环引用。需要注意的是，对象在调用此方法后必须仍是有效的（不能对引用只调用 `Py_DECREF()` 方法）。当垃圾回收器检测到该对象在循环引用中时，此方法会被调用。

### 12.11.1 控制垃圾回收器状态

这个 C-API 提供了以下函数用于控制垃圾回收的运行。

```
Py_ssize_t PyGC_Collect (void)
```

属于稳定 ABI。执行完全的垃圾回收，如果垃圾回收器已启用的话。（请注意 `gc.collect()` 会无条件地执行它。）

返回已回收的 + 无法回收的不可获取对象的数量。如果垃圾回收器被禁用或已在执行回收，则立即返回 0。在垃圾回收期间发生的错误会被传给 `sys.unraisablehook`。此函数不会引发异常。

```
int PyGC_Enable (void)
```

属于稳定 ABI 自 3.10 版起。启用垃圾回收器：类似于 `gc.enable()`。返回之前的状态，0 为禁用而 1 为启用。

在 3.10 版新加入。

```
int PyGC_Disable (void)
```

属于稳定 ABI 自 3.10 版起。禁用垃圾回收器：类似于 `gc.disable()`。返回之前的状态，0 为禁用而 1 为启用。

在 3.10 版新加入。

```
int PyGC_IsEnabled (void)
```

属于稳定 ABI 自 3.10 版起。查询垃圾回收器的状态：类似于 `gc.isenabled()`。返回当前的状态，0 为禁用而 1 为启用。

在 3.10 版新加入。



API 和 ABI 版本管理

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，**不一定**是**執行環境 (run time)** 所使用的版本。

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 **則**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**則**單一整數的 Python 版本號。

所代表的版本資訊可以用以下規則將其看做是一個 32 位元數字來獲得：

位 元 組 串	位元 (大端位元組序 (big endian order))	意義	3.4.1a2 中的值
1	1-8	PY_MAJOR_VERSION	0x03
2	9-16	PY_MINOR_VERSION	0x04
3	17-24	PY_MICRO_VERSION	0x01
4	25-28	PY_RELEASE_LEVEL	0xA
	29-32	PY_RELEASE_SERIAL	0x2

因此 3.4.1a2 代表 hexversion 0x030401a2、3.10.0 代表 hexversion 0x030a00f0。

使用它進行數值比較，例如 `#if PY_VERSION_HEX >= ...`。

该版本还可通过符号 `Py_Version` 获取。

const unsigned long **Py\_Version**

属于稳定 ABI 自 3.11 版起。編碼⾏單個常數整數的 Python 執行環境版本號，格式與 `PY_VERSION_HEX` 巨集相同。這包含在執行環境使用的 Python 版本。

在 3.11 版新加入。

所有提到的巨集都定義在 `Include/patchlevel.h`。

## 術語表

&gt;&gt;&gt;

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) 內部, 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。
- 建立常數 Ellipsis。

**2to3**

一個試著將 Python 2.x 程式碼轉換成 Python 3.x 程式碼的工具, 它是透過處理大部分的不相容性來達成此目的, 而這些不相容性能透過剖析原始碼和遍歷剖析樹而被檢測出來。

2to3 在標準函式庫中以 `lib2to3` 被使用; 它提供了一個獨立的入口點, 在 `Tools/scripts/2to3`。請參閱 `2to3-reference`。

**abstract base class (抽象基底類)**

抽象基底類 (又稱 ABC) 提供了一種定義界面的方法, 作為 *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 用擬定的 subclass (子類), 它們不繼承自另一個 class (類), 但仍可被 `isinstance()` 及 `issubclass()` 辨識; 請參閱 abc 模組的說明文件。Python 有許多建立的 ABC, 用於資料結構 (在 `collections.abc` 模組)、數字 (在 `numbers` 模組)、串流 (在 `io` 模組) 及 import 尋檢器和載入器 (在 `importlib.abc` 模組)。你可以使用 abc 模組建立自己的 ABC。

**annotation (註釋)**

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

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

請參閱 *variable annotation*、*function annotation*、**PEP 484** 和 **PEP 526**, 這些章節皆有此功能的說明。關於註釋的最佳實踐方法也請參閱 `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, 記住位置執行狀態 (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。

### 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 (二進制檔案)**

*file object* 能够读写字节型对象。二进制文件的例子包括以二进制模式 ('rb', 'wb' 或 'rb+') 打开的文件、`sys.stdin.buffer`、`sys.stdout.buffer` 以及 `io.BytesIO` 和 `gzip.GzipFile` 的实例。

另請參閱 *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, 借用引用是指一种对象引用, 使用该对象的代码并不持有该引用。如果对象被销毁则它就会变成一个悬空指针。例如, 垃圾回收器可以移除对象的最后一个 *strong reference* 来销毁它。

對 *borrowed reference* 呼叫 `Py_INCREF()` 以將它原地 (in-place) 轉為 *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毀。 `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制資料的各種運算; 這些運算包括壓縮、儲存至二進制檔案和透過 `socket` (插座) 發送。

有些運算需要二進制資料是可變的。明文文件通常會將這些物件稱為「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制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 直譯器中的內部表示法。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 (可以不用從原始碼重新編譯位元組碼)。這種「中間語言 (intermediate language)」據說是運行在一個 *virtual machine* (虛擬機器) 上, 該虛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 虛擬機器之間運作的, 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。

**complex number (複數)**

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

**context manager (情境管理器)**

在 `with` 语句中通过定义 `__enter__()` 和 `__exit__()` 方法来控制环境状态的对象。参见 [PEP 343](#)。

**context variable (情境變數)**

一個變數，其值可以根據上下文的情境而有所不同。這類似執行緒區域儲存區 (Thread-Local Storage)，在其中，一個變數在每個執行緒可能具有不同的值。然而，關於情境變數，在一個執行緒中可能會有多個情境，而情境變數的主要用途，是在行的非同步任務 (concurrent asynchronous task) 中，對於變數狀態的追蹤。請參 [contextvars](#)。

**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訪問各個項目時，最後一個索引的變化最快。然而，在 Fortran contiguous 陣列中，第一個索引的變化最快。

**coroutine (協程)**

協程是副程式 (subroutine) 的一種更廣義的形式。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，可能包含 `await`、`async for` 和 `async with` 關鍵字。這些關鍵字由 [PEP 492](#) 引入。

**CPython**

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

**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。關於裝飾器的更多內容，請參函式定義和 class 定義的明文件。

**descriptor (描述器)**

任何定义了 `__get__()`、`__set__()` 或 `__delete__()` 方法的对象。当一个类属性为描述器时，它的特殊绑定行为就会在属性查找时被触发。通常情况下，使用 `a.b` 来获取、设置或删除一个属性时会在 `a` 类的字典中查找名称为 `b` 的对象，但如果 `b` 是一个描述器，则会调用对应的描述器方法。理解描述器的概念是更深层次理解 Python 的关键，因为这是许多重要特性的基础，包括函数、方法、特征属性、类方法、静态方法以及对超类的引用等等。

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)**

一个关联数组，其中的任意键都映射到相应的值。键可以是任何具有 `__hash__()` 和 `__eq__()` 方法的对象。在 Perl 中称为 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` 映射到值 `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。若要限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)**

作為類、函數或模組之內的第一个表达式出現的字符串字面值。它在代碼被執行時會被忽略，但會被編譯器識別並放入所在類、函數或模組的 `__doc__` 屬性中。由於它可用於代碼內省，因此是存放對象的文檔的規範位置。

**duck-typing (鴨子型)**

一種程式設計風格，它不是藉由檢查一個物件的型來確定它是否具有正確的介面；取而代之的是，method 或屬性會單純地被呼叫或使用。（「如果它看起來像一鴨子而且叫起來像一鴨子，那它一定是一鴨子。」）因調介面而非特定型，精心設計的程式碼能讓多形替代 (polymorphic substitution) 來增進它的靈活性。鴨子型要避免使用 `type()` 或 `isinstance()` 進行測試。（但是請注意，鴨子型可以用抽象基底類 (*abstract base class*) 來補充。）然而，它通常會用 `hasattr()` 測試，或是 *EAFP* 程式設計風格。

**EAFP**

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

**expression (運算式)**

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

**extension module (擴充模組)**

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

**f-string (f 字串)**

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

**file object (檔案物件)**

對外公開面向文件的 API（帶有 `read()` 或 `write()` 等方法）以使用下层资源的对象。根据其创建方式的不同，文件对象可以处理对真实磁盘文件、其他类型的存储或通信设备的访问（例如标准输入/输出、内存缓冲区、套接字、管道等）。文件对象也被称为文件型对象或流。

實際上，有三種檔案物件：原始的二進制檔案、緩衝的二進制檔案和文字檔案。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 啟動時由 `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）。

從 Python 3.3 開始，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`。

請參閱 PEP 302、PEP 420 和 PEP 451 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` 的計算結果是 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，因為 -2.75 被向下無條件舍去。請參閱 PEP 238。

### 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。另請參閱 *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。

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

### \_\_future\_\_

`future` 陳述式：`from __future__ import <feature>`，會指示編譯器使用那些在 Python 未來的發布版本中將成標準的語法或語義，來編譯當前的模組。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 (生成器)

一個會回傳 *generator iterator* (生成器代器) 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成器函式，但在某些情境中，也可能是表示生成器代器。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 (生成器代器)

一個由 *generator* (生成器) 函式所建立的物件。

每個 `yield` 會暫停處理程序，記住位置執行狀態 (包括區域變數及擱置中的 `try` 陳述式)。當生成器代器回復時，它會從停止的地方繼續執行 (與那些每次調用時都要重新開始的函式有所不同)。

### generator expression (生成器運算式)

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))           # sum of squares 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](#)、[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。

過去對於建立「無限制執行緒」直譯器 (以更高的精細度鎖定共享資料的直譯器) 的努力未成功，因在一般的單一處理器情況下，效能會有所損失。一般認為，若要克服這個效能問題，會使實作變得雜許多，進而付出更高的維護成本。

**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)**

一個對象如果具有在其生命期內絕不改變的哈希值 (它需要有 `__hash__()` 方法)，並可以同其他對象進行比較 (它需要有 `__eq__()` 方法) 就被稱為可哈希對象。可哈希對象必須具有相同的哈希值比較結果才會相等。

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。

**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。它們在需要定雜值的地方，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，立即執行它們且看到它們的結果。只要啟動 python，不需要任何引數（可能藉由從你的電腦的主選單選擇它）。這是測試新想法或檢查模塊和包的非常巨大的方法（請記住 `help(x)`）。

**interpreted (直譯的)**

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

**interpreter shutdown (直譯器關閉)**

當 Python 直譯器被要求關閉時，它會進入一個特殊階段，在此它逐漸釋放所有被配置的資源，例如模組和各種關鍵內部結構。它也會多次呼叫垃圾回收器 (garbage collector)。這能觸發使用者自定的解構函式 (destructor) 或弱引用的回呼 (weakref callback)，執行其中的程式碼。在關閉階段被執行的程式碼會遇到各種例外，因為它所依賴的資源可能不再有作用了（常見的例子是函式庫模組或是警告機制）。

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

**iterable (可迭代物件)**

一種能夠逐個返回其成員項的對象。可迭代對象的例子包括所有序列類型（如 `list`、`str` 和 `tuple` 等）以及某些非序列類型如 `dict`、文件對象以及任何定義了 `__iter__()` 方法或實現了 *sequence* 語義的 `__getitem__()` 方法的自定義類的對象。

可迭代對象可被用於 `for` 循環以及許多其他需要一個序列的地方 (`zip()`、`map()`、...)。當一個可迭代對象作為參數被傳給內置函數 `iter()` 時，它會返回該對象的迭代器。這種迭代器適用於對值集合的一次性遍歷。在使用可迭代對象時，你通常不需要調用 `iter()` 或者自己處理迭代器對象。`for` 語句會自動為你處理那些操作，創建一個臨時的未命名變量用來在循環期間保存迭代器。參見 [iterator](#)、[sequence](#) 和 [generator](#)。

**iterator (迭代器)**

用來表示一連串數據流的對象。重複調用迭代器的 `__next__()` 方法（或將其傳給內置函數 `next()`）將逐個返回流中的項。當沒有數據可用時則將引發 `StopIteration` 異常。到這時迭代器對象中的數據項已耗盡，繼續調用其 `__next__()` 方法只會再次引發 `StopIteration`。迭代器必須具有 `__iter__()` 方法用來返回該迭代器對象自身，因此迭代器必定也是可迭代對象，可被用於其他可迭代對象適用的大部分場合。一個顯著的例外是那些會多次重複訪問迭代項的代碼。容器對象（例如 `list`）在你每次將其傳入 `iter()` 函數或是在 `for` 循環中使用時都會產生一個新的迭代器。如果在此情況下你嘗試用迭代器則會返回在之前迭代過程中被耗盡的同一迭代器對象，使其看起來就像是一個空容器。

在 `typeiter` 文中可以找到更多資訊。

**CPython 實作細節：**CPython 沒有統一應用迭代器定義 `__iter__()` 的要求。

**key function (鍵函式)**

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

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

有幾種方法可以建立一個鍵函式。例如，`str.lower()` method 可以作為不分大小寫排序的鍵函式。或者，一個鍵函式也可以從 `lambda` 運算式被建造，例如 `lambda r: (r[0], r[2])`。另外，`operator.attrgetter()`、`operator.itemgetter()` 和 `operator.methodcaller()` 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，請參見 [如何排序](#)。

**keyword argument (關鍵字引數)**

請參見 [argument \(引數\)](#)。

**lambda**

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



**LBYL**

Look before you leap. (三思而後行。) 這種編碼風格會在進行呼叫或查找之前，明確地測試先條件。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。

**list (串列)**

一種 Python 內置 *sequence*。雖然名為列表，但它更类似于其他语言中的数组而非链表，因为访问元素的时间复杂度为  $O(1)$ 。

**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。它必須定義一個名 `load_module()` 的 method (方法)。載入器通常是被 *finder* (尋檢器) 回傳。更多細節請參 PEP 302，關於 *abstract base class* (抽象基底類)，請參 `importlib.abc.Loader`。

**locale encoding (區域編碼)**

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

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

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

`locale.getencoding()` 可被用来获取语言区域编码格式。

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。關於第 2.3 版至今，Python 直譯器所使用的演算法細節，請參 Python 2.3 版方法解析順序。

**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。

**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。

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

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

有些具名元組是內置類型 (比如上面的例子)。此外，具名元組還可通過常規類定義從 tuple 繼承並定義指定名稱的字段的方式來創建。這樣的類可以手工編號，或者也可以通過繼承 `typing.NamedTuple`，或者使用工廠函數 `collections.namedtuple()` 來創建。後一種方式還會添加一些手工編寫或內置的具名元組所沒有的額外方法。

**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 PEP 420 *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。

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，在預設情況下，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)**

對目前已被應用於所有類對象的類形式的舊稱謂。在較早的 Python 版本中，只有新式類能够使用 Python 新增的更靈活我，如 `__slots__`、描述器、特征屬性、`__getattr__()`、類方法和靜態方法等。

**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)。

**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)，或在某些情況下指示多個引數。共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別、`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)**

一種可調用對象，它在知道如何查找特定 *path entry* 中的模块的情况下能够使用 `sys.path_hooks` 列表返回一个 *path entry finder*。

**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

的物件可以被轉成 `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。

這個過程使得標準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__
```

(繼續下一頁)



(繼續上一頁)

```
'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 (*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)。參照計數通常在 Python 程式碼中看不到，但它 [是 CPython](#) 實作的一個關鍵元素。程式設計師可以呼叫 `getrefcount()` 函式來回傳一個特定物件的參照計數。

### regular package (正規套件)

一個傳統的 *package* (套件)，例如一個包含 `__init__.py` 檔案的目錄。

另請參 [namespace package](#) (命名空間套件)。

### \_\_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況。

### sequence (序列)

一種 *iterable*，它支持通过 `__getitem__()` 特殊方法来使用整数索引进行高效的元素访问，并定义了一个返回序列长度的 `__len__()` 方法。内置的序列类型有 `list`, `str`, `tuple` 和 `bytes` 等。请注意虽然 `dict` 也支持 `__getitem__()` 和 `__len__()`，但它被归类为映射而非序列，因为它使用任意 *immutable* 键而不是整数来查找元素。

`collections.abc.Sequence` 抽象基类定义了一个更丰富的接口，它在 `__getitem__()` 和 `__len__()` 之外，还添加了 `count()`, `index()`, `__contains__()` 和 `__reversed__()`。实现此扩展接口的类型可以使用 `register()` 来显式地注册。要获取有关通用序列方法的更多文档，请参阅 [通用序列操作](#)。

