The Python/C API

發🄵 *3.12.3*

Guido van Rossum and the Python development team

5 月 02, 2024

Python Software Foundation
Email: docs@python.org

Contents

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

CHAPTER 1

簡介

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

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

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

1.1 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🄵： 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🄵明，請參🄵剖析引數與建置數值。

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

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

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

要引入標頭，請將兩個（如果不同）目🄵放在編譯器的引入搜索路徑 (search path) 中。不要將父目🄵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🄵 prefix 下獨立於平台的標頭包括來自 exec_prefix 的平台特定標頭。

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

1.3 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

声明扩展模块 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++ 将函数为声明为 extern "C"。

初始化函数必须命名为 PyInit_*name*，其中 *name* 是模块名称，并且应为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🄵對值。

Added in version 3.3.

Py_ALWAYS_INLINE

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

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

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

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

它必須在函式回傳型🄵之前被指定。用法：

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

Added in version 3.11.

Py_CHARMASK(c)

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

Py_DEPRECATED(version)

將其用於已🄵用的聲明。巨集必須放在符號名稱之前。

範例：

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

在 3.8 版的變更: 新增了 MSVC 支援。

Py_GETENV(s)

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

Py_MAX(x, y)

回傳 `x` 和 `y` 之間的最大值。

Added in version 3.3.

Py_MEMBER_SIZE(type, member)

以位元組🄵單位回傳結構 (`type`) `member` 的大小。

Added in version 3.6.

Py_MIN(x, y)

回傳 `x` 和 `y` 之間的最小值。

Added in version 3.3.

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; }
```

Added in version 3.11.

Py_STRINGIFY(x)

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

Added in version 3.4.

Py_UNREACHABLE()

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

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

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

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

Added in version 3.7.

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; }`。

Added in version 3.4.

PyDoc_STRVAR(name, str)

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

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

範例：

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

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

PyDoc_STR(str)

🄵給定的輸入字串建立一個文件字串，如果文件字串被禁用則建立空字串。

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

範例：

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

1.4 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🄵下，Python 語言以相同的方式處理所有 Python 物件型🄵（例如賦值、作用域規則和引數傳遞），因此它們應該由單個 C 型🄵來表示。幾乎所有的 Python 物件都存在於堆積 (heap) 中：你永遠不會聲明*PyObject* 型🄵的自動變數或🄵態變數，只能聲明 *PyObject** 型🄵的指標變數。唯一的例外是型🄵物件；由於它們🄵不能被釋放，因此它們通常是🄵態*PyTypeObject* 物件。

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

1.4.1 參照計數

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

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

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

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

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

參照計數詳細資訊

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

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

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

```
PyObject *t;

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

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

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");
```

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

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

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

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

重要的是要意識到你是否擁有一個函式回傳的參照只取🄵於你呼叫哪個函式 --- 羽毛 *(plumage)** （作🄵引數傳遞給函式的物件之型🄵）* 不會進入它！因此，如果你使用*PyList_GetItem()* 從 list 中提取一個項目，你不會擁有其參照 --- 但如果你使用*PySequence_GetItem()* 從同一 list 中獲取相同的項目（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)
        return -1; /* Not a list */
```

(繼續下一頁)

(繼續上一頁)

```
    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**

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

1.5 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🄵它們透過堆🄵回溯 (stack trace) 回報給使用者。

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

例外的狀態會在個🄵執行緒的存儲空間 (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（🄵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🄵幀 (stack frame) 參照的物件🄵少了通常不需要的生命🄵期延長。

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

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

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

這是相應的 C 程式碼：

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

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

        /* Clear the error and use zero: */
        PyErr_Clear();
```

(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🄴 `NULL` 的參照（注意名稱中的 `'X'`；*Py_DECREF()* 在遇到 `NULL` 參照時會崩潰）。重要的是，用於保存擁有的參照的變數被初始化🄵 `NULL` 以使其能🄴順利作用；同樣地，回傳值被初始化🄵 `-1`（失敗），🄴且僅在最後一次呼叫成功後才設定🄴成功。

1.6 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🄴是在 Unix 和 Windows 上，🄴管細節略有不同)，*Py_Initialize()* 會假設 Python 函式庫相對於 Python 直譯器可執行檔案的位置固定，🄴根據其對標準 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🄴，它會在 shell 命令搜索路徑（環境變數 `PATH`）中找到名🄵 `python` 的可執行檔案，🄴在其父目🄴中查找一個名🄵 `lib/pythonX.Y` 的目🄴的相對位置。

例如，如果在 `/usr/local/bin/python` 中找到 Python 可執行檔案，它將假定函式庫位於 `/usr/local/lib/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🄵 `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🄵。

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🄵下不🄴用它們。

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

`Py_DEBUG`

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

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

定義 `Py_TRACE_REFS` 來🄴用參照追🄵（參見調用 `--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🄴。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`。

CHAPTER 2

C API 穩定性

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

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

C API 有兩層級，有不同的穩定性期望：

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

下面將更詳細地討論這些🄵容。

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

2.1 不穩定的 C API

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

它通常用於專門的低階工具，例如偵錯器。

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

2.2 穩定的應用程式二進位介面

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

2.2.1 受限 C API

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

Py_LIMITED_API

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

將 `Py_LIMITED_API` 定義🄵對應於你的擴充有支援的最低 Python 版本的*PY_VERSION_HEX* 值。該擴充無需重新編譯即可與從指定版本開始的所有 Python 3 版本一起使用，🄵且可以使用過去版本有引入的受限 API。

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

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

2.2.2 穩定 ABI

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

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

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

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

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

2.2.3 受限 API 范圍和性能

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

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

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

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🄵生一個擴充，可以在特定版本的擴充不可用的地方發布—例如，用於即將發布的 Python 版本的預發布版本 (prerelease)。

2.2.4 受限 API 注意事項

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

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

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

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

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

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

2.3 平台注意事項

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

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

2.4 受限 API 的🄵容

目前，*受限 API* 包括以下項目：

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

- *PyBuffer_FromContiguous()*
- *PyBuffer_GetPointer()*
- *PyBuffer_IsContiguous()*
- *PyBuffer_Release()*
- *PyBuffer_SizeFromFormat()*
- *PyBuffer_ToContiguous()*
- PyByteArrayIter_Type
- *PyByteArray_AsString()*
- *PyByteArray_Concat()*
- *PyByteArray_FromObject()*
- *PyByteArray_FromStringAndSize()*
- *PyByteArray_Resize()*
- *PyByteArray_Size()*
- *PyByteArray_Type*
- PyBytesIter_Type
- *PyBytes_AsString()*
- *PyBytes_AsStringAndSize()*
- *PyBytes_Concat()*
- *PyBytes_ConcatAndDel()*
- PyBytes_DecodeEscape()
- *PyBytes_FromFormat()*
- *PyBytes_FromFormatV()*
- *PyBytes_FromObject()*
- *PyBytes_FromString()*
- *PyBytes_FromStringAndSize()*
- PyBytes_Repr()
- *PyBytes_Size()*
- *PyBytes_Type*
- *PyCFunction*
- *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_DisplayException()*
- *PyErr_ExceptionMatches()*
- *PyErr_Fetch()*
- *PyErr_Format()*
- *PyErr_FormatV()*
- *PyErr_GetExcInfo()*
- *PyErr_GetHandledException()*
- *PyErr_GetRaisedException()*
- *PyErr_GivenExceptionMatches()*
- *PyErr_NewException()*
- *PyErr_NewExceptionWithDoc()*
- *PyErr_NoMemory()*
- *PyErr_NormalizeException()*
- *PyErr_Occurred()*
- *PyErr_Print()*
- *PyErr_PrintEx()*
- PyErr_ProgramText()
- *PyErr_ResourceWarning()*
- *PyErr_Restore()*
- *PyErr_SetExcFromWindowsErr()*
- *PyErr_SetExcFromWindowsErrWithFilename()*
- *PyErr_SetExcFromWindowsErrWithFilenameObject()*
- *PyErr_SetExcFromWindowsErrWithFilenameObjects()*
- *PyErr_SetExcInfo()*
- *PyErr_SetFromErrno()*
- *PyErr_SetFromErrnoWithFilename()*
- *PyErr_SetFromErrnoWithFilenameObject()*
- *PyErr_SetFromErrnoWithFilenameObjects()*
- *PyErr_SetFromWindowsErr()*
- *PyErr_SetFromWindowsErrWithFilename()*
- *PyErr_SetHandledException()*
- *PyErr_SetImportError()*
- *PyErr_SetImportErrorSubclass()*
- *PyErr_SetInterrupt()*
- *PyErr_SetInterruptEx()*
- *PyErr_SetNone()*

- *PyErr_SetObject()*
- *PyErr_SetRaisedException()*
- *PyErr_SetString()*
- *PyErr_SyntaxLocation()*
- *PyErr_SyntaxLocationEx()*
- *PyErr_WarnEx()*
- *PyErr_WarnExplicit()*
- *PyErr_WarnFormat()*
- *PyErr_WriteUnraisable()*
- *PyEval_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_GetArgs()*
- *PyException_GetCause()*
- *PyException_GetContext()*
- *PyException_GetTraceback()*
- *PyException_SetArgs()*
- *PyException_SetCause()*
- *PyException_SetContext()*
- *PyException_SetTraceback()*
- *PyFile_FromFd()*
- *PyFile_GetLine()*
- *PyFile_WriteObject()*
- *PyFile_WriteString()*
- PyFilter_Type
- *PyFloat_AsDouble()*
- *PyFloat_FromDouble()*
- *PyFloat_FromString()*
- *PyFloat_GetInfo()*

- *PyFloat_GetMax()*
- *PyFloat_GetMin()*
- *PyFloat_Type*
- *PyFrameObject*
- *PyFrame_GetCode()*
- *PyFrame_GetLineNumber()*
- *PyFrozenSet_New()*
- *PyFrozenSet_Type*
- *PyGC_Collect()*
- *PyGC_Disable()*
- *PyGC_Enable()*
- *PyGC_IsEnabled()*
- *PyGILState_Ensure()*
- *PyGILState_GetThisThreadState()*
- *PyGILState_Release()*
- PyGILState_STATE
- *PyGetSetDef*
- PyGetSetDescr_Type
- *PyImport_AddModule()*
- *PyImport_AddModuleObject()*
- *PyImport_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
- *PyMember_GetOne()*
- *PyMember_SetOne()*
- *PyMemoryView_FromBuffer()*
- *PyMemoryView_FromMemory()*
- *PyMemoryView_FromObject()*
- *PyMemoryView_GetContiguous()*
- PyMemoryView_Type
- *PyMethodDef*
- PyMethodDescr_Type
- *PyModuleDef*
- PyModuleDef_Base
- *PyModuleDef_Init()*

- PyModuleDef_Type
- *PyModule_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_GetTypeData()*
- *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()*
- *PyObject_Vectorcall()*
- *PyObject_VectorcallMethod()*
- *PyProperty_Type*
- PyRangeIter_Type
- PyRange_Type
- PyReversed_Type
- *PySeqIter_New()*
- *PySeqIter_Type*
- *PySequence_Check()*
- *PySequence_Concat()*
- *PySequence_Contains()*
- *PySequence_Count()*
- *PySequence_DelItem()*
- *PySequence_DelSlice()*

- *PySequence_Fast()*
- *PySequence_GetItem()*
- *PySequence_GetSlice()*
- PySequence_In()
- *PySequence_InPlaceConcat()*
- *PySequence_InPlaceRepeat()*
- *PySequence_Index()*
- *PySequence_Length()*
- *PySequence_List()*
- *PySequence_Repeat()*
- *PySequence_SetItem()*
- *PySequence_SetSlice()*
- *PySequence_Size()*
- *PySequence_Tuple()*
- PySetIter_Type
- *PySet_Add()*
- *PySet_Clear()*
- *PySet_Contains()*
- *PySet_Discard()*
- *PySet_New()*
- *PySet_Pop()*
- *PySet_Size()*
- *PySet_Type*
- *PySlice_AdjustIndices()*
- *PySlice_GetIndices()*
- *PySlice_GetIndicesEx()*
- *PySlice_New()*
- *PySlice_Type*
- *PySlice_Unpack()*
- *PyState_AddModule()*
- *PyState_FindModule()*
- *PyState_RemoveModule()*
- *PyStructSequence_Desc*
- *PyStructSequence_Field*
- *PyStructSequence_GetItem()*
- *PyStructSequence_New()*
- *PyStructSequence_NewType()*
- *PyStructSequence_SetItem()*
- *PyStructSequence_UnnamedField*

- PySuper_Type
- *PySys_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_FromMetaclass()*
- *PyType_FromModuleAndSpec()*
- *PyType_FromSpec()*
- *PyType_FromSpecWithBases()*
- *PyType_GenericAlloc()*
- *PyType_GenericNew()*
- *PyType_GetFlags()*
- *PyType_GetModule()*
- *PyType_GetModuleState()*
- *PyType_GetName()*
- *PyType_GetQualName()*
- *PyType_GetSlot()*

- *PyType_GetTypeDataSize()*
- *PyType_IsSubtype()*
- *PyType_Modified()*
- *PyType_Ready()*
- *PyType_Slot*
- *PyType_Spec*
- *PyType_Type*
- *PyUnicodeDecodeError_Create()*
- *PyUnicodeDecodeError_GetEncoding()*
- *PyUnicodeDecodeError_GetEnd()*
- *PyUnicodeDecodeError_GetObject()*
- *PyUnicodeDecodeError_GetReason()*
- *PyUnicodeDecodeError_GetStart()*
- *PyUnicodeDecodeError_SetEnd()*
- *PyUnicodeDecodeError_SetReason()*
- *PyUnicodeDecodeError_SetStart()*
- *PyUnicodeEncodeError_GetEncoding()*
- *PyUnicodeEncodeError_GetEnd()*
- *PyUnicodeEncodeError_GetObject()*
- *PyUnicodeEncodeError_GetReason()*
- *PyUnicodeEncodeError_GetStart()*
- *PyUnicodeEncodeError_SetEnd()*
- *PyUnicodeEncodeError_SetReason()*
- *PyUnicodeEncodeError_SetStart()*
- PyUnicodeIter_Type
- *PyUnicodeTranslateError_GetEnd()*
- *PyUnicodeTranslateError_GetObject()*
- *PyUnicodeTranslateError_GetReason()*
- *PyUnicodeTranslateError_GetStart()*
- *PyUnicodeTranslateError_SetEnd()*
- *PyUnicodeTranslateError_SetReason()*
- *PyUnicodeTranslateError_SetStart()*
- PyUnicode_Append()
- PyUnicode_AppendAndDel()
- *PyUnicode_AsASCIIString()*
- *PyUnicode_AsCharmapString()*
- PyUnicode_AsDecodedObject()
- PyUnicode_AsDecodedUnicode()
- PyUnicode_AsEncodedObject()

- *PyUnicode_AsEncodedString()*
- PyUnicode_AsEncodedUnicode()
- *PyUnicode_AsLatin1String()*
- *PyUnicode_AsMBCSString()*
- *PyUnicode_AsRawUnicodeEscapeString()*
- *PyUnicode_AsUCS4()*
- *PyUnicode_AsUCS4Copy()*
- *PyUnicode_AsUTF16String()*
- *PyUnicode_AsUTF32String()*
- *PyUnicode_AsUTF8AndSize()*
- *PyUnicode_AsUTF8String()*
- *PyUnicode_AsUnicodeEscapeString()*
- *PyUnicode_AsWideChar()*
- *PyUnicode_AsWideCharString()*
- PyUnicode_BuildEncodingMap()
- *PyUnicode_Compare()*
- *PyUnicode_CompareWithASCIIString()*
- *PyUnicode_Concat()*
- *PyUnicode_Contains()*
- *PyUnicode_Count()*
- *PyUnicode_Decode()*
- *PyUnicode_DecodeASCII()*
- *PyUnicode_DecodeCharmap()*
- PyUnicode_DecodeCodePageStateful()
- *PyUnicode_DecodeFSDefault()*
- *PyUnicode_DecodeFSDefaultAndSize()*
- *PyUnicode_DecodeLatin1()*
- *PyUnicode_DecodeLocale()*
- *PyUnicode_DecodeLocaleAndSize()*
- *PyUnicode_DecodeMBCS()*
- *PyUnicode_DecodeMBCSStateful()*
- *PyUnicode_DecodeRawUnicodeEscape()*
- *PyUnicode_DecodeUTF16()*
- *PyUnicode_DecodeUTF16Stateful()*
- *PyUnicode_DecodeUTF32()*
- *PyUnicode_DecodeUTF32Stateful()*
- *PyUnicode_DecodeUTF7()*
- *PyUnicode_DecodeUTF7Stateful()*
- *PyUnicode_DecodeUTF8()*

- *PyUnicode_DecodeUTF8Stateful()*
- *PyUnicode_DecodeUnicodeEscape()*
- *PyUnicode_EncodeCodePage()*
- *PyUnicode_EncodeFSDefault()*
- *PyUnicode_EncodeLocale()*
- *PyUnicode_FSConverter()*
- *PyUnicode_FSDecoder()*
- *PyUnicode_Find()*
- *PyUnicode_FindChar()*
- *PyUnicode_Format()*
- *PyUnicode_FromEncodedObject()*
- *PyUnicode_FromFormat()*
- *PyUnicode_FromFormatV()*
- *PyUnicode_FromObject()*
- PyUnicode_FromOrdinal()
- *PyUnicode_FromString()*
- *PyUnicode_FromStringAndSize()*
- *PyUnicode_FromWideChar()*
- PyUnicode_GetDefaultEncoding()
- *PyUnicode_GetLength()*
- *PyUnicode_InternFromString()*
- *PyUnicode_InternInPlace()*
- *PyUnicode_IsIdentifier()*
- *PyUnicode_Join()*
- PyUnicode_Partition()
- PyUnicode_RPartition()
- PyUnicode_RSplit()
- *PyUnicode_ReadChar()*
- *PyUnicode_Replace()*
- PyUnicode_Resize()
- *PyUnicode_RichCompare()*
- *PyUnicode_Split()*
- *PyUnicode_Splitlines()*
- *PyUnicode_Substring()*
- *PyUnicode_Tailmatch()*
- *PyUnicode_Translate()*
- *PyUnicode_Type*
- *PyUnicode_WriteChar()*
- *PyVarObject*

- PyVarObject.ob_base
- *PyVarObject.ob_size*
- *PyVectorcall_Call()*
- *PyVectorcall_NARGS()*
- PyWeakReference
- *PyWeakref_GetObject()*
- *PyWeakref_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*
- *getbufferproc*
- *getiterfunc*
- *getter*
- *hashfunc*
- *initproc*
- *inquiry*
- *iternextfunc*
- *lenfunc*
- *newfunc*
- *objobjargproc*
- *objobjproc*
- *releasebufferproc*
- *reprfunc*
- *richcmpfunc*
- *setattrfunc*
- *setattrofunc*
- *setter*
- *ssizeargfunc*
- *ssizeobjargproc*
- ssizessizeargfunc
- ssizessizeobjargproc
- symtable
- *ternaryfunc*
- *traverseproc*
- *unaryfunc*
- *vectorcallfunc*
- *visitproc*

CHAPTER 3

极高层级 API

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)

稳定 ABI 的一部分. 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*PyConfig.inspect* 为零就会这样。

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

稳定 ABI 的一部分 *自 3.8 版本開始.* 类似于*Py_Main()* 但 *argv* 是一个包含字节串的数组。

Added in version 3.8.

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*PyConfig.inspect* 为零就会这样。

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🄵: 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)

🄵稳定 ABI 的一部分. 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。

在 3.12 版的變更: 此函数只能被主解释器 调用。

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。

在 3.12 版的變更: 此函数只能被主解释器 调用。

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。🄵穩定 ABI 的一部分. 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)。

Added in version 3.4.

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。

Added in version 3.2.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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”。

CHAPTER 4

參照計數

本節中的函式與巨集用於管理 Python 物件的參照計數。

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

取得物件 *o* 的參照計數。

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

使用*Py_SET_REFCNT()* 函式設定物件參照計數。

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

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

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

設定物件 *o* 的參照計數。

請注意，此函式對不滅的物件🄵有影響。

Added in version 3.9.

在 3.12 版的變更: 不滅的物件不會被修改。

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

代表取得對於物件 *o* 的新🄵參照，即它正在使用且不應被銷🄴。

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

使用完該物件後，透過呼叫*Py_DECREF()* 來釋放它。

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

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

在 3.12 版的變更: 不滅的物件不會被修改。

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

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

另請見*Py_XNewRef()*。

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

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

當不再需要🄵參照時，應對其呼叫*Py_DECREF()* 以釋放該參照。

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

舉例來🄵：

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

可以寫成：

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

另請參🄵*Py_INCREF()*。

Added in version 3.10.

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

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

如果物件 *o* 🄵 NULL，則該函式僅回傳 NULL。

Added in version 3.10.

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

釋放一個對物件 *o* 的🄵參照，代表該參照不會再被使用。

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

此函式通常用於在退出作用域之前🄵除🄵參照。

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

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

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

在 3.12 版的變更: 不滅的物件不會被修改。

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

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

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

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

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

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

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

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

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

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

Py_SETREF(dst, src)

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

與*Py_CLEAR()* 的情🄵一樣，「明顯的」程式碼可能是致命的：

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

安全的方法是：

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

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

Added in version 3.6.

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

Py_XSETREF(dst, src)

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

Added in version 3.6.

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

CHAPTER 5

例外處理

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会它。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🄴: 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**()

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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* 为非零值，则变量 `sys.last_exc` 将被设为要打印的异常。出于向下兼容性考虑，已弃用的变量 `sys.last_type`, `sys.last_value` 和 `sys.last_traceback` 也会被分别设为该异常的类型, 值和回溯。

在 3.12 版的變更: 增加了对 `sys.last_exc` 的设置。

void **PyErr_Print**()

🄵穩定 ABI 的一部分. `PyErr_PrintEx(1)` 的🄵名。

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

🄵穩定 ABI 的一部分. 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。

该函数调用时将传入单个参数 *obj*，它标识发生不可引发的异常所在的上下文。如果可能，*obj* 的表示形式将打印在警告消息中。如果 *obj* 为 `NULL`，将只打印回溯。

调用此函数时必须设置一个异常。

在 3.4 版的變更: 打印回溯信息。如果 *obj* 为 `NULL` 将只打印回溯。

在 3.8 版的變更: 使用 `sys.unraisablehook()`。

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

🄵穩定 ABI 的一部分 *自 3.12 版本開始.* 将 `exc` 的标准回溯显示打印到 `sys.stderr`，包括链式异常和注释。

Added in version 3.12.

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，以便用于 `return` 语句。

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

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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*。🄵穩定 ABI 的一部分. 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*。🄵穩定 ABI 的一部分 *自 3.5 版本開始.* 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。

Added in version 3.5.

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

🄵穩定 ABI 的一部分. 这是 `PyErr_SetObject(type, Py_None)` 的简写。

int **PyErr_BadArgument**()

🄵穩定 ABI 的一部分. 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*。🄵穩定 ABI 的一部分. 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*。🄵穩定 ABI 的一部分. 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*。🄵穩定 ABI 的一部分. 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*。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.4.

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

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分. 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_SetFromWindowsErr**(int ierr)

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 这是一个用于引发 `OSError` 的便捷函数。如果调用时传入的 *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 `OSError` 对象，其中 `winerror` 属性将设为该错误代码，`strerror` 属性将设为相应的错误消息（从 `FormatMessage()` 获得），然后再调用 `PyErr_SetObject(PyExc_OSError, object)`。该函数将总是返回 `NULL`。

適用：Windows。

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

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 类似于*PyErr_SetFromWindowsErr()*，额外的参数指定要触发的异常类型。

適用：Windows。

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

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 与*PyErr_SetFromWindowsErr()* 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_SetExcFromWindowsErrWithFilenameObject**(*PyObject* *type, int ierr, *PyObject* *filename)

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 与*PyErr_SetExcFromWindowsErr()* 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_SetExcFromWindowsErrWithFilenameObjects**(*PyObject* *type, int ierr, *PyObject* *filename, *PyObject* *filename2)

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 类似于*PyErr_SetExcFromWindowsErrWithFilenameObject()*，但是接受第二个 filename 对象。

適用：Windows。

Added in version 3.4.

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

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 类似于*PyErr_SetFromWindowsErrWithFilename()*，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*。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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`。

Added in version 3.3.

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

回傳值：總是🄵 *NULL*。🄵穩定 ABI 的一部分 自 *3.6* 版本開始. 和*PyErr_SetImportError()* 很类似，但这个函数允许指定一个 `ImportError` 的子类来触发。

Added in version 3.6.

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`。

Added in version 3.4.

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

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 类似于*PyErr_SyntaxLocationObject()*，但 *filename* 是用*filesystem encoding and error handler* 解码的字节串。

Added in version 3.2.

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

🄵穩定 ABI 的一部分. 类似于*PyErr_SyntaxLocationEx()*，但省略了 *col_offset* parameter 形参。

void ***PyErr_BadInternalCall***()

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。

Added in version 3.4.

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

🄵稳定 ABI 的一部分. 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, ...)

🄵稳定 ABI 的一部分. 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。

Added in version 3.2.

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

🄵稳定 ABI 的一部分 自 *3.6* 版本開始. 类似于*PyErr_WarnFormat()* 的函数，但 *category* 是 `ResourceWarning` 并且它会将 *source* 传给 `warnings.WarningMessage`。

Added in version 3.6.

5.4 查询错误指示器

PyObject ***PyErr_Occurred**()

回傳值：借用參照。🄵稳定 ABI 的一部分. 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()*。

调用时必须持有 GIL。

備🄵: 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)

🄵稳定 ABI 的一部分. 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)

🄵稳定 ABI 的一部分. 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。

PyObject ***PyErr_GetRaisedException**(void)

回傳值：新的參照。🄵稳定 ABI 的一部分 自 *3.12* 版本開始. 返回当前被引发的异常，同时清除错误指示器。如果错误指示器尚未设置则返回 `NULL`。

此函数会被需要捕获异常的代码，或需要临时保存和恢复错误指示器的代码所使用。

例如：

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

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

   PyErr_SetRaisedException(exc);
}
```

也參考:

PyErr_GetHandledException()，保存当前正在处理的异常。

Added in version 3.12.

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

🄵稳定 ABI 的一部分 自 *3.12* 版本開始. 将 *exc* 设为当前被引发的异常，如果已设置则清空现有的异常。

警告: 此调用将偷取一个对 *exc* 的引用，它必须是一个有效的异常。

Added in version 3.12.

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

🄵稳定 ABI 的一部分. 在 3.12 版之後被🄵用: 使用*PyErr_GetRaisedException()* 代替。

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🄵: 此函数通常只被需要捕获异常或临时保存和恢复错误指示符的旧式代码所使用。

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)

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

根据 *type*, *value* 和 *traceback* 这三个对象设置错误指示符，如果已设置了错误指示符则先清除它。如果三个对象均为 NULL，则清除错误指示符。请不要传入 NULL 类型和非 NULL 的值或回溯。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继问题。)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并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🄵: 此函数通常只被需要临时保存和恢复错误指示符的旧代码所使用。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)

🄵稳定 ABI 的一部分. 在 3.12 版之後被🄵用: 请改用*PyErr_GetRaisedException()*，以避免任何可能的去正规化。

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🄵: 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)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. 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🄵: 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。

Added in version 3.11.

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

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. 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🄵: 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。

Added in version 3.11.

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

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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🄵: 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。

Added in version 3.3.

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

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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🄵: 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。

Added in version 3.3.

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**()

🄵稳定 ABI 的一部分. 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🄵: 针对 `SIGINT` 的默认 Python 信号处理器会引发 `KeyboardInterrupt` 异常。

---

void **PyErr_SetInterrupt**()

🄵稳定 ABI 的一部分. 模拟一个 `SIGINT` 信号到达的效果。这等价于 `PyErr_SetInterruptEx(SIGINT)`。

備🄵: 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)

🄵稳定 ABI 的一部分 自 *3.10* 版本開始. 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🄵: 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。

Added in version 3.10.

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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。

Added in version 3.2.

5.7 例外物件

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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`。

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

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

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

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 将异常 *ex* 的 `args` 设为 *args*。

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

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

解释器的 `except*` 实现的具体实现部分。*orig* 是被捕获的原始异常，而 *excs* 是需要被引发的异常组成的列表。该列表包含 *orig* 可能存在的未被处理的部分，以及在 `except*` 子句中被引发的异常

（因而它们具有与 *orig* 不同的回溯数据）和被重新引发的异常（因而它们具有与 *orig* 相同的回溯）。返回需要被最终引发的 `ExceptionGroup`，或者如果没有要被引发的异常则返回 `None`。

Added in version 3.12.

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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵稳定 ABI 的一部分 自 *3.9* 版本開始. 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)

🄵稳定 ABI 的一部分 自 *3.9* 版本開始. 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)

🄵稳定 ABI 的一部分. 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)

🄵稳定 ABI 的一部分. 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
`PyExc_BaseException`	`BaseException`	1
`PyExc_Exception`	`Exception`	Page 58, 1
`PyExc_ArithmeticError`	`ArithmeticError`	Page 58, 1
`PyExc_AssertionError`	`AssertionError`	
`PyExc_AttributeError`	`AttributeError`	
`PyExc_BlockingIOError`	`BlockingIOError`	
`PyExc_BrokenPipeError`	`BrokenPipeError`	
`PyExc_BufferError`	`BufferError`	
`PyExc_ChildProcessError`	`ChildProcessError`	

繼續下一頁

表格 1 – 繼續上一頁

C 名称	Python 名称	🄵解
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 58, 1
PyExc_MemoryError	MemoryError	
PyExc_ModuleNotFoundErro	ModuleNotFoundError	
PyExc_NameError	NameError	
PyExc_NotADirectoryError	NotADirectoryError	
PyExc_NotImplementedErro	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	

Added in version 3.3: `PyExc_BlockingIOError`, `PyExc_BrokenPipeError`, `PyExc_ChildProcessError`, `PyExc_ConnectionError`, `PyExc_ConnectionAbortedError`, `PyExc_ConnectionRefusedError`, `PyExc_ConnectionResetError`, `PyExc_FileExistsError`, `PyExc_FileNotFoundError`, `PyExc_InterruptedError`, `PyExc_IsADirectoryError`, `PyExc_NotADirectoryError`, `PyExc_PermissionError`, `PyExc_ProcessLookupError` 和 `PyExc_TimeoutError` 是在 **PEP 3151** 被引入。

Added in version 3.5: `PyExc_StopAsyncIteration` 和 `PyExc_RecursionError`。

Added in version 3.6: `PyExc_ModuleNotFoundError`。

[1] 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	[2]

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	[3]
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	

Added in version 3.2: PyExc_ResourceWarning.

🄵解:

[2] 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。
[3] 这是其他标准警告类别的基类。

CHAPTER 6

工具

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

6.1 作業系統工具

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

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.6* 版本開始. 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`。

Added in version 3.6.

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

如果名称为 *filename* 的标准 Return true (nonzero) if the standard I/O 文件 *fp* 被确认为可交互的则返回真（非零）值。所有 `isatty(fileno(fp))` 为真值的文件都属于这种情况。如果*PyConfig.interactive* 为非零值，此函数在 *filename* 指针为 `NULL` 或者其名称等于字符串 `'<stdin>'` 或 `'???'` 之一时也将返回真值。

此函数不可在 Python 被初始化之前调用。

void **PyOS_BeforeFork**()

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

Added in version 3.7.

void **PyOS_AfterFork_Parent**()

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

Added in version 3.7.

void **PyOS_AfterFork_Child**()

🄴稳定 ABI 的一部分 *on platforms with fork()* 自 *3.7* 版本開始. 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。

Added in version 3.7.

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**()

🄴稳定 ABI 的一部分 *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**()

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

🄴稳定 ABI 的一部分.

PyOS_sighandler_t **PyOS_getsig**(int i)

🄴稳定 ABI 的一部分. 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)

🄴稳定 ABI 的一部分. 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)

🄴稳定 ABI 的一部分 自 *3.7* 版本開始.

> **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。

Added in version 3.5.

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*PyPreConfig.legacy_windows_fs_encoding* 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.5.

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*PyPreConfig.legacy_windows_fs_encoding* 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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**()

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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🄵用.

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

🄵穩定 ABI 的一部分. 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🄵用.

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

🄵穩定 ABI 的一部分. 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🄵用.

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

🄵穩定 ABI 的一部分. 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, ...)

🄵穩定 ABI 的一部分. 类似*PySys_WriteStdout()*，但改为写入到 sys.stderr 或 *stderr*。

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

🄵穩定 ABI 的一部分. 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。

Added in version 3.2.

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

🄵穩定 ABI 的一部分. 类似*PySys_FormatStdout()*，但改为写入到 `sys.stderr` 或 *stderr*。

Added in version 3.2.

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

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.2.

在 3.11 版之後被🄵用.

PyObject ***PySys_GetXOptions**()

回傳值：借用參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.2.

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用入传入的元组。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。

Added in version 3.8.

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。

Added in version 3.8.

6.3 行程控制

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

🄵稳定 ABI 的一部分. 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)

🄵稳定 ABI 的一部分. 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)())

🄵稳定 ABI 的一部分. 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。🄵稳定 ABI 的一部分. 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。🄵稳定 ABI 的一部分. 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。🄵稳定 ABI 的一部分 自 *3.7* 版本開始. 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*。

Added in version 3.3.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分 *自 3.7 版本開始.* 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🄵: 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。

Added in version 3.3.

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

回傳值：借用參照。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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识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()*。

在 3.12 版的變更: __cached__ 和 __loader__ 的设置已被弃用。替代设置参见 ModuleSpec。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.3.

在 3.12 版的變更: __cached__ 的设置已被弃用。替代设置参见 ModuleSpec。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 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。

Added in version 3.2.

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。

在 3.12 版的變更: 不再使用已被移除的 imp 模組。

long **PyImport_GetMagicNumber**()

🄵穩定 ABI 的一部分. 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**()

🄵穩定 ABI 的一部分. 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。

Added in version 3.2.

PyObject ***PyImport_GetModuleDict**()

回傳值：借用參照。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分 自 *3.8* 版本開始. 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。

Added in version 3.7.

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。)

Added in version 3.3.

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)

🄵稳定 ABI 的一部分. 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))

🄵稳定 ABI 的一部分. 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。

数字值在存储时会将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)

将一个 `long` 整数 *value* 以 marshal 格式写入 *file*。这将只写入 *value* 中最低的 32 个比特位；无论本机的 `long` 类型的大小如何。*version* 指明文件格式的版本。

此函数可能失败，在这种情况下它半设置错误提示符。请使用*PyErr_Occurred()* 进行检测。

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。

此函数可能失败，在这种情况下它半设置错误提示符。请使用*PyErr_Occurred()* 进行检测。

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)

从打开用于读取的 `FILE*` 对应的数据流返回一个 C `long`。使用此函数只能读取 32 位的值，无论本机 `long` 类型的大小如何。

发生错误时，将设置适当的异常 (`EOFError`) 并返回 `-1`。

int **PyMarshal_ReadShortFromFile**(FILE *file)

从打开用于读取的 `FILE*` 对应的数据流返回一个 C `short`。使用此函数只能读取 16 位的值，无论本机 `short` 类型的大小如何。

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。 从打开用于读取的 `FILE*` 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`。

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

回傳值：新的參照。 从打开用于读取的 `FILE*` 对应的数据流返回一个 Python 对象。不同于*PyMarshal_ReadObjectFromFile()*，此函数假定将不再从该文件读取更多的对象，允许其将文件数据积极地载入内存，以便反序列化过程可以在内存中的数据上操作而不是每次从文件读取一个字节。只有当你确定不会再从文件读取任何内容时方可使用此形式。

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🄴: 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🄴:** 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) [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，它将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。

在 3.12 版的變更: u, u#, Z 和 Z# 已被移除因为它们只用于旧式的 Py_UNICODE* 表示形式。

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]
將一個 Python 整數轉🄵成 C 的 short int。

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

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

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

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

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

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]
將一个 Python 字节类型，如一个长度为 1 的 bytes 或 bytearray 对象，转换为 C char 。

在 3.3 版的變更: 允許 bytearray 物件。

C（長度🄵 1 的 str）[int]
將一个 Python 字符，如一个长度为 1 的 str 对象，转换为 C int。

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

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

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

其他物件

O（物件）[PyObject *]
将 Python 对象（未经任何转换）存储到一个 C 对象指针中。这样 C 程序就能接收到实际传递的对象。对象的新*strong reference* 不会被创建（即其引用计数不会增加）。存储的指针将不为 NULL。

O!（物件）[*typeobject*, PyObject *]
将一个 Python 对象存入一个 C 对象指针。这类似于 O，但是接受两个 C 参数：第一个是 Python 类型对象的地址，第二个是存储对象指针的 C 变量 (类型为 *PyObject**)。如果 Python 对象不具有所要求的类型，则会引发 TypeError。

O&（物件）[*converter*, *anything*]
通过 *converter* 函数将 Python 对象转换为 C 变量。这需要两个参数：第一个是函数，第二个是 C 变量（任意类型）的地址，转换为 void*。转换器函数依次调用如下：

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

其中 *object* 是待转换的 Python 对象而 *address* 为传给 PyArg_Parse* 函数的 void* 参数。返回的 *status* 应当以 1 代表转换成功而以 0 代表转换失败。当转换失败时，*converter* 函数应当引发异常并让 *address* 的内容保持未修改状态。

如果 *converter* 返回 Py_CLEANUP_SUPPORTED，则如果参数解析最终失败，它可能会再次调用该函数，从而使转换器有机会释放已分配的任何内存。在第二个调用中，*object* 参数将为 NULL；因此，该参数将为 NULL；因此，该参数将为 NULL，因此，该参数将为 NULL（如果值）为 NULL *address* 的值与原始呼叫中的值相同。

在 3.1 版的變更: 加入 Py_CLEANUP_SUPPORTED。

p (bool) [int]
测试传入的值是否为真 (一个布尔判断) 并且将结果转化为相对应的 C true/false 整型值。如果表达式为真置 1，假则置 0。它接受任何合法的 Python 值。参见 truth 获取更多关于 Python 如何测试值为真的信息。

Added in version 3.3.

(items) (tuple) [*matching-items*]
对象必须是 Python 序列，它的长度是 *items* 中格式单元的数量。C 参数必须对应 *items* 中每一个独立的格式单元。序列中的格式单元可能有嵌套。

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。

Added in version 3.3.

:
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, ...)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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[], ...)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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**)

🄵穩定 ABI 的一部分. 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。

Added in version 3.2.

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

🄵穩定 ABI 的一部分. 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, ...)

🄵穩定 ABI 的一部分. 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；它们将以来自 *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;
```

(繼續下一頁)

(繼續上一頁)

```
    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：新的參照。🄵穩定 ABI 的一部分. 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🄵一的 `bytes`。

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

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

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

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

O（物件）[PyObject *]
原封不动地传递一个 Python 对象，但为其创建一个新的*strong reference* (即其引用计数加一)。如果传入的对象是一个 `NULL` 指针，则会假定这是因为产生该参数的调用发现了错误并设置了异常。因此，*Py_BuildValue()* 将返回 `NULL` 但不会引发异常。如果尚未引发异常，则会设置 `SystemError`。

S（物件）[PyObject *]
和 `O` 相同。

N（物件）[PyObject *]
: 与 O 相同，但它不会创建新的*strong reference*。如果对象是通过调用参数列表中的对象构造器来创建的则该方法将很有用处。

O&（物件）[*converter*, *anything*]
: 通过 *converter* 函数将 *anything* 转换为 Python 对象。该函数在调用时附带 *anything* (它应当兼容 `void*`) 作为其参数并且应返回一个" 新的" Python 对象，或者如果发生错误则返回 `NULL`。

(items) (**tuple**) [*matching-items*]
: 将一个 C 变量序列转换成 Python 元组并保持相同的元素数量。

[items] (**list**) [*matching-items*]
: 将一个 C 变量序列转换成 Python 列表并保持相同的元素数量。

{items} (**dict**) [*matching-items*]
: 将一个 C 变量序列转换成 Python 字典。每一对连续的 C 变量对作为一个元素插入字典中，分别作为关键字和值。

如果格式字符串中出现错误，则设置 `SystemError` 异常并返回 `NULL`。

PyObject ***Py_VaBuildValue**(const char *format, va_list vargs)
: 回傳值：新的參照。🄵穩定 ABI 的一部分. 和*Py_BuildValue()* 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。

6.7 字串轉🄵與格式化

用於數字轉🄵和格式化字串輸出的函式。

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

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

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

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

這些函式的回傳值 (*rv*) 應如下被直譯：

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

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

unsigned long **PyOS_strtoul**(const char *str, char **ptr, int base)
: 🄵穩定 ABI 的一部分. 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。

也請見 Unix 手🄵頁面 *strtoul(3)*。

Added in version 3.2.

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

🄵穩定 ABI 的一部分. 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。

也請見 Unix 手🄵頁面 *strtol(3)*。

Added in version 3.2.

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

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

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

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

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

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

Added in version 3.1.

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

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

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

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

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

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

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

Added in version 3.1.

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

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

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

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

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。

Added in version 3.2.

type **Py_uhash_t**

哈希值类型：无符号整数。

Added in version 3.2.

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。

Added in version 3.4.

PyHash_FuncDef ***PyHash_GetFuncDef**(void)

获取哈希函数定义。

也參考:

PEP 456 ”安全且可互换的哈希算法”。

Added in version 3.4.

6.9 反射

PyObject ***PyEval_GetBuiltins**(void)

回傳值：借用參照。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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 編解碼器🄵🄵表和支援函式

int __PyCodec_Register__(*PyObject* *search_function)

🄵穩定 ABI 的一部分. 🄵🄵一個新的編解碼器搜索函式。

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

int __PyCodec_Unregister__(*PyObject* *search_function)

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

Added in version 3.10.

int __PyCodec_KnownEncoding__(const char *encoding)

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

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

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

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

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

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

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

6.10.1 編解碼器查找 API

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

PyObject *__PyCodec_Encoder__(const char *encoding)

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

PyObject *__PyCodec_Decoder__(const char *encoding)

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

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

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

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

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

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

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

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

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

6.10.2 用於 Unicode 編碼錯誤處理程式的🄵🄵 API

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

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

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

成功時回傳 `0`，錯誤時回傳 `-1`。

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Added in version 3.5.

6.11 对 Perf Maps 的支持

在受支持的平台上（在撰写本文档时，只有 Linux），运行时可以利用 *perf map* 文件来使得 Python 函数对于外部性能分析工具可见（例如 perf 等）。正在运行的进行可以在 `/tmp` 目录中创建一个文件，其中包含可将部分可执行代码映射到特定名称的条目。本接口的描述参见 Linux Perf 工具文档。

在 Python 中，这些辅助 API 可供依赖于动态生成机器码的库和特性使用。

请注意这些 API 并不要求持有全局解释器锁（GIL）。

int **PyUnstable_PerfMapState_Init**(void)

这是不稳定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

打开 `/tmp/perf-$pid.map` 文件，除非它已经被打开，并创建一个锁来确保线程安全地写入该文件（如果写入是通过*PyUnstable_WritePerfMapEntry()* 执行的）。通常，没有必要显式地调用此函数；只需使用*PyUnstable_WritePerfMapEntry()* 这样它将在第一次调用时初始化状态。

成功时返回 0，创建/打开 perf map 文件失败时返回 -1，或者创建锁失败时返回 -2。可检查 errno 获取有关失败原因的更多信息。

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

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

向 /tmp/perf-$pid.map 文件写入一个单独条目。此函数是线程安全的。下面显示了一个示例条目:

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

将在写入条目之前调用*PyUnstable_PerfMapState_Init()*，如果 perf map 文件尚未打开。成功时返回 0，或者在失败时返回与*PyUnstable_PerfMapState_Init()* 相同的错误代码。

void **PyUnstable_PerfMapState_Fini**(void)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

关闭*PyUnstable_PerfMapState_Init()* 所打开的 perf map 文件。此函数会在解释器关闭期间由运行时本身调用。通常，应该没有理由显式地调用此函数，除了处理特殊场景例如分叉操作。

CHAPTER 7

抽象物件層 (Abstract Objects Layer)

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

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

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()*) 一起使用的旗标。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)

🄵穩定 ABI 的一部分. 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🄵: 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)

🄵穩定 ABI 的一部分. 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🄵: 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处理，请改用*PyObject_GetAttrString()*。

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

回傳值：新的參照。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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，则将比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。🄵穩定 ABI 的一部分 自 *3.10* 版本開始. `__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。

此函数还可能会被调用以获取对象 *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。

Added in version 3.3.

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

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. `__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。

Added in version 3.3.

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：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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🄴: 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)

🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)`。

Added in version 3.4.

PyObject ***PyObject_GetItem**(*PyObject* *o, *PyObject* *key)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 相当于 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。🄵穩定 ABI 的一部分. 等同于 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。🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 等同于 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。

Added in version 3.10.

void ***PyObject_GetTypeData**(*PyObject* *o, *PyTypeObject* *cls)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 获取一个指向为 *cls* 保留的子类专属数据的指针。

对象 *o* 必须为 *cls* 的实例，而 *cls* 必须使用负的*PyType_Spec.basicsize* 来创建。Python 不会检查这一点。

发生错误时，将设置异常并返回 NULL。

Added in version 3.12.

Py_ssize_t **PyType_GetTypeDataSize**(*PyTypeObject* *cls)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 返回为 *cls* 保留的实例内存空间大小，即*PyObject_GetTypeData()* 所返回的内存大小。

这可能会大于使用*-PyType_Spec.basicsize* 请求到的大小；可以安全地使用这个更大的值 (例如通过 `memset()`)。

类型 *cls* **必须**使用负的*PyType_Spec.basicsize* 来创建。Python 不会检查这一点。

当失败时，将设置异常并返回一个负值。

Added in version 3.12.

void ***PyObject_GetItemData**(*PyObject* *o)

使用*Py_TPFLAGS_ITEMS_AT_END* 获取一个指向类的单独条目数据的指针。

出错时，将设置异常并返回 NULL。如果 *o* 没有设置*Py_TPFLAGS_ITEMS_AT_END* 则会引发 TypeError。

Added in version 3.12.

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🄵 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

Added in version 3.9.

Vectorcall 協定是在 **PEP 590** 被引入的，它是使函式呼叫更加有效率的附加協定。

經驗法則上，如果可呼叫物件有支援，CPython 於🄵部呼叫中會更傾向使用 vectorcall。然而，這🄴不是一個硬性規定。此外，有些第三方擴充套件會直接使用 *tp_call*（而不是使用*PyObject_Call()*）。因此，一個支援 vectorcall 的類🄴也必須實作*tp_call*。此外，無論使用哪種協定，可呼叫物件的行🄵都必須是相同的。要達成這個目的的推薦做法是將*tp_call* 設定🄵*PyVectorcall_Call()*。這值得一再提醒：

> **警告：** 一個支援 vectorcall 的類🄴**必須**也實作具有相同語義的*tp_call*。

在 3.12 版的變更: 現在*Py_TPFLAGS_HAVE_VECTORCALL* 旗标在类的 `__call__()` 方法被重新赋值时将会从类中移除。（这将仅在内部设置*tp_call*，因此可能使其行为不同于 vectorcall 函数。）在更早的 Python 版本中，vectorcall 应当仅被用于不可变对象 或静态类型。

如果一個類🄴的 vectorcall 比 *tp_call* 慢，就不應該實作 vectorcall。例如，如果被呼叫者需要將引數轉🄵🄴 args tuple（引數元組）和 kwargs dict（關鍵字引數字典），那🄵實作 vectorcall 就🄵有意義。

類🄴可以透過🄵用*Py_TPFLAGS_HAVE_VECTORCALL* 旗標🄵將*tp_vectorcall_offset* 設定🄵物件結構中有出現 *vectorcallfunc* 的 offset 來實作 vectorcall 協定。這是一個指向具有以下簽章之函式的指標：

typedef *PyObject* *(***vectorcallfunc**)(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始.

- *callable* 是指被呼叫的物件。
- ***args* 是一個 C 語言陣列 (array)，包含位置引數與後面**
  關鍵字引數的值。如果🄴有引數，這個值可以是 *NULL*。
- ***nargsf* 是位置引數的數量加上可能會有的**
  PY_VECTORCALL_ARGUMENTS_OFFSET 旗標。如果要從 *nargsf* 獲得實際的位置引數數量，請使用*PyVectorcall_NARGS()*。
- ***kwnames* 是一個包含所有關鍵字引數名稱的 tuple；**
  🄵句話🄵，就是 kwargs 字典的鍵。這些名字必須是字串（`str` 或其子類🄴的實例），🄵且它們必須是不重🄴的。如果🄴有關鍵字引數，那🄵 *kwnames* 可以用 *NULL* 代替。

PY_VECTORCALL_ARGUMENTS_OFFSET

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 如果在 vectorcall 的 *nargsf* 引數中設定了此旗標，則允許被呼叫者臨時更改 `args[-1]` 的值。🄵句話🄵，*args* 指向向量中的引數 1（不是 0）。被呼叫方必須在回傳之前還原 `args[-1]` 的值。

對於*PyObject_VectorcallMethod()*，這個旗標的改變意味著可能是 `args[0]` 被改變。

當可以以幾乎無代價的方式（無需🄵據額外的記憶體）來達成，那🄵會推薦呼叫者使用*PY_VECTORCALL_ARGUMENTS_OFFSET*。這樣做會讓如 bound method（🄵結方法）之類的可呼叫函式非常有效地繼續向前呼叫（這類函式包含一個在首位的 *self* 引數）。

Added in version 3.8.

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🄵: 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🄵名。

遞🄵控制

在使用 *tp_call* 時，被呼叫者不必擔心遞🄵: CPython 對於使用 *tp_call* 的呼叫會使用*Py_EnterRecursiveCall()* 和*Py_LeaveRecursiveCall()*。

🄵保證效率，這不適用於使用 vectorcall 的呼叫：被呼叫方在需要時應當使用 *Py_EnterRecursiveCall* 和 *Py_LeaveRecursiveCall*。

Vectorcall 支援 API

Py_ssize_t **PyVectorcall_NARGS**(size_t nargsf)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 給定一個 vectorcall *nargsf* 引數，回傳引數的實際數量。目前等同於：