### 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。

### 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。

### 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 -- 静态类型检查器

读取 Python 代码并进行分析，以查找问题例如拼写错误的外部工具。另请参阅 [类型提示](#) 以及 `typing` 模块。

**strong reference (參照)**

在 Python 的 C API 中，強引用是指為持有引用的代碼所擁有的對象的引用。在創建引用時可通過調用 `Py_INCREF()` 來獲取強引用而在刪除引用時可通過 `Py_DECREF()` 來釋放它。

`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)。

**text encoding (文字編碼)**

Python 中的字串是一個 Unicode 碼點 (code point) 的序列 (範圍在 U+0000 -- U+10FFFF 之間)。若要儲存或傳送一個字串，它必須被序列化一個位元組序列。

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](#) (二進制檔案)，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。

**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別有用。

**type (型)**

一個 Python 物件的型定了它是什麼類型的物件；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。

類型提示是可选的而不是 Python 的强制要求，但它們對靜態類型檢查器很有用處。它們還能協助 IDE 實現代碼補全與重構。

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 應用程式的行產生干擾。

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。

### 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示字元後輸入「`import this`」來找到它。



---

### 關於這些📄明文件

---

這些📄明文件是透過 [Sphinx](#)（一個專📄 Python 📄明文件所撰寫的文件處理器）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 標準函式庫和 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！



## 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，目的是作一種稱作 ABC 語言的後繼者。儘管 Python 包含了許多來自其他人的貢獻，Guido 仍是其主要作者。

1995 年，Guido 在維吉尼亞州雷斯頓的國家創新研究公司（CNRI，見 <https://www.cnri.reston.va.us/>）繼續他在 Python 的工作，在那發行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（現 Zope Corporation；見 <https://www.zope.org/>）。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 相關的智慧財產權而創立的非營利組織。Zope Corporation 是 PSF 的一個贊助會員。

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

發行版本	源自	年份	擁有者	GPL 相容性？
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	否
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

**備註：**GPL 相容不表示我們是在 GPL 下發行 Python。不像 GPL，所有的 Python 授權都可以讓您發行修改後的版本，但不一定要使您的變更成開源。GPL 相容的授權使得 Python 可以結合其他在 GPL 下發行的軟體一起使用；但其它的授權則不行。

感謝許多的外部志工，在 Guido 指導下的付出，使得這些版本的發行可能。

## C.2 關於存取或以其他方式使用 Python 的合約條款

Python 軟體和說明文件的授權是基於 *PSF 授權合約*。

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

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

### C.2.1 用於 PYTHON 3.11.11 的 PSF 授權合約

1. This LICENSE AGREEMENT is between the Python Software Foundation,  
→ ("PSF"), and  
the Individual or Organization ("Licensee") accessing and otherwise  
→ using Python  
3.11.11 software in source or binary form and its associated  
→ documentation.
2. Subject to the terms and conditions of this License Agreement, PSF  
→ hereby  
grants Licensee a nonexclusive, royalty-free, world-wide license to  
→ reproduce,  
analyze, test, perform and/or display publicly, prepare derivative  
→ works,  
distribute, and otherwise use Python 3.11.11 alone or in any derivative  
version, provided, however, that PSF's License Agreement and PSF's  
→ notice of  
copyright, i.e., "Copyright © 2001–2023 Python Software Foundation; All  
→ Rights  
Reserved" are retained in Python 3.11.11 alone or in any derivative  
→ version  
prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or  
incorporates Python 3.11.11 or any part thereof, and wants to make the  
derivative work available to others as provided herein, then Licensee  
→ hereby  
agrees to include in any such work a brief summary of the changes made  
→ to Python  
3.11.11.
4. PSF is making Python 3.11.11 available to Licensee on an "AS IS" basis.  
PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY  
→ OF  
EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY  
→ REPRESENTATION OR  
WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR  
→ THAT THE  
USE OF PYTHON 3.11.11 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.11.11  
FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A  
→ RESULT OF



MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.11.11, OR ANY  
 ↳DERIVATIVE  
 THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

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

7. Nothing in this License Agreement shall be deemed to create any  
 ↳relationship  
 of agency, partnership, or joint venture between PSF and Licensee. ↳  
 ↳This License  
 Agreement does not grant permission to use PSF trademarks or trade name ↳  
 ↳in a  
 trademark sense to endorse or promote products or services of Licensee, ↳  
 ↳or any  
 third party.

8. By copying, installing or otherwise using Python 3.11.11, Licensee ↳  
 ↳agrees  
 to be bound by the terms and conditions of this License Agreement.

## C.2.2 用於 PYTHON 2.0 的 BEOPEN.COM 授權合約

### BEOPEN PYTHON 開源授權合約第 1 版

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

(繼續下一頁)

(繼續上一頁)

<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 用於 PYTHON 1.6.1 的 CNRI 授權合約

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the internet using the following URL: <http://hdl.handle.net/1895.22/1013>."
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement

(繼續下一頁)

(繼續上一頁)

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 用於 PYTHON 0.9.0 至 1.2 的 CWI 授權合約

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

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

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

### C.2.5 用於 PYTHON 3.11.11 <sub>F</sub>明文件<sub>F</sub>程式碼的 ZERO-CLAUSE BSD 授權

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<sub>F</sub>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<sub>F</sub>版本中所收<sub>F</sub>的第三方軟體。

### C.3.1 Mersenne Twister

`_random` 模組包含了以 <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html> 的下載內容為基礎的程式碼。以下是原始程式碼的完整聲明：

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

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

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

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote
   products derived from this software without specific prior written
   permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

### C.3.2 Sockets

`socket` 使用了 `getaddrinfo()` 和 `getnameinfo()` WIDE 项目的不同源文件中: <https://www.wide.ad.jp/>

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
```

(繼續下一頁)

(繼續上一頁)

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

asynchat 和 asyncore 模組包含以下聲明：

Copyright 1996 by Sam Rushing

All Rights Reserved

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

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

### C.3.4 Cookie 管理

http.cookies 模組包含以下聲明：

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

(繼續下一頁)

(繼續上一頁)

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, 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 模組包含以下聲明：

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.

(繼續下一頁)



(繼續上一頁)

```

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 MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

### C.3.8 test\_epoll

test.test\_epoll 模块包含以下说明:

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

(繼續下一頁)

(繼續上一頁)

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

(繼續下一頁)

(繼續上一頁)

```

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<sub>F</sub>。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

如果操作系统提供支持，则 `hashlib`, `posix`, `ssl`, `crypt` 模块会使用 OpenSSL 库来提高性能。此外，Python 的 Windows 和 macOS 安装程序可能会包括 OpenSSL 库的副本，所以我们也在此包括一份 OpenSSL 许可证的副本。对于 OpenSSL 3.0 版及其后续衍生版本，均使用 Apache 许可证 v2:

```

                        Apache License
                        Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all

```

(繼續下一頁)

(繼續上一頁)

other entities that control, are controlled by, or are under common control with that entity. For the purposes of this definition, "control" means (i) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (ii) ownership of fifty percent (50%) or more of the outstanding shares, or (iii) beneficial ownership of such entity.

"You" (or "Your") shall mean an individual or Legal Entity exercising permissions granted by this License.

"Source" form shall mean the preferred form for making modifications, including but not limited to software source code, documentation source, and configuration files.

"Object" form shall mean any form resulting from mechanical transformation or translation of a Source form, including but not limited to compiled object code, generated documentation, and conversions to other media types.

"Work" shall mean the work of authorship, whether in Source or Object form, made available under the License, as indicated by a copyright notice that is included in or attached to the work (an example is provided in the Appendix below).

"Derivative Works" shall mean any work, whether in Source or Object form, that is based on (or derived from) the Work and for which the editorial revisions, annotations, elaborations, or other modifications represent, as a whole, an original work of authorship. For the purposes of this License, Derivative Works shall not include works that remain separable from, or merely link (or bind by name) to the interfaces of, the Work and Derivative Works thereof.

"Contribution" shall mean any work of authorship, including the original version of the Work and any modifications or additions to that Work or Derivative Works thereof, that is intentionally submitted to Licensor for inclusion in the Work by the copyright owner or by an individual or Legal Entity authorized to submit on behalf of the copyright owner. For the purposes of this definition, "submitted" means any form of electronic, verbal, or written communication sent to the Licensor or its representatives, including but not limited to communication on electronic mailing lists, source code control systems, and issue tracking systems that are managed by, or on behalf of, the Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.
3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made,

(繼續下一頁)

(繼續上一頁)

use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:
  - (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and
  - (b) You must cause any modified files to carry prominent notices stating that You changed the files; and
  - (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and
  - (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed as modifying the License.

You may add Your own copyright statement to Your modifications and may provide additional or different license terms and conditions for use, reproduction, or distribution of Your modifications, or for any such Derivative Works as a whole, provided Your use, reproduction, and distribution of the Work otherwise complies with the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise, any Contribution intentionally submitted for inclusion in the Work by You to the Licensor shall be under the terms and conditions of this License, without any additional terms or conditions. Notwithstanding the above, nothing herein shall supersede or modify the terms of any separate license agreement you may have executed with Licensor regarding such Contributions.

(繼續下一頁)

(繼續上一頁)

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

pyexpat 擴展是使用所包括的 expat 源副本來構建的，除非配置了 `--with-system-expat`:

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd  
and Clark Cooper

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

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

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,

(繼續下一頁)



(繼續上一頁)

```
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` 擴充時設定 `--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
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.

(繼續下一頁)

(繼續上一頁)

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly  
jloup@gzip.org

Mark Adler  
madler@alumni.caltech.edu

## C.3.16 cfuhash

tracemalloc 使用的雜表 (hash table) 實作，是以 cfuhash 專案為基礎：

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.

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.

(繼續下一頁)

(繼續上一頁)

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<sub>F</sub>：

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 audioop

audioop 模块使用了 SoX 项目的 g771.c 文件中的基础代码。<https://sourceforge.net/projects/sox/files/sox/12.17.7/sox-12.17.7.tar.gz>

此源代码是 Sun Microsystems, Inc. 的产品并可供无限制地使用。用户可以拷贝或修改此源代码而无须付费。

SUN SOURCE CODE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING THE WARRANTIES OF DESIGN, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.

提供的 Sun 源代码不附带技术支持并且 Sun Microsystems, Inc. 也没有义务协助其使用、排错、修改或增强。

SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT TO THE INFRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS SOFTWARE OR ANY PART THEREOF.

在任何情况下 Sun Microsystems, Inc. 均不对任何收入或利润损失或其他特殊的、间接的和后续的损害负责，即使 Sun 已被告知可能发生此类损害。

Sun Microsystems, Inc. 2550 Garcia Avenue Mountain View, California 94043

### C.3.20 asyncio

asyncio 模块的某些部分来自 uvloop 0.16，它是基于 MIT 许可证发行的：

```
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.
```

---

### 版權宣告

---

Python 和這份圖明文件的版權：

Copyright © 2001-2023 Python Software Foundation 保留一切權利。

Copyright © 2000 BeOpen.com 保留一切權利。

Copyright © 1995-2000 Corporation for National Research Initiatives 保留一切權利。

Copyright © 1991-1995 Stichting Mathematisch Centrum 保留一切權利。

---

完整的授權條款資訊請參見[沿革與授權](#)。





## 非依字母順序

..., 265  
 2to3, 265  
 >>>, 265  
 \_\_all\_\_ (套件變數), 62  
 \_\_dict\_\_ (模組屬性), 146  
 \_\_doc\_\_ (模組屬性), 146  
 \_\_file\_\_ (模組屬性), 146  
 \_\_future\_\_, 270  
 \_\_import\_\_  
     built-in function (F建函式), 62  
 \_\_loader\_\_ (模組屬性), 146  
 \_\_main\_\_  
     module (模組), 11  
     模組, 170, 181  
 \_\_name\_\_ (模組屬性), 146  
 \_\_package\_\_ (模組屬性), 146  
 \_\_PYENV\_LAUNCHER\_\_, 194, 200  
 \_\_slots\_\_, 277  
 \_frozen (C struct), 65  
 \_inittab (C struct), 65  
 \_inittab.initfunc (C member), 65  
 \_inittab.name (C member), 65  
 \_Py\_c\_diff (C function), 112  
 \_Py\_c\_neg (C function), 112  
 \_Py\_c\_pow (C function), 112  
 \_Py\_c\_prod (C function), 112  
 \_Py\_c\_quot (C function), 112  
 \_Py\_c\_sum (C function), 112  
 \_Py\_InitializeMain (C function), 207  
 \_Py\_NoneStruct (C var), 219  
 \_PyBytes\_Resize (C function), 115  
 \_PyCFunctionFast (C type), 222  
 \_PyCFunctionFastWithKeywords (C type), 222  
 \_PyFrameEvalFunction (C type), 179  
 \_PyInterpreterState\_GetEvalFrameFunc  
     (C function), 179  
 \_PyInterpreterState\_SetEvalFrameFunc  
     (C function), 180  
 \_PyObject\_GetDictPtr (C function), 80  
 \_PyObject\_New (C function), 219  
 \_PyObject\_NewVar (C function), 219  
 \_PyTuple\_Resize (C function), 133

\_thread  
     模組, 176  
 環境變數  
     \_\_PYENV\_LAUNCHER\_\_, 194, 200  
     PATH, 11  
     PYTHONCOERCECLOCALE, 205  
     PYTHONDEBUG, 168, 199  
     PYTHONDEVMODE, 196  
     PYTHONDONTWRITEBYTECODE, 168, 202  
     PYTHONDUMPREFS, 196, 233  
     PYTHONEXECUTABLE, 200  
     PYTHONFAULTHANDLER, 196  
     PYTHONHASHSEED, 168, 197  
     PYTHONHOME, 11, 168, 174, 197  
     PYTHONINSPECT, 169, 197  
     PYTHONIOENCODING, 171, 201  
     PYTHONLEGACYWINDOWSFSENCODING, 169,  
         191  
     PYTHONLEGACYWINDOWSTDIO, 169, 198  
     PYTHONMALLOC, 210, 213, 215  
     PYTHONMALLOC` (例 如:  
         `PYTHONMALLOC=malloc`, 216  
     PYTHONMALLOCSTATS, 198, 210  
     PYTHONNODEBUGRANGES, 195  
     PYTHONNOUSERSITE, 169, 202  
     PYTHONOPTIMIZE, 169, 199  
     PYTHONPATH, 11, 168, 198  
     PYTHONPLATLIBDIR, 198  
     PYTHONPROFILEIMPORTTIME, 197  
     PYTHONPYCACHEPREFIX, 200  
     PYTHONSAFEPATH, 194  
     PYTHONTRACEMALLOC, 201  
     PYTHONUNBUFFERED, 169, 195  
     PYTHONUTF8, 191, 205  
     PYTHONVERBOSE, 169, 202  
     PYTHONWARNINGS, 202

## A

abort (C 函数), 62  
 abs  
     built-in function (F建函式), 88  
 abstract base class (抽象基底類F), 265  
 allocfunc (C type), 255  
 annotation (F釋), 265

argument (引數), 265  
 argv (sys 模組中), 173  
 ascii  
   built-in function (內建函式), 81  
 asynchronous context manager (非同步情境管理器), 266  
 asynchronous generator iterator (非同步生成器迭代器), 266  
 asynchronous generator (非同步生成器), 266  
 asynchronous iterable (非同步可迭代物件), 266  
 asynchronous iterator (非同步迭代器), 266  
 attribute (屬性), 266  
 awaitable (可等待物件), 266