```
(Py_ssize_t)(nargsf & ~PY_VECTORCALL_ARGUMENTS_OFFSET)
```

然而，應使用 `PyVectorcall_NARGS` 函式以便將來需要擴充。

Added in version 3.8.

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。

Added in version 3.9.

PyObject ***PyVectorcall_Call**(*PyObject* *callable, *PyObject* *tuple, *PyObject* *dict)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 呼叫 *callable* 的*vectorcallfunc*，其位置引數和關鍵字引數分🄵以 tuple 和 dict 格式給定。

這是一個專門函式，其目的是被放入*tp_call* 擴充槽或是用於 `tp_call` 的實作。它不會檢查*Py_TPFLAGS_HAVE_VECTORCALL* 旗標🄵且它不會退回 (fall back) 使用 `tp_call`。

Added in version 3.8.

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🄵🄵被呼叫物件所支援的慣用形式–可以是 *tp_call* 或 vectorcall。🄵了🄵可能🄵少轉🄵的進行，請選擇一個適合你所擁有資料格式的函式。

下表總結了可用的函式；請參🄵各個🄵明文件以🄵解詳情。

函式	callable	args	kwargs
PyObject_Call()	`PyObject *`	tuple	dict/`NULL`
PyObject_CallNoArgs()	`PyObject *`	---	---
PyObject_CallOneArg()	`PyObject *`	一個物件	---
PyObject_CallObject()	`PyObject *`	tuple/`NULL`	---
PyObject_CallFunction()	`PyObject *`	format	---
PyObject_CallMethod()	物件 + `char*`	format	---
PyObject_CallFunctionObjArgs()	`PyObject *`	可變引數	---
PyObject_CallMethodObjArgs()	物件 + 名稱	可變引數	---
PyObject_CallMethodNoArgs()	物件 + 名稱	---	---
PyObject_CallMethodOneArg()	物件 + 名稱	一個物件	---
PyObject_Vectorcall()	`PyObject *`	vectorcall	vectorcall
PyObject_VectorcallDict()	`PyObject *`	vectorcall	dict/`NULL`
PyObject_VectorcallMethod()	引數 + 名稱	vectorcall	vectorcall

PyObject ***PyObject_Call**(*PyObject* *callable, *PyObject* *args, *PyObject* *kwargs)

回傳值：新的參照。🄵穩定 ABI 的一部分. 呼叫一個可呼叫的 Python 物件 *callable*，附帶由 tuple *args* 所給定的引數及由字典 *kwargs* 所給定的關鍵字引數。

args 必須不🄵 *NULL*；如果不需要引數，請使用一個空 tuple。如果不需要關鍵字引數，則 *kwargs* 可以🄵 *NULL*。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

這等價於 Python 運算式 `callable(*args, **kwargs)`。

PyObject ***PyObject_CallNoArgs**(*PyObject* *callable)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 呼叫一個可呼叫的 Python 物件 *callable* 🄵不附帶任何引數。這是不帶引數呼叫 Python 可呼叫物件的最有效方式。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

Added in version 3.9.

PyObject ***PyObject_CallOneArg**(*PyObject* *callable, *PyObject* *arg)

回傳值：新的參照。 呼叫一個可呼叫的 Python 物件 *callable* 🄵附帶正好一個位置引數 *arg* 而🄵有關鍵字引數。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

Added in version 3.9.

PyObject ***PyObject_CallObject**(*PyObject* *callable, *PyObject* *args)

回傳值：新的參照。🄵穩定 ABI 的一部分. 呼叫一個可呼叫的 Python 物件 *callable*，附帶由 tuple *args* 所給定的引數。如果不需要傳入引數，則 *args* 可以🄵 *NULL*。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

這等價於 Python 運算式 `callable(*args)`。

PyObject ***PyObject_CallFunction**(*PyObject* *callable, const char *format, ...)

回傳值：新的參照。🄵穩定 ABI 的一部分. 呼叫一個可呼叫的 Python 物件 *callable*，附帶數量可變的 C 引數。這些 C 引數使用*Py_BuildValue()* 風格的格式字串來描述。格式可以🄵 *NULL*，表示🄵有提供任何引數。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

這等價於 Python 運算式 `callable(*args)`。

注意，如果你只傳入 *PyObject** 引數，則*PyObject_CallFunctionObjArgs()* 是另一個更快速的選擇。

在 3.4 版的變更: 這個 *format* 的型🄵已從 `char *` 更改。

PyObject ***PyObject_CallMethod**(*PyObject* *obj, const char *name, const char *format, ...)

回傳值：新的參照。🄵穩定 ABI 的一部分. 呼叫 *obj* 物件中名🄵 *name* 的 method 🄵附帶數量可變的 C 引數。這些 C 引數由*Py_BuildValue()* 格式字串來描述，🄵應當生成一個 tuple。

格式可以🄵 *NULL*，表示🄵有提供任何引數。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

這等價於 Python 運算式 `obj.name(arg1, arg2, ...)`。

注意，如果你只傳入 *PyObject** 引數，則*PyObject_CallMethodObjArgs()* 是另一個更快速的選擇。

在 3.4 版的變更: *name* 和 *format* 的型🄵已從 `char *` 更改。

PyObject ***PyObject_CallFunctionObjArgs**(*PyObject* *callable, ...)

回傳值：新的參照。🄵穩定 ABI 的一部分. 呼叫一個可呼叫的 Python 物件 *callable*，附帶數量可變的 *PyObject** 引數。這些引數是以位置在 *NULL* 後面、數量可變的參數來提供。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

這等價於 Python 運算式 `callable(arg1, arg2, ...)`。

PyObject ***PyObject_CallMethodObjArgs**(*PyObject* *obj, *PyObject* *name, ...)

回傳值：新的參照。🄵穩定 ABI 的一部分. 呼叫 Python 物件 *obj* 中的一個 method，其中 method 名稱由 *name* 中的 Python 字串物件給定。被呼叫時會附帶數量可變的 *PyObject** 引數。這些引數是以位置在 *NULL* 後面、且數量可變的參數來提供。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

PyObject ***PyObject_CallMethodNoArgs**(*PyObject* *obj, *PyObject* *name)

不附帶任何引數地呼叫 Python 物件 *obj* 中的一個 method，其中 method 名稱由 *name* 中的 Python 字串物件給定。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

Added in version 3.9.

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*。

Added in version 3.9.

PyObject ***PyObject_Vectorcall**(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 呼叫一個可呼叫的 Python 物件 *callable*。附帶引數與*vectorcallfunc* 的相同。如果 *callable* 支援*vectorcall*，則它會直接呼叫存放在 *callable* 中的 vectorcall 函式。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

Added in version 3.9.

PyObject ***PyObject_VectorcallDict**(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwdict)

附帶與在*vectorcall* 協定中傳入的相同位置引數來呼叫 *callable*，但會加上以字典 *kwdict* 格式傳入的關鍵字引數。*args* 陣列將只包含位置引數。

無論🄵部使用了哪一種協定，都會需要進行引數的轉🄵。因此，此函式應該只有在呼叫方已經擁有一個要作🄵關鍵字引數的字典、但🄵有作🄵位置引數的 tuple 時才被使用。

Added in version 3.9.

PyObject ***PyObject_VectorcallMethod**(*PyObject* *name, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 使用 vectorcall 呼叫慣例來呼叫一個 method。method 的名稱以 Python 字串 *name* 的格式給定。被呼叫 method 的物件🄵 *args[0]*，而 *args* 陣列從 *args[1]* 開始的部分則代表呼叫的引數。必須傳入至少一個位置引數。*nargsf* 🄵包括 *args[0]* 在🄵的位置引數的數量，如果 `args[0]` 的值可能被臨時改變則要再加上*PY_VECTORCALL_ARGUMENTS_OFFSET*。關鍵字引數可以像在*PyObject_Vectorcall()* 中一樣被傳入。

如果物件具有*Py_TPFLAGS_METHOD_DESCRIPTOR* 特性，這將以完整的 *args* 向量作🄵引數來呼叫 unbound method（未🄵結方法）物件。

成功時回傳結果，或在失敗時引發一個例外🄵回傳 *NULL*。

Added in version 3.9.

7.2.4 呼叫支援 API

int **PyCallable_Check**(*PyObject* *o)

🄵穩定 ABI 的一部分. 判定物件 *o* 是否🄵可呼叫的。如果物件是可呼叫物件則回傳 `1`，其他情🄵回傳 `0`。這個函式不會呼叫失敗。

7.3 数字协议

int **PyNumber_Check**(*PyObject* *o)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 返回 *o1* 减去 *o2* 的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 - o2`。

PyObject ***PyNumber_Multiply**(*PyObject* *o1, *PyObject* *o2)

回傳值: 新的參照。🄵穩定 ABI 的一部分. 返回 *o1* 、*o2* 相乘的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 * o2`。

PyObject ***PyNumber_MatrixMultiply**(*PyObject* *o1, *PyObject* *o2)

回傳值: 新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 返回 *o1* 、*o2* 做矩阵乘法的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 @ o2`。

Added in version 3.5.

PyObject ***PyNumber_FloorDivide**(*PyObject* *o1, *PyObject* *o2)

回傳值: 新的參照。🄵穩定 ABI 的一部分. 返回 *o1* 除以 *o2* 向下取整的值，失败时返回 `NULL`。这等价于 Python 表达式 `o1 // o2`。

PyObject ***PyNumber_TrueDivide**(*PyObject* *o1, *PyObject* *o2)

回傳值: 新的參照。🄵穩定 ABI 的一部分. 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值。此函数等价于 Python 表达式 `o1 / o2`。

PyObject ***PyNumber_Remainder**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回 *o1* 除以 *o2* 得到的余数，如果失败，返回 NULL。等价于 Python 表达式 `o1 % o2`。

PyObject ***PyNumber_Divmod**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. 参考内置函数 `divmod()`。如果失败，返回 NULL。等价于 Python 表达式 `divmod(o1, o2)`。

PyObject ***PyNumber_Power**(*PyObject* *o1, *PyObject* *o2, *PyObject* *o3)

回傳值：新的參照。🄵穩定 ABI 的一部分. 请参阅内置函数 `pow()`。如果失败，返回 NULL。等价于 Python 中的表达式 `pow(o1, o2, o3)`，其中 *o3* 是可选的。如果要忽略 *o3*，则需传入*Py_None* 作为代替（如果传入 NULL 会导致非法内存访问）。

PyObject ***PyNumber_Negative**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回 *o* 的负值，如果失败，返回 NULL 。等价于 Python 表达式 `-o`。

PyObject ***PyNumber_Positive**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回 *o*，如果失败，返回 NULL 。等价于 Python 表达式 `+o`。

PyObject ***PyNumber_Absolute**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回 *o* 的绝对值，如果失败，返回 NULL。等价于 Python 表达式 `abs(o)`。

PyObject ***PyNumber_Invert**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回 *o* 的按位取反后的结果，如果失败，返回 NULL。等价于 Python 表达式 `~o`。

PyObject ***PyNumber_Lshift**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回 *o1* 左移 *o2* 个比特后的结果，如果失败，返回 NULL。等价于 Python 表达式 `o1 << o2`。

PyObject ***PyNumber_Rshift**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回 *o1* 右移 *o2* 个比特后的结果，如果失败，返回 NULL 。等价于 Python 表达式 `o1 >> o2`。

PyObject ***PyNumber_And**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回 *o1* 和 *o2* “按位与”的结果，如果失败，返回 NULL 。等价于 Python 表达式 `o1 & o2`。

PyObject ***PyNumber_Xor**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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`。

Added in version 3.5.

PyObject ***PyNumber_InPlaceFloorDivide**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。等价与 Python 语句 `o1 &= o2`。

PyObject ***PyNumber_InPlaceXor**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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。等价与 Python 语句 `o1 ^= o2`。

PyObject ***PyNumber_InPlaceOr**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. 成功时返回 *o1* 和 *o2* " 按位或" 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_Long**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 成功时返回 *o* 转换为整数对象后的结果，失败时返回 `NULL`。等价于 Python 表达式 `int(o)`。

PyObject ***PyNumber_Float**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 成功时返回 *o* 转换为浮点对象后的结果，失败时返回 `NULL`。等价于 Python 表达式 `float(o)`。

PyObject ***PyNumber_Index**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 成功时返回 *o* 转换为 Python int 类型后的结果，失败时返回 `NULL` 并引发 `TypeError` 异常。

在 3.10 版的變更: 结果总是为 `int` 类型。在之前版本中，结果可能为 `int` 的子类的实例。

PyObject ***PyNumber_ToBase**(*PyObject* *n, int base)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回整数 *n* 转换成以 *base* 为基数的字符串后的结果。这个 *base* 参数必须是 2，8，10 或者 16 。对于基数 2，8，或 16 ，返回的字符串将分别加上基数标识 `'0b'`, `'0o'`, or `'0x'`。如果 *n* 不是 Python 中的整数 *int* 类型，就先通过*PyNumber_Index()* 将它转换成整数类型。

Py_ssize_t **PyNumber_AsSsize_t**(*PyObject* *o, *PyObject* *exc)

🄵穩定 ABI 的一部分. 如果 *o* 可以被解读为一个整数则返回 *o* 转换成的*Py_ssize_t* 值。如果调用失败，则会引发一个异常并返回 `-1`。

如果 *o* 可以被转换为 Python 的 int 值但尝试转换为*Py_ssize_t* 值则会引发 `OverflowError`，则 *exc* 参数将为所引发的异常类型 (通常为 `IndexError` 或 `OverflowError`)。如果 *exc* 为 `NULL`，则异常会被清除并且值会在为负整数时被裁剪为 `PY_SSIZE_T_MIN` 而在为正整数时被裁剪为 `PY_SSIZE_T_MAX`。

int **PyIndex_Check**(*PyObject* *o)

🄵穩定 ABI 的一部分 自 *3.8* 版本開始. 返回 `1` 如果 *o* 是一个索引整数（将 `nb_index` 槽位填充到 `tp_as_number` 结构体），或者在其他情况下返回 `0`。此函数总是会成功执行。

7.4 序列协议

int **PySequence_Check**(*PyObject* *o)

🄵穩定 ABI 的一部分. 如果对象提供了序列协议则返回 `1`，否则返回 `0`。请注意对于具有 `__getitem__()` 方法的 Python 类返回 `1` ，除非它们是 `dict` 的子类，因为在通常情况下无法确定这种类支持哪种键类型。该函数总是会成功执行。

Py_ssize_t **PySequence_Size**(*PyObject* *o)

Py_ssize_t **PySequence_Length**(*PyObject* *o)

🄵穩定 ABI 的一部分. 成功时返回序列中 *o* 的对象数, 失败时返回 `-1` . 相当于 Python 的 `len(o)` 表达式.

PyObject ***PySequence_Concat**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. 成功时返回 *o1* 和 *o2* 的拼接，失败时返回 `NULL`。这等价于 Python 表达式 `o1 + o2`。

PyObject ***PySequence_Repeat**(*PyObject* *o, *Py_ssize_t* count)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回序列对象 *o* 重复 *count* 次的结果，失败时返回 `NULL`。这等价于 Python 表达式 `o * count`。

PyObject ***PySequence_InPlaceConcat**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。🄵穩定 ABI 的一部分. 成功时返回 *o1* 和 *o2* 的拼接，失败时返回 `NULL`。在 *o1* 支持的情况下操作将 原地完成。这等价于 Python 表达式 `o1 += o2`。

PyObject ***PySequence_InPlaceRepeat**(*PyObject* *o, *Py_ssize_t* count)

回傳值：新的參照。🄵穩定 ABI 的一部分. Return the result of repeating sequence object 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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 `NULL`，并且 *i* d 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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🄵： 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)

🄵穩定 ABI 的一部分. 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🄵： 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 成功時，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 🄵代器協議

有兩個專門用於🄵代器的函式。

int **PyIter_Check**(*PyObject* *o)

🄵穩定 ABI 的一部分 自 *3.8* 版本開始. 如果物件 *o* 可以安全地傳遞給*PyIter_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 **PyAIter_Check**(*PyObject* *o)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 如果物件 *o* 有提供 AsyncIterator 協議，則回傳非零，否則回傳 0。這個函式一定會執行成功。

Added in version 3.10.

PyObject ***PyIter_Next**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 回傳🄵代器 *o* 的下一個值。根據*PyIter_Check()*，該物件必須是一個🄵代器（由呼叫者檢查）。如果🄵有剩余值，則回傳 NULL 且不設定例外。如果檢索項目時發生錯誤，則回傳 NULL 🄵傳遞例外。

要編寫一個🄵代於🄵代器的🄵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 */
}
else {
    /* continue doing useful work */
}
```

type **PySendResult**

用於表示*PyIter_Send()* 不同結果的列舉 (enum) 值。

Added in version 3.10.

PySendResult **PyIter_Send**(*PyObject* *iter, *PyObject* *arg, *PyObject* **presult)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 將 *arg* 值發送到🄵代器 *iter* 中。回傳：

- 如果🄵代器有回傳則🄵 PYGEN_RETURN。回傳值透過 *presult* 回傳。
- 如果🄵代器有🄵生 (yield) 則🄵 PYGEN_NEXT。🄵生值透過 *presult* 回傳。

- 如果🄵代器引發例外則🄵 PYGEN_ERROR。*presult* 被設定🄵 NULL。

Added in version 3.10.

7.7 緩沖協定 (Buffer Protocol)

在 Python 中可使用一些对象来包装对底层内存数组或称 *缓冲* 的访问。此类对象包括内置的 bytes 和 bytearray 以及一些如 array.array 这样的扩展类型。第三方库也可能会为了特殊的目的而定义它们自己的类型，例如用于图像处理和数值分析等。

虽然这些类型中的每一种都有自己的语义，但它们具有由可能较大的内存缓冲区支持的共同特征。在某些情况下，希望直接访问该缓冲区而无需中间复制。

Python 以*缓冲协议* 的形式在 C 层级上提供这样的功能。此协议包括两个方面:

- 在生产者这一方面，该类型的协议可以导出一个 “缓冲区接口”，允许公开它的底层缓冲区信息。该接口的描述信息在*缓冲区对象结构体* 一节中；
- 在消费者一侧，有几种方法可用于获得指向对象的原始底层数据的指针（例如一个方法的形参）。

一些简单的对象例如 bytes 和 bytearray 会以面向字节的形式公开它们的底层缓冲区。也可能会用其他形式；例如 array.array 所公开的元素可以是多字节值。

缓冲区接口的消费者的一个例子是文件对象的 write() 方法：任何可以输出为一系列字节流的对象都可以被写入文件。然而 write() 只需要对传入对象内容的只读权限，其他的方法如 readinto() 需要对参数内容的写入权限。缓冲区接口使用对象可以选择性地允许或拒绝读写或只读缓冲区的导出。

对于缓冲区接口的使用者而言，有两种方式来获取一个目的对象的缓冲：

- 使用正确的参数来调用*PyObject_GetBuffer()* 函数；
- 调用*PyArg_ParseTuple()* (或其同级对象之一) 并传入 y*, w* or s* *格式代码* 中的一个。

在这两种情况下，当不再需要缓冲区时必须调用*PyBuffer_Release()* 。如果此操作失败，可能会导致各种问题，例如资源泄漏。

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切片机制。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 解释器公开的大多部数据类型不同，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**

🄵*稳定 ABI 的一部分* （包含所有成員）自 *3.11* 版本開始.

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
PyBUF_INDIRECT	是	是	如果需要的话
PyBUF_STRIDES	是	是	NULL
PyBUF_ND	是	NULL	NULL
PyBUF_SIMPLE	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
PyBUF_C_CONTIGUOUS	是	是	NULL	C
PyBUF_F_CONTIGUOUS	是	是	NULL	F
PyBUF_ANY_CONTIGUOUS	是	是	NULL	C 或 F
PyBUF_ND	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	只读	format
PyBUF_FULL	是	是	如果需要的话	U	0	是
PyBUF_FULL_RO	是	是	如果需要的话	U	1 或 0	是
PyBUF_RECORDS	是	是	NULL	U	0	是
PyBUF_RECORDS_RO	是	是	NULL	U	1 或 0	是
PyBUF_STRIDED	是	是	NULL	U	0	NULL
PyBUF_STRIDED_RO	是	是	NULL	U	1 或 0	NULL
PyBUF_CONTIG	是	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	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

    if ndim <= 0:
        return ndim == 0 and not shape and not strides
    if 0 in shape:
        return True

    imin = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] <= 0)
    imax = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] > 0)

    return 0 <= offset+imin and offset+imax+itemsize <= memlen
```

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];
        }
```

(繼續下一頁)

(繼續上一頁)

```
    }
    return (void*)pointer;
}
```

7.7.4 缓冲区相关函数

int **PyObject_CheckBuffer**(*PyObject* *obj)

🄵稳定 ABI 的一部分 自 *3.11* 版本開始. 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)

🄵稳定 ABI 的一部分 自 *3.11* 版本開始. 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)

🄵稳定 ABI 的一部分 自 *3.11* 版本開始. 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)

🄵稳定 ABI 的一部分 自 *3.11* 版本開始. 从*format* 返回隐含的*itemsize*。如果出错，则引发异常并返回 -1。

Added in version 3.9.

int **PyBuffer_IsContiguous**(const *Py_buffer* *view, char order)

🄵稳定 ABI 的一部分 自 *3.11* 版本開始. 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)

🄵稳定 ABI 的一部分 自 *3.11* 版本開始. 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)

🄵稳定 ABI 的一部分 自 *3.11* 版本開始. 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)

🄵稳定 ABI 的一部分 自 *3.11* 版本開始. 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)

🄵稳定 ABI 的一部分 自 *3.11* 版本開始. 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)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. 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)

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. 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 舊式緩沖協定 (Buffer Protocol)

在 3.0 版之後被🄵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🄵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)

🄵穩定 ABI 的一部分. 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🄵記憶體位置、將 *buffer_len* 設定🄵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)

🄵穩定 ABI 的一部分. 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`，🄵將 *buffer* 設定🄵記憶體位置、將 *buffer_len* 設定🄵緩衝區長度。回傳 `-1` 🄵在錯誤時設定 `TypeError`。

int **PyObject_CheckReadBuffer**(*PyObject* *o)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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`，🄵將 *buffer* 設定🄵記憶體位置，且將 *buffer_len* 設定🄵緩衝區長度。回傳 `-1` 🄵在錯誤時設定 `TypeError`。

CHAPTER 8

具體物件層

此章節列出的函式僅能接受某些特定的 Python 物件型🄵，將錯誤型🄵的物件傳遞給它們🄵不是什🄵好事，如果你從 Python 程式當中接收到一個不確定是否🄵正確型🄵的物件，那🄵請一定要先做型🄵檢查。例如使用*PyDict_Check()* 來確認一個物件是否🄵字典。本章結構類似於 Python 物件型🄵的" 族譜圖 (family tree)"。

警告： 雖然本章所述之函式仔細地檢查了傳入物件的型🄵，但大多🄵無檢查是否🄵 NULL。允許 NULL 的傳入可能造成記憶體的不合法存取和直譯器的立即中止。

8.1 基礎物件

此段落描述 Python 型🄵物件與單例 (singleton) 物件 None。

8.1.1 类型对象

type **PyTypeObject**

🄵受限 API 的一部分 （做🄵一個不透明結構 *(opaque struct)*）. 对象的 C 结构用于描述 built-in 类型。

PyTypeObject **PyType_Type**

🄵穩定 ABI 的一部分. 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**()

🄵穩定 ABI 的一部分. 清空内部查找缓存。返回当前版本标签。

unsigned long **PyType_GetFlags**(*PyTypeObject* *type)

🄵穩定 ABI 的一部分. 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。

Added in version 3.2.

在 3.4 版的變更: 返回类型现在是 unsigned long 而不是 long。

PyObject ***PyType_GetDict**(*PyTypeObject* *type)

返回类型对象的内部命名空间，它在其他情况下只能通过只读代理 (cls.__dict__) 公开。这可以代替直接访问*tp_dict* 的方式。返回的字典必须当作是只读的。

该函数用于特定的嵌入和语言绑定场景，在这些场景下需要直接访问该字典而间接访问（例如通过代理或*PyObject_GetAttr()* 访问）并不足够。

扩展模块在设置它们自己的类型时应当继续直接或间接地使用 tp_dict。

Added in version 3.12.

void **PyType_Modified**(*PyTypeObject* *type)

🄵穩定 ABI 的一部分. 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_AddWatcher**(*PyType_WatchCallback* callback)

注册 *callback* 作为类型监视器。返回一个非负的整数 ID，它必须传给将来对*PyType_Watch()* 的调用。如果出错（例如没有足够的可用监视器 ID），则返回 -1 并设置一个异常。

Added in version 3.12.

int **PyType_ClearWatcher**(int watcher_id)

清除由 *watcher_id* (之前从*PyType_AddWatcher()* 返回) 所标识的 watcher。成功时返回 0，出错时（例如 *watcher_id* 未被注册）返回 -1。

扩展在调用 PyType_ClearWatcher 时绝不能使用不是之前调用*PyType_AddWatcher()* 所返回的 *watcher_id*。

Added in version 3.12.

int **PyType_Watch**(int watcher_id, *PyObject* *type)

将 *type* 标记为已监视。每当*PyType_Modified()* 报告 *type* 发生变化时*PyType_AddWatcher()* 赋予 *watcher_id* 的回调将被调用。（如果在 *type* 的一系列连续修改之间没有调用 _PyType_Lookup()，则回调只能被调用一次；这是一个实现细节并可能发生变化）。

扩展在调用 PyType_Watch 时绝不能使用不是之前调用*PyType_AddWatcher()* 所返回的 *watcher_id*。

Added in version 3.12.

typedef int (***PyType_WatchCallback**)(*PyObject* *type)

类型监视器回调函数的类型。

回调不可以修改 *type* 或是导致*PyType_Modified()* 在 *type* 或其 MRO 中的任何类型上被调用；违反此规则可能导致无限递归。

Added in version 3.12.

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)

🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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* *kwds)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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🄵: 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 **必須**自己实现 GC 协议，至少要实现*tp_traverse* 句柄。

PyObject ***PyType_GetName**(*PyTypeObject* *type)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.11* 版本開始. 返回类型名称。等同于获取类型的 `__name__` 属性。

Added in version 3.11.

PyObject ***PyType_GetQualName**(*PyTypeObject* *type)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.11* 版本開始. 返回类型的限定名称。等同于获取类型的 `__qualname__` 属性。

Added in version 3.11.

void ***PyType_GetSlot**(*PyTypeObject* *type, int slot)

🄵穩定 ABI 的一部分 自 *3.4* 版本開始. 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。

Added in version 3.4.

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)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 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。

Added in version 3.9.

void *__PyType_GetModuleState__(*PyTypeObject* *type)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 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。

Added in version 3.9.

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。

Added in version 3.11.

int __PyUnstable_Type_AssignVersionTag__(*PyTypeObject* *type)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

尝试为给定的类型设置一个版本标签。

如果类型已有合法的版本标签或已设置了新的版本标签则返回 1，或者如果无法设置新的标签则返回 0。

Added in version 3.12.

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_FromMetaclass__(*PyTypeObject* *metaclass, *PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始. 根据 *spec* (参见`Py_TPFLAGS_HEAPTYPE`) 创建并返回一个堆类型。

元类 *metaclass* 用于构建结果类型对象。当 *metaclass* 为 `NULL` 时，元类将派生自 *bases* (或者如果 *bases* 为 `NULL` 则派生自 *Py_tp_base[s]* 槽位，见下文)。

不支持重写`tp_new` 的元类，除非 `tp_new` 为 `NULL`。(为了向下兼容，其他 `PyType_From*` 函数允许这样的元类。它们将忽略 `tp_new`，可能导致不完整的初始化。这样的元类已被弃用并在 Python 3.14+ 中停止支持。)

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()`。

请注意此函数 不能完全匹配调用 `type()` 或使用 `class` 语句的行为。对于用户提供的类型或元类，推荐调用 `type` (或元类) 而不是 `PyType_From*` 函数。特别地:

- `__new__()` 不会在新类上被调用 (它必须被设为 `type.__new__`)。
- `__init__()` 不会在新类上被调用。
- `__init_subclass__()` 不会在任何基类上调用。

- `__set_name__()` 不会在新的描述器上调用。

Added in version 3.12.

PyObject ***PyType_FromModuleAndSpec**(*PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 等價於 `PyType_FromMetaclass(NULL, module, spec, bases)`。

Added in version 3.9.

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。

在 3.12 版的變更: 该函数现在可以找到并使用与所提供的基类相对应的元类。在此之前，只会返回 `type` 实例。

元类的*tp_new* 将被 忽略。这可能导致不完整的初始化。创建元类重写*tp_new* 的类的做法已被弃用并且在 Python 3.14+ 中将不再被允许。

PyObject ***PyType_FromSpecWithBases**(*PyType_Spec* *spec, *PyObject* *bases)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.3* 版本開始. 等價於 `PyType_FromMetaclass(NULL, NULL, spec, bases)`。

Added in version 3.3.

在 3.12 版的變更: 该函数现在可以找到并使用与所提供的基类相对应的元类。在此之前，只会返回 `type` 实例。

元类的*tp_new* 将被 忽略。这可能导致不完整的初始化。创建元类重写*tp_new* 的类的做法已被弃用并且在 Python 3.14+ 中将不再被允许。

PyObject ***PyType_FromSpec**(*PyType_Spec* *spec)

回傳值：新的參照。🄵穩定 ABI 的一部分. 等價於 `PyType_FromMetaclass(NULL, NULL, spec, NULL)`。

在 3.12 版的變更: 该函数现在可以找到并使用与 *Py_tp_base[s]* 槽位中提供的基类相对应的元类。在此之前，只会返回 `type` 实例。

元类的*tp_new* 将被 忽略。这可能导致不完整的初始化。创建元类重写*tp_new* 的类的做法已被弃用并且在 Python 3.14+ 中将不再被允许。

type **PyType_Spec**

🄵穩定 ABI 的一部分 （包含所有成員）. 定义一个类型的行为的结构体。

const char ***name**

类型的名称，用来设置*PyTypeObject.tp_name*。

int **basicsize**

如果为正数，则以字节为单位指定实例的大小。它用于设置*PyTypeObject.tp_basicsize*。

如果为零，则指定应当继承*tp_basicsize*。

如果为负数，则以其绝对值指定该类的实例在超类的基础之上还需要多少空间。使用*PyObject_GetTypeData()* 来获取通过此方式保留的子类专属内存的指针。

在 3.12 版的變更: 在之前版本中，此字段不能为负数。

int **itemsize**

可变大小类型中一个元素的大小，以字节为单位。用于设置*PyTypeObject.tp_itemsize*。注意事项请参阅 `tp_itemsize` 文档。

如果为零，则会继承*tp_itemsize*。扩展任意可变大小的类是很危险的，因为某些类型使用固定偏移量来标识可变大小的内存，这样就会与子类使用的固定大小的内存相重叠。为了防止出错，只有在以下情况下才可以继承 `itemsize`:

- 基类不是可变大小的 (即其*tp_itemsize*)。
- 所请求的*PyType_Spec.basicsize* 为正值，表明基类的内存布局是已知的。

- 所请求的*PyType_Spec.basicsize* 为零，表明子类不会直接访问实例的内存。
- 具有*Py_TPFLAGS_ITEMS_AT_END* 旗标。

unsigned int **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 ***slots**

PyType_Slot 结构体的数组。以特殊槽位值 {0, NULL} 来结束。

每个槽位 ID 应当只被指定一次。

type **PyType_Slot**

🄵稳定 ABI 的一部分 （包含所有成員）. 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 **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“offset”字段不可使用*PyType_Slot* 来设置:

- *tp_weaklistoffset*(如果可能请改用*Py_TPFLAGS_MANAGED_WEAKREF*)
- *tp_dictoffset*（如果可能，請改用*Py_TPFLAGS_MANAGED_DICT*）
- *tp_vectorcall_offset*（請用*PyMemberDef* 中的"__vectorcalloffset__"）

如果无法转为 MANAGED 旗标 (例如，对于 vectorcall 或是为了支持早于 Python 3.12 的版本)，请在*Py_tp_members* 中指定 offset。详情参见*PyMemberDef documentation*。

以下字段在创建堆类型时完全不可设置:

- *tp_vectorcall* (请使用*tp_new* 和/或*tp_init*)
- 内部字段: *tp_dict*, *tp_mro*, *tp_cache*, *tp_subclasses* 和*tp_weaklist*。

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 ***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，表示空值。该对象没有任何方法并且是 永久性对象。

在 3.12 版的變更: *Py_None* 🄵不滅的 (immortal)。

Py_RETURN_NONE

从一个函数返回*Py_None*。

8.2 數值物件

8.2.1 整數物件

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**

🄵受限 API 的一部分 （做🄵一個不透明結構 *(opaque struct)*）. 表示 Python 整数对象的*PyObject* 子类型。

PyTypeObject **PyLong_Type**

🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 由 *v* 的整数部分返回一个新的*PyLongObject* 对象，失败则返回 `NULL` 。

PyObject ***PyLong_FromString**(const char *str, char **pend, int base)

回傳值：新的參照。🄵穩定 ABI 的一部分. 根据 *str* 字符串值返回一个新的*PyLongObject*，它将根据 *base* 指定的基数来解读，或是在失败时返回 `NULL`。如果 *pend* 不为 `NULL`，则在成功时 **pend* 将指向 *str* 中末尾而在出错时将指向第一个无法处理的字符。如果 *base* 为 `0`，则 *str* 将使用 integers 定义来解读；在此情况下，非零十进制数以零开头将会引发 `ValueError`。如果 *base* 不为 `0`，则必须在 `2` 和 `36`，包括这两个值。开头和末尾的空格以及基数标示符之后和数码之间的单下划线将被忽略。如果没有数码或 *str* 中数码和末尾空格之后不以 NULL 结束，则将引发 `ValueError`。

也參考:

Python 方法 `int.to_bytes()` 和 `int.from_bytes()` 用于*PyLongObject* 到/从字节数组之间以 `256` 为基数进行转换。你可以使用*PyObject_CallMethod()* 从 C 调用它们。

PyObject ***PyLong_FromUnicodeObject**(*PyObject* *u, int base)

回傳值：新的參照。 将字符串 *u* 中的 Unicode 数字序列转换为 Python 整数值。

Added in version 3.3.

PyObject ***PyLong_FromVoidPtr**(void *p)

回傳值：新的參照。🄵穩定 ABI 的一部分. 从指针 *p* 创建一个 Python 整数。可以使用*PyLong_AsVoidPtr()* 返回的指针值。

long **PyLong_AsLong**(*PyObject* *obj)

🄵穩定 ABI 的一部分. 返回 *obj* 的 C `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 `long` 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。

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__()`。

long **PyLong_AsLongAndOverflow**(*PyObject* *obj, int *overflow)

🄵穩定 ABI 的一部分. 返回 *obj* 的 C `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* 的值大于 `LONG_MAX` 或小于 `LONG_MIN`，则会把 **overflow* 分别置为 `1` 或 `-1`，并返回 `-1`；否则，将 **overflow* 置为 `0`。如果发生其他异常则按常规把 **overflow* 置为 `0` 并返回 `-1`。

出错则返回 `-1` 。请用*PyErr_Occurred()* 找出具体问题。

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__()`。

long long **PyLong_AsLongLong**(*PyObject* *obj)

🄵穩定 ABI 的一部分. 返回 *obj* 的 C `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* 值超出 `long long` 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。

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__()`。

long long **PyLong_AsLongLongAndOverflow**(*PyObject* *obj, int *overflow)

🄵穩定 ABI 的一部分. 返回 *obj* 的 C `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* 的值大于 `LLONG_MAX` 或小于 `LLONG_MIN`，则会把 **overflow* 分别置为 `1` 或 `-1`，并返回 `-1`；否则，将 **overflow* 置为 `0`。如果发生其他异常则按常规把 **overflow* 置为 `0` 并返回 `-1`。

出错则返回 `-1` 。请用*PyErr_Occurred()* 找出具体问题。

Added in version 3.2.

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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。

int **PyUnstable_Long_IsCompact**(const *PyLongObject* *op)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

如果 *op* 为紧凑形式则返回 1，否则返回 0。

此函数使得显著影响性能的关键代码可以实现小整数的“快速路径”。对于紧凑形式的值使用*PyUnstable_Long_CompactValue()*；对于其他值则回退为*PyLong_As** 函数或者调用 `int.to_bytes()`。

此项加速对于大多数用户来说是可以忽略的。

具体有哪些值会被视为紧凑形式属于实现细节并可能发生改变。

Py_ssize_t **PyUnstable_Long_CompactValue**(const *PyLongObject* *op)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

如果 *op* 为紧凑形式，如*PyUnstable_Long_IsCompact()* 所确定的，则返回它的值。

在其他情况下，返回值是未定义的。

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。

PyTypeObject **PyBool_Type**

🄵穩定 ABI 的一部分. 这个*PyTypeObject* 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🄵有任何方法且🄵不滅的 (immortal)。

在 3.12 版的變更: *Py_False* 🄵不滅的。

PyObject ***Py_True**

Python 的 `True` 物件。此物件🄵有任何方法且🄵不滅的 (immortal)。

在 3.12 版的變更: *Py_True* 🄵不滅的。

Py_RETURN_FALSE

從函式回傳*Py_False*。

Py_RETURN_TRUE

從函式回傳*Py_True*。

PyObject ***PyBool_FromLong**(long v)

回傳值：新的參照。🄵穩定 ABI 的一部分. 根據 *v* 的實際值來回傳*Py_True* 或者*Py_False*。

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。

PyTypeObject **PyFloat_Type**

🄵穩定 ABI 的一部分. 这是个属于 C 类型*PyTypeObject* 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 根据 *v* 创建一个*PyFloatObject* 对象，失败时返回 `NULL`。

double **PyFloat_AsDouble**(*PyObject* *pyfloat)

🄵穩定 ABI 的一部分. 返回 *pyfloat* 的内容的 C `double` 表示形式。如果 *pyfloat* 不是一个 Python 浮点数对象但是具有 `__float__()` 方法，则会先调用此方法来将 *pyfloat* 转换为浮点数。如果 `__float__()` 未定义则将回退至 `__index__()`。此方法在失败时将返回 `-1.0`，因此开发者应当调用*PyErr_Occurred()* 来检测错误。

在 3.8 版的變更: 如果可能将使用 `__index__()`。

double **PyFloat_AS_DOUBLE**(*PyObject* *pyfloat)

返回 *pyfloat* 的 C `double` 表示形式，但不带错误检测。

PyObject ***PyFloat_GetInfo**(void)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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**()

🄵穩定 ABI 的一部分. 返回 C `double` 形式的最大可表示有限浮点数 *DBL_MAX*。

double **PyFloat_GetMin**()

🄵穩定 ABI 的一部分. 返回 C `double` 形式的最小正规化正浮点数 *DBL_MIN*。

打包与解包函数

打包与解包函数提供了独立于平台的高效方式来将浮点数值存储为字节串。Pack 例程根据 C `double` 产生字节串，而 Unpack 例程根据这样的字节串产生 C `double`。后缀 (2, 4 or 8) 指明字节串中的字节数。

在明显使用 IEEE 754 格式的平台上这些函数是通过拷贝比特位来实现的。在其他平台上，2 字节格式与 IEEE 754 binary16 半精度格式相同，4 字节格式 (32 位) 与 IEEE 754 binary32 单精度格式相同，而 8 字节格式则与 IEEE 754 双精度格式相同，不过 INF 和 NaN (如果平台存在这两种值) 未得到正确处理，而试图对包含 IEEE INF 或 NaN 的字节串执行解包将会引发一个异常。

在具有比 IEEE 754 所支持的更高精度，或更大动态范围的非 IEEE 平台上，不是所有的值都能被打包；在具有更低精度，或更小动态范围的非 IEEE 平台上，则不是所有的值都能被解包。在这种情况下发生的事情有一部分将是偶然的（无奈）。

Added in version 3.11.