## B

BDFL, 267  
 binary file (二進制檔案), 267  
 binaryfunc (C type), 256  
 borrowed reference (借用參照), 267  
 buffer interface (緩衝介面)  
   (請見緩衝協定), 94  
 buffer object (緩衝物件)  
   (請見緩衝協定), 94  
 buffer protocol (緩衝協定), 94  
 built-in function (內建函式)  
   \_\_import\_\_, 62  
   abs, 88  
   classmethod, 224  
   compile (編譯), 63  
   divmod, 88  
   float, 90  
   hash (雜項), 236  
   int, 90  
   len, 92, 135, 138, 140  
   pow, 88, 89  
   repr, 235  
   staticmethod, 224  
   tuple (元組), 136  
 built-in function (內建函式)  
   len, 90  
   tuple (元組), 91  
 builtins (內建)  
   module (模組), 11  
   模組, 170, 181  
 built-in function (內建函式)  
   ascii, 81  
   bytes (位元組), 81  
   hash (雜項), 82  
   len, 82  
   repr, 81  
   type (型別), 82  
 bytearray (位元組串列)  
   object (物件), 115  
 bytecode (位元組碼), 267  
 bytes-like object (類位元組串物件), 267  
 bytes (位元組)  
   built-in function (內建函式), 81

object (物件), 113

## C

callable (可呼叫物件), 267  
 callback (回呼), 267  
 calloc (C 函數), 209  
 Capsule  
   object (物件), 157  
 C-contiguous (C 連續的), 97, 268  
 class variable (類變數), 267  
 classmethod  
   built-in function (內建函式), 224  
 class (類), 267  
 cleanup functions (清理函式), 62  
 close (在 os 模組中), 181  
 CO\_FUTURE\_DIVISION (C var), 42  
 code object (程式碼物件), 143  
 compile (編譯)  
   built-in function (內建函式), 63  
 complex number (複數), 267  
   object (物件), 112  
 context manager (情境管理器), 268  
 context variable (情境變數), 268  
 contiguous (連續的), 97, 268  
 copyright (sys 模組中), 173  
 coroutine function (協程函式), 268  
 coroutine (協程), 268  
 CPython, 268

## D

decorator (裝飾器), 268  
 descrgetfunc (C type), 256  
 descriptor (描述器), 268  
 descrsetfunc (C type), 256  
 destructor (C type), 255  
 dictionary comprehension (字典綜合運算), 268  
 dictionary view (字典檢視), 269  
 dictionary (字典), 268  
   object (物件), 136  
 divmod  
   built-in function (內建函式), 88  
 docstring (說明字串), 269  
 duck-typing (鴨子型別), 269

## E

EAFP, 269  
 EOFError (內建例外), 145  
 exc\_info (在 sys 模組中), 10  
 executable (sys 模組中), 172  
 exit (C 函數), 62  
 expression (運算式), 269  
 extension module (擴充模組), 269

## F

f-string (f 字串), 269  
 file object (檔案物件), 269  
 file-like object (類檔案物件), 269

filesystem encoding and error  
     handler (檔案系統編碼和錯誤處理函式), 269

file (檔案)  
     object (物件), 145

finder (尋檢器), 269

float  
     built-in function (F建函式), 90

floating point (浮點)  
     object (物件), 110

floor division (向下取整除法), 270

Fortran contiguous (Fortran 連續的), 97, 268

free (C 函數), 209

freefunc (C type), 255

freeze utility (凍結工具), 65

frozenset (凍結集合)  
     object (物件), 139

function annotation (函式F釋), 270

function (函式), 270  
     object (物件), 141

## G

garbage collection (垃圾回收), 270

generator expression (F生器運算式), 270

generator iterator (F生器F代器), 270

generator (F生器), 270

generic function (泛型函式), 271

generic type (泛型型F), 271

getattrfunc (C type), 256

getattrofunc (C type), 256

getbufferproc (C type), 256

getiterfunc (C type), 256

GIL, 271

global interpreter lock (全域直譯器鎖), 174, 271

## H

hash-based pyc (雜F架構的 pyc), 271

hashable (可雜F的), 271

hashfunc (C type), 256

hash (雜F)  
     built-in function (F建函式), 236  
     bulit-in function (F建函式), 82

## I

IDLE, 271

immutable (不可變物件), 271

import path (引入路徑), 271

importer (引入器), 271

importing (引入), 271

incr\_item(), 10, 11

initproc (C type), 255

inquiry (C type), 261

instancemethod  
     object (物件), 142

int  
     built-in function (F建函式), 90

integer (整數)  
     object (物件), 107

interactive (互動的), 272

interpreted (直譯的), 272

interpreter lock (直譯器鎖), 174

interpreter shutdown (直譯器關閉), 272

iterable (可F代物件), 272

iterator (F代器), 272

iternextfunc (C type), 256

## K

key function (鍵函式), 272

KeyboardInterrupt (F建例外), 51

keyword argument (關鍵字引數), 272

## L

lambda, 272

LBYL, 273

len  
     built-in function (F建函式), 92, 135, 138, 140  
     built-in function (內建函式), 90  
     bulit-in function (F建函式), 82

lenfunc (C type), 256

list comprehension (串列綜合運算), 273

list (串列), 273  
     object (物件), 135

loader (載入器), 273

locale encoding (區域編碼), 273

lock, interpreter (鎖、直譯器), 174

long integer (長整數)  
     object (物件), 107

LONG\_MAX (C 宏), 108

## M

magic  
     method (方法), 273

magic method (魔術方法), 273

main(), 171, 173

malloc (C 函數), 209

mapping (對映), 273  
     object (物件), 136

memoryview (記憶體視圖)  
     object (物件), 155

meta path finder (元路徑尋檢器), 273

metaclass (元類F), 273

METH\_CLASS (C macro), 224

METH\_COEXIST (C macro), 224

METH\_FASTCALL (C macro), 223

METH\_KEYWORDS (C macro), 223

METH\_METHOD (C macro), 223

METH\_NOARGS (C macro), 223

METH\_O (C macro), 223

METH\_STATIC (C macro), 224

METH\_VARARGS (C macro), 223

method resolution order (方法解析順序), 274

MethodType (types 模組中), 141, 142, 146

method (方法), 273  
     magic, 273  
     object (物件), 142  
     special, 277  
 module spec (模組規格), 274  
 modules (sys 模組中), 62, 170  
 module (模組), 274  
     \_\_main\_\_, 11  
     builtins (F建), 11  
     object (模組), 146  
     search (搜尋) path (路徑), 11  
     signal (訊號), 51  
     sys, 11  
 MRO, 274  
 mutable (可變物件), 274

## N

named tuple (附名元組), 274  
 namespace package (命名空間套件), 274  
 namespace (命名空間), 274  
 nested scope (巢狀作用域), 274  
 new-style class (新式類F), 274  
 newfunc (C type), 255  
 None  
     object (物件), 107  
 numeric (數值)  
     object (物件), 107

## O

object (模組)  
     module (模組), 146  
 object (物件), 275  
     bytearray (位元組串列), 115  
     bytes (位元組), 113  
     Capsule, 157  
     code (程式碼), 143  
     complex number (F數), 112  
     dictionary (字典), 136  
     file (檔案), 145  
     floating point (浮點), 110  
     frozenset (凍結集合), 139  
     function (函式), 141  
     instancemethod, 142  
     integer (整數), 107  
     list (串列), 135  
     long integer (長整數), 107  
     mapping (對映), 136  
     memoryview (記憶體視圖), 155  
     method (方法), 142  
     None, 107  
     numeric (數值), 107  
     sequence (序列), 113  
     set (集合), 139  
     tuple (元組), 133  
     type (型F), 6, 103  
 objobjargproc (C type), 257  
 objobjproc (C type), 256  
 OverflowError (內建例外), 108, 109

## P

package variable (套件變數)  
     \_\_all\_\_, 62  
 package (套件), 275  
 parameter (參數), 275  
 PATH, 11  
 path based finder (基於路徑的尋檢器), 275  
 path entry finder (路徑項目尋檢器), 275  
 path entry hook (路徑項目F), 275  
 path entry (路徑項目), 275  
 path-like object (類路徑物件), 275  
 path (sys 模組中), 11, 170, 172  
 path (路徑)  
     module (模組) search (搜尋), 11  
     模組 search (搜尋), 170, 172  
 PEP, 276  
 platform (sys 模組中), 173  
 portion (部分), 276  
 positional argument (位置引數), 276  
 pow  
     built-in function (F建函式), 88, 89  
 provisional API (暫行 API), 276  
 provisional package (暫行套件), 276  
 Py\_ABS (C macro), 4  
 Py\_AddPendingCall (C function), 182  
 Py\_ALWAYS\_INLINE (C macro), 4  
 Py\_AtExit (C function), 62  
 Py\_AuditHookFunction (C type), 61  
 Py\_BEGIN\_ALLOW\_THREADS (C macro), 178  
 Py\_BEGIN\_ALLOW\_THREADS (C 宏), 175  
 Py\_BLOCK\_THREADS (C macro), 178  
 Py\_buffer (C type), 95  
 Py\_buffer.buf (C member), 95  
 Py\_buffer.format (C member), 95  
 Py\_buffer.internal (C member), 96  
 Py\_buffer.itemsize (C member), 95  
 Py\_buffer.len (C member), 95  
 Py\_buffer.ndim (C member), 95  
 Py\_buffer.obj (C member), 95  
 Py\_buffer.readonly (C member), 95  
 Py\_buffer.shape (C member), 95  
 Py\_buffer.strides (C member), 96  
 Py\_buffer.suboffsets (C member), 96  
 Py\_BuildValue (C function), 72  
 Py\_BytesMain (C function), 39  
 Py\_BytesWarningFlag (C var), 168  
 Py\_CHARMASK (C macro), 5  
 Py\_CLEAR (C function), 44  
 Py\_CompileString (C function), 41  
 Py\_CompileString (C 函數), 42  
 Py\_CompileStringExFlags (C function), 41  
 Py\_CompileStringFlags (C function), 41  
 Py\_CompileStringObject (C function), 41  
 Py\_complex (C type), 112  
 Py\_DEBUG (C macro), 12  
 Py\_DebugFlag (C var), 168  
 Py\_DecodeLocale (C function), 58  
 Py\_DECREF (C function), 44

- Py\_DecRef (*C function*), 44
- Py\_DECREF (*C 函数*), 7
- Py\_DEPRECATED (*C macro*), 5
- Py\_DontWriteBytecodeFlag (*C var*), 168
- Py\_Ellipsis (*C var*), 155
- Py\_EncodeLocale (*C function*), 59
- Py\_END\_ALLOW\_THREADS (*C macro*), 178
- Py\_END\_ALLOW\_THREADS (*C 宏*), 175
- Py\_EndInterpreter (*C function*), 181
- Py\_EnterRecursiveCall (*C function*), 54
- Py\_EQ (*C macro*), 243
- Py\_eval\_input (*C var*), 42
- Py\_Exit (*C function*), 62
- Py\_ExitStatusException (*C function*), 189
- Py\_False (*C var*), 110
- Py\_FatalError (*C function*), 62
- Py\_FatalError(), 173
- Py\_FdIsInteractive (*C function*), 57
- Py\_file\_input (*C var*), 42
- Py\_Finalize (*C function*), 170
- Py\_FinalizeEx (*C function*), 170
- Py\_FinalizeEx (*C 函数*), 62, 170, 181
- Py\_FrozenFlag (*C var*), 168
- Py\_GE (*C macro*), 243
- Py\_GenericAlias (*C function*), 165
- Py\_GenericAliasType (*C var*), 165
- Py\_GetArgcArgv (*C function*), 206
- Py\_GetBuildInfo (*C function*), 173
- Py\_GetCompiler (*C function*), 173
- Py\_GetCopyright (*C function*), 173
- Py\_GETENV (*C macro*), 5
- Py\_GetExecPrefix (*C function*), 171
- Py\_GetExecPrefix (*C 函数*), 11
- Py\_GetPath (*C function*), 172
- Py\_GetPath (*C 函数*), 11
- Py\_GetPath(), 171, 172
- Py\_GetPlatform (*C function*), 173
- Py\_GetPrefix (*C function*), 171
- Py\_GetPrefix (*C 函数*), 11
- Py\_GetProgramFullPath (*C function*), 172
- Py\_GetProgramFullPath (*C 函数*), 11
- Py\_GetProgramName (*C function*), 171
- Py\_GetPythonHome (*C function*), 174
- Py\_GetVersion (*C function*), 172
- Py\_GT (*C macro*), 243
- Py\_hash\_t (*C type*), 76
- Py\_HashRandomizationFlag (*C var*), 168
- Py\_IgnoreEnvironmentFlag (*C var*), 168
- Py\_INCREF (*C function*), 43
- Py\_IncRef (*C function*), 44
- Py\_INCREF (*C 函数*), 7
- Py\_Initialize (*C function*), 170
- Py\_Initialize (*C 函数*), 11, 181
- Py\_Initialize(), 171
- Py\_InitializeEx (*C function*), 170
- Py\_InitializeFromConfig (*C function*), 203
- Py\_InspectFlag (*C var*), 168
- Py\_InteractiveFlag (*C var*), 169
- Py\_Is (*C function*), 220
- Py\_IS\_TYPE (*C function*), 221
- Py\_IsFalse (*C function*), 220
- Py\_IsInitialized (*C function*), 170
- Py\_IsInitialized (*C 函数*), 11
- Py\_IsNone (*C function*), 220
- Py\_IsolatedFlag (*C var*), 169
- Py\_IsTrue (*C function*), 220
- Py\_LE (*C macro*), 243
- Py\_LeaveRecursiveCall (*C function*), 54
- Py\_LegacyWindowsFSEncodingFlag (*C var*), 169
- Py\_LegacyWindowsStdioFlag (*C var*), 169
- Py\_LIMITED\_API (*C macro*), 13
- Py\_LT (*C macro*), 243
- Py\_Main (*C function*), 39
- PY\_MAJOR\_VERSION (*C macro*), 263
- Py\_MAX (*C macro*), 5
- Py\_MEMBER\_SIZE (*C macro*), 5
- PY\_MICRO\_VERSION (*C macro*), 263
- Py\_MIN (*C macro*), 5
- PY\_MINOR\_VERSION (*C macro*), 263
- Py\_mod\_create (*C macro*), 149
- Py\_mod\_exec (*C macro*), 149
- Py\_NE (*C macro*), 243
- Py\_NewInterpreter (*C function*), 181
- Py\_NewRef (*C function*), 43
- Py\_NO\_INLINE (*C macro*), 5
- Py\_None (*C var*), 107
- Py\_NoSiteFlag (*C var*), 169
- Py\_NotImplemented (*C var*), 79
- Py\_NoUserSiteDirectory (*C var*), 169
- Py\_OpenCodeHookFunction (*C type*), 145
- Py\_OptimizeFlag (*C var*), 169
- Py\_PreInitialize (*C function*), 192
- Py\_PreInitializeFromArgs (*C function*), 192
- Py\_PreInitializeFromBytesArgs (*C function*), 192
- Py\_PRINT\_RAW (*C macro*), 79
- Py\_PRINT\_RAW (*C 宏*), 145
- Py\_QuietFlag (*C var*), 169
- Py\_REFCNT (*C function*), 221
- PY\_RELEASE\_LEVEL (*C macro*), 263
- PY\_RELEASE\_SERIAL (*C macro*), 263
- Py\_ReprEnter (*C function*), 54
- Py\_ReprLeave (*C function*), 54
- Py\_RETURN\_FALSE (*C macro*), 110
- Py\_RETURN\_NONE (*C macro*), 107
- Py\_RETURN\_NOTIMPLEMENTED (*C macro*), 79
- Py\_RETURN\_RICHCOMPARE (*C macro*), 243
- Py\_RETURN\_TRUE (*C macro*), 110
- Py\_RunMain (*C function*), 206
- Py\_SET\_REFCNT (*C function*), 221
- Py\_SET\_SIZE (*C function*), 221
- Py\_SET\_TYPE (*C function*), 221
- Py\_SetPath (*C function*), 172
- Py\_SetPath(), 172
- Py\_SetProgramName (*C function*), 171



- Py\_SetProgramName (C 函数), 11
- Py\_SetProgramName (), 170172
- Py\_SetPythonHome (C function), 174
- Py\_SetStandardStreamEncoding (C function), 171
- Py\_single\_input (C var), 42
- Py\_SIZE (C function), 221
- Py\_ssize\_t (C type), 9
- PY\_SSIZE\_T\_MAX (C 宏), 109
- Py\_STRINGIFY (C macro), 5
- Py\_TPFLAGS\_BASE\_EXC\_SUBCLASS (C macro), 239
- Py\_TPFLAGS\_BASETYPE (C macro), 238
- Py\_TPFLAGS\_BYTES\_SUBCLASS (C macro), 239
- Py\_TPFLAGS\_DEFAULT (C macro), 239
- Py\_TPFLAGS\_DICT\_SUBCLASS (C macro), 239
- Py\_TPFLAGS\_DISALLOW\_INSTANTIATION (C macro), 240
- Py\_TPFLAGS\_HAVE\_FINALIZE (C macro), 239
- Py\_TPFLAGS\_HAVE\_GC (C macro), 238
- Py\_TPFLAGS\_HAVE\_VECTORCALL (C macro), 240
- Py\_TPFLAGS\_HEAPTYPE (C macro), 238
- Py\_TPFLAGS\_IMMUTABLETYPE (C macro), 240
- Py\_TPFLAGS\_LIST\_SUBCLASS (C macro), 239
- Py\_TPFLAGS\_LONG\_SUBCLASS (C macro), 239
- Py\_TPFLAGS\_MAPPING (C macro), 240
- Py\_TPFLAGS\_METHOD\_DESCRIPTOR (C macro), 239
- Py\_TPFLAGS\_READY (C macro), 238
- Py\_TPFLAGS\_READYING (C macro), 238
- Py\_TPFLAGS\_SEQUENCE (C macro), 240
- Py\_TPFLAGS\_TUPLE\_SUBCLASS (C macro), 239
- Py\_TPFLAGS\_TYPE\_SUBCLASS (C macro), 239
- Py\_TPFLAGS\_UNICODE\_SUBCLASS (C macro), 239
- Py\_tracefunc (C type), 183
- Py\_True (C var), 110
- Py\_tss\_NEEDS\_INIT (C macro), 185
- Py\_tss\_t (C type), 185
- Py\_TYPE (C function), 221
- Py\_UCS1 (C type), 117
- Py\_UCS2 (C type), 117
- Py\_UCS4 (C type), 117
- Py\_uhash\_t (C type), 76
- Py\_UNBLOCK\_THREADS (C macro), 178
- Py\_UnbufferedStdioFlag (C var), 169
- Py\_UNICODE (C type), 117
- Py\_UNICODE\_IS\_HIGH\_SURROGATE (C macro), 120
- Py\_UNICODE\_IS\_LOW\_SURROGATE (C macro), 120
- Py\_UNICODE\_IS\_SURROGATE (C macro), 120
- Py\_UNICODE\_ISALNUM (C function), 119
- Py\_UNICODE\_ISALPHA (C function), 119
- Py\_UNICODE\_ISDECIMAL (C function), 119
- Py\_UNICODE\_ISDIGIT (C function), 119
- Py\_UNICODE\_ISLINEBREAK (C function), 119
- Py\_UNICODE\_ISLOWER (C function), 119
- Py\_UNICODE\_ISNUMERIC (C function), 119
- Py\_UNICODE\_ISPRINTABLE (C function), 119
- Py\_UNICODE\_ISSPACE (C function), 119
- Py\_UNICODE\_ISTITLE (C function), 119
- Py\_UNICODE\_ISUPPER (C function), 119
- Py\_UNICODE\_JOIN\_SURROGATES (C macro), 120
- Py\_UNICODE\_TODECIMAL (C function), 120
- Py\_UNICODE\_TODIGIT (C function), 120
- Py\_UNICODE\_TOLOWER (C function), 119
- Py\_UNICODE\_TONUMERIC (C function), 120
- Py\_UNICODE\_TOTITLE (C function), 120
- Py\_UNICODE\_Toupper (C function), 120
- Py\_UNREACHABLE (C macro), 5
- Py\_UNUSED (C macro), 6
- Py\_VaBuildValue (C function), 74
- PY\_VECTORCALL\_ARGUMENTS\_OFFSET (C macro), 84
- Py\_VerboseFlag (C var), 169
- Py\_Version (C var), 264
- PY\_VERSION\_HEX (C macro), 263
- Py\_Visit (C function), 260
- Py\_XDECREF (C function), 44
- Py\_XDECREF (C 函数), 11
- Py\_XINCR (C function), 43
- Py\_XNewRef (C function), 43
- PyAIter\_Check (C function), 93
- PyAnySet\_Check (C function), 139
- PyAnySet\_CheckExact (C function), 139
- PyArg\_Parse (C function), 72
- PyArg\_ParseTuple (C function), 71
- PyArg\_ParseTupleAndKeywords (C function), 71
- PyArg\_UnpackTuple (C function), 72
- PyArg\_ValidateKeywordArguments (C function), 71
- PyArg\_VaParse (C function), 71
- PyArg\_VaParseTupleAndKeywords (C function), 71
- PyASCIIObject (C type), 117
- PyAsyncMethods (C type), 254
- PyAsyncMethods.am\_aiter (C member), 255
- PyAsyncMethods.am\_anext (C member), 255
- PyAsyncMethods.am\_await (C member), 255
- PyAsyncMethods.am\_send (C member), 255
- PyBool\_Check (C function), 110
- PyBool\_FromLong (C function), 110
- PyBool\_Type (C var), 110
- PYBUF\_ANY\_CONTIGUOUS (C macro), 97
- PYBUF\_C\_CONTIGUOUS (C macro), 97
- PYBUF\_CONTIG (C macro), 98
- PYBUF\_CONTIG\_RO (C macro), 98
- PYBUF\_F\_CONTIGUOUS (C macro), 97
- PYBUF\_FORMAT (C macro), 96
- PYBUF\_FULL (C macro), 98
- PYBUF\_FULL\_RO (C macro), 98
- PYBUF\_INDIRECT (C macro), 97
- PYBUF\_MAX\_NDIM (C macro), 96
- PYBUF\_ND (C macro), 97



- PyBUF\_READ (*C macro*), 155
- PyBUF\_RECORDS (*C macro*), 98
- PyBUF\_RECORDS\_RO (*C macro*), 98
- PyBUF\_SIMPLE (*C macro*), 97
- PyBUF\_STRIDED (*C macro*), 98
- PyBUF\_STRIDED\_RO (*C macro*), 98
- PyBUF\_STRIDES (*C macro*), 97
- PyBUF\_WRITABLE (*C macro*), 96
- PyBUF\_WRITE (*C macro*), 155
- PyBuffer\_FillContiguousStrides (*C function*), 100
- PyBuffer\_FillInfo (*C function*), 100
- PyBuffer\_FromContiguous (*C function*), 100
- PyBuffer\_GetPointer (*C function*), 100
- PyBuffer\_IsContiguous (*C function*), 100
- PyBuffer\_Release (*C function*), 99
- PyBuffer\_SizeFromFormat (*C function*), 99
- PyBuffer\_ToContiguous (*C function*), 100
- PyBufferProcs (*C type*), 253
- PyBufferProcs (*C 类型*), 94
- PyBufferProcs.bf\_getbuffer (*C member*), 253
- PyBufferProcs.bf\_releasebuffer (*C member*), 254
- PyByteArray\_AS\_STRING (*C function*), 116
- PyByteArray\_AsString (*C function*), 116
- PyByteArray\_Check (*C function*), 115
- PyByteArray\_CheckExact (*C function*), 115
- PyByteArray\_Concat (*C function*), 116
- PyByteArray\_FromObject (*C function*), 116
- PyByteArray\_FromStringAndSize (*C function*), 116
- PyByteArray\_GET\_SIZE (*C function*), 116
- PyByteArray\_Resize (*C function*), 116
- PyByteArray\_Size (*C function*), 116
- PyByteArray\_Type (*C var*), 115
- PyByteArrayObject (*C type*), 115
- PyBytes\_AS\_STRING (*C function*), 114
- PyBytes\_AsString (*C function*), 114
- PyBytes\_AsStringAndSize (*C function*), 114
- PyBytes\_Check (*C function*), 113
- PyBytes\_CheckExact (*C function*), 113
- PyBytes\_Concat (*C function*), 115
- PyBytes\_ConcatAndDel (*C function*), 115
- PyBytes\_FromFormat (*C function*), 114
- PyBytes\_FromFormatV (*C function*), 114
- PyBytes\_FromObject (*C function*), 114
- PyBytes\_FromString (*C function*), 114
- PyBytes\_FromStringAndSize (*C function*), 114
- PyBytes\_GET\_SIZE (*C function*), 114
- PyBytes\_Size (*C function*), 114
- PyBytes\_Type (*C var*), 113
- PyBytesObject (*C type*), 113
- PyCallable\_Check (*C function*), 87
- PyCallIter\_Check (*C function*), 153
- PyCallIter\_New (*C function*), 153
- PyCallIter\_Type (*C var*), 153
- PyCapsule (*C type*), 157
- PyCapsule\_CheckExact (*C function*), 157
- PyCapsule\_Destructor (*C type*), 157
- PyCapsule\_GetContext (*C function*), 157
- PyCapsule\_GetDestructor (*C function*), 157
- PyCapsule\_GetName (*C function*), 157
- PyCapsule\_GetPointer (*C function*), 157
- PyCapsule\_Import (*C function*), 157
- PyCapsule\_IsValid (*C function*), 157
- PyCapsule\_New (*C function*), 157
- PyCapsule\_SetContext (*C function*), 158
- PyCapsule\_SetDestructor (*C function*), 158
- PyCapsule\_SetName (*C function*), 158
- PyCapsule\_SetPointer (*C function*), 158
- PyCell\_Check (*C function*), 143
- PyCell\_GET (*C function*), 143
- PyCell\_Get (*C function*), 143
- PyCell\_New (*C function*), 143
- PyCell\_SET (*C function*), 143
- PyCell\_Set (*C function*), 143
- PyCell\_Type (*C var*), 143
- PyCellObject (*C type*), 143
- PyCFunction (*C type*), 222
- PyCFunction\_New (*C function*), 224
- PyCFunction\_NewEx (*C function*), 224
- PyCFunctionWithKeywords (*C type*), 222
- PyCMethod (*C type*), 222
- PyCMethod\_New (*C function*), 224
- PyCode\_Addr2Line (*C function*), 144
- PyCode\_Addr2Location (*C function*), 144
- PyCode\_Check (*C function*), 143
- PyCode\_GetCellvars (*C function*), 144
- PyCode\_GetCode (*C function*), 144
- PyCode\_GetFreevars (*C function*), 144
- PyCode\_GetNumFree (*C function*), 143
- PyCode\_GetVarnames (*C function*), 144
- PyCode\_New (*C function*), 143
- PyCode\_NewEmpty (*C function*), 144
- PyCode\_NewWithPosOnlyArgs (*C function*), 143
- PyCode\_Type (*C var*), 143
- PyCodec\_BackslashReplaceErrors (*C function*), 78
- PyCodec\_Decompile (*C function*), 77
- PyCodec\_Decoder (*C function*), 78
- PyCodec\_Encode (*C function*), 77
- PyCodec\_Encoder (*C function*), 78
- PyCodec\_IgnoreErrors (*C function*), 78
- PyCodec\_IncrementalDecoder (*C function*), 78
- PyCodec\_IncrementalEncoder (*C function*), 78
- PyCodec\_KnownEncoding (*C function*), 77
- PyCodec\_LookupError (*C function*), 78
- PyCodec\_NameReplaceErrors (*C function*), 78
- PyCodec\_Register (*C function*), 77
- PyCodec\_RegisterError (*C function*), 78
- PyCodec\_ReplaceErrors (*C function*), 78
- PyCodec\_StreamReader (*C function*), 78
- PyCodec\_StreamWriter (*C function*), 78
- PyCodec\_StrictErrors (*C function*), 78
- PyCodec\_Unregister (*C function*), 77

- PyCodec\_XMLCharRefReplaceErrors (*C function*), 78
- PyCodeObject (*C type*), 143
- PyCompactUnicodeObject (*C type*), 117
- PyCompilerFlags (*C struct*), 42
- PyCompilerFlags.cf\_feature\_version (*C member*), 42
- PyCompilerFlags.cf\_flags (*C member*), 42
- PyComplex\_AsCComplex (*C function*), 113
- PyComplex\_Check (*C function*), 113
- PyComplex\_CheckExact (*C function*), 113
- PyComplex\_FromCComplex (*C function*), 113
- PyComplex\_FromDoubles (*C function*), 113
- PyComplex\_ImagAsDouble (*C function*), 113
- PyComplex\_RealAsDouble (*C function*), 113
- PyComplex\_Type (*C var*), 113
- PyComplexObject (*C type*), 113
- PyConfig (*C type*), 193
- PyConfig\_Clear (*C function*), 193
- PyConfig\_InitIsolatedConfig (*C function*), 193
- PyConfig\_InitPythonConfig (*C function*), 193
- PyConfig\_Read (*C function*), 193
- PyConfig\_SetArgv (*C function*), 193
- PyConfig\_SetBytesArgv (*C function*), 193
- PyConfig\_SetBytesString (*C function*), 193
- PyConfig\_SetString (*C function*), 193
- PyConfig\_SetWideStringList (*C function*), 193
- PyConfig.argv (*C member*), 194
- PyConfig.base\_exec\_prefix (*C member*), 194
- PyConfig.base\_executable (*C member*), 194
- PyConfig.base\_prefix (*C member*), 194
- PyConfig.buffered\_stdio (*C member*), 195
- PyConfig.bytes\_warning (*C member*), 195
- PyConfig.check\_hash\_pycs\_mode (*C member*), 195
- PyConfig.code\_debug\_ranges (*C member*), 195
- PyConfig.configure\_c\_stdio (*C member*), 195
- PyConfig.dev\_mode (*C member*), 195
- PyConfig.dump\_refs (*C member*), 196
- PyConfig.exec\_prefix (*C member*), 196
- PyConfig.executable (*C member*), 196
- PyConfig.faulthandler (*C member*), 196
- PyConfig.filesystem\_encoding (*C member*), 196
- PyConfig.filesystem\_errors (*C member*), 196
- PyConfig.hash\_seed (*C member*), 197
- PyConfig.home (*C member*), 197
- PyConfig.import\_time (*C member*), 197
- PyConfig.inspect (*C member*), 197
- PyConfig.install\_signal\_handlers (*C member*), 197
- PyConfig.interactive (*C member*), 197
- PyConfig.isolated (*C member*), 197
- PyConfig.legacy\_windows\_stdio (*C member*), 198
- PyConfig.malloc\_stats (*C member*), 198
- PyConfig.module\_search\_paths (*C member*), 198
- PyConfig.module\_search\_paths\_set (*C member*), 198
- PyConfig.optimization\_level (*C member*), 199
- PyConfig.orig\_argv (*C member*), 199
- PyConfig.parse\_argv (*C member*), 199
- PyConfig.parser\_debug (*C member*), 199
- PyConfig.pathconfig\_warnings (*C member*), 199
- PyConfig.platlibdir (*C member*), 198
- PyConfig.prefix (*C member*), 199
- PyConfig.program\_name (*C member*), 200
- PyConfig.pycache\_prefix (*C member*), 200
- PyConfig.pythonpath\_env (*C member*), 198
- PyConfig.quiet (*C member*), 200
- PyConfig.run\_command (*C member*), 200
- PyConfig.run\_filename (*C member*), 200
- PyConfig.run\_module (*C member*), 200
- PyConfig.safe\_path (*C member*), 194
- PyConfig.show\_ref\_count (*C member*), 200
- PyConfig.site\_import (*C member*), 201
- PyConfig.skip\_source\_first\_line (*C member*), 201
- PyConfig.stdio\_encoding (*C member*), 201
- PyConfig.stdio\_errors (*C member*), 201
- PyConfig.tracemalloc (*C member*), 201
- PyConfig.use\_environment (*C member*), 201
- PyConfig.use\_hash\_seed (*C member*), 197
- PyConfig.user\_site\_directory (*C member*), 201
- PyConfig.verbose (*C member*), 202
- PyConfig.warn\_default\_encoding (*C member*), 195
- PyConfig.warnoptions (*C member*), 202
- PyConfig.write\_bytecode (*C member*), 202
- PyConfig.xoptions (*C member*), 202
- PyContext (*C type*), 160
- PyContext\_CheckExact (*C function*), 160
- PyContext\_Copy (*C function*), 161
- PyContext\_CopyCurrent (*C function*), 161
- PyContext\_Enter (*C function*), 161
- PyContext\_Exit (*C function*), 161
- PyContext\_New (*C function*), 161
- PyContext\_Type (*C var*), 160
- PyContextToken (*C type*), 160
- PyContextToken\_CheckExact (*C function*), 161
- PyContextToken\_Type (*C var*), 160
- PyContextVar (*C type*), 160
- PyContextVar\_CheckExact (*C function*), 160
- PyContextVar\_Get (*C function*), 161
- PyContextVar\_New (*C function*), 161
- PyContextVar\_Reset (*C function*), 161
- PyContextVar\_Set (*C function*), 161