打包函数

打包例程会写入 2, 4 或 8 个字节，从 *p* 开始。*le* 是一个 `int` 参数，如果你想要字节串为小端序格式 (指数部分放在后面，位于 `p+1`, `p+3` 或 `p+6` `p+7`) 则其应为非零值，如果你想要大端序格式 (指数部分放在前面，位于 *p*) 则其应为零。`PY_BIG_ENDIAN` 常量可被用于使用本机端序：在大端序处理器上等于 `1`，在小端序处理器上则等于 `0`。

返回值: 如果一切正常则为 `0`，如果出错则为 `-1` (并会设置一个异常，最大可能为 `OverflowError`)。

在非 IEEE 平台上存在两个问题:

- 如果 *x* 为 NaN 或无穷大则此函数的行为是未定义的。
- `-0.0` 和 `+0.0` 将产生相同的字节串。

int **PyFloat_Pack2**(double x, unsigned char *p, int le)

将 C double 打包为 IEEE 754 binary16 半精度格式。

int **PyFloat_Pack4**(double x, unsigned char *p, int le)

将 C double 打包为 IEEE 754 binary32 单精度格式。

int **PyFloat_Pack8**(double x, unsigned char *p, int le)

将 C double 打包为 IEEE 754 binary64 双精度格式。

解包函数

解包例程会读取 2, 4 或 8 个字节，从 *p* 开始。*le* 是一个 `int` 参数，如果字节串为小端序格式 (指数部门放在后面，位于 `p+1`, `p+3` 或 `p+6` 和 `p+7`) 则其应为非零值，如果为大端序格式 (指数部分放在前面，位于 *p*) 则其应为零。`PY_BIG_ENDIAN` 常量可被用于使用本机端序：在大端序处理器上等于 `1`，在小端序处理器上则等于 `0`。

返回值: 解包后的 double。出错时，返回值为 `-1.0` 且*PyErr_Occurred()* 为真值 (并且会设置一个异常，最大可能为 `OverflowError`)。

请注意在非 IEEE 平台上此函数将拒绝解包表示 NaN 或无穷大的字节串。

double **PyFloat_Unpack2**(const unsigned char *p, int le)

将 IEEE 754 binary16 半精度格式解包为 C double。

double **PyFloat_Unpack4**(const unsigned char *p, int le)

将 IEEE 754 binary32 单精度格式解包为 C double。

double **PyFloat_Unpack8**(const unsigned char *p, int le)

将 IEEE 754 binary64 双精度格式解包为 C double。

8.2.4 🄵數物件

從 C API 來看，Python 的🄵數物件被實作🄵兩種不同的型🄵：一種是公開給 Python 程式的 Python 物件，另一種是表示實際🄵數值的 C 結構。API 提供了與兩者一起作用的函式。

作🄵 C 結構的🄵數

請注意，接受這些結構作🄵參數🄵將它們作🄵結果回傳的函式是按值 *(by value)* 執行的，而不是透過指標取消參照 (dereference) 它們。這在整個 API 中都是一致的。

type **Py_complex**

相對於 Python 🄵數物件之數值部分的 C 結構。大多數處理🄵數物件的函式根據需求會使用這種型🄵的結構作🄵輸入或輸出值。它定義🄵：

```
typedef struct {
   double real;
   double imag;
} Py_complex;
```

Py_complex **_Py_c_sum**(*Py_complex* left, *Py_complex* right)

以 C 的*Py_complex* 表示形式來回傳兩個🄵數之和。

Py_complex **_Py_c_diff**(*Py_complex* left, *Py_complex* right)

以 C 的*Py_complex* 表示形式來回傳兩個🄵數間的差。

Py_complex **_Py_c_neg**(*Py_complex* num)

以 C 的*Py_complex* 表示形式來回傳🄵數 *num* 的相反數 (negation)。

Py_complex **_Py_c_prod**(*Py_complex* left, *Py_complex* right)

以 C 的*Py_complex* 表示形式來回傳兩個🄵數的乘積。

Py_complex **_Py_c_quot**(*Py_complex* dividend, *Py_complex* divisor)

以 C 的*Py_complex* 表示形式來回傳兩個🄵數的商。

如果 *divisor* 🄵 null，則此方法會回傳零🄵將 `errno` 設定🄵 `EDOM`。

Py_complex **_Py_c_pow**(*Py_complex* num, *Py_complex* exp)

以 C 的*Py_complex* 表示形式來回傳 *num* 的 *exp* 次方的結果。

如果 *num* 🄵 null 且 *exp* 不是正實數，則此方法會回傳零🄵將 `errno` 設定🄵 `EDOM`。

作🄵 Python 物件的🄵數

type **PyComplexObject**

這個*PyObject* 的子型🄵代表一個 Python 🄵數物件。

PyTypeObject **PyComplex_Type**

🄵穩定 ABI 的一部分. 這個*PyTypeObject* 的實例代表 Python 🄵數型🄵。它與 Python 層中的 `complex` 是同一個物件。

int **PyComplex_Check**(*PyObject* *p)

如果其引數是一個*PyComplexObject* 或者是*PyComplexObject* 的子型🄵，則會回傳 true。這個函式不會失敗。

int **PyComplex_CheckExact**(*PyObject* *p)

如果其引數是一個*PyComplexObject*，但不是*PyComplexObject* 的子型🄵，則會回傳 true。這個函式不會失敗。

PyObject ***PyComplex_FromCComplex**(*Py_complex* v)

回傳值：新的參照。 從 C 的*Py_complex* 值建立一個新的 Python 🄵數物件。

PyObject ***PyComplex_FromDoubles**(double real, double imag)

回傳值：新的參照。🄵穩定 ABI 的一部分. 從 *real* 和 *imag* 回傳一個新的*PyComplexObject* 物件。

double **PyComplex_RealAsDouble**(*PyObject* *op)

🄵穩定 ABI 的一部分. 以 C 的 double 形式回傳 *op* 的實部。

double **PyComplex_ImagAsDouble**(*PyObject* *op)

🄵穩定 ABI 的一部分. 將 *op* 的🄵部作🄵 C 的 double 回傳。

Py_complex **PyComplex_AsCComplex**(*PyObject* *op)

回傳🄵數 *op* 的*Py_complex* 值。

如果 *op* 不是 Python 🄵數物件，但有一個 __complex__() 方法，則首先會呼叫該方法將 *op* 轉🄵🄵 Python 🄵數物件。如果 __complex__() 🄵未定義，那🄵它會回退到 __float__()。如果 __float__() 未定義，則它將繼續回退🄵 __index__()。失敗時，此方法回傳 -1.0 作🄵實部值。

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**

🄵穩定 ABI 的一部分. *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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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
%%	*n/a*	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").[1]
%u	unsigned int	等價於 printf("%u").Page 123, 1
%ld	long	等價於 printf("%ld").[1]
%lu	unsigned long	等價於 printf("%lu").[1]
%zd	*Py_ssize_t*	等價於 printf("%zd").[1]
%zu	size_t	等價於 printf("%zu").[1]
%i	int	等價於 printf("%i").[1]
%x	int	等價於 printf("%x").[1]
%s	const char*	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 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 返回字节表示实现缓冲区协议的对象 *o*。

Py_ssize_t **PyBytes_Size**(*PyObject* *o)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。

int **PyBytes_AsStringAndSize**(*PyObject* *obj, char **buffer, *Py_ssize_t* *length)

🄵穩定 ABI 的一部分. 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。

[1] 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。

void **PyBytes_Concat**(*PyObject* **bytes, *PyObject* *newpart)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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一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 位元組陣列物件 (Byte Array Objects)

type **PyByteArrayObject**

這個*`PyObject`* 的子型🄵代表了 Python 的位元組陣列物件。

PyTypeObject **PyByteArray_Type**

🄵穩定 ABI 的一部分. 這個*`PyTypeObject`* 的實例代表了 Python 的位元組陣列型🄵；在 Python 層中的 `bytearray` 🄵同一個物件。

型🄵檢查巨集

int **PyByteArray_Check**(*PyObject* *o)

如果物件 *o* 是一個位元組陣列物件，或者是位元組陣列型🄵的子型🄵的實例，則回傳真值。此函式總是會成功執行。

int **PyByteArray_CheckExact**(*PyObject* *o)

如果物件 *o* 是一個位元組陣列物件，但不是位元組陣列型🄵的子型🄵的實例，則回傳真值。此函式總是會成功執行。

直接 API 函式

PyObject ***PyByteArray_FromObject**(*PyObject* *o)

回傳值：新的參照。🄵穩定 ABI 的一部分. 由任意物件 *o* 回傳一個新的位元組陣列物件，🄵實作了緩衝協議 *(buffer protocol)*。

PyObject ***PyByteArray_FromStringAndSize**(const char *string, *Py_ssize_t* len)

回傳值：新的參照。🄵穩定 ABI 的一部分. 從 *string* 及其長度 *len* 建立一個新的位元組陣列物件。若失敗則回傳 `NULL`。

PyObject ***PyByteArray_Concat**(*PyObject* *a, *PyObject* *b)

回傳值：新的參照。🄵穩定 ABI 的一部分. 連接位元組陣列 *a* 和 *b*，🄵回傳一個包含結果的新位元組陣列。

Py_ssize_t **PyByteArray_Size**(*PyObject* *bytearray)

🄵穩定 ABI 的一部分. 在檢查🄵 `NULL` 指標後，回傳 *bytearray* 的大小。

char ***PyByteArray_AsString**(*PyObject* *bytearray)

🄵穩定 ABI 的一部分. 在檢查是否🄵 `NULL` 指標後，將 *bytearray* 的🄵容回傳🄵字元陣列。回傳的陣列總是會多附加一個空位元組。

int **PyByteArray_Resize**(*PyObject* *bytearray, *Py_ssize_t* len)
🄵穩定 ABI 的一部分. 將 *bytearray* 的🄴部緩衝區大小調整🄴 *len*。

巨集

這些巨集犧牲了安全性以🄴取速度，🄵且它們不會檢查指標。

char ***PyByteArray_AS_STRING***(*PyObject* *bytearray)
與`PyByteArray_AsString()` 類似，但🄵有錯誤檢查。

Py_ssize_t **PyByteArray_GET_SIZE**(*PyObject* *bytearray)
與`PyByteArray_Size()` 類似，但🄵有錯誤檢查。

8.3.3 Unicode 物件與編解碼器

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）。

UTF-8 表示将按需创建并缓存在 Unicode 对象中。

備🄵: `Py_UNICODE` 表示形式在 Python 3.12 中同被弃用的 API 一起被移除了，查阅 **PEP 623** 以获得更多信息。

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**
🄵穩定 ABI 的一部分. 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`。

Added in version 3.3.

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。

Added in version 3.3.

PyTypeObject **PyUnicode_Type**
🄵穩定 ABI 的一部分. 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)

返回 `0`。此 API 仅为向下兼容而保留。

Added in version 3.3.

在 3.10 版之後被🄵用: 此 API 从 Python 3.12 起将不做任何事。

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）。

Added in version 3.3.

Py_UCS1 ***PyUnicode_1BYTE_DATA**(*PyObject* *unicode)

Py_UCS2 ***PyUnicode_2BYTE_DATA**(*PyObject* *unicode)

Py_UCS4 ***PyUnicode_4BYTE_DATA**(*PyObject* *unicode)

返回一个用于直接字符访问的指向转换为 UCS1、UCS2 或 UCS4 整数类型的规范表示的指针。如果规范表示具有正确的字符大小，则不执行检查；使用*PyUnicode_KIND()* 选择正确的函数。

Added in version 3.3.

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND

PyUnicode_4BYTE_KIND

返回*PyUnicode_KIND()* 宏的值。

Added in version 3.3.

在 3.12 版的變更: `PyUnicode_WCHAR_KIND` 已被移除。

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）。

Added in version 3.3.

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）。

Added in version 3.3.

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。

Added in version 3.3.

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。

Added in version 3.3.

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()*。

Added in version 3.3.

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。

Added in version 3.3.

int **PyUnicode_IsIdentifier**(*PyObject* *unicode)

🄵穩定 ABI 的一部分. 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* 是否为十进制数字符返回 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)
将字符 *ch* 转换为十进制正整数返回。如果无法转换则返回 `-1`。此函数不会引发异常。

int **Py_UNICODE_TODIGIT**(*Py_UCS4* ch)
将字符 *ch* 转换为单个数码位的整数返回。如果无法转换则返回 `-1`。此函数不会引发异常。

double **Py_UNICODE_TONUMERIC**(*Py_UCS4* ch)
将字符 *ch* 转换为双精度浮点数返回。如果无法转换则返回 `-1.0`。此函数不会引发异常。

这些 API 可被用来操作代理项:

int **Py_UNICODE_IS_SURROGATE**(*Py_UCS4* ch)
检测 *ch* 是否为代理项 (`0xD800 <= ch <= 0xDFFF`)。

int **Py_UNICODE_IS_HIGH_SURROGATE**(*Py_UCS4* ch)
检测 *ch* 是否为高代理项 (`0xD800 <= ch <= 0xDBFF`)。

int **Py_UNICODE_IS_LOW_SURROGATE**(*Py_UCS4* ch)
检测 *ch* 是否为低代理项 (`0xDC00 <= ch <= 0xDFFF`)。

Py_UCS4 **Py_UNICODE_JOIN_SURROGATES**(*Py_UCS4* high, *Py_UCS4* low)
合并两个代理项字符并返回单个*Py_UCS4* 值。*high* 和 *low* 分别为一个代理项对的开头和末尾代理项。*high* 取值范围必须为 [0xD800; 0xDBFF] 而 *low* 取值范围必须为 [0xDC00; 0xDFFF]。

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。

Added in version 3.3.

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*)。

Added in version 3.3.

PyObject ***PyUnicode_FromStringAndSize**(const char *str, *Py_ssize_t* size)

回傳值：新的參照。🄵穩定 ABI 的一部分. 根据字符缓冲区 *str* 创建一个 Unicode 对象。字节数据将按 UTF-8 编码格式来解读。缓冲区会被拷贝到新的对象中。返回值可以是一个共享对象，即其数据不允许修改。

此函数会因以下情况而引发 `SystemError`:

- *size* < 0,
- *str* 为 `NULL` 且 *size* > 0

在 3.12 版的變更: *str* == `NULL` 且 *size* > 0 不再被允许。

PyObject ***PyUnicode_FromString**(const char *str)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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 Unicode 字符串的大小并返回包含已格式化值的字符串。可变数量的参数必须均为 C 类型并且必须恰好与 *format* ASCII 编码字符串中的格式字符相对应。

转换标记符包含两个或更多字符并具有以下组成，且必须遵循此处规定的顺序：

1. `'%'` 字符，用于标记转换符的起始。
2. 转换旗标（可选），用于影响某些转换类型的结果。
3. 最小字段宽度（可选）。如果指定为 `'*'` (星号)，则实际宽度会在下一参数中给出，该参数必须为 `int` 类型，要转换的对象则放在最小字段宽度和可选精度之后。
4. 精度（可选），以在 `'.'` (点号) 之后加精度值的形式给出。如果指定为 `'*'` (星号)，则实际精度会在下一参数中给出，该参数必须为 `int` 类型，要转换的对象则放在精度之后。
5. 长度修饰符（可选）。
6. 转换类型。

转换旗标为：

标志	含意
`0`	转换将为数字值填充零字符。
`-`	转换值将靠左对齐（如果同时给出则会覆盖 `0` 旗标）。

以下整数转换的长度修饰符 (`d`, `i`, `o`, `u`, `x`, or `X`) 指明参数的类型 (默认为 `int`):

修饰符	类型
`l`	`long` 或 `unsigned long`
`ll`	`long long` 或 `unsigned long long`
`j`	`intmax_t` 或 `uintmax_t`
`z`	`size_t` 或 `ssize_t`
`t`	`ptrdiff_t`

针对以下转换 `s` 或 `V` 的长度修饰符 `l` 指明参数的类型为 `const wchar_t*`。

转换指示符如下：

转换指示符	类型	注释
`%`	*n/a*	字面的 `%` 字符。
`d`, `i`	由长度修饰符指明	有符号 C 整数的十进制表示。
`u`	由长度修饰符指明	无符号 C 整数的十进制表示。
`o`	由长度修饰符指明	无符号 C 整数的八进制表示。
`x`	由长度修饰符指明	无符号 C 整数的十六进制表示（小写）。
`X`	由长度修饰符指明	无符号 C 整数的十六进制表示（大写）。
`c`	`int`	单个字符。
`s`	`const char*` 或 `const wchar_t*`	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 `printf("%p")` 但它会确保以字面值 `0x` 开头而不管系统平台上的 `printf` 输出是什么。
`A`	*PyObject*`*`	`ascii()` 调用的结果。
`U`	*PyObject*`*`	一 Unicode 物件。
`V`	*PyObject*`*`、`const char*` 或 `const wchar_t*`	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*PyObject_Str()* 的结果。
`R`	*PyObject*`*`	调用*PyObject_Repr()* 的结果。

備🄵: 格式符的宽度单位是字符数而不是字节数。格式符的精度单位对于 `"%s"` 和 `"%V"` (如果 `PyObject*` 参数为 `NULL`) 是字节数或 `wchar_t` 项数 (如果使用了长度修饰符 `l`)，而对于 `"%A"`, `"%U"`, `"%S"`, `"%R"` 和 `"%V"` (如果 `PyObject*` 参数不为 `NULL`) 则为字符数。

備🄵: 与 C `printf()` 不同的是 `0` 旗标即使在为整数转换 (`d`, `i`, `u`, `o`, `x`, or `X`) 指定精度时也是有效的。

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。

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。

在 3.12 版的變更: 支持转换说明符 `o` 和 `X`。支持长度修饰符 `j` 和 `t`。长度修饰符现在将应用于所有整数转换。长度修饰符 `l` 现在将应用于转换说明符 `s` 和 `V`。支持可变宽度和精度 `*`。支持旗标 `-`。

不可识别的格式字符现在会设置一个 `SystemError`。在之前版本中它会导致所有剩余格式字符串被原样拷贝到结果字符串，并丢弃任何额外的参数。

PyObject ***PyUnicode_FromFormatV**(const char *format, va_list vargs)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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（请参阅内置编解码器 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)

🄵稳定 ABI 的一部分 自 *3.7* 版本開始. 返回 Unicode 对象码位的长度。

Added in version 3.3.

Py_ssize_t **PyUnicode_CopyCharacters**(*PyObject* *to, *Py_ssize_t* to_start, *PyObject* *from, *Py_ssize_t* from_start, *Py_ssize_t* how_many)

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。

Added in version 3.3.

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。

Added in version 3.3.

int **PyUnicode_WriteChar**(*PyObject* *unicode, *Py_ssize_t* index, *Py_UCS4* character)

🄵稳定 ABI 的一部分 自 *3.7* 版本開始. 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）。

Added in version 3.3.

Py_UCS4 **PyUnicode_ReadChar**(*PyObject* *unicode, *Py_ssize_t* index)

🄵稳定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.3.

PyObject ***PyUnicode_Substring**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* end)

回傳值：新的參照。🄵稳定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.3.

Py_UCS4 ***PyUnicode_AsUCS4**(*PyObject* *unicode, *Py_UCS4* *buffer, *Py_ssize_t* buflen, int copy_null)

🄵稳定 ABI 的一部分 自 *3.7* 版本開始. 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*。

Added in version 3.3.

Py_UCS4 ***PyUnicode_AsUCS4Copy**(*PyObject* *unicode)

🄵稳定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.3.

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。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。

使用*PyUnicode_DecodeFSDefaultAndSize()* 以*filesystem encoding and error handler* 来解码字符串。

此函数将忽略 Python UTF-8 模式。

也參考:

Py_DecodeLocale() 函式。

Added in version 3.3.

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。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 类似于*PyUnicode_DecodeLocaleAndSize()*，但会使用 strlen() 来计算字符串长度。

Added in version 3.3.

PyObject ***PyUnicode_EncodeLocale**(*PyObject* *unicode, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。

使用*PyUnicode_EncodeFSDefault()* 将字符串编码为*filesystem encoding and error handler*。

此函数将忽略 Python UTF-8 模式。

也參考:

Py_EncodeLocale() 函式。

Added in version 3.3.

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

使用*filesystem encoding and error handler* 的编码和解码函数 (**PEP 383** 和 **PEP 529**)。

要在参数解析期间将文件名编码为 bytes，应当使用 "O&" 转换器，传入*PyUnicode_FSConverter()* 作为转换函数：

int **PyUnicode_FSConverter**(*PyObject* *obj, void *result)

🄵穩定 ABI 的一部分. 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。

Added in version 3.1.

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)

🄵稳定 ABI 的一部分. 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。

Added in version 3.2.

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。🄵稳定 ABI 的一部分. 使用*filesystem encoding and error handler* 解码字符串。

如果你需要以当前语言区域编码格式解码字符串，请使用*PyUnicode_DecodeLocaleAndSize()*。

也參考:

Py_DecodeLocale() 函式。

在 3.6 版的變更: 现在将使用文件系统编码格式和错误处理器。

PyObject ***PyUnicode_DecodeFSDefault**(const char *str)

回傳值：新的參照。🄵稳定 ABI 的一部分. 使用*filesystem encoding and error handler* 解码以空值结尾的字符串。

如果字符串长度已知，则使用*PyUnicode_DecodeFSDefaultAndSize()*。

在 3.6 版的變更: 现在将使用文件系统编码格式和错误处理器。

PyObject ***PyUnicode_EncodeFSDefault**(*PyObject* *unicode)

回傳值：新的參照。🄵稳定 ABI 的一部分. 使用*filesystem encoding and error handler* 编码一个 Unicode 对象，并返回 `bytes`。请注意结果 `bytes` 对象可以包含空字节。

如果你需要以当前语言区域编码格式编码字符串，请使用*PyUnicode_EncodeLocale()*。

也參考:

Py_EncodeLocale() 函式。

Added in version 3.2.

在 3.6 版的變更: 现在将使用文件系统编码格式和错误处理器。

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。🄵稳定 ABI 的一部分. 根据给定的 *size* 的 `wchar_t` 缓冲区 *wstr* 创建一个 Unicode 对象。传入 `-1` 作为 *size* 表示该函数必须使用 `wcslen()` 自动计算缓冲区长度。失败时将返回 `NULL`。

Py_ssize_t **PyUnicode_AsWideChar**(*PyObject* *unicode, wchar_t *wstr, *Py_ssize_t* size)

🄵稳定 ABI 的一部分. 将 Unicode 对象的内容拷贝到 `wchar_t` 缓冲区 *wstr* 中。至多拷贝 *size* 个 `wchar_t` 字符（不包括可能存在的末尾空结束字符）。返回拷贝的 `wchar_t` 字符数或在出错时返回 `-1`。

当 *wstr* 为 `NULL` 时，则改为返回存储包括结束空值在内的所有 *unicode* 内容所需的 *size*。

请注意结果 `wchar_t*` 字符串可能是以空值结束也可能不是。调用方要负责确保 `wchar_t*` 字符串以空值结束以防应用有此要求。此外，请注意 `wchar_t*` 字符串有可能包含空字符，这将导致字符串在与大多数 C 函数一起使用时被截断。

wchar_t *__PyUnicode_AsWideCharString__(*PyObject* *unicode, *Py_ssize_t* *size)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 将 Unicode 对象转换为宽字符串。输出字符串将总是以空字符结尾。如果 *size* 不为 NULL，则会将宽字符的数量（不包括结尾空字符）写入到 **size* 中。请注意结果 wchar_t 字符串可能包含空字符，这将导致在大多数 C 函数中使用时字符串被截断。如果 *size* 为 NULL 并且 wchar_t* 字符串包含空字符则将引发 ValueError。

成功时返回由*PyMem_New* 分配的缓冲区（使用*PyMem_Free()* 来释放它）。发生错误时，则返回 NULL 并且 **size* 将是未定义的。如果内存分配失败则会引发 MemoryError。

Added in version 3.2.

在 3.7 版的變更: 如果 *size* 为 NULL 且 wchar_t* 字符串包含空字符则会引发 ValueError。

内置编解码器

Python 提供了一组以 C 编写以保证运行速度的内置编解码器。所有这些编解码器均可通过下列函数直接使用。

下列 API 大都接受 encoding 和 errors 两个参数，它们具有与在内置 str() 字符串对象构造器中同名参数相同的语义。

将 encoding 设为 NULL 将使用默认编码格式即 UTF-8。文件系统调用应当使用*PyUnicode_FSConverter()* 来编码文件名。这将在内部使用*filesystem encoding and error handler*。

错误处理方式由 errors 设置并且也可以设为 NULL 表示使用为编解码器定义的默认处理方式。所有内置编解码器的默认错误处理方式是"strict" (会引发 ValueError)。

编解码器都使用类似的接口。为了保持简单只有与下列泛型编解码器的差异才会记录在文档中。

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: 新的參照。🄵穩定 ABI 的一部分. 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: 新的參照。🄵穩定 ABI 的一部分. 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 編解碼器

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: 新的參照。🄵穩定 ABI 的一部分. 通过解码 UTF-8 编码的字节串 *str* 的的 *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: 新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 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。

Added in version 3.3.

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。

Added in version 3.3.