- PyContextVar\_Type (*C var*), 160  
 PyCoro\_CheckExact (*C function*), 160  
 PyCoro\_New (*C function*), 160  
 PyCoro\_Type (*C var*), 160  
 PyCoroObject (*C type*), 160  
 PyDate\_Check (*C function*), 162  
 PyDate\_CheckExact (*C function*), 162  
 PyDate\_FromDate (*C function*), 163  
 PyDate\_FromTimestamp (*C function*), 165  
 PyDateTime\_Check (*C function*), 162  
 PyDateTime\_CheckExact (*C function*), 162  
 PyDateTime\_Date (*C type*), 161  
 PyDateTime\_DATE\_GET\_FOLD (*C function*), 164  
 PyDateTime\_DATE\_GET\_HOUR (*C function*), 164  
 PyDateTime\_DATE\_GET\_MICROSECOND (*C function*), 164  
 PyDateTime\_DATE\_GET\_MINUTE (*C function*), 164  
 PyDateTime\_DATE\_GET\_SECOND (*C function*), 164  
 PyDateTime\_DATE\_GET\_TZINFO (*C function*), 164  
 PyDateTime\_DateTime (*C type*), 161  
 PyDateTime\_DateTimeType (*C var*), 162  
 PyDateTime\_DateType (*C var*), 162  
 PyDateTime\_Delta (*C type*), 162  
 PyDateTime\_DELTA\_GET\_DAYS (*C function*), 164  
 PyDateTime\_DELTA\_GET\_MICROSECONDS (*C function*), 165  
 PyDateTime\_DELTA\_GET\_SECONDS (*C function*), 164  
 PyDateTime\_DeltaType (*C var*), 162  
 PyDateTime\_FromDateAndTime (*C function*), 163  
 PyDateTime\_FromDateAndTimeAndFold (*C function*), 163  
 PyDateTime\_FromTimestamp (*C function*), 165  
 PyDateTime\_GET\_DAY (*C function*), 164  
 PyDateTime\_GET\_MONTH (*C function*), 163  
 PyDateTime\_GET\_YEAR (*C function*), 163  
 PyDateTime\_Time (*C type*), 162  
 PyDateTime\_TIME\_GET\_FOLD (*C function*), 164  
 PyDateTime\_TIME\_GET\_HOUR (*C function*), 164  
 PyDateTime\_TIME\_GET\_MICROSECOND (*C function*), 164  
 PyDateTime\_TIME\_GET\_MINUTE (*C function*), 164  
 PyDateTime\_TIME\_GET\_SECOND (*C function*), 164  
 PyDateTime\_TIME\_GET\_TZINFO (*C function*), 164  
 PyDateTime\_TimeType (*C var*), 162  
 PyDateTime\_TimeZone\_UTC (*C var*), 162  
 PyDateTime\_TZInfoType (*C var*), 162  
 PyDelta\_Check (*C function*), 162  
 PyDelta\_CheckExact (*C function*), 163  
 PyDelta\_FromDSU (*C function*), 163  
 PyDescr\_IsData (*C function*), 153  
 PyDescr\_NewClassMethod (*C function*), 153  
 PyDescr\_NewGetSet (*C function*), 153  
 PyDescr\_NewMember (*C function*), 153  
 PyDescr\_NewMethod (*C function*), 153  
 PyDescr\_NewWrapper (*C function*), 153  
 PyDict\_Check (*C function*), 136  
 PyDict\_CheckExact (*C function*), 136  
 PyDict\_Clear (*C function*), 137  
 PyDict\_Contains (*C function*), 137  
 PyDict\_Copy (*C function*), 137  
 PyDict\_DelItem (*C function*), 137  
 PyDict\_DelItemString (*C function*), 137  
 PyDict\_GetItem (*C function*), 137  
 PyDict\_GetItemString (*C function*), 137  
 PyDict\_GetItemWithError (*C function*), 137  
 PyDict\_Items (*C function*), 138  
 PyDict\_Keys (*C function*), 138  
 PyDict\_Merge (*C function*), 138  
 PyDict\_MergeFromSeq2 (*C function*), 139  
 PyDict\_New (*C function*), 137  
 PyDict\_Next (*C function*), 138  
 PyDict\_SetDefault (*C function*), 137  
 PyDict\_SetItem (*C function*), 137  
 PyDict\_SetItemString (*C function*), 137  
 PyDict\_Size (*C function*), 138  
 PyDict\_Type (*C var*), 136  
 PyDict\_Update (*C function*), 138  
 PyDict\_Values (*C function*), 138  
 PyDictObject (*C type*), 136  
 PyDictProxy\_New (*C function*), 137  
 PyDoc\_STR (*C macro*), 6  
 PyDoc\_STRVAR (*C macro*), 6  
 PyErr\_BadArgument (*C function*), 46  
 PyErr\_BadInternalCall (*C function*), 48  
 PyErr\_CheckSignals (*C function*), 51  
 PyErr\_Clear (*C function*), 45  
 PyErr\_Clear (*C 函数*), 10, 11  
 PyErr\_ExceptionMatches (*C function*), 49  
 PyErr\_ExceptionMatches (*C 函数*), 11  
 PyErr\_Fetch (*C function*), 49  
 PyErr\_Format (*C function*), 46  
 PyErr\_FormatV (*C function*), 46  
 PyErr\_GetExcInfo (*C function*), 50  
 PyErr\_GetHandledException (*C function*), 50  
 PyErr\_GivenExceptionMatches (*C function*), 49  
 PyErr\_NewException (*C function*), 52  
 PyErr\_NewExceptionWithDoc (*C function*), 52  
 PyErr\_NoMemory (*C function*), 46  
 PyErr\_NormalizeException (*C function*), 50  
 PyErr\_Occurred (*C function*), 49  
 PyErr\_Occurred (*C 函数*), 10  
 PyErr\_Print (*C function*), 45  
 PyErr\_PrintEx (*C function*), 45  
 PyErr\_ResourceWarning (*C function*), 49  
 PyErr\_Restore (*C function*), 49  
 PyErr\_SetExcFromWindowsErr (*C function*), 47

- PyErr\_SetExcFromWindowsErrWithFilename (C function), 47
- PyErr\_SetExcFromWindowsErrWithFilename (C function), 47
- PyErr\_SetExcFromWindowsErrWithFilename (C function), 47
- PyErr\_SetExcInfo (C function), 50
- PyErr\_SetFromErrno (C function), 46
- PyErr\_SetFromErrnoWithFilename (C function), 47
- PyErr\_SetFromErrnoWithFilenameObject (C function), 46
- PyErr\_SetFromErrnoWithFilenameObjects (C function), 46
- PyErr\_SetFromWindowsErr (C function), 47
- PyErr\_SetFromWindowsErrWithFilename (C function), 47
- PyErr\_SetHandledException (C function), 50
- PyErr\_SetImportError (C function), 47
- PyErr\_SetImportErrorSubclass (C function), 48
- PyErr\_SetInterrupt (C function), 51
- PyErr\_SetInterruptEx (C function), 51
- PyErr\_SetNone (C function), 46
- PyErr\_SetObject (C function), 46
- PyErr\_SetString (C function), 46
- PyErr\_SetString (C 函数), 10
- PyErr\_SyntaxLocation (C function), 48
- PyErr\_SyntaxLocationEx (C function), 48
- PyErr\_SyntaxLocationObject (C function), 48
- PyErr\_WarnEx (C function), 48
- PyErr\_WarnExplicit (C function), 48
- PyErr\_WarnExplicitObject (C function), 48
- PyErr\_WarnFormat (C function), 49
- PyErr\_WriteUnraisable (C function), 45
- PyEval\_AcquireLock (C function), 180
- PyEval\_AcquireThread (C function), 180
- PyEval\_AcquireThread(), 176
- PyEval\_EvalCode (C function), 41
- PyEval\_EvalCodeEx (C function), 42
- PyEval\_EvalFrame (C function), 42
- PyEval\_EvalFrameEx (C function), 42
- PyEval\_GetBuiltins (C function), 77
- PyEval\_GetFrame (C function), 77
- PyEval\_GetFuncDesc (C function), 77
- PyEval\_GetFuncName (C function), 77
- PyEval\_GetGlobals (C function), 77
- PyEval\_GetLocals (C function), 77
- PyEval\_InitThreads (C function), 176
- PyEval\_InitThreads(), 170
- PyEval\_MergeCompilerFlags (C function), 42
- PyEval\_ReleaseLock (C function), 181
- PyEval\_ReleaseThread (C function), 180
- PyEval\_ReleaseThread(), 176
- PyEval\_RestoreThread (C function), 177
- PyEval\_RestoreThread (C 函数), 175
- PyEval\_RestoreThread(), 176
- PyEval\_SaveThread (C function), 176
- PyEval\_SaveThread (C 函数), 175
- PyEval\_SaveThread(), 176
- PyEval\_SetProfile (C function), 183
- PyEval\_SetTrace (C function), 184
- PyEval\_ThreadsInitialized (C function), 176
- PyExc\_ArithmeticError (C 变量), 54
- PyExc\_AssertionError (C 变量), 54
- PyExc\_AttributeError (C 变量), 54
- PyExc\_BaseException (C 变量), 54
- PyExc\_BlockingIOError (C 变量), 54
- PyExc\_BrokenPipeError (C 变量), 54
- PyExc\_BufferError (C 变量), 54
- PyExc\_BytesWarning (C 变量), 56
- PyExc\_ChildProcessError (C 变量), 54
- PyExc\_ConnectionAbortedError (C 变量), 54
- PyExc\_ConnectionError (C 变量), 54
- PyExc\_ConnectionRefusedError (C 变量), 54
- PyExc\_ConnectionResetError (C 变量), 54
- PyExc\_DeprecationWarning (C 变量), 56
- PyExc\_EnvironmentError (C 变量), 56
- PyExc\_EOFError (C 变量), 54
- PyExc\_Exception (C 变量), 54
- PyExc\_FileExistsError (C 变量), 54
- PyExc\_FileNotFoundError (C 变量), 54
- PyExc\_FloatingPointError (C 变量), 54
- PyExc\_FutureWarning (C 变量), 56
- PyExc\_GeneratorExit (C 变量), 54
- PyExc\_ImportError (C 变量), 54
- PyExc\_ImportWarning (C 变量), 56
- PyExc\_IndentationError (C 变量), 54
- PyExc\_IndexError (C 变量), 54
- PyExc\_InterruptedError (C 变量), 54
- PyExc\_IOError (C 变量), 56
- PyExc\_IsADirectoryError (C 变量), 54
- PyExc\_KeyboardInterrupt (C 变量), 54
- PyExc\_KeyError (C 变量), 54
- PyExc\_LookupError (C 变量), 54
- PyExc\_MemoryError (C 变量), 54
- PyExc\_ModuleNotFoundError (C 变量), 54
- PyExc\_NameError (C 变量), 54
- PyExc\_NotADirectoryError (C 变量), 54
- PyExc\_NotImplementedError (C 变量), 54
- PyExc\_OSError (C 变量), 54
- PyExc\_OverflowError (C 变量), 54
- PyExc\_PendingDeprecationWarning (C 变量), 56
- PyExc\_PermissionError (C 变量), 54
- PyExc\_ProcessLookupError (C 变量), 54
- PyExc\_RecursionError (C 变量), 54
- PyExc\_ReferenceError (C 变量), 54
- PyExc\_ResourceWarning (C 变量), 56
- PyExc\_RuntimeError (C 变量), 54
- PyExc\_RuntimeWarning (C 变量), 56
- PyExc\_StopAsyncIteration (C 变量), 54
- PyExc\_StopIteration (C 变量), 54
- PyExc\_SyntaxError (C 变量), 54
- PyExc\_SyntaxWarning (C 变量), 56
- PyExc\_SystemError (C 变量), 54



- PyExc\_SystemExit (C 变量), 54
- PyExc\_TabError (C 变量), 54
- PyExc\_TimeoutError (C 变量), 54
- PyExc\_TypeError (C 变量), 54
- PyExc\_UnboundLocalError (C 变量), 54
- PyExc\_UnicodeDecodeError (C 变量), 54
- PyExc\_UnicodeEncodeError (C 变量), 54
- PyExc\_UnicodeError (C 变量), 54
- PyExc\_UnicodeTranslateError (C 变量), 54
- PyExc\_UnicodeWarning (C 变量), 56
- PyExc\_UserWarning (C 变量), 56
- PyExc\_ValueError (C 变量), 54
- PyExc\_Warning (C 变量), 56
- PyExc\_WindowsError (C 变量), 56
- PyExc\_ZeroDivisionError (C 变量), 54
- PyException\_GetCause (C function), 52
- PyException\_GetContext (C function), 52
- PyException\_GetTraceback (C function), 52
- PyException\_SetCause (C function), 52
- PyException\_SetContext (C function), 52
- PyException\_SetTraceback (C function), 52
- PyFile\_FromFd (C function), 145
- PyFile\_GetLine (C function), 145
- PyFile\_SetOpenCodeHook (C function), 145
- PyFile\_WriteObject (C function), 145
- PyFile\_WriteString (C function), 145
- PyFloat\_AS\_DOUBLE (C function), 111
- PyFloat\_AsDouble (C function), 110
- PyFloat\_Check (C function), 110
- PyFloat\_CheckExact (C function), 110
- PyFloat\_FromDouble (C function), 110
- PyFloat\_FromString (C function), 110
- PyFloat\_GetInfo (C function), 111
- PyFloat\_GetMax (C function), 111
- PyFloat\_GetMin (C function), 111
- PyFloat\_Pack2 (C function), 111
- PyFloat\_Pack4 (C function), 111
- PyFloat\_Pack8 (C function), 111
- PyFloat\_Type (C var), 110
- PyFloat\_Unpack2 (C function), 112
- PyFloat\_Unpack4 (C function), 112
- PyFloat\_Unpack8 (C function), 112
- PyFloatObject (C type), 110
- PyFrame\_Check (C function), 158
- PyFrame\_GetBack (C function), 158
- PyFrame\_GetBuiltins (C function), 158
- PyFrame\_GetCode (C function), 158
- PyFrame\_GetGenerator (C function), 159
- PyFrame\_GetGlobals (C function), 159
- PyFrame\_GetLasti (C function), 159
- PyFrame\_GetLineNumber (C function), 159
- PyFrame\_GetLocals (C function), 159
- PyFrame\_Type (C var), 158
- PyFrameObject (C type), 158
- PyFrozenSet\_Check (C function), 139
- PyFrozenSet\_CheckExact (C function), 139
- PyFrozenSet\_New (C function), 140
- PyFrozenSet\_Type (C var), 139
- PyFunction\_Check (C function), 141
- PyFunction\_GetAnnotations (C function), 141
- PyFunction\_GetClosure (C function), 141
- PyFunction\_GetCode (C function), 141
- PyFunction\_GetDefaults (C function), 141
- PyFunction\_GetGlobals (C function), 141
- PyFunction\_GetModule (C function), 141
- PyFunction\_New (C function), 141
- PyFunction\_NewWithQualName (C function), 141
- PyFunction\_SetAnnotations (C function), 142
- PyFunction\_SetClosure (C function), 141
- PyFunction\_SetDefaults (C function), 141
- PyFunction\_Type (C var), 141
- PyFunctionObject (C type), 141
- PyGC\_Collect (C function), 261
- PyGC\_Disable (C function), 261
- PyGC\_Enable (C function), 261
- PyGC\_IsEnabled (C function), 261
- PyGen\_Check (C function), 159
- PyGen\_CheckExact (C function), 159
- PyGen\_New (C function), 159
- PyGen\_NewWithQualName (C function), 159
- PyGen\_Type (C var), 159
- PyGenObject (C type), 159
- PyGetSetDef (C type), 225
- PyGILState\_Check (C function), 177
- PyGILState\_Ensure (C function), 177
- PyGILState\_GetThisThreadState (C function), 177
- PyGILState\_Release (C function), 177
- PyHash\_FuncDef (C type), 76
- PyHash\_FuncDef.hash\_bits (C member), 76
- PyHash\_FuncDef.name (C member), 76
- PyHash\_FuncDef.seed\_bits (C member), 76
- PyHash\_GetFuncDef (C function), 76
- PyImport\_AddModule (C function), 63
- PyImport\_AddModuleObject (C function), 63
- PyImport\_AppendInittab (C function), 65
- PyImport\_ExecCodeModule (C function), 63
- PyImport\_ExecCodeModuleEx (C function), 63
- PyImport\_ExecCodeModuleObject (C function), 64
- PyImport\_ExecCodeModuleWithPathnames (C function), 64
- PyImport\_ExtendInittab (C function), 65
- PyImport\_FrozenModules (C var), 65
- PyImport\_GetImporter (C function), 64
- PyImport\_GetMagicNumber (C function), 64
- PyImport\_GetMagicTag (C function), 64
- PyImport\_GetModule (C function), 64
- PyImport\_GetModuleDict (C function), 64
- PyImport\_Import (C function), 63
- PyImport\_ImportFrozenModule (C function), 64
- PyImport\_ImportFrozenModuleObject (C function), 64
- PyImport\_ImportModule (C function), 62

- PyImport\_ImportModuleEx (*C function*), 62
- PyImport\_ImportModuleLevel (*C function*), 63
- PyImport\_ImportModuleLevelObject (*C function*), 62
- PyImport\_ImportModuleNoBlock (*C function*), 62
- PyImport\_ReloadModule (*C function*), 63
- PyIndex\_Check (*C function*), 90
- PyInstanceMethod\_Check (*C function*), 142
- PyInstanceMethod\_Function (*C function*), 142
- PyInstanceMethod\_GET\_FUNCTION (*C function*), 142
- PyInstanceMethod\_New (*C function*), 142
- PyInstanceMethod\_Type (*C var*), 142
- PyInterpreterState (*C type*), 176
- PyInterpreterState\_Clear (*C function*), 178
- PyInterpreterState\_Delete (*C function*), 178
- PyInterpreterState\_Get (*C function*), 179
- PyInterpreterState\_GetDict (*C function*), 179
- PyInterpreterState\_GetID (*C function*), 179
- PyInterpreterState\_Head (*C function*), 184
- PyInterpreterState\_Main (*C function*), 184
- PyInterpreterState\_New (*C function*), 178
- PyInterpreterState\_Next (*C function*), 184
- PyInterpreterState\_ThreadHead (*C function*), 184
- PyIter\_Check (*C function*), 93
- PyIter\_Next (*C function*), 93
- PyIter\_Send (*C function*), 94
- PyList\_Append (*C function*), 136
- PyList\_AsTuple (*C function*), 136
- PyList\_Check (*C function*), 135
- PyList\_CheckExact (*C function*), 135
- PyList\_GET\_ITEM (*C function*), 135
- PyList\_GET\_SIZE (*C function*), 135
- PyList\_GetItem (*C function*), 135
- PyList\_GetItem (*C 函数*), 8
- PyList\_GetSlice (*C function*), 136
- PyList\_Insert (*C function*), 136
- PyList\_New (*C function*), 135
- PyList\_Reverse (*C function*), 136
- PyList\_SET\_ITEM (*C function*), 136
- PyList\_SetItem (*C function*), 135
- PyList\_SetItem (*C 函数*), 7
- PyList\_SetSlice (*C function*), 136
- PyList\_Size (*C function*), 135
- PyList\_Sort (*C function*), 136
- PyList\_Type (*C var*), 135
- PyListObject (*C type*), 135
- PyLong\_AsDouble (*C function*), 109
- PyLong\_AsLong (*C function*), 108
- PyLong\_AsLongAndOverflow (*C function*), 108
- PyLong\_AsLongLong (*C function*), 108
- PyLong\_AsLongLongAndOverflow (*C function*), 108
- PyLong\_AsSize\_t (*C function*), 109
- PyLong\_AsSsize\_t (*C function*), 109
- PyLong\_AsUnsignedLong (*C function*), 109
- PyLong\_AsUnsignedLongLong (*C function*), 109
- PyLong\_AsUnsignedLongLongMask (*C function*), 109
- PyLong\_AsUnsignedLongMask (*C function*), 109
- PyLong\_AsVoidPtr (*C function*), 109
- PyLong\_Check (*C function*), 107
- PyLong\_CheckExact (*C function*), 107
- PyLong\_FromDouble (*C function*), 107
- PyLong\_FromLong (*C function*), 107
- PyLong\_FromLongLong (*C function*), 107
- PyLong\_FromSize\_t (*C function*), 107
- PyLong\_FromSsize\_t (*C function*), 107
- PyLong\_FromString (*C function*), 108
- PyLong\_FromUnicodeObject (*C function*), 108
- PyLong\_FromUnsignedLong (*C function*), 107
- PyLong\_FromUnsignedLongLong (*C function*), 107
- PyLong\_FromVoidPtr (*C function*), 108
- PyLong\_Type (*C var*), 107
- PyLongObject (*C type*), 107
- PyMapping\_Check (*C function*), 92
- PyMapping\_DelItem (*C function*), 92
- PyMapping\_DelItemString (*C function*), 92
- PyMapping\_GetItemString (*C function*), 92
- PyMapping\_HasKey (*C function*), 92
- PyMapping\_HasKeyString (*C function*), 92
- PyMapping\_Items (*C function*), 93
- PyMapping\_Keys (*C function*), 93
- PyMapping\_Length (*C function*), 92
- PyMapping\_SetItemString (*C function*), 92
- PyMapping\_Size (*C function*), 92
- PyMapping\_Values (*C function*), 93
- PyMappingMethods (*C type*), 252
- PyMappingMethods.mp\_ass\_subscript (*C member*), 252
- PyMappingMethods.mp\_length (*C member*), 252
- PyMappingMethods.mp\_subscript (*C member*), 252
- PyMarshal\_ReadLastObjectFromFile (*C function*), 66
- PyMarshal\_ReadLongFromFile (*C function*), 66
- PyMarshal\_ReadObjectFromFile (*C function*), 66
- PyMarshal\_ReadObjectFromString (*C function*), 66
- PyMarshal\_ReadShortFromFile (*C function*), 66
- PyMarshal\_WriteLongToFile (*C function*), 65
- PyMarshal\_WriteObjectToFile (*C function*), 66
- PyMarshal\_WriteObjectToString (*C function*), 66
- PyMem\_Calloc (*C function*), 211
- PyMem\_Del (*C function*), 212
- PYMEM\_DOMAIN\_MEM (*C macro*), 214
- PYMEM\_DOMAIN\_OBJ (*C macro*), 214



- PYMEM\_DOMAIN\_RAW (*C macro*), 214  
 PyMem\_Free (*C function*), 211  
 PyMem\_GetAllocator (*C function*), 214  
 PyMem\_Malloc (*C function*), 211  
 PyMem\_New (*C macro*), 211  
 PyMem\_RawCalloc (*C function*), 210  
 PyMem\_RawFree (*C function*), 211  
 PyMem\_RawMalloc (*C function*), 210  
 PyMem\_RawRealloc (*C function*), 210  
 PyMem\_Realloc (*C function*), 211  
 PyMem\_Resize (*C macro*), 211  
 PyMem\_SetAllocator (*C function*), 214  
 PyMem\_SetupDebugHooks (*C function*), 214  
 PyMemAllocatorDomain (*C type*), 213  
 PyMemAllocatorEx (*C type*), 213  
 PyMember\_GetOne (*C function*), 225  
 PyMember\_SetOne (*C function*), 225  
 PyMemberDef (*C type*), 224  
 PyMemoryView\_Check (*C function*), 155  
 PyMemoryView\_FromBuffer (*C function*), 155  
 PyMemoryView\_FromMemory (*C function*), 155  
 PyMemoryView\_FromObject (*C function*), 155  
 PyMemoryView\_GET\_BASE (*C function*), 155  
 PyMemoryView\_GET\_BUFFER (*C function*), 155  
 PyMemoryView\_GetContiguous (*C function*), 155  
 PyMethod\_Check (*C function*), 142  
 PyMethod\_Function (*C function*), 142  
 PyMethod\_GET\_FUNCTION (*C function*), 142  
 PyMethod\_GET\_SELF (*C function*), 142  
 PyMethod\_New (*C function*), 142  
 PyMethod\_Self (*C function*), 142  
 PyMethod\_Type (*C var*), 142  
 PyMethodDef (*C type*), 222  
 PyMethodDef.ml\_doc (*C member*), 222  
 PyMethodDef.ml\_flags (*C member*), 222  
 PyMethodDef.ml\_meth (*C member*), 222  
 PyMethodDef.ml\_name (*C member*), 222  
 PyMODINIT\_FUNC (*C macro*), 4  
 PyModule\_AddFunctions (*C function*), 150  
 PyModule\_AddIntConstant (*C function*), 151  
 PyModule\_AddIntMacro (*C macro*), 152  
 PyModule\_AddObject (*C function*), 151  
 PyModule\_AddObjectRef (*C function*), 150  
 PyModule\_AddStringConstant (*C function*), 151  
 PyModule\_AddStringMacro (*C macro*), 152  
 PyModule\_AddType (*C function*), 152  
 PyModule\_Check (*C function*), 146  
 PyModule\_CheckExact (*C function*), 146  
 PyModule\_Create (*C function*), 148  
 PyModule\_Create2 (*C function*), 148  
 PyModule\_ExecDef (*C function*), 150  
 PyModule\_FromDefAndSpec (*C function*), 149  
 PyModule\_FromDefAndSpec2 (*C function*), 149  
 PyModule\_GetDef (*C function*), 146  
 PyModule\_GetDict (*C function*), 146  
 PyModule\_GetFilename (*C function*), 146  
 PyModule\_GetFilenameObject (*C function*), 146  
 PyModule\_GetName (*C function*), 146  
 PyModule\_GetNameObject (*C function*), 146  
 PyModule\_GetState (*C function*), 146  
 PyModule\_New (*C function*), 146  
 PyModule\_NewObject (*C function*), 146  
 PyModule\_SetDocString (*C function*), 150  
 PyModule\_Type (*C var*), 146  
 PyModuleDef (*C type*), 147  
 PyModuleDef\_Init (*C function*), 148  
 PyModuleDef\_Slot (*C type*), 149  
 PyModuleDef\_Slot.slot (*C member*), 149  
 PyModuleDef\_Slot.value (*C member*), 149  
 PyModuleDef.m\_base (*C member*), 147  
 PyModuleDef.m\_clear (*C member*), 147  
 PyModuleDef.m\_doc (*C member*), 147  
 PyModuleDef.m\_free (*C member*), 148  
 PyModuleDef.m\_methods (*C member*), 147  
 PyModuleDef.m\_name (*C member*), 147  
 PyModuleDef.m\_size (*C member*), 147  
 PyModuleDef.m\_slots (*C member*), 147  
 PyModuleDef.m\_slots.m\_reload (*C member*), 147  
 PyModuleDef.m\_traverse (*C member*), 147  
 PyNumber\_Absolute (*C function*), 88  
 PyNumber\_Add (*C function*), 87  
 PyNumber\_And (*C function*), 88  
 PyNumber\_AsSsize\_t (*C function*), 90  
 PyNumber\_Check (*C function*), 87  
 PyNumber\_Divmod (*C function*), 88  
 PyNumber\_Float (*C function*), 90  
 PyNumber\_FloorDivide (*C function*), 88  
 PyNumber\_Index (*C function*), 90  
 PyNumber\_InPlaceAdd (*C function*), 89  
 PyNumber\_InPlaceAnd (*C function*), 89  
 PyNumber\_InPlaceFloorDivide (*C function*), 89  
 PyNumber\_InPlaceLshift (*C function*), 89  
 PyNumber\_InPlaceMatrixMultiply (*C function*), 89  
 PyNumber\_InPlaceMultiply (*C function*), 89  
 PyNumber\_InPlaceOr (*C function*), 89  
 PyNumber\_InPlacePower (*C function*), 89  
 PyNumber\_InPlaceRemainder (*C function*), 89  
 PyNumber\_InPlaceRshift (*C function*), 89  
 PyNumber\_InPlaceSubtract (*C function*), 89  
 PyNumber\_InPlaceTrueDivide (*C function*), 89  
 PyNumber\_InPlaceXor (*C function*), 89  
 PyNumber\_Invert (*C function*), 88  
 PyNumber\_Long (*C function*), 90  
 PyNumber\_Lshift (*C function*), 88  
 PyNumber\_MatrixMultiply (*C function*), 88  
 PyNumber\_Multiply (*C function*), 88  
 PyNumber\_Negative (*C function*), 88  
 PyNumber\_Or (*C function*), 89  
 PyNumber\_Positive (*C function*), 88  
 PyNumber\_Power (*C function*), 88

- PyNumber\_Remainder (*C function*), 88
- PyNumber\_Rshift (*C function*), 88
- PyNumber\_Subtract (*C function*), 87
- PyNumber\_ToBase (*C function*), 90
- PyNumber\_TrueDivide (*C function*), 88
- PyNumber\_Xor (*C function*), 88
- PyNumberMethods (*C type*), 250
- PyNumberMethods.nb\_absolute (*C member*), 251
- PyNumberMethods.nb\_add (*C member*), 251
- PyNumberMethods.nb\_and (*C member*), 251
- PyNumberMethods.nb\_bool (*C member*), 251
- PyNumberMethods.nb\_divmod (*C member*), 251
- PyNumberMethods.nb\_float (*C member*), 252
- PyNumberMethods.nb\_floor\_divide (*C member*), 252
- PyNumberMethods.nb\_index (*C member*), 252
- PyNumberMethods.nb\_inplace\_add (*C member*), 252
- PyNumberMethods.nb\_inplace\_and (*C member*), 252
- PyNumberMethods.nb\_inplace\_floor\_divide (*C member*), 252
- PyNumberMethods.nb\_inplace\_lshift (*C member*), 252
- PyNumberMethods.nb\_inplace\_matrix\_multiply (*C member*), 252
- PyNumberMethods.nb\_inplace\_multiply (*C member*), 252
- PyNumberMethods.nb\_inplace\_or (*C member*), 252
- PyNumberMethods.nb\_inplace\_power (*C member*), 252
- PyNumberMethods.nb\_inplace\_remainder (*C member*), 252
- PyNumberMethods.nb\_inplace\_rshift (*C member*), 252
- PyNumberMethods.nb\_inplace\_subtract (*C member*), 252
- PyNumberMethods.nb\_inplace\_true\_divide (*C member*), 252
- PyNumberMethods.nb\_inplace\_xor (*C member*), 252
- PyNumberMethods.nb\_int (*C member*), 251
- PyNumberMethods.nb\_invert (*C member*), 251
- PyNumberMethods.nb\_lshift (*C member*), 251
- PyNumberMethods.nb\_matrix\_multiply (*C member*), 252
- PyNumberMethods.nb\_multiply (*C member*), 251
- PyNumberMethods.nb\_negative (*C member*), 251
- PyNumberMethods.nb\_or (*C member*), 251
- PyNumberMethods.nb\_positive (*C member*), 251
- PyNumberMethods.nb\_power (*C member*), 251
- PyNumberMethods.nb\_remainder (*C member*), 251
- PyNumberMethods.nb\_reserved (*C member*), 252
- PyNumberMethods.nb\_rshift (*C member*), 251
- PyNumberMethods.nb\_subtract (*C member*), 251
- PyNumberMethods.nb\_true\_divide (*C member*), 252
- PyNumberMethods.nb\_xor (*C member*), 251
- PyObject (*C type*), 220
- PyObject\_AsCharBuffer (*C function*), 100
- PyObject\_ASCII (*C function*), 81
- PyObject\_AsFileDescriptor (*C function*), 145
- PyObject\_AsReadBuffer (*C function*), 101
- PyObject\_AsWriteBuffer (*C function*), 101
- PyObject\_Bytes (*C function*), 81
- PyObject\_Call (*C function*), 85
- PyObject\_CallFunction (*C function*), 86
- PyObject\_CallFunctionObjArgs (*C function*), 86
- PyObject\_CallMethod (*C function*), 86
- PyObject\_CallMethodNoArgs (*C function*), 86
- PyObject\_CallMethodObjArgs (*C function*), 86
- PyObject\_CallMethodOneArg (*C function*), 86
- PyObject\_CallNoArgs (*C function*), 85
- PyObject\_CallObject (*C function*), 86
- PyObject\_Calloc (*C function*), 212
- PyObject\_CallOneArg (*C function*), 85
- PyObject\_CheckBuffer (*C function*), 99
- PyObject\_CheckReadBuffer (*C function*), 101
- PyObject\_ClearWeakRefs (*C function*), 156
- PyObject\_CopyData (*C function*), 100
- PyObject\_Del (*C function*), 219
- PyObject\_DelAttr (*C function*), 80
- PyObject\_DelAttrString (*C function*), 80
- PyObject\_DelItem (*C function*), 82
- PyObject\_Dir (*C function*), 82
- PyObject\_Format (*C function*), 81
- PyObject\_Free (*C function*), 213
- PyObject\_GC\_Del (*C function*), 260
- PyObject\_GC\_IsFinalized (*C function*), 260
- PyObject\_GC\_IsTracked (*C function*), 260
- PyObject\_GC\_New (*C macro*), 259
- PyObject\_GC\_NewVar (*C macro*), 259
- PyObject\_GC\_Resize (*C macro*), 259
- PyObject\_GC\_Track (*C function*), 260
- PyObject\_GC\_UnTrack (*C function*), 260
- PyObject\_GenericGetAttr (*C function*), 80
- PyObject\_GenericGetDict (*C function*), 80
- PyObject\_GenericSetAttr (*C function*), 80
- PyObject\_GenericSetDict (*C function*), 80
- PyObject\_GetAIter (*C function*), 83
- PyObject\_GetArenaAllocator (*C function*), 216
- PyObject\_GetAttr (*C function*), 80
- PyObject\_GetAttrString (*C function*), 80
- PyObject\_GetBuffer (*C function*), 99
- PyObject\_GetItem (*C function*), 82
- PyObject\_GetIter (*C function*), 83