在 3.7 版的變更: 返回类型现在是 `const char *` 而不是 `char *`。

UTF-32 編解碼器

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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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 編解碼器

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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 如果 *consumed* 为 `NULL`，则行为类似于*PyUnicode_DecodeUTF16()*。如果 *consumed* 不为 `NULL`，则*PyUnicode_DecodeUTF16Stateful()* 将不把末尾的不完整 UTF-16 字节序列（如为奇数个字节或为分开的替代对）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_AsUTF16String**(*PyObject* *unicode)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回使用 UTF-16 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-7 編解碼器

以下是 UTF-7 编解码器 API:

PyObject ***PyUnicode_DecodeUTF7**(const char *str, *Py_ssize_t* size, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分. 通过解码 UTF-7 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_DecodeUTF7Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

回傳值：新的參照。🄵穩定 ABI 的一部分. 如果 *consumed* 为 `NULL`，则行为类似于*PyUnicode_DecodeUTF7()*。如果 *consumed* 不为 `NULL`，则末尾的不完整 UTF-7 base-64 部分将不被视为错误。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。

Unicode-Escape 编解码器

以下是"Unicode Escape" 编解码器的 API:

PyObject ***PyUnicode_DecodeUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分. 通过解码 Unicode-Escape 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_AsUnicodeEscapeString**(*PyObject* *unicode)

回傳值：新的參照。🄵穩定 ABI 的一部分. 使用 Unicode-Escape 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。

Raw-Unicode-Escape 编解码器

以下是"Raw Unicode Escape" 编解码器的 API:

PyObject ***PyUnicode_DecodeRawUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

回傳值：新的參照。🄵穩定 ABI 的一部分. 通过解码 Raw-Unicode-Escape 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_AsRawUnicodeEscapeString**(*PyObject* *unicode)

回傳值：新的參照。🄵穩定 ABI 的一部分. 使用 Raw-Unicode-Escape 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。

Latin-1 編解碼器

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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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 編解碼器

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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 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。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 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。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 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。🄵穩定 ABI 的一部分 *on Windows* 自 *3.7* 版本開始. 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。

Added in version 3.3.

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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.3.

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)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 將 *unicode* 中 *substr* 替換为 *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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. *PyUnicode_FromString()* 和*PyUnicode_InternInPlace()* 的组合操作，返回一个已内部化的新 Unicode 字符串对象，或一个指向具有相同值的原有内部化字符串对象的新的（“拥有的”）引用。

8.3.4 Tuple（元組）物件

type **PyTupleObject**

这个*PyObject* 的子类型代表一个 Python 的元组对象。

PyTypeObject **PyTuple_Type**

🄵穩定 ABI 的一部分. *PyTypeObject* 的实例代表一个 Python 元组类型，这与 Python 层面的 `tuple` 是相同的对象。

int **PyTuple_Check**(*PyObject* *p)

如果 *p* 是一个 tuple 对象或者 tuple 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 **PyTuple_CheckExact**(*PyObject* *p)

如果 *p* 是一个 tuple 对象但不是 tuple 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 ***PyTuple_New**(*Py_ssize_t* len)

回傳值：新的參照。🄵穩定 ABI 的一部分. 成功时返回一个新的元组对象，长度为 *len*，失败时返回 NULL。

PyObject ***PyTuple_Pack**(*Py_ssize_t* n, ...)

回傳值：新的參照。🄵穩定 ABI 的一部分. 成功时返回一个新的元组对象，大小为 *n*，失败时返回 NULL。元组值初始化为指向 Python 对象的后续 *n* 个 C 参数。`PyTuple_Pack(2, a, b)` 和 `Py_BuildValue("(OO)", a, b)` 相等。

Py_ssize_t **PyTuple_Size**(*PyObject* *p)

🄵穩定 ABI 的一部分. 获取指向元组对象的指针，并返回该元组的大小。

Py_ssize_t **PyTuple_GET_SIZE**(*PyObject* *p)

返回元组 *p* 的大小，它必须为非 NULL 并且指向一个元组；不执行错误检查。

PyObject ***PyTuple_GetItem**(*PyObject* *p, *Py_ssize_t* pos)

回傳值：借用參照。🄵穩定 ABI 的一部分. 返回 *p* 所指向的元组中位于 *pos* 处的对象。如果 *pos* 为负值或超出范围，则返回 NULL 并设置一个 IndexError 异常。

返回的引用是从元组 *p* 借入的（也就是说：它只在你持有对 *p* 的引用时才是可用的）。要获取*strong reference*，请使用*Py_NewRef(PyTuple_GetItem(...))* 或*PySequence_GetItem()*。

PyObject ***PyTuple_GET_ITEM**(*PyObject* *p, *Py_ssize_t* pos)

回傳值：借用參照。 类似于*PyTuple_GetItem()*，但不检查其参数。

PyObject ***PyTuple_GetSlice**(*PyObject* *p, *Py_ssize_t* low, *Py_ssize_t* high)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回 *p* 所指向的元组的从 *low* 到 *high* 的切片，或者在失败时返回 NULL。这等价于 Python 表达式 `p[low:high]`。不支持从元组末尾进行索引。

int **PyTuple_SetItem**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

🄵穩定 ABI 的一部分. 在 *p* 指向的元组的 *pos* 位置插入对对象 *o* 的引用。成功时返回 0；如果 *pos* 越界，则返回 -1，并抛出一个 IndexError 异常。

備🄵： 此函数会“窃取”对 *o* 的引用，并丢弃对元组中已在受影响位置的条目的引用。

---

void **PyTuple_SET_ITEM**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

类似于*PyTuple_SetItem()*，但不进行错误检查，并且应该只是被用来填充全新的元组。

備🄵： 这个函数会“窃取”一个对 *o* 的引用，但是，不与*PyTuple_SetItem()* 不同，它不会丢弃对任何被替换项的引用；元组中位于 *pos* 位置的任何引用都将被泄漏。

---

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。

PyTypeObject ***PyStructSequence_NewType**(*PyStructSequence_Desc* *desc)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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` 。

Added in version 3.4.

type **PyStructSequence_Desc**

🄵穩定 ABI 的一部分 （包含所有成員）. 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**

🄵穩定 ABI 的一部分 （包含所有成員）. 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**

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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🄵: 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🄵: 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**

PyObject 的這個子型🄵表示 Python 的 list（串列）物件。

PyTypeObject **PyList_Type**

🄵穩定 ABI 的一部分. 此*PyTypeObject* 實例表示 Python 的 list 型🄵。這與 Python 層中的 `list` 是同一個物件。

int **PyList_Check**(*PyObject* *p)

如果 *p* 是一個 list 物件或者是 list 型🄵之子型🄵的實例，就回傳 true。這個函式永遠會成功執行。

int **PyList_CheckExact**(*PyObject* *p)

如果 *p* 是一個 list 物件但不是 list 型🄵的子型🄵的實例，就回傳 true。這個函式永遠會成功執行。

PyObject ***PyList_New**(*Py_ssize_t* len)

回傳值：新的參照。🄵穩定 ABI 的一部分. 成功時回傳長度🄵 *len* 的新串列，失敗時回傳 `NULL`。

備🄵: 如果 *len* 大於零，則回傳的串列物件之項目將被設定🄵 `NULL`。因此，在使用*PyList_SetItem()* 來將所有項目設定🄵一個真實物件前，你無法使用像是*PySequence_SetItem()* 的使用抽象 API 函式，也不能將物件暴露 (expose) 給 Python 程式碼。

Py_ssize_t **PyList_Size**(*PyObject* *list)

🄵穩定 ABI 的一部分. 回傳 *list* 串列物件的長度；這相當於串列物件的 `len(list)`。

Py_ssize_t **PyList_GET_SIZE**(*PyObject* *list)

與*PyList_Size()* 類似，但🄵有錯誤檢查。

PyObject ***PyList_GetItem**(*PyObject* *list, *Py_ssize_t* index)

回傳值：借用參照。🄵穩定 ABI 的一部分. 回傳 *list* 指向的串列中位於 *index* 位置的物件。該位置不可🄵負數；🄵不支援從串列尾末開始索引。如果 *index* 超出邊界範圍（<0 或 >=len(list)）則回傳 `NULL` 🄵設定 `IndexError` 例外。

PyObject ***PyList_GET_ITEM**(*PyObject* *list, *Py_ssize_t* i)

回傳值：借用參照。與*PyList_GetItem()* 類似，但🄵有錯誤檢查。

int **PyList_SetItem**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

🄵穩定 ABI 的一部分. 將串列中索引 *index* 處的項目設定🄵 *item*。成功時回傳 `0`。如果 *index* 超出邊界範圍則回傳 `-1` 🄵設定一個 `IndexError` 例外。

備🄵: 此函式「竊取」對 *item* 的參照，🄵🄵🄵對串列中受影響位置上已存在項目的參照。

---

void **PyList_SET_ITEM**(*PyObject* *list, *Py_ssize_t* i, *PyObject* *o)

PyList_SetItem() 的巨集形式，🄵有錯誤檢查。這通常僅用於填充🄵有已存在🄵容的新串列。

備🄵： 該巨集「竊取」對 *item* 的參照，🄵且與*PyList_SetItem()* 不同的是，它不會🄵🄵對任意被替🄵項目的參照；*list* 中位置 *i* 的任何參照都將被🄵漏 (leak)。

int **PyList_Insert**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

🄵穩定 ABI 的一部分. 將項目 *item* 插入串列 *list* 中索引 *index* 的位置之前。如果成功則回傳 `0`；如果失敗則回傳 `-1` 🄵設定例外。類似於 `list.insert(index, item)`。

int **PyList_Append**(*PyObject* *list, *PyObject* *item)

🄵穩定 ABI 的一部分. 將物件 *item* 附加到串列 *list* 的最後面。如果成功則回傳 `0`；如果不成功，則回傳 `-1` 🄵設定例外。類似於 `list.append(item)`。

PyObject ***PyList_GetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high)

回傳值：新的參照。🄵穩定 ABI 的一部分. 回傳 *list* 中的物件串列，其中包含 *low* 和 *high* 之間的物件。如果🄵有成功則回傳 `NULL` 🄵設定例外。類似於 `list[low:high]`。不支援從串列尾末開始索引。

int **PyList_SetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high, *PyObject* *itemlist)

🄵穩定 ABI 的一部分. 將 *low* 和 *high* 之間的 *list* 切片設定🄵 *itemlist* 的🄵容。類似於 `list[low:high] = itemlist`。*itemlist* 可能🄵 `NULL`，表示分配一個空串列（切片🄵除）。成功時回傳 `0`，失敗時則回傳 `-1`。不支援從串列尾末開始索引。

int **PyList_Sort**(*PyObject* *list)

🄵穩定 ABI 的一部分. 對 *list* 的項目進行原地 (in place) 排序。成功時回傳 `0`，失敗時回傳 `-1`。這相當於 `list.sort()`。

int **PyList_Reverse**(*PyObject* *list)

🄵穩定 ABI 的一部分. 原地反轉 *list* 的項目。成功時回傳 `0`，失敗時回傳 `-1`。這相當於 `list.reverse()`。

PyObject ***PyList_AsTuple**(*PyObject* *list)

回傳值：新的參照。🄵穩定 ABI 的一部分. 回傳一個新的 tuple（元組）物件，其中包含 *list* 的🄵容；相當於 `tuple(list)`。

8.4 容器物件

8.4.1 字典物件

type **PyDictObject**

PyObject 子型態代表一個 Python 字典物件。

PyTypeObject **PyDict_Type**

🄵穩定 ABI 的一部分. *PyTypeObject* 實例代表一個 Python 字典型態。此與 Python 層中的 `dict` 🄵同一個物件。

int **PyDict_Check**(*PyObject* *p)

若 *p* 是一個字典物件或字典的子型態實例則會回傳 true。此函式每次都會執行成功。

int **PyDict_CheckExact**(*PyObject* *p)

若 *p* 是一個字典物件但🄵不是一個字典子型態的實例，則回傳 true。此函式每次都會執行成功。

PyObject ***PyDict_New**()

回傳值：新的參照。🄵穩定 ABI 的一部分. 回傳一個新的空字典，或在失敗時回傳 `NULL`。

PyObject ***PyDictProxy_New**(*PyObject* *mapping)

回傳值：新的參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 清空现有字典的所有键值对。

int **PyDict_Contains**(*PyObject* *p, *PyObject* *key)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 返回与 *p* 包含相同键值对的新字典。

int **PyDict_SetItem**(*PyObject* *p, *PyObject* *key, *PyObject* *val)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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🄵: 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。🄵穩定 ABI 的一部分. *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。🄵穩定 ABI 的一部分. 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🄵: 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()* 函数并附带你自己的*PyUnicode_FromString()* *key*。

PyObject ***PyDict_SetDefault**(*PyObject* *p, *PyObject* *key, *PyObject* *defaultobj)

回傳值：借用參照。 这跟 Python 层面的 `dict.setdefault()` 一样。如果键 *key* 存在，它返回在字典 *p* 里面对应的值。如果键不存在，它会和值 *defaultobj* 一起插入并返回 *defaultobj* 。这个函数只计算 *key* 的哈希函数一次，而不是在查找和插入时分别计算它。

Added in version 3.4.

PyObject ***PyDict_Items**(*PyObject* *p)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回一个包含字典中所有键值项的*PyListObject*。

PyObject ***PyDict_Keys**(*PyObject* *p)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回一个包含字典中所有键 (keys) 的*PyListObject*。

PyObject ***PyDict_Values**(*PyObject* *p)

回傳值：新的參照。🄵穩定 ABI 的一部分. 返回一个包含字典中所有值 (values) 的*PyListObject*。

Py_ssize_t **PyDict_Size**(*PyObject* *p)

🄵穩定 ABI 的一部分. 返回字典中项目数，等价于对字典 *p* 使用 `len(p)`。

int **PyDict_Next**(*PyObject* *p, *Py_ssize_t* *ppos, *PyObject* **pkey, *PyObject* **pvalue)

🄵穩定 ABI 的一部分. 迭代字典 *p* 中的所有键值对。在第一次调用此函数开始迭代之前，由 *ppos* 所引用的*Py_ssize_t* 必须被初始化为 `0`；该函数将为字典中的每个键值对返回真值，一旦所有键值对都报告完毕则返回假值。形参 *pkey* 和 *pvalue* 应当指向 *PyObject** 变量，它们将分别使用每个键和值来填充，或者也可以为 `NULL`。通过它们返回的任何引用都是暂借的。*ppos* 在迭代期间不应被更改。它的值表示内部字典结构中的偏移量，并且由于结构是稀疏的，因此偏移量并不连续。

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* 不应该在遍历期间发生改变。在遍历字典时，改变键中的值是安全的，但仅限于键的集合不发生改变。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)

🄵穩定 ABI 的一部分. 对映射对象 *b* 进行迭代，将键值对添加到字典 *a*。*b* 可以是一个字典，或任何支持*PyMapping_Keys()* 和*PyObject_GetItem()* 的对象。如果 *override* 为真值，则如果在 *b* 中找到相同的键则 *a* 中已存在的相应键值对将被替换，否则如果在 *a* 中没有相同的键则只是添加键值对。当成功时返回 `0` 或者当引发异常时返回 `-1`。

int **PyDict_Update**(*PyObject* *a, *PyObject* *b)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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
```

int **PyDict_AddWatcher**(*PyDict_WatchCallback* callback)

在字典上注册 *callback* 来作为 watcher。返回值为非负数的整数 id，作为将来调用*PyDict_Watch()* 的时候使用。如果出现错误（比如没有足够的可用 watcher ID），返回 `-1` 并且设置异常。

Added in version 3.12.

int **PyDict_ClearWatcher**(int watcher_id)

清空由之前从*PyDict_AddWatcher()* 返回的 *watcher_id* 所标识的 watcher。成功时返回 `0`，出错时（例如当给定的 *watcher_id* 未被注册）返回 `-1`。

Added in version 3.12.

int **PyDict_Watch**(int watcher_id, *PyObject* *dict)

将字典 *dict* 标记为已被监视。由*PyDict_AddWatcher()* 授权 *watcher_id* 对应的回调将在 *dict* 被修改或释放时被调用。成功时返回 `0`，出错时返回 `-1`。

Added in version 3.12.

int **PyDict_Unwatch**(int watcher_id, *PyObject* *dict)

将字典 *dict* 标记为不再被监视。由*PyDict_AddWatcher()* 授权 *watcher_id* 对应的回调在 *dict* 被修改或释放时将不再被调用。该字典在此之前必须已被此监视器所监视。成功时返回 `0`，出错时返回 `-1`。

Added in version 3.12.

type **PyDict_WatchEvent**

由以下可能的字典监视器事件组成的枚举: `PyDict_EVENT_ADDED`, `PyDict_EVENT_MODIFIED`, `PyDict_EVENT_DELETED`, `PyDict_EVENT_CLONED`, `PyDict_EVENT_CLEARED` 或 `PyDict_EVENT_DEALLOCATED`。

Added in version 3.12.

typedef int (***PyDict_WatchCallback**)(*PyDict_WatchEvent* event, *PyObject* *dict, *PyObject* *key, *PyObject* *new_value)

字典监视器回调函数的类型。

如果 *event* 是 `PyDict_EVENT_CLEARED` 或 `PyDict_EVENT_DEALLOCATED`，则 *key* 和 *new_value* 都将为 `NULL`。如果 *event* 是 `PyDict_EVENT_ADDED` 或 `PyDict_EVENT_MODIFIED`，则 *new_value* 将为 *key* 的新值。如果 *event* 是 `PyDict_EVENT_DELETED`，则将从字典中删除 *key* 而 *new_value* 将为 `NULL`。

`PyDict_EVENT_CLONED` 会在另一个字典合并到之前为空的 *dict* 时发生。为保证此操作的效率，该场景不会发出针对单个键的 `PyDict_EVENT_ADDED` 事件；而是发出单个 `PyDict_EVENT_CLONED`，而 *key* 将为源字典。

该回调可以检查但不能修改 *dict*；否则会产生不可预料的影响，包括无限递归。请不要在该回调中触发 Python 代码的执行，因为它可能产生修改 dict 的附带影响。

如果 *event* 是 `PyDict_EVENT_DEALLOCATED`，则在回调中接受一个对即将销毁的字典的新引用将使其重生并阻止其在此时被释放。当重生的对象以后再被销毁时，任何在当时已激活的监视器回调将再次被调用。

回调会在已通知的对 *dict* 的修改完成之前执行，这样在此之前的 *dict* 状态可以被检查。

如果该回调设置了一个异常，则它必须返回 -1；此异常将作为不可引发的异常使用*PyErr_WriteUnraisable()* 打印出来。在其他情况下它应当返回 0。

在进入回调时可能已经设置了尚未处理的异常。在此情况下，回调应当返回 0 并仍然设置同样的异常。这意味着该回调可能不会调用任何其他可设置异常的 API 除非它先保存并清空异常状态，并在返回之前恢复它。

Added in version 3.12.

8.4.2 集合物件

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**

🄵穩定 ABI 的一部分. 这是一个*PyTypeObject* 实例，表示 Python set 类型。

PyTypeObject **PyFrozenSet_Type**

🄵穩定 ABI 的一部分. 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。

Added in version 3.10.

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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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🄵，Python 程式設計者可透過 `types.FunctionType` 使用它。

int **PyFunction_Check**(*PyObject* *o)

如果 *o* 是個函式物件（擁有*PyFunction_Type* 的型🄵）則回傳 true。參數必須不🄵 NULL。此函式必能成功執行。

PyObject ***PyFunction_New**(*PyObject* *code, *PyObject* *globals)

回傳值：新的參照。 回傳一個與程式碼物件 *code* 相關聯的函式物件。*globals* 必須是一個帶有函式能🄵存取的全域變數的字典。

函式的文件字串 (docstring) 和名稱是從程式碼物件所取得，__module__ 是自 *globals* 所取得。引數預設值、標🄵 (annotation) 和閉包 (closure) 被設🄵 NULL，__qualname__ 被設🄵和程式碼物件 co_qualname 欄位相同的值。

PyObject ***PyFunction_NewWithQualName**(*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)

回傳值：新的參照。 和*PyFunction_New()* 相似，但也允許函式物件 __qualname__ 屬性的設定，*qualname* 應🄵一個 unicode 物件或是 NULL；如🄵 NULL，__qualname__ 屬性會被設🄵與程式碼物件 co_qualname 欄位相同的值。

Added in version 3.3.

PyObject ***PyFunction_GetCode**(*PyObject* *op)

回傳值：借用參照。 回傳與程式碼物件相關的函式物件 *op*。

PyObject ***PyFunction_GetGlobals**(*PyObject* *op)

回傳值：借用參照。 回傳與全域函式字典相關的函式物件 *op*。

PyObject ***PyFunction_GetModule**(*PyObject* *op)

回傳值：借用參照。 回傳一個函式物件 *op* 之 __module__ 屬性的*borrowed reference*，它可以是 *NULL*。

這通常是個包含模組名稱的字串，但可以被 Python 程式設🄵任何其他物件。

PyObject ***PyFunction_GetDefaults**(*PyObject* *op)

回傳值：借用參照。 回傳函式物件 *op* 的引數預設值，這可以是一個含有多個引數的 tuple（元組）或 NULL。

int **PyFunction_SetDefaults**(*PyObject* *op, *PyObject* *defaults)

設定函式物件 *op* 的引數預設值。*defaults* 必須是 Py_None 或一個 tuple。

引發 SystemError 且在失敗時回傳 -1。

void **PyFunction_SetVectorcall**(*PyFunctionObject* *func, *vectorcallfunc* vectorcall)

🄵一個給定的函式物件 *func* 設定 vectorcall 欄位。

警告：使用此 API 的扩展必须保留未修改的（默认）vectorcall 函数的行为！

Added in version 3.12.

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。

int **PyFunction_AddWatcher**(*PyFunction_WatchCallback* callback)

注册 *callback* 作为当前解释器的函数监视器。返回一个可被传给*PyFunction_ClearWatcher()* 的 ID。如果出现错误（比如没有足够的可用监视器 ID），则返回 `-1` 并设置一个异常。

Added in version 3.12.

int **PyFunction_ClearWatcher**(int watcher_id)

清空当前解释器在之前从 Clear watcher identified by previously returned from *PyFunction_AddWatcher()* 返回的由 *watcher_id* 所标识的监视器。成功时返回 `0`，或者出错时（比如当给定的 *watcher_id* 未被注册）返回 `-1` 并设置一个异常。

Added in version 3.12.

type **PyFunction_WatchEvent**

由以下可能的函数监视器事件组成的枚举: - PyFunction_EVENT_CREATE - PyFunction_EVENT_DESTROY - PyFunction_EVENT_MODIFY_CODE - PyFunction_EVENT_MODIFY_DEFAULTS - PyFunction_EVENT_MODIFY_KWDEFAULTS

Added in version 3.12.

typedef int (***PyFunction_WatchCallback**)(*PyFunction_WatchEvent* event, *PyFunctionObject* *func, *PyObject* *new_value)

函数监视器回调函数的类型。

如果 *event* 为 `PyFunction_EVENT_CREATE` 或 `PyFunction_EVENT_DESTROY` 则 *new_value* 将为 `NULL`。在其他情况下，*new_value* 将为被修改的属性持有一个指向要保存在 *func* 中的新值的*borrowed reference*。

该回调可以检查但不能修改 *func*; 这样做可能具有不可预知的影响，包括无限递归。

如果 *event* 是 `PyFunction_EVENT_CREATE`，则该回调会在 *func* 完成初始化之后被发起调用。在其他情况下，该回调会在对 *func* 进行修改之前被发起调用，这样就可以检查 *func* 之前的状态。如有可能函数对象的创建允许被运行时优化掉。在此情况下将不发出任何事件。虽然根据不同的优化决定这会产生可被观察到的运行时行为变化，但是它不会改变被运行的 Python 代码的语义。

如果 *event* 是 `PyFunction_EVENT_DESTROY`，则在回调中接受一个即将销毁的函数的引用将使其重生，并阻止其在此时被释放。当重生的对象以后再被销毁时，任何在当时已激活的监视器回调将再次被调用。

如果该回调设置了一个异常，则它必须返回 `-1`；此异常将作为不可引发的异常使用*PyErr_WriteUnraisable()* 打印出来。在其他情况下它应当返回 `0`。

在进入回调时可能已经设置了尚未处理的异常。在此情况下，回调应当返回 `0` 并仍然设置同样的异常。这意味着该回调可能不会调用任何其他可设置异常的 API 除非它先保存并清空异常状态，并在返回之前恢复它。

Added in version 3.12.

8.5.2 實例方法物件 (Instance Method Objects)

實例方法是*PyCFunction* 的包裝器 (wrapper)，也是將*PyCFunction* 🄵結 (bind) 到類🄵物件的一種新方式。它替代了原先對 `PyMethod_New(func, NULL, class)` 的呼叫。

PyTypeObject **PyInstanceMethod_Type**

PyTypeObject 的實例代表 Python 實例方法型🄵。它不會公開 (expose) 給 Python 程式。

int **PyInstanceMethod_Check**(*PyObject* *o)

如果 *o* 是一個實例方法物件（型🄵🄵*PyInstanceMethod_Type*）則回傳 true。參數必須不🄵 `NULL`。此函式總是會成功執行。

PyObject ***PyInstanceMethod_New**(*PyObject* *func)

回傳值：新的參照。 回傳一個新的實例方法物件，*func* 🄵任意可呼叫物件，在實例方法被呼叫時 *func* 函式也會被呼叫。

PyObject ***PyInstanceMethod_Function**(*PyObject* *im)

回傳值：借用參照。 回傳關聯到實例方法 *im* 的函式物件。

PyObject ***PyInstanceMethod_GET_FUNCTION**(*PyObject* *im)

回傳值：借用參照。 巨集 (macro) 版本的*PyInstanceMethod_Function()*，忽略了錯誤檢查。

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 **PyCellObject**

Cell 物件所用之 C 結構。

PyTypeObject **PyCell_Type**

對應 cell 物件的物件型🄵。

int **PyCell_Check**(*PyObject* *ob)

如果 *ob* 是一個 cell 物件則回傳真值；*ob* 必須不🄵 `NULL`。此函式總是會成功執行。

PyObject ***PyCell_New**(*PyObject* *ob)

回傳值：新的參照。 建立🄵回傳一個包含 *ob* 的新 cell 物件。參數可以🄵 `NULL`。

PyObject ***PyCell_Get**(*PyObject* *cell)

回傳值：新的參照。 回傳 cell 🄵容中的 *cell*。

PyObject ***PyCell_GET**(*PyObject* *cell)

回傳值：借用參照。 回傳 cell 物件 *cell* 的🄴容，但是不檢查 *cell* 是否非 NULL 🄵且🄴一個 cell 物件。

int **PyCell_Set**(*PyObject* *cell, *PyObject* *value)

將 cell 物件 *cell* 的🄴容設🄵 *value*。這將釋放任何對 cell 物件當前🄴容的參照。*value* 可以🄵 NULL。*cell* 必須不🄵 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🄵 *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

代码对象是 CPython 实现的低层级细节。每个代表一块尚未绑定到函数中的可执行代码。

type **PyCodeObject**

用于描述代码对象的对象的 C 结构。此类型字段可随时更改。

PyTypeObject **PyCode_Type**

这一个代表 Python 代码对象的*PyTypeObject* 实例。

int **PyCode_Check**(*PyObject* *co)

如果 *co* 是一个 代码对象则返回真值。此函数总是会成功执行。

Py_ssize_t **PyCode_GetNumFree**(*PyCodeObject* *co)

返回代码对象中的自由变量数。

int **PyCode_GetFirstFree**(*PyCodeObject* *co)

返回代码对象中第一个自由变量的位置。

PyCodeObject ***PyUnstable_Code_New**(int argcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)

這是不穩定 *API*，它可能在小版本發布中🄴有任何警告地被變更。

返回一个新的代码对象。如果你需要一个用空代码对象来创建帧，请改用*PyCode_NewEmpty()*。

由于字节码的定义经常变化，可以直接调用*PyUnstable_Code_New()* 来绑定某个确定的 Python 版本。

此函数的许多参数以复杂的方式相互依赖，这意味着参数值的细微改变可能导致不正确的执行或 VM 崩溃。使用此函数需要极度小心。

在 3.11 版的變更: 新增 qualname 和 exceptiontable 参数。

在 3.12 版的變更: 由 PyCode_New 更名而来，是不穩定的 *C API* 的一部分。旧名称已被弃用，但在签名再次更改之前仍然可用。

PyCodeObject ***PyUnstable_Code_NewWithPosOnlyArgs**(int argcount, int posonlyargcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)

這是不穩定 *API*，它可能在小版本發布中🄴有任何警告地被變更。

与*PyUnstable_Code_New()* 类似，但额外增加了一个针对仅限位置参数的"posonlyargcount"。适用于 PyUnstable_Code_New 的适用事项同样适用于这个函数。

Added in version 3.8: 作为 PyCode_NewWithPosOnlyArgs

在 3.11 版的變更: 新增 qualname 和 exceptiontable 參數。

在 3.12 版的變更: 重命名为 PyUnstable_Code_NewWithPosOnlyArgs。旧名称已被弃用，但在签名再次更改之前将保持可用。

PyCodeObject ***PyCode_NewEmpty**(const char *filename, const char *funcname, int firstlineno)

回傳值：新的參照。 返回一个具有指定用户名、函数名和首行行号的空代码对象。结果代码对象如果被执行则将引发一个 Exception。

int **PyCode_Addr2Line**(*PyCodeObject* *co, int byte_offset)

返回在 byte_offset 位置或之前以及之后发生的指令的行号。如果你只需要一个帧的行号，请改用*PyFrame_GetLineNumber()*。

要高效地对代码对象中的行号进行迭代，请使用 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)

将传入的 int 指针设为 byte_offset 处的指令的源代码行编号和列编号。当没有任何特定元素的信息时则将值设为 0。

如果函数执行成功则返回 1 否则返回 0。

Added in version 3.11.

PyObject ***PyCode_GetCode**(*PyCodeObject* *co)

等价于 Python 代码 getattr(co, 'co_code')。返回一个指向表示代码对象中的字节码的*PyBytesObject* 的强引用。当出错时，将返回 NULL 并引发一个异常。

这个 PyBytesObject 可以由解释器按需创建并且不必代表 CPython 所实际执行的字节码。此函数的主要用途是调试器和性能分析工具。

Added in version 3.11.

PyObject ***PyCode_GetVarnames**(*PyCodeObject* *co)

等价于 Python 代码 getattr(co, 'co_varnames')。返回一个指向包含局部变量名称的*PyTupleObject* 的新引用。当出错时，将返回 NULL 并引发一个异常。

Added in version 3.11.

PyObject ***PyCode_GetCellvars**(*PyCodeObject* *co)

等价于 Python 代码 getattr(co, 'co_cellvars')。返回一个包含被嵌套的函数所引用的局部变量名称的*PyTupleObject* 的新引用。当出错时，将返回 NULL 并引发一个异常。

Added in version 3.11.

PyObject ***PyCode_GetFreevars**(*PyCodeObject* *co)

等价于 Python 代码 getattr(co, 'co_freevars')。返回一个指向包含自由变量名称的*PyTupleObject* 的新引用。当出错时，将返回 NULL 并引发一个异常。

Added in version 3.11.

int **PyCode_AddWatcher**(*PyCode_WatchCallback* callback)

注册 *callback* 作为当前解释器的代码对象监视器。返回一个可被传给*PyCode_ClearWatcher()* 的 ID。如果出现错误（例如没有足够的可用监视器 ID），则返回 -1 并设置一个异常。

Added in version 3.12.

int **PyCode_ClearWatcher**(int watcher_id)

清除之前从*PyCode_AddWatcher()* 返回的当前解释器中由 *watcher_id* 所标识的监视器。成功时返回 0，或者出错时（例如当给定的 *watcher_id* 未被注册）返回 -1 并设置异常。

Added in version 3.12.

type **PyCodeEvent**

由可能的代码对象监视器事件组成的枚举：- PY_CODE_EVENT_CREATE - PY_CODE_EVENT_DESTROY

Added in version 3.12.

typedef int (***PyCode_WatchCallback**)(*PyCodeEvent* event, *PyCodeObject* *co)

代码对象监视器回调函数的类型。

如果 *event* 为 PY_CODE_EVENT_CREATE，则回调会在 *co* 完全初始化后被发起调用。否则，回调会在 *co* 执行销毁之前被发起调用，这样就可以检查 *co* 之前的状态。

如果 *event* 为 PY_CODE_EVENT_DESTROY，则在回调中接受一个即将被销毁的代码对象的引用将使其重生，并阻止其在此时被释放。当重生的对象以后再被销毁时，任何在当时已激活的监视器回调将再次被调用。

本 API 的用户不应依赖内部运行时的实现细节。这类细节可能包括但不限于创建和销毁代码对象的确切顺序和时间。虽然这些细节的变化可能会导致监视器可观察到的差异（包括回调是否被发起调用），但不会改变正在执行的 Python 代码的语义。

如果该回调设置了一个异常，则它必须返回 -1；此异常将作为不可引发的异常使用*PyErr_WriteUnraisable()* 打印出来。在其他情况下它应当返回 0。

在进入回调时可能已经设置了尚未处理的异常。在此情况下，回调应当返回 0 并仍然设置同样的异常。这意味着该回调可能不会调用任何其他可设置异常的 API 除非它先保存并清空异常状态，并在返回之前恢复它。

Added in version 3.12.

8.5.6 附加信息

为了支持对帧求值的低层级扩展，如外部即时编译器等，可以在代码对象上附加任意的额外数据。

这些函数是不稳定 C API 层的一部分：该功能是 CPython 的实现细节，此 API 可能随时改变而不发出弃用警告。

Py_ssize_t **PyUnstable_Eval_RequestCodeExtraIndex**(*freefunc* free)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

返回一个新的不透明索引值用于向代码对象添加数据。

通常情况下（对于每个解释器）你只需调用该函数一次然后将调用结果与 PyCode_GetExtra 和 PyCode_SetExtra 一起使用以操作单个代码对象上的数据。

如果 *free* 没有不为 NULL: 当代码对象被释放时，*free* 将在存储于新索引下的非 NULL 数据上被调用。当存储*PyObject* 时使用*Py_DecRef()*。

Added in version 3.6: 作为 _PyEval_RequestCodeExtraIndex

在 3.12 版的變更: 重命名为 PyUnstable_Eval_RequestCodeExtraIndex。旧的私有名称已被弃用，但在 API 更改之前仍将可用。

int **PyUnstable_Code_GetExtra**(*PyObject* *code, *Py_ssize_t* index, void **extra)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

將 *extra* 设为存储在给定索引下的额外数据。成功时将返回 0。失败时将设置一个异常并返回 -1。

如果未在索引下设置数据，则将 *extra* 设为 `NULL` 并返回 0 而不设置异常。

Added in version 3.6: 作为 `_PyCode_GetExtra`

在 3.12 版的變更: 重命名为 `PyUnstable_Code_GetExtra`。旧的私有名称已被弃用，但在 API 更改之前仍将可用。

int **PyUnstable_Code_SetExtra**(*PyObject* *code, *Py_ssize_t* index, void *extra)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

將存储在给定索引下的额外数据设为 *extra*。成功时将返回 0。失败时将设置一个异常并返回 -1。

Added in version 3.6: 作为 `_PyCode_SetExtra`

在 3.12 版的變更: 重命名为 `PyUnstable_Code_SetExtra`。旧的私有名称已被弃用，但在 API 更改之前仍将可用。

8.6 其他物件

8.6.1 檔案物件 (File Objects)

這些 API 是用於🄵建檔案物件的 Python 2 C API 的最小模擬 (minimal emulation)，它過去依賴於 C 標准函式庫對於緩衝 I/O (`FILE*`) 的支援。在 Python 3 中，檔案和串流使用新的 `io` 模組，它在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🄵部錯誤報告；建議第三方程式碼改🄵存取 `io` API。

PyObject ***PyFile_FromFd**(int fd, const char *name, const char *mode, int buffering, const char *encoding, const char *errors, const char *newline, int closefd)

回傳值：新的參照。🄵穩定 ABI 的一部分. 根据已打开文件 *fd* 的文件描述符创建一个 Python 文件对象。参数 *name*, *encoding*, *errors* 和 *newline* 可以为 `NULL` 表示使用默认值；*buffering* 可以为 *-1* 表示使用默认值。*name* 会被忽略仅保留用于向下兼容。失败时返回 `NULL`。有关参数的更全面描述，请参阅 `io.open()` 函数的文档。

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🄵生各種問題（例如資料的排序不符合預期）。

在 3.2 版的變更: 忽略 *name* 屬性。

int **PyObject_AsFileDescriptor**(*PyObject* *p)

🄵穩定 ABI 的一部分. 回傳與 *p* 關聯的檔案描述器作🄵 `int`。如果物件是整數，則回傳其值。如果不是整數，則呼叫物件的 `fileno()` 方法（如果存在）；該方法必須回傳一個整數，它作🄵檔案描述器值回傳。設定例外🄵在失敗時回傳 `-1`。

PyObject ***PyFile_GetLine**(*PyObject* *p, int n)

回傳值：新的參照。🄵穩定 ABI 的一部分. 等价于 `p.readline([n])`，这个函数从对象 *p* 中读取一行。*p* 可以是文件对象或具有 `readline()` 方法的任何对象。如果 *n* 是 `0`，则无论该行的长度如何，都会读取一行。如果 *n* 大于 `0`，则从文件中读取不超过 *n* 个字节；可以返回行的一部分。在这两种情况下，如果立即到达文件末尾，则返回空字符串。但是，如果 *n* 小于 `0`，则无论长度如何都会读取一行，但是如果立即到达文件末尾，则引发 `EOFError`。

int **PyFile_SetOpenCodeHook**(*Py_OpenCodeHookFunction* handler)

覆盖 `io.open_code()` 的正常行🄵以透過提供的處理程式 (handler) 傳遞其參數。

处理器函数的类型为：

type **Py_OpenCodeHookFunction**

等价于 *PyObject* *(*)(*PyObject* *path, void *userData)，其中 *path* 会确保为*PyUnicodeObject*。

userData 指標被傳遞到🄵🄵函式 (hook function) 中。由於可能會從不同的執行環境 (runtime) 呼叫🄵🄵函式，因此該指標不應直接指向 Python 狀態。

由於此🄵🄵函式是在導入期間有意使用的，因此請避免在其執行期間導入新模組，除非它們已知有被凍結或在 `sys.modules` 中可用。

一旦钩子被设定，它就不能被移除或替换，之后对*PyFile_SetOpenCodeHook()* 的调用也将失败，如果解释器已经被初始化，函数将返回 -1 并设置一个异常。

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`。

Added in version 3.8.

int **PyFile_WriteObject**(*PyObject* *obj, *PyObject* *p, int flags)

🄵穩定 ABI 的一部分. 將物件 *obj* 寫入檔案物件 *p*。*flags* 唯一支援的旗標是*Py_PRINT_RAW*；如果有給定，則寫入物件的 `str()` 而不是 `repr()`。在成功回傳 `0` 或在失敗回傳 `-1`；將設定適當的例外。

int **PyFile_WriteString**(const char *s, *PyObject* *p)

🄵穩定 ABI 的一部分. 寫入字串 *s* 到檔案物件 *p*。當成功時回傳 0，而當失敗時回傳 -1，🄵會設定合適的例外狀🄵。

8.6.2 模組物件模組

PyTypeObject **PyModule_Type**

🄵穩定 ABI 的一部分. 这个 C 类型实例*PyTypeObject* 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。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.3.

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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.3.

const char ***PyModule_GetName**(*PyObject* *module)

🄵穩定 ABI 的一部分. 类似于*PyModule_GetNameObject()* 但返回 'utf-8' 编码的名称。

void ***PyModule_GetState**(*PyObject* *module)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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。

Added in version 3.2.

const char ***PyModule_GetFilename**(*PyObject* *module)

🄵穩定 ABI 的一部分. 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**

🄵穩定 ABI 的一部分 （包含所有成員）. 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。

將其设置为非负值，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。🄵穩定 ABI 的一部分. 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🄵: 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。🄵穩定 ABI 的一部分 自 *3.5* 版本開始. 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`。

Added in version 3.5.

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。

Added in version 3.5.

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。

Py_mod_multiple_interpreters

指定以下的值之一:

Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED

该模块不支持在子解释器中导入。

Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED

该模块支持在子解释器中导入，但是它们必须要共享主解释器的 GIL。(参见 isolating-extensions-howto。)

Py_MOD_PER_INTERPRETER_GIL_SUPPORTED

该模块支持在子解释器中导入，即使它们有自己的 GIL。(参见 isolating-extensions-howto。)

此槽位决定在子解释器中导入此模块是否会失败。

在一个模块定义中不能指定多个 `Py_mod_multiple_interpreters` 槽位。

如果未指定 `Py_mod_multiple_interpreters`，则导入机制默认为 `Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED`。

Added in version 3.12.

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`。

Added in version 3.5.

PyObject ***PyModule_FromDefAndSpec2**(*PyModuleDef* *def, *PyObject* *spec, int module_api_version)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 创建一个新的模块对象，在参数 *def* 和 *spec* 中给出定义，设置 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🄵: 大多数时候应该使用*PyModule_FromDefAndSpec()* 代替使用此函数，除非你确定需要使用它。

Added in version 3.5.

int **PyModule_ExecDef**(*PyObject* *module, *PyModuleDef* *def)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 执行参数 *def* 中给出的任意执行槽（*Py_mod_exec*）。

Added in version 3.5.

int **PyModule_SetDocString**(*PyObject* *module, const char *docstring)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 将 *module* 的文档字符串设置为 *docstring*。当使用 `PyModule_Create` 或 `PyModule_FromDefAndSpec` 从 `PyModuleDef` 创建模块时，会自动调用此函数。

Added in version 3.5.

int **PyModule_AddFunctions**(*PyObject* *module, *PyMethodDef* *functions)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 将以 NULL 结尾的 *functions* 数组中的函数添加到 *module* 模块中。有关单个条目的更多细节，请参与*PyMethodDef* 文档（由于缺少共享的模块命名空间，在 C 中实现的模块级“函数”通常将模块作为它的第一个参数，与 Python 类的实例方法类似）。当使用 PyModule_Create 或 PyModule_FromDefAndSpec 从 PyModuleDef 创建模块时，会自动调用此函数。

Added in version 3.5.

支持函数

模块初始化函数（单阶段初始化）或通过模块的执行槽位调用的函数（多阶段初始化），可以使用以下函数，来帮助初始化模块的状态：

int **PyModule_AddObjectRef**(*PyObject* *module, const char *name, *PyObject* *value)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 将一个名称为 *name* 的对象添加到 *module* 模块中。这是一个方便的函数，可以在模块的初始化函数中使用。

如果成功，返回 0。如果发生错误，引发异常并返回 -1。

如果 *value* 为 NULL，返回 NULL。在调用它时发生这种情况，必须抛出异常。

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;
}
```

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);
    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。

Added in version 3.10.

int **PyModule_AddObject**(*PyObject* *module, const char *name, *PyObject* *value)

🄵穩定 ABI 的一部分. 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🄵: 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 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`。

Added in version 3.9.

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)

回傳值：借用參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.3* 版本開始. 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。

Added in version 3.3.

int **PyState_RemoveModule**(*PyModuleDef* *def)

🄵穩定 ABI 的一部分 自 *3.3* 版本開始. 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。

Added in version 3.3.

8.6.3 🄵代器（Iterator）物件

Python 提供了兩種通用的🄵代器 (iterator) 物件，第一種是序列🄵代器 (sequence iterator)，適用於支援 __getitem__() 方法的任意序列，第二種是與可呼叫 (callable) 物件和哨兵值 (sentinel value) 一起使用，會呼叫序列中的每個可呼叫物件，當回傳哨兵值時就結束🄵代。

PyTypeObject **PySeqIter_Type**

🄵穩定 ABI 的一部分. 此型🄵物件用於由*PySeqIter_New()* 所回傳的🄵代器物件以及用於🄵建序列型🄵的🄵建函式 iter() 的單引數形式。

int **PySeqIter_Check**(*PyObject* *op)

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。🄵穩定 ABI 的一部分. 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**

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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 可调用对象；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" 是描述物件某些屬性的物件，它們存在於型🄵物件的 dictionary（字典）中。

PyTypeObject **PyProperty_Type**

🄵穩定 ABI 的一部分. 🄵建 descriptor 型🄵的型🄵物件。

PyObject ***PyDescr_NewGetSet**(*PyTypeObject* *type, struct *PyGetSetDef* *getset)

回傳值：新的參照。🄵穩定 ABI 的一部分.

PyObject ***PyDescr_NewMember**(*PyTypeObject* *type, struct *PyMemberDef* *meth)

回傳值：新的參照。🄵穩定 ABI 的一部分.

PyObject ***PyDescr_NewMethod**(*PyTypeObject* *type, struct *PyMethodDef* *meth)

回傳值：新的參照。🄵穩定 ABI 的一部分.

PyObject ***PyDescr_NewWrapper**(*PyTypeObject* *type, struct wrapperbase *wrapper, void *wrapped)

回傳值：新的參照。

PyObject ***PyDescr_NewClassMethod**(*PyTypeObject* *type, *PyMethodDef* *method)

回傳值：新的參照。🄵穩定 ABI 的一部分.

int **PyDescr_IsData**(*PyObject* *descr)

如果 descriptor 物件 *descr* 描述的是一個資料屬性則回傳非零值，或者如果它描述的是一個方法則返回 `0`。*descr* 必須🄵一個 descriptor 物件；🄵有錯誤檢查。

PyObject ***PyWrapper_New**(*PyObject**, *PyObject**)

回傳值：新的參照。🄵穩定 ABI 的一部分.

8.6.5 切片物件

PyTypeObject **PySlice_Type**

🄵穩定 ABI 的一部分. 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。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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。

成功时返回 `0`，出错时返回 `-1` 并且不设置异常（除非某个索引号不为 `None` 且无法被转换为整数，在这种情况下将返回 `-1` 并且设置一个异常）。

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)

🄵穩定 ABI 的一部分. *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🄵: 此函数对于可变大小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🄴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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.6.1.

Py_ssize_t **PySlice_AdjustIndices**(*Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* step)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.6.1.

Ellipsis 对象

PyObject ***Py_Ellipsis***

Python `Ellipsis` 对象。此没有没有任何方法。像*Py_None* 一样，它是一个 永久性对象 immortal。并且属于单例对象。

在 3.12 版的變更: *Py_Ellipsis* 🄵不滅的 (immortal)。

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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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.

Added in version 3.3.

PyObject ***PyMemoryView_FromBuffer***(const *Py_buffer* *view)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.11* 版本開始. 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：新的參照。🄵穩定 ABI 的一部分. 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'F'ortran *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 弱參照物件

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：新的參照。🄵穩定 ABI 的一部分. 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：新的參照。🄵穩定 ABI 的一部分. 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敌；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：借用參照。🄵穩定 ABI 的一部分. 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🄵： 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：借用參照。 类似于*`PyWeakref_GetObject()`*，但是不带错误检测。

void **PyObject_ClearWeakRefs**(*PyObject* *object)

🄵穩定 ABI 的一部分. 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。

Added in version 3.1.