- PyObject\_HasAttr (*C function*), 79
- PyObject\_HasAttrString (*C function*), 79
- PyObject\_Hash (*C function*), 82
- PyObject\_HashNotImplemented (*C function*), 82
- PyObject\_HEAD (*C macro*), 220
- PyObject\_HEAD\_INIT (*C macro*), 221
- PyObject\_Init (*C function*), 219
- PyObject\_InitVar (*C function*), 219
- PyObject\_IS\_GC (*C function*), 260
- PyObject\_IsInstance (*C function*), 81
- PyObject\_IsSubclass (*C function*), 81
- PyObject\_IsTrue (*C function*), 82
- PyObject\_Length (*C function*), 82
- PyObject\_LengthHint (*C function*), 82
- PyObject\_Malloc (*C function*), 212
- PyObject\_New (*C macro*), 219
- PyObject\_NewVar (*C macro*), 219
- PyObject\_Not (*C function*), 82
- PyObject.\_ob\_next (*C member*), 233
- PyObject.\_ob\_prev (*C member*), 233
- PyObject\_Print (*C function*), 79
- PyObject\_Realloc (*C function*), 212
- PyObject\_Repr (*C function*), 81
- PyObject\_RichCompare (*C function*), 81
- PyObject\_RichCompareBool (*C function*), 81
- PyObject\_SetArenaAllocator (*C function*), 216
- PyObject\_SetAttr (*C function*), 80
- PyObject\_SetAttrString (*C function*), 80
- PyObject\_SetItem (*C function*), 82
- PyObject\_Size (*C function*), 82
- PyObject\_Str (*C function*), 81
- PyObject\_Type (*C function*), 82
- PyObject\_TypeCheck (*C function*), 82
- PyObject\_VAR\_HEAD (*C macro*), 220
- PyObject\_Vectorcall (*C function*), 87
- PyObject\_VectorcallDict (*C function*), 87
- PyObject\_VectorcallMethod (*C function*), 87
- PyObjectArenaAllocator (*C type*), 216
- PyObject.ob\_refcnt (*C member*), 232
- PyObject.ob\_type (*C member*), 232
- PyOS\_AfterFork (*C function*), 58
- PyOS\_AfterFork\_Child (*C function*), 58
- PyOS\_AfterFork\_Parent (*C function*), 57
- PyOS\_BeforeFork (*C function*), 57
- PyOS\_CheckStack (*C function*), 58
- PyOS\_double\_to\_string (*C function*), 75
- PyOS\_FSPath (*C function*), 57
- PyOS\_getsig (*C function*), 58
- PyOS\_InputHook (*C var*), 40
- PyOS\_ReadlineFunctionPointer (*C var*), 40
- PyOS\_setsig (*C function*), 58
- PyOS\_sighandler\_t (*C type*), 58
- PyOS\_snprintf (*C function*), 74
- PyOS\_stricmp (*C function*), 76
- PyOS\_string\_to\_double (*C function*), 75
- PyOS\_strncmp (*C function*), 76
- PyOS\_strtol (*C function*), 75
- PyOS\_strtoul (*C function*), 75
- PyOS\_vsnprintf (*C function*), 74
- PyPreConfig (*C type*), 190
- PyPreConfig\_InitIsolatedConfig (*C function*), 190
- PyPreConfig\_InitPythonConfig (*C function*), 190
- PyPreConfig.allocator (*C member*), 190
- PyPreConfig.coerce\_c\_locale (*C member*), 190
- PyPreConfig.coerce\_c\_locale\_warn (*C member*), 191
- PyPreConfig.configure\_locale (*C member*), 190
- PyPreConfig.dev\_mode (*C member*), 191
- PyPreConfig.isolated (*C member*), 191
- PyPreConfig.legacy\_windows\_fs\_encoding (*C member*), 191
- PyPreConfig.parse\_argv (*C member*), 191
- PyPreConfig.use\_environment (*C member*), 191
- PyPreConfig.utf8\_mode (*C member*), 191
- PyProperty\_Type (*C var*), 153
- PyRun\_AnyFile (*C function*), 39
- PyRun\_AnyFileEx (*C function*), 39
- PyRun\_AnyFileExFlags (*C function*), 39
- PyRun\_AnyFileFlags (*C function*), 39
- PyRun\_File (*C function*), 41
- PyRun\_FileEx (*C function*), 41
- PyRun\_FileExFlags (*C function*), 41
- PyRun\_FileFlags (*C function*), 41
- PyRun\_InteractiveLoop (*C function*), 40
- PyRun\_InteractiveLoopFlags (*C function*), 40
- PyRun\_InteractiveOne (*C function*), 40
- PyRun\_InteractiveOneFlags (*C function*), 40
- PyRun\_SimpleFile (*C function*), 40
- PyRun\_SimpleFileEx (*C function*), 40
- PyRun\_SimpleFileExFlags (*C function*), 40
- PyRun\_SimpleString (*C function*), 40
- PyRun\_SimpleStringFlags (*C function*), 40
- PyRun\_String (*C function*), 41
- PyRun\_StringFlags (*C function*), 41
- PySendResult (*C type*), 94
- PySeqIter\_Check (*C function*), 153
- PySeqIter\_New (*C function*), 153
- PySeqIter\_Type (*C var*), 153
- PySequence\_Check (*C function*), 90
- PySequence\_Concat (*C function*), 90
- PySequence\_Contains (*C function*), 91
- PySequence\_Count (*C function*), 91
- PySequence\_DelItem (*C function*), 91
- PySequence\_DelSlice (*C function*), 91
- PySequence\_Fast (*C function*), 91
- PySequence\_Fast\_GET\_ITEM (*C function*), 92
- PySequence\_Fast\_GET\_SIZE (*C function*), 91
- PySequence\_Fast\_ITEMS (*C function*), 92
- PySequence\_GetItem (*C function*), 91

- PySequence\_GetItem (*C 函数*), 8
- PySequence\_GetSlice (*C function*), 91
- PySequence\_Index (*C function*), 91
- PySequence\_InPlaceConcat (*C function*), 90
- PySequence\_InPlaceRepeat (*C function*), 90
- PySequence\_ITEM (*C function*), 92
- PySequence\_Length (*C function*), 90
- PySequence\_List (*C function*), 91
- PySequence\_Repeat (*C function*), 90
- PySequence\_SetItem (*C function*), 91
- PySequence\_SetSlice (*C function*), 91
- PySequence\_Size (*C function*), 90
- PySequence\_Tuple (*C function*), 91
- PySequenceMethods (*C type*), 253
- PySequenceMethods.sq\_ass\_item (*C member*), 253
- PySequenceMethods.sq\_concat (*C member*), 253
- PySequenceMethods.sq\_contains (*C member*), 253
- PySequenceMethods.sq\_inplace\_concat (*C member*), 253
- PySequenceMethods.sq\_inplace\_repeat (*C member*), 253
- PySequenceMethods.sq\_item (*C member*), 253
- PySequenceMethods.sq\_length (*C member*), 253
- PySequenceMethods.sq\_repeat (*C member*), 253
- PySet\_Add (*C function*), 140
- PySet\_Check (*C function*), 139
- PySet\_CheckExact (*C function*), 139
- PySet\_Clear (*C function*), 140
- PySet\_Contains (*C function*), 140
- PySet\_Discard (*C function*), 140
- PySet\_GET\_SIZE (*C function*), 140
- PySet\_New (*C function*), 140
- PySet\_Pop (*C function*), 140
- PySet\_Size (*C function*), 140
- PySet\_Type (*C var*), 139
- PySetObject (*C type*), 139
- PySignal\_SetWakeUpFd (*C function*), 51
- PySlice\_AdjustIndices (*C function*), 155
- PySlice\_Check (*C function*), 154
- PySlice\_GetIndices (*C function*), 154
- PySlice\_GetIndicesEx (*C function*), 154
- PySlice\_New (*C function*), 154
- PySlice\_Type (*C var*), 154
- PySlice\_Unpack (*C function*), 154
- PyState\_AddModule (*C function*), 152
- PyState\_FindModule (*C function*), 152
- PyState\_RemoveModule (*C function*), 152
- PyStatus (*C type*), 189
- PyStatus\_Error (*C function*), 189
- PyStatus\_Exception (*C function*), 189
- PyStatus\_Exit (*C function*), 189
- PyStatus\_IsError (*C function*), 189
- PyStatus\_IsExit (*C function*), 189
- PyStatus\_NoMemory (*C function*), 189
- PyStatus\_Ok (*C function*), 189
- PyStatus.err\_msg (*C member*), 189
- PyStatus.exitcode (*C member*), 189
- PyStatus.func (*C member*), 189
- PyStructSequence\_Desc (*C type*), 134
- PyStructSequence\_Desc.doc (*C member*), 134
- PyStructSequence\_Desc.fields (*C member*), 134
- PyStructSequence\_Desc.n\_in\_sequence (*C member*), 134
- PyStructSequence\_Desc.name (*C member*), 134
- PyStructSequence\_Field (*C type*), 134
- PyStructSequence\_Field.doc (*C member*), 134
- PyStructSequence\_Field.name (*C member*), 134
- PyStructSequence\_GET\_ITEM (*C function*), 135
- PyStructSequence\_GetItem (*C function*), 134
- PyStructSequence\_InitType (*C function*), 134
- PyStructSequence\_InitType2 (*C function*), 134
- PyStructSequence\_New (*C function*), 134
- PyStructSequence\_NewType (*C function*), 134
- PyStructSequence\_SET\_ITEM (*C function*), 135
- PyStructSequence\_SetItem (*C function*), 135
- PyStructSequence\_UnnamedField (*C var*), 134
- PySys\_AddAuditHook (*C function*), 61
- PySys\_AddWarnOption (*C function*), 60
- PySys\_AddWarnOptionUnicode (*C function*), 60
- PySys\_AddXOption (*C function*), 61
- PySys\_Audit (*C function*), 61
- PySys\_FormatStderr (*C function*), 61
- PySys\_FormatStdout (*C function*), 60
- PySys\_GetObject (*C function*), 60
- PySys\_GetXOptions (*C function*), 61
- PySys\_ResetWarnOptions (*C function*), 60
- PySys\_SetArgv (*C function*), 174
- PySys\_SetArgv (*C 函数*), 170
- PySys\_SetArgvEx (*C function*), 173
- PySys\_SetArgvEx (*C 函数*), 170
- PySys\_SetObject (*C function*), 60
- PySys\_SetPath (*C function*), 60
- PySys\_WriteStderr (*C function*), 60
- PySys\_WriteStdout (*C function*), 60
- Python 3000, 276
- Python Enhancement Proposals
  - PEP 1, 276
  - PEP 7, 3, 6
  - PEP 238, 42, 270
  - PEP 278, 279
  - PEP 302, 270, 273
  - PEP 343, 268
  - PEP 353, 9
  - PEP 362, 266, 275
  - PEP 383, 124, 125



- PEP 387, 13
- PEP 393, 116, 123
- PEP 411, 276
- PEP 420, 270, 274, 276
- PEP 432, 207
- PEP 442, 249
- PEP 443, 271
- PEP 451, 149, 270
- PEP 456, 76
- PEP 483, 271
- PEP 484, 265, 270, 271, 278, 279
- PEP 492, 266, 268
- PEP 498, 269
- PEP 519, 276
- PEP 523, 180
- PEP 525, 266
- PEP 526, 265, 279
- PEP 528, 169, 198
- PEP 529, 125, 169
- PEP 538, 205
- PEP 539, 185
- PEP 540, 205
- PEP 552, 195
- PEP 578, 61
- PEP 585, 271
- PEP 587, 187
- PEP 590, 83
- PEP 623, 116
- PEP 634, 240, 241
- PEP 3116, 279
- PEP 3119, 81, 82
- PEP 3121, 147
- PEP 3147, 64
- PEP 3151, 55
- PEP 3155, 276
- PYTHONCOERCECLOCALE, 205
- PYTHONDEBUG, 168, 199
- PYTHONDEVMODE, 196
- PYTHONDONTWRITEBYTECODE, 168, 202
- PYTHONDUMPREFS, 196, 233
- PYTHONEXECUTABLE, 200
- PYTHONFAULTHANDLER, 196
- PYTHONHASHSEED, 168, 197
- PYTHONHOME, 11, 168, 174, 197
- Pythonic (Python 風格的), 276
- PYTHONINSPECT, 169, 197
- PYTHONIOENCODING, 171, 201
- PYTHONLEGACYWINDOWSFSENCODING, 169, 191
- PYTHONLEGACYWINDOWSSTDIO, 169, 198
- PYTHONMALLOC, 210, 213, 215
- PYTHONMALLOC` (例
  - `PYTHONMALLOC=malloc`, 216
- PYTHONMALLOCSTATS, 198, 210
- PYTHONNODEBUGRANGES, 195
- PYTHONNOUSERSITE, 169, 202
- PYTHONOPTIMIZE, 169, 199
- PYTHONPATH, 11, 168, 198
- PYTHONPLATLIBDIR, 198
- PYTHONPROFILEIMPORTTIME, 197
- PYTHONPYCACHEPREFIX, 200
- PYTHONSAFEPATH, 194
- PYTHONTRACEMALLOC, 201
- PYTHONUNBUFFERED, 169, 195
- PYTHONUTF8, 191, 205
- PYTHONVERBOSE, 169, 202
- PYTHONWARNINGS, 202
- PyThread\_create\_key (C function), 186
- PyThread\_delete\_key (C function), 186
- PyThread\_delete\_key\_value (C function), 186
- PyThread\_get\_key\_value (C function), 186
- PyThread\_ReInitTLS (C function), 186
- PyThread\_set\_key\_value (C function), 186
- PyThread\_tss\_alloc (C function), 185
- PyThread\_tss\_create (C function), 185
- PyThread\_tss\_delete (C function), 185
- PyThread\_tss\_free (C function), 185
- PyThread\_tss\_get (C function), 185
- PyThread\_tss\_is\_created (C function), 185
- PyThread\_tss\_set (C function), 185
- PyThreadState (C type), 176
- PyThreadState (C 类型), 174
- PyThreadState\_Clear (C function), 178
- PyThreadState\_Delete (C function), 178
- PyThreadState\_DeleteCurrent (C function), 178
- PyThreadState\_EnterTracing (C function), 179
- PyThreadState\_Get (C function), 177
- PyThreadState\_GetDict (C function), 180
- PyThreadState\_GetFrame (C function), 178
- PyThreadState\_GetID (C function), 179
- PyThreadState\_GetInterpreter (C function), 179
- PyThreadState\_LeaveTracing (C function), 179
- PyThreadState\_New (C function), 178
- PyThreadState\_Next (C function), 184
- PyThreadState\_SetAsyncExc (C function), 180
- PyThreadState\_Swap (C function), 177
- PyThreadState.interp (C member), 176
- PyTime\_Check (C function), 162
- PyTime\_CheckExact (C function), 162
- PyTime\_FromTime (C function), 163
- PyTime\_FromTimeAndFold (C function), 163
- PyTimeZone\_FromOffset (C function), 163
- PyTimeZone\_FromOffsetAndName (C function), 163
- PyTrace\_C\_CALL (C var), 183
- PyTrace\_C\_EXCEPTION (C var), 183
- PyTrace\_C\_RETURN (C var), 183
- PyTrace\_CALL (C var), 183
- PyTrace\_EXCEPTION (C var), 183
- PyTrace\_LINE (C var), 183
- PyTrace\_OPCODE (C var), 183
- PyTrace\_RETURN (C var), 183
- PyTraceMalloc\_Track (C function), 217

如：

- PyTraceMalloc\_Untrack (*C function*), 217
- PyTuple\_Check (*C function*), 133
- PyTuple\_CheckExact (*C function*), 133
- PyTuple\_GET\_ITEM (*C function*), 133
- PyTuple\_GET\_SIZE (*C function*), 133
- PyTuple\_GetItem (*C function*), 133
- PyTuple\_GetSlice (*C function*), 133
- PyTuple\_New (*C function*), 133
- PyTuple\_Pack (*C function*), 133
- PyTuple\_SET\_ITEM (*C function*), 133
- PyTuple\_SetItem (*C function*), 133
- PyTuple\_SetItem (*C 函数*), 7
- PyTuple\_Size (*C function*), 133
- PyTuple\_Type (*C var*), 133
- PyTupleObject (*C type*), 133
- PyType\_Check (*C function*), 103
- PyType\_CheckExact (*C function*), 103
- PyType\_ClearCache (*C function*), 103
- PyType\_FromModuleAndSpec (*C function*), 105
- PyType\_FromSpec (*C function*), 105
- PyType\_FromSpecWithBases (*C function*), 105
- PyType\_GenericAlloc (*C function*), 104
- PyType\_GenericNew (*C function*), 104
- PyType\_GetFlags (*C function*), 103
- PyType\_GetModule (*C function*), 104
- PyType\_GetModuleByDef (*C function*), 105
- PyType\_GetModuleState (*C function*), 105
- PyType\_GetName (*C function*), 104
- PyType\_GetQualName (*C function*), 104
- PyType\_GetSlot (*C function*), 104
- PyType\_HasFeature (*C function*), 104
- PyType\_IS\_GC (*C function*), 104
- PyType\_IsSubtype (*C function*), 104
- PyType\_Modified (*C function*), 104
- PyType\_Ready (*C function*), 104
- PyType\_Slot (*C type*), 106
- PyType\_Slot.PyType\_Slot.pfunc (*C member*), 106
- PyType\_Slot.PyType\_Slot.slot (*C member*), 106
- PyType\_Spec (*C type*), 105
- PyType\_Spec.PyType\_Spec.basicsize (*C member*), 106
- PyType\_Spec.PyType\_Spec.flags (*C member*), 106
- PyType\_Spec.PyType\_Spec.itemsize (*C member*), 106
- PyType\_Spec.PyType\_Spec.name (*C member*), 106
- PyType\_Spec.PyType\_Spec.slots (*C member*), 106
- PyType\_Type (*C var*), 103
- PyTypeObject (*C type*), 103
- PyTypeObject.tp\_alloc (*C member*), 247
- PyTypeObject.tp\_as\_async (*C member*), 235
- PyTypeObject.tp\_as\_buffer (*C member*), 237
- PyTypeObject.tp\_as\_mapping (*C member*), 236
- PyTypeObject.tp\_as\_number (*C member*), 236
- PyTypeObject.tp\_as\_sequence (*C member*), 236
- PyTypeObject.tp\_base (*C member*), 245
- PyTypeObject.tp\_bases (*C member*), 248
- PyTypeObject.tp\_basicsize (*C member*), 233
- PyTypeObject.tp\_cache (*C member*), 248
- PyTypeObject.tp\_call (*C member*), 236
- PyTypeObject.tp\_clear (*C member*), 242
- PyTypeObject.tp\_dealloc (*C member*), 234
- PyTypeObject.tp\_del (*C member*), 249
- PyTypeObject.tp\_descr\_get (*C member*), 245
- PyTypeObject.tp\_descr\_set (*C member*), 245
- PyTypeObject.tp\_dict (*C member*), 245
- PyTypeObject.tp\_dictoffset (*C member*), 246
- PyTypeObject.tp\_doc (*C member*), 241
- PyTypeObject.tp\_finalize (*C member*), 249
- PyTypeObject.tp\_flags (*C member*), 238
- PyTypeObject.tp\_free (*C member*), 247
- PyTypeObject.tp\_getattr (*C member*), 235
- PyTypeObject.tp\_getattro (*C member*), 237
- PyTypeObject.tp\_getset (*C member*), 244
- PyTypeObject.tp\_hash (*C member*), 236
- PyTypeObject.tp\_init (*C member*), 246
- PyTypeObject.tp\_is\_gc (*C member*), 248
- PyTypeObject.tp\_itemsize (*C member*), 233
- PyTypeObject.tp\_iter (*C member*), 244
- PyTypeObject.tp\_ternext (*C member*), 244
- PyTypeObject.tp\_members (*C member*), 244
- PyTypeObject.tp\_methods (*C member*), 244
- PyTypeObject.tp\_mro (*C member*), 248
- PyTypeObject.tp\_name (*C member*), 233
- PyTypeObject.tp\_new (*C member*), 247
- PyTypeObject.tp\_repr (*C member*), 235
- PyTypeObject.tp\_richcompare (*C member*), 242
- PyTypeObject.tp\_setattr (*C member*), 235
- PyTypeObject.tp\_setattro (*C member*), 237
- PyTypeObject.tp\_str (*C member*), 237
- PyTypeObject.tp\_subclasses (*C member*), 248
- PyTypeObject.tp\_traverse (*C member*), 241
- PyTypeObject.tp\_vectorcall (*C member*), 249
- PyTypeObject.tp\_vectorcall\_offset (*C member*), 234
- PyTypeObject.tp\_version\_tag (*C member*), 249
- PyTypeObject.tp\_weaklist (*C member*), 249
- PyTypeObject.tp\_weaklistoffset (*C member*), 243
- PyTZInfo\_Check (*C function*), 163
- PyTZInfo\_CheckExact (*C function*), 163
- PyUnicode\_1BYTE\_DATA (*C function*), 117
- PyUnicode\_1BYTE\_KIND (*C macro*), 118
- PyUnicode\_2BYTE\_DATA (*C function*), 117
- PyUnicode\_2BYTE\_KIND (*C macro*), 118



- PyUnicode\_4BYTE\_DATA (*C function*), 117  
 PyUnicode\_4BYTE\_KIND (*C macro*), 118  
 PyUnicode\_AS\_DATA (*C function*), 118  
 PyUnicode\_AS\_UNICODE (*C function*), 118  
 PyUnicode\_AsASCIIString (*C function*), 130  
 PyUnicode\_AsCharmapString (*C function*), 130  
 PyUnicode\_AsEncodedString (*C function*), 127  
 PyUnicode\_AsLatin1String (*C function*), 130  
 PyUnicode\_AsMBCSString (*C function*), 131  
 PyUnicode\_AsRawUnicodeEscapeString (*C function*), 129  
 PyUnicode\_AsUCS4 (*C function*), 122  
 PyUnicode\_AsUCS4Copy (*C function*), 123  
 PyUnicode\_AsUnicode (*C function*), 123  
 PyUnicode\_AsUnicodeAndSize (*C function*), 123  
 PyUnicode\_AsUnicodeEscapeString (*C function*), 129  
 PyUnicode\_AsUTF8 (*C function*), 127  
 PyUnicode\_AsUTF8AndSize (*C function*), 127  
 PyUnicode\_AsUTF8String (*C function*), 127  
 PyUnicode\_AsUTF16String (*C function*), 129  
 PyUnicode\_AsUTF32String (*C function*), 128  
 PyUnicode\_AsWideChar (*C function*), 126  
 PyUnicode\_AsWideCharString (*C function*), 126  
 PyUnicode\_Check (*C function*), 117  
 PyUnicode\_CheckExact (*C function*), 117  
 PyUnicode\_Compare (*C function*), 132  
 PyUnicode\_CompareWithASCIIString (*C function*), 132  
 PyUnicode\_Concat (*C function*), 131  
 PyUnicode\_Contains (*C function*), 132  
 PyUnicode\_CopyCharacters (*C function*), 122  
 PyUnicode\_Count (*C function*), 132  
 PyUnicode\_DATA (*C function*), 118  
 PyUnicode\_Decode (*C function*), 127  
 PyUnicode\_DecodeASCII (*C function*), 130  
 PyUnicode\_DecodeCharmap (*C function*), 130  
 PyUnicode\_DecodeFSDefault (*C function*), 125  
 PyUnicode\_DecodeFSDefaultAndSize (*C function*), 125  
 PyUnicode\_DecodeLatin1 (*C function*), 130  
 PyUnicode\_DecodeLocale (*C function*), 124  
 PyUnicode\_DecodeLocaleAndSize (*C function*), 124  
 PyUnicode\_DecodeMBCS (*C function*), 131  
 PyUnicode\_DecodeMBCSStateful (*C function*), 131  
 PyUnicode\_DecodeRawUnicodeEscape (*C function*), 129  
 PyUnicode\_DecodeUnicodeEscape (*C function*), 129  
 PyUnicode\_DecodeUTF7 (*C function*), 129  
 PyUnicode\_DecodeUTF7Stateful (*C function*), 129  
 PyUnicode\_DecodeUTF8 (*C function*), 127  
 PyUnicode\_DecodeUTF8Stateful (*C function*), 127  
 PyUnicode\_DecodeUTF16 (*C function*), 128  
 PyUnicode\_DecodeUTF16Stateful (*C function*), 128  
 PyUnicode\_DecodeUTF32 (*C function*), 128  
 PyUnicode\_DecodeUTF32Stateful (*C function*), 128  
 PyUnicode\_EncodeCodePage (*C function*), 131  
 PyUnicode\_EncodeFSDefault (*C function*), 125  
 PyUnicode\_EncodeLocale (*C function*), 124  
 PyUnicode\_Fill (*C function*), 122  
 PyUnicode\_Find (*C function*), 131  
 PyUnicode\_FindChar (*C function*), 132  
 PyUnicode\_Format (*C function*), 132  
 PyUnicode\_FromEncodedObject (*C function*), 122  
 PyUnicode\_FromFormat (*C function*), 121  
 PyUnicode\_FromFormatV (*C function*), 122  
 PyUnicode\_FromKindAndData (*C function*), 120  
 PyUnicode\_FromObject (*C function*), 122  
 PyUnicode\_FromString (*C function*), 121  
 PyUnicode\_FromStringAndSize (*C function*), 120  
 PyUnicode\_FromUnicode (*C function*), 123  
 PyUnicode\_FromWideChar (*C function*), 126  
 PyUnicode\_FSConverter (*C function*), 125  
 PyUnicode\_FSDecoder (*C function*), 125  
 PyUnicode\_GET\_DATA\_SIZE (*C function*), 118  
 PyUnicode\_GET\_LENGTH (*C function*), 117  
 PyUnicode\_GET\_SIZE (*C function*), 118  
 PyUnicode\_GetLength (*C function*), 122  
 PyUnicode\_GetSize (*C function*), 123  
 PyUnicode\_InternFromString (*C function*), 132  
 PyUnicode\_InternInPlace (*C function*), 132  
 PyUnicode\_IsIdentifier (*C function*), 119  
 PyUnicode\_Join (*C function*), 131  
 PyUnicode\_KIND (*C function*), 118  
 PyUnicode\_MAX\_CHAR\_VALUE (*C function*), 118  
 PyUnicode\_New (*C function*), 120  
 PyUnicode\_READ (*C function*), 118  
 PyUnicode\_READ\_CHAR (*C function*), 118  
 PyUnicode\_ReadChar (*C function*), 122  
 PyUnicode\_READY (*C function*), 117  
 PyUnicode\_Replace (*C function*), 132  
 PyUnicode\_RichCompare (*C function*), 132  
 PyUnicode\_Split (*C function*), 131  
 PyUnicode\_Splitlines (*C function*), 131  
 PyUnicode\_Substring (*C function*), 122  
 PyUnicode\_Tailmatch (*C function*), 131  
 PyUnicode\_Translate (*C function*), 130  
 PyUnicode\_Type (*C var*), 117  
 PyUnicode\_WCHAR\_KIND (*C macro*), 118  
 PyUnicode\_WRITE (*C function*), 118  
 PyUnicode\_WriteChar (*C function*), 122  
 PyUnicodeDecodeError\_Create (*C function*), 53

- PyUnicodeDecodeError\_GetEncoding (C function), 53
  - PyUnicodeDecodeError\_GetEnd (C function), 53
  - PyUnicodeDecodeError\_GetObject (C function), 53
  - PyUnicodeDecodeError\_GetReason (C function), 53
  - PyUnicodeDecodeError\_GetStart (C function), 53
  - PyUnicodeDecodeError\_SetEnd (C function), 53
  - PyUnicodeDecodeError\_SetReason (C function), 53
  - PyUnicodeDecodeError\_SetStart (C function), 53
  - PyUnicodeEncodeError\_GetEncoding (C function), 53
  - PyUnicodeEncodeError\_GetEnd (C function), 53
  - PyUnicodeEncodeError\_GetObject (C function), 53
  - PyUnicodeEncodeError\_GetReason (C function), 53
  - PyUnicodeEncodeError\_GetStart (C function), 53
  - PyUnicodeEncodeError\_SetEnd (C function), 53
  - PyUnicodeEncodeError\_SetReason (C function), 53
  - PyUnicodeEncodeError\_SetStart (C function), 53
  - PyUnicodeObject (C type), 117
  - PyUnicodeTranslateError\_GetEnd (C function), 53
  - PyUnicodeTranslateError\_GetObject (C function), 53
  - PyUnicodeTranslateError\_GetReason (C function), 53
  - PyUnicodeTranslateError\_GetStart (C function), 53
  - PyUnicodeTranslateError\_SetEnd (C function), 53
  - PyUnicodeTranslateError\_SetReason (C function), 53
  - PyUnicodeTranslateError\_SetStart (C function), 53
  - PyVarObject (C type), 220
  - PyVarObject\_HEAD\_INIT (C macro), 221
  - PyVarObject.ob\_size (C member), 233
  - PyVectorcall\_Call (C function), 85
  - PyVectorcall\_Function (C function), 84
  - PyVectorcall\_NARGS (C function), 84
  - PyWeakref\_Check (C function), 156
  - PyWeakref\_CheckProxy (C function), 156
  - PyWeakref\_CheckRef (C function), 156
  - PyWeakref\_GET\_OBJECT (C function), 156
  - PyWeakref\_GetObject (C function), 156
  - PyWeakref\_NewProxy (C function), 156
  - PyWeakref\_NewRef (C function), 156
  - PyWideStringList (C type), 188
  - PyWideStringList\_Append (C function), 188
  - PyWideStringList\_Insert (C function), 188
  - PyWideStringList.items (C member), 188
  - PyWideStringList.length (C member), 188
  - PyWrapper\_New (C function), 153
- ## Q
- qualified name (限定名稱), 276
- ## R
- realloc (C 函数), 209
  - reference count (參照計數), 277
  - regular package (正規套件), 277
  - releasebufferproc (C type), 256
  - repr
    - built-in function (C 函数), 235
    - built-in function (C 函数), 81
  - reprfunc (C type), 256
  - richcmpfunc (C type), 256
- ## S
- stderr
    - stdin stdout, 171
  - search (搜尋)
    - path (路徑), module (模組), 11
    - path (路徑), 模組, 170, 172
  - sendfunc (C type), 256
  - sequence (序列), 277
    - object (物件), 113
  - set comprehension (集合綜合運算), 277
  - set\_all(), 8
  - setattrfunc (C type), 256
  - setattrofunc (C type), 256
  - setswitchinterval (在 sys 模組中), 174
  - set (集合)
    - object (物件), 139
  - SIGINT (C 宏), 51
  - signal (訊號)
    - module (模組), 51
  - single dispatch (單一調度), 277
  - SIZE\_MAX (C 宏), 109
  - slice (切片), 277
  - special
    - method (方法), 277
  - special method (特殊方法), 277
  - ssizeargfunc (C type), 256
  - ssizeobjargproc (C type), 256
  - statement (陳述式), 277
  - static type checker -- 静态类型检查器, 277
  - staticmethod
    - built-in function (C 函数), 224
  - stderr (sys 模組中), 181
  - stdin
    - stdout stderr, 171

stdin (sys 模組中), 181  
 stdout  
     sdterr, stdin, 171  
 stdout (sys 模組中), 181  
 strerror (C 函數), 46  
 string (字串)  
     PyObject\_Str (C 函式), 81  
 strong reference (F 參照), 278  
 sum\_list(), 9  
 sum\_sequence(), 9, 10  
 sys  
     module (模組), 11  
     模組, 170, 181  
 SystemError (F 建例外), 146

## T

ternaryfunc (C type), 256  
 text encoding (文字編碼), 278  
 text file (文字檔案), 278  
 traverseproc (C type), 260  
 triple-quoted string (三引號 F 字串), 278  
 tuple (元組)  
     built-in function (F 建函式), 136  
     object (物件), 133  
 tuple (元組)  
     built-in function (內建函式), 91  
 type alias (型 F 名), 278  
 type hint (型 F 提示), 278  
 type (型 F), 278  
     built-in function (F 建函式), 82  
     object (物件), 6, 103

## U

ULONG\_MAX (C 宏), 109  
 unaryfunc (C type), 256  
 universal newlines (通用 F 行字元), 278  
 USE\_STACKCHECK (C 宏), 58

## V

variable annotation (變數 F 釋), 279  
 vectorcallfunc (C type), 83  
 version (sys 模組中), 173  
 virtual environment (F 擬環境), 279  
 virtual machine (F 擬機器), 279  
 visitproc (C type), 260

## W

模組

\_\_main\_\_, 170, 181  
 \_thread, 176  
 builtins (F 建), 170, 181  
 search (搜尋) path (路徑), 170, 172  
 sys, 170, 181

## Z

Zen of Python (Python 之 F), 279