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**

🄵穩定 ABI 的一部分. Capsule 的析构器回调的类型。定义如下：

```
typedef void (*PyCapsule_Destructor)(PyObject *);
```

參阅*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：新的參照。🄵穩定 ABI 的一部分. 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.attributename`。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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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 Frame 物件

type **PyFrameObject**

🄵受限 API 的一部分 （做🄵一個不透明結構 *(opaque struct)*）. 用來描述 frame 物件的 C 結構。

在這個結構中🄵有公開的成員。

在 3.11 版的變更: 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。

Added in version 3.9.

PyObject ***PyFrame_GetBuiltins**(*PyFrameObject* *frame)

取得 *frame* 的 f_builtins 屬性。

回傳*strong reference*。結果不能🄵 NULL。

Added in version 3.11.

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

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 获取 *frame* 的代码。

回傳*strong reference*。

结果（帧代码）不可为 NULL。

Added in version 3.9.

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`。

回傳*strong reference* 或 `NULL`。

Added in version 3.11.

PyObject ***PyFrame_GetGlobals**(*PyFrameObject* *frame)

取得 *frame* 的 `f_globals` 屬性。

回傳*strong reference*。結果不能🄵 `NULL`。

Added in version 3.11.

int **PyFrame_GetLasti**(*PyFrameObject* *frame)

取得 *frame* 的 `f_lasti` 屬性。

如果 `frame.f_lasti` 是 `None` 則回傳 -1。

Added in version 3.11.

PyObject ***PyFrame_GetVar**(*PyFrameObject* *frame, *PyObject* *name)

取得 *frame* 的變數 *name*。

- 在成功時回傳變數值的*strong reference*。
- 如果變數不存在，則引發 `NameError` 🄵回傳 `NULL`。
- 在錯誤時引發例外🄴回傳 `NULL`。

name 的型🄵必須是 `str`。

Added in version 3.12.

PyObject ***PyFrame_GetVarString**(*PyFrameObject* *frame, const char *name)

和*PyFrame_GetVar()* 相似，但该变量名是一个使用 UTF-8 编码的 C 字符串。

Added in version 3.12.

PyObject ***PyFrame_GetLocals**(*PyFrameObject* *frame)

获取 *frame* 的 `f_locals` 属性 (`dict`)。

回傳*strong reference*。

Added in version 3.11.

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

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 返回 *frame* 当前正在执行的行号。

内部帧

除非使用:pep:*523*，否则你不会需要它。

struct **_PyInterpreterFrame**

解释器的内部帧表示。

Added in version 3.11.

PyObject ***PyUnstable_InterpreterFrame_GetCode**(struct *_PyInterpreterFrame* *frame);

這是不穩定 *API*，它可能在小版本發布中🄴有任何警告地被變更。

返回一个指向帧的代码对象的*strong reference*。

Added in version 3.12.

int **PyUnstable_InterpreterFrame_GetLasti**(struct *_PyInterpreterFrame* *frame);

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

将字节偏移量返回到最后执行的指令中。

Added in version 3.12.

int **PyUnstable_InterpreterFrame_GetLine**(struct *_PyInterpreterFrame* *frame);

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

返回正在执行的指令的行数，如果没有行数，则返回-1。

Added in version 3.12.

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🄵 NULL。此函式總是會成功執行。

int **PyGen_CheckExact**(*PyObject* *ob)

如果 *ob* 的型🄵是*PyGen_Type* 則回傳真值；*ob* 必須不🄵 NULL。此函式總是會成功執行。

PyObject ***PyGen_New**(*PyFrameObject* *frame)

回傳值：新的參照。 基於 *frame* 物件建立🄵回傳一個新的🄵生器物件。此函式會取走一個對 *frame* 的參照 (reference)。引數必須不🄵 NULL。

PyObject ***PyGen_NewWithQualName**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

回傳值：新的參照。 基於 *frame* 物件建立🄵回傳一個新的🄵生器物件，其中 __name__ 和 __qualname__ 設🄵 *name* 和 *qualname*。此函式會取走一個對 *frame* 的參照。*frame* 引數必須不🄵 NULL。

8.6.11 Coroutine（協程）物件

Added in version 3.5.

Coroutine 物件是那些以 `async` 關鍵字來宣告的函式所回傳的物件。

type **PyCoroObject**

用於 coroutine 物件的 C 結構。

PyTypeObject **PyCoro_Type**

與 coroutine 物件對應的型🄵物件。

int **PyCoro_CheckExact**(*PyObject* *ob)

如果 *ob* 的型🄵是*PyCoro_Type* 則回傳真值；*ob* 必須不🄵 NULL。此函式總是會執行成功。

PyObject ***PyCoro_New**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

回傳值：新的參照。 基於 *frame* 物件來建立🄴回傳一個新的 coroutine 物件，其中 `__name__` 和 `__qualname__` 被設🄴 *name* 和 *qualname*。此函式會取得一個對 *frame* 的參照 (reference)。*frame* 引數必須不🄴 `NULL`。

8.6.12 上下文变量对象

Added in version 3.7.

在 3.7.1 版的變更:

備🄴: 在 Python 3.7.1 中，所有上下文变量 C API 的签名被 **更改** 为使用 *PyObject* 指针而不是 *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 模組提供各種日期和時間物件。在使用任何這些函式之前，必須將標頭檔 datetime.h 引入於原始碼中（請注意，Python.h 🄵無引入該標頭檔），且巨集 PyDateTime_IMPORT 必須被調用，而這通常作🄵模組初始化函式的一部分。該巨集將指向 C 結構的指標放入🄵態變數 PyDateTimeAPI 中，該變數會被以下巨集使用。

type **PyDateTime_Date**

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。

Added in version 3.7.

型🄵檢查巨集：

int **PyDate_Check**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_DateType* 或 PyDateTime_DateType 的子型🄵，則回傳 true。*ob* 不得🄵 NULL。這個函式一定會執行成功。

int **PyDate_CheckExact**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_DateType*，則回傳 true。*ob* 不得🄵 NULL。這個函式一定會執行成功。

int **PyDateTime_Check**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_DateTimeType* 或 PyDateTime_DateTimeType 的子型🄵，則回傳 true。*ob* 不得🄵 NULL。這個函式一定會執行成功。

int **PyDateTime_CheckExact**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_DateTimeType*，則回傳 true。*ob* 不得🄵 NULL。這個函式一定會執行成功。

int **PyTime_Check**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_TimeType* 或 PyDateTime_TimeType 的子型🄵，則回傳 true。*ob* 不得🄵 NULL。這個函式一定會執行成功。

int **PyTime_CheckExact**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_TimeType*，則回傳 true。*ob* 不得🄵 NULL。這個函式一定會執行成功。

int **PyDelta_Check**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_DeltaType* 或 PyDateTime_DeltaType 的子型🄵，則回傳 true。*ob* 不得🄵 NULL。這個函式一定會執行成功。

int **PyDelta_CheckExact**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_DeltaType*，則回傳 true。*ob* 不得🄵 NULL。這個函式一定會執行成功。

int **PyTZInfo_Check**(*PyObject* *ob)

如果 *ob* 的型🄵🄵*PyDateTime_TZInfoType* 或 PyDateTime_TZInfoType 的子型🄵，則回傳 true。*ob* 不得🄵 NULL。這個函式一定會執行成功。

int **PyTZInfo_CheckExact**(*PyObject* *ob)

如果 *ob* 的型FF*PyDateTime_TZInfoType*，則回傳 true。*ob* 不得F 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F）的物件 datetime.datetime。

Added in version 3.6.

PyObject ***PyTime_FromTime**(int hour, int minute, int second, int usecond)

回傳值：新的參照。 回傳一個有特定時、分、秒、微秒的物件 datetime.date。

PyObject ***PyTime_FromTimeAndFold**(int hour, int minute, int second, int usecond, int fold)

回傳值：新的參照。回傳一個有特定時、分、秒、微秒與 fold（時間折F）的物件 datetime.time。

Added in version 3.6.

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F的範圍F。

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。

Added in version 3.7.

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，F帶有 tzname *name*。

Added in version 3.7.

從 date 物件中提取欄位的巨集。引數必須是個*PyDateTime_Date* 的實例，包括子類F（例如*PyDateTime_DateTime*）。引數不得F NULL，F且不會檢查型F：

int **PyDateTime_GET_YEAR**(*PyDateTime_Date* *o)

回傳年份，F正整數。

int **PyDateTime_GET_MONTH**(*PyDateTime_Date* *o)

回傳月份，F正整數，從 1 到 12。

int **PyDateTime_GET_DAY**(*PyDateTime_Date* *o)

回傳日期，F正整數，從 1 到 31。

從 datetime 物件中提取欄位的巨集。引數必須是個*PyDateTime_DateTime* 的實例，包括子類F。引數不得F NULL，F且不會檢查型F：

int **PyDateTime_DATE_GET_HOUR**(*PyDateTime_DateTime* *o)

回傳小時，F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。

Added in version 3.6.

PyObject ***PyDateTime_DATE_GET_TZINFO**(*PyDateTime_DateTime* *o)
回傳 tzinfo（可能是 `None`）。

Added in version 3.10.

從 time 物件中提取欄位的巨集。引數必須是個*`PyDateTime_Time`* 的實例，包括子類🄵。引數不得🄵 `NULL`，🄵且不會檢查型🄵：

int **PyDateTime_TIME_GET_HOUR**(*PyDateTime_Time* *o)
回傳小時，🄵正整數，從 0 到 23。

int **PyDateTime_TIME_GET_MINUTE**(*PyDateTime_Time* *o)
回傳分鐘，🄵正整數，從 0 到 59。

int **PyDateTime_TIME_GET_SECOND**(*PyDateTime_Time* *o)
回傳秒，🄵正整數，從 0 到 59。

int **PyDateTime_TIME_GET_MICROSECOND**(*PyDateTime_Time* *o)
回傳微秒，🄵正整數，從 0 到 999999。

int **PyDateTime_TIME_GET_FOLD**(*PyDateTime_Time* *o)
回傳 fold，🄵 0 或 1 的正整數。

Added in version 3.6.

PyObject ***PyDateTime_TIME_GET_TZINFO**(*PyDateTime_Time* *o)
回傳 tzinfo（可能是 `None`）。

Added in version 3.10.

從 time delta 物件中提取欄位的巨集。引數必須是個*`PyDateTime_Delta`* 的實例，包括子類🄵。引數不能🄵 `NULL`，🄵且不會檢查型🄵：

int **PyDateTime_DELTA_GET_DAYS**(*PyDateTime_Delta* *o)
以 -999999999 到 999999999 之間的整數形式回傳天數。

Added in version 3.3.

int **PyDateTime_DELTA_GET_SECONDS**(*PyDateTime_Delta* *o)
以 0 到 86399 之間的整數形式回傳秒數。

Added in version 3.3.

int **PyDateTime_DELTA_GET_MICROSECONDS**(*PyDateTime_Delta* *o)
以 0 到 999999 之間的整數形式回傳微秒數。

Added in version 3.3.

🄵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🄵建型🄵。目前有兩種 -- GenericAlias 和 Union。只有 `GenericAlias` 有公開 (expose) 給 C。

PyObject ***Py_GenericAlias**(*PyObject* *origin, *PyObject* *args)

🄵穩定 ABI 的一部分 自 *3.9* 版本開始. 建立一個 GenericAlias 物件，等同於呼叫 Python 的 `types.GenericAlias` class。*origin* 和 *args* 引數分🄵設定了 `GenericAlias` 的 `__origin__` 與 `__args__` 屬性。*origin* 應該要是個 *PyTypeObject** 且 *args* 可以是個 *PyTupleObject** 或任意 `PyObject*`。如果傳入的 *args* 不是個 tuple（元組），則會自動建立一個長度🄵 1 的 tuple 且 `__args__` 會被設🄵 `(args,)`。只會進行最少的引數檢查，所以即便 *origin* 不是個型🄵，函式也會不會失敗。`GenericAlias` 的 `__parameters__` 屬性會自 `__args__` 惰性地建立 (constructed lazily)。當失敗時，會引發一個例外🄵回傳 `NULL`。

以下是個讓一個擴充型🄵泛用化 (generic) 的例子：

```
...
static PyMethodDef my_obj_methods[] = {
    // Other methods.
    ...
    {"__class_getitem__", Py_GenericAlias, METH_O|METH_CLASS, "See PEP 585"}
    ...
}
```

也參考:

資料模型方法 `__class_getitem__()`。

Added in version 3.9.

PyTypeObject **Py_GenericAliasType**

🄵穩定 ABI 的一部分 自 *3.9* 版本開始. *Py_GenericAlias()* 所回傳該物件的 C 型🄵。等價於 Python 中的 `types.GenericAlias`。

Added in version 3.9.

CHAPTER 9

初始化，终结和线程

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 🄵: 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**

此 API 仅为向下兼容而保留：应当改为设置*PyConfig.bytes_warning*，参见*Python 初始化配置*。

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` 選項設定。

在 3.12 版之後被🄵用.

int **Py_DebugFlag**

此 API 仅为向下兼容而保留：应当改为设置*PyConfig.parser_debug*，参见*Python 初始化配置*。

开启解析器调试输出（限专家使用，依赖于编译选项）。

由 `-d` 選項與 `PYTHONDEBUG` 環境變數設定。

在 3.12 版之後被🄵用.

int **Py_DontWriteBytecodeFlag**

此 API 仅为向下兼容而保留：应当改为设置*PyConfig.write_bytecode*，参见*Python 初始化配置*。

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` 選項與 `PYTHONDONTWRITEBYTECODE` 環境變數設定。

在 3.12 版之後被🄵用.

int **Py_FrozenFlag**

此 API 仅为向下兼容而保留：应当改为设置*PyConfig.pathconfig_warnings*，参见*Python 初始化配置*。

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。

在 3.12 版之後被🄵用.

int **Py_HashRandomizationFlag**

此 API 仅为向下兼容而保留：应当改为设置*PyConfig.hash_seed* 和*PyConfig.use_hash_seed*，参见*Python 初始化配置*。

如果環境變數 PYTHONHASHSEED 被設定🄵一個非空字串則設🄵 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。

在 3.12 版之後被🄵用.

int **Py_IgnoreEnvironmentFlag**

此 API 仅为向下兼容而保留：应当改为设置*PyConfig.use_environment*，参见*Python 初始化配置*。

忽略所有可能被設定的 PYTHON* 環境變數，例如 PYTHONPATH 與 PYTHONHOME。

由 -E 與 -I 選項設定。

在 3.12 版之後被🄵用.

int **Py_InspectFlag**

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

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。

在 3.12 版之後被🄵用.

int **Py_InteractiveFlag**

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

由 -i 選項設定。

在 3.12 版之後被🄵用.

int **Py_IsolatedFlag**

此 API 被保留用于向下兼容：应当改为设置*PyConfig.isolated*，参见*Python 初始化配置*。

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。

Added in version 3.4.

在 3.12 版之後被🄵用.

int **Py_LegacyWindowsFSEncodingFlag**

此 API 被保留用于向下兼容：应当改为设置*PyPreConfig.legacy_windows_fs_encoding*，参见*Python 初始化配置*。

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。

在 3.12 版之後被🄴用.

int **Py_LegacyWindowsStdioFlag**

此 API 被保留用于向下兼容：应当改为设置*PyConfig.legacy_windows_stdio*，参见*Python 初始化配置*。

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。

在 3.12 版之後被🄴用.

int **Py_NoSiteFlag**

此 API 被保留用于向下兼容：应当改为设置*PyConfig.site_import*，参见*Python 初始化配置*。

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。

在 3.12 版之後被🄴用.

int **Py_NoUserSiteDirectory**

此 API 被保留用于向下兼容：应当改为设置*PyConfig.user_site_directory*，参见*Python 初始化配置*。

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。

在 3.12 版之後被🄴用.

int **Py_OptimizeFlag**

此 API 被保留用于向下兼容：应当改为*PyConfig.optimization_level*，参见*Python 初始化配置*。

由 `-O` 選項與 `PYTHONOPTIMIZE` 環境變數設定。

在 3.12 版之後被🄴用.

int **Py_QuietFlag**

此 API 被保留用于向下兼容：应当改为设置*PyConfig.quiet*，参见*Python 初始化配置*。

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。

Added in version 3.2.

在 3.12 版之後被🄴用.

int **Py_UnbufferedStdioFlag**

此 API 被保留用于向下兼容：应当改为设置*PyConfig.buffered_stdio*，参见*Python 初始化配置*。

强制 stdout 和 stderr 流不带缓冲。

由 `-u` 選項與 `PYTHONUNBUFFERED` 環境變數設定。

在 3.12 版之後被🄴用.

int **Py_VerboseFlag**

此 API 被保留用于向下兼容：应当改为设置*PyConfig.verbose*，参见*Python 初始化配置*。

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。

在 3.12 版之後被🄵用.

9.3 初始化和最终化解释器

void **Py_Initialize**()

🄵穩定 ABI 的一部分. 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。

使用*Py_InitializeFromConfig()* 函数自定义*Python 初始化配置*。

備🄵: 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)

🄵穩定 ABI 的一部分. 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。

使用*Py_InitializeFromConfig()* 函数自定义*Python 初始化配置*。

int **Py_IsInitialized**()

🄵穩定 ABI 的一部分. 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**()

🄵穩定 ABI 的一部分 *自 3.6 版本開始.* 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。

Added in version 3.6.

void **Py_Finalize**()

🄵穩定 ABI 的一部分. 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）。

Added in version 3.4.

在 3.11 版之後被🄵用.

void **Py_SetProgramName**(const wchar_t *name)

🄵穩定 ABI 的一部分. 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()* 对字节串进行解码以得到一个 `wchar_t*` 字符串。

在 3.11 版之後被🄵用.

wchar_t ***Py_GetProgramName**()

🄵穩定 ABI 的一部分. 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。

此函式不應該在*Py_Initialize()* 之前呼叫，否則會回傳 `NULL`。

在 3.10 版的變更: 如果在*Py_Initialize()* 之前呼叫，現在會回傳 `NULL`。

wchar_t ***Py_GetPrefix**()

🄵穩定 ABI 的一部分. 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。

此函式不應該在*Py_Initialize()* 之前呼叫，否則會回傳 `NULL`。

在 3.10 版的變更: 如果在*Py_Initialize()* 之前呼叫，現在會回傳 `NULL`。

wchar_t ***Py_GetExecPrefix**()

🄵穩定 ABI 的一部分. 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**

變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。

此函式不應該在*Py_Initialize()* 之前呼叫，否則會回傳 `NULL`。

在 3.10 版的變更: 如果在*Py_Initialize()* 之前呼叫，現在會回傳 `NULL`。

wchar_t ***Py_GetProgramFullPath**()

🄵穩定 ABI 的一部分. 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。

此函式不應該在*Py_Initialize()* 之前呼叫，否則會回傳 `NULL`。

在 3.10 版的變更: 如果在*Py_Initialize()* 之前呼叫，現在會回傳 `NULL`。

wchar_t ***Py_GetPath**()

🄵穩定 ABI 的一部分. 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。

此函式不應該在*Py_Initialize()* 之前呼叫，否則會回傳 `NULL`。

在 3.10 版的變更: 如果在*Py_Initialize()* 之前呼叫，現在會回傳 `NULL`。

void **Py_SetPath**(const wchar_t*)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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_*` 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🄵用.

const char ***Py_GetVersion**()

🄵穩定 ABI 的一部分. 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**()

🄴稳定 ABI 的一部分. 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**()

🄴稳定 ABI 的一部分. 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**()

🄴稳定 ABI 的一部分. 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_GetBuildInfo**()

🄴稳定 ABI 的一部分. 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)

🄴稳定 ABI 的一部分. 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_*` 字符串。

另请参阅*Python 初始化配置* 的*PyConfig.orig_argv* 和*PyConfig.argv* 成员。

備🄴: 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");
```

Added in version 3.1.3.

在 3.11 版之後被🄵用.

void **PySys_SetArgv**(int argc, wchar_t **argv)

🄵穩定 ABI 的一部分. 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_*` 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)

🄵穩定 ABI 的一部分. 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_*` 字符串。

在 3.11 版之後被🄵用.

wchar_t ***Py_GetPythonHome**()

🄵穩定 ABI 的一部分. 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。

此函式不應該在*Py_Initialize()* 之前呼叫，否則會回傳 `NULL`。

在 3.10 版的變更: 如果在*Py_Initialize()* 之前呼叫，現在會回傳 `NULL`。

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🄴: 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（这通常会是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**

🄵受限 API 的一部分 （做🄵一個不透明結構 *(opaque struct)*）. 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**

🄵受限 API 的一部分 （做🄵一個不透明結構 *(opaque struct)*）. 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**()

🄵穩定 ABI 的一部分. 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🄵用.

int **PyEval_ThreadsInitialized**()

🄵穩定 ABI 的一部分. 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🄵用.

PyThreadState ***PyEval_SaveThread**()

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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🄵: 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**()

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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**()

🄵穩定 ABI 的一部分. 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🄵: 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)

🄵穩定 ABI 的一部分. 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**()

🄵穩定 ABI 的一部分. 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。

Added in version 3.4.

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

🄵穩定 ABI 的一部分. 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

🄵穩定 ABI 的一部分. 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

🄵穩定 ABI 的一部分. 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

🄵穩定 ABI 的一部分. 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**()

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 *自 3.10 版本開始.* 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。

Added in version 3.9.

uint64_t **PyThreadState_GetID**(*PyThreadState* *tstate)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 获取 Python 线程状态 *tstate* 的唯一线程状态标识符。

tstate 不可🄵 NULL。

Added in version 3.9.

PyInterpreterState ***PyThreadState_GetInterpreter**(*PyThreadState* *tstate)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 获取 Python 线程状态 *tstate* 对应的解释器。

tstate 不可🄵 NULL。

Added in version 3.9.

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。

Added in version 3.11.

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。

Added in version 3.11.

PyInterpreterState ***PyInterpreterState_Get**(void)

🄵穩定 ABI 的一部分 自 *3.9* 版本開始. 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。

呼叫者必须持有 GIL。

Added in version 3.9.

int64_t **PyInterpreterState_GetID**(*PyInterpreterState* *interp)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。

呼叫者必须持有 GIL。

Added in version 3.7.

PyObject ***PyInterpreterState_GetDict**(*PyInterpreterState* *interp)

🄵穩定 ABI 的一部分 自 *3.8* 版本開始. 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。

Added in version 3.8.

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"。

Added in version 3.9.

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"。

Added in version 3.9.

PyObject ***PyThreadState_GetDict**()

回傳值：借用參照。🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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）。这将不会引发异常。

在 3.7 版的變更: *id* 形参的类型已从 long 变为 unsigned long。

void **PyEval_AcquireThread**(*PyThreadState* *tstate)

🄵穩定 ABI 的一部分. 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🄵: 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 版的變更: 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)

🄵穩定 ABI 的一部分. 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**()

🄵穩定 ABI 的一部分. 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🄵用: 此函数不会更新当前线程状态。请改用*PyEval_RestoreThread()* 或*PyEval_AcquireThread()*。

備🄵: 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 版的變更: 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**()

🄵穩定 ABI 的一部分. 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:

type **PyInterpreterConfig**

包含用于配置子解释器的大部分形参的结构体。其值仅在*Py_NewInterpreterFromConfig()* 中被使用而绝不会被运行时所修改。

Added in version 3.12.

结构体字段:

int **use_main_obmalloc**

如果该值为 0 则子解释器将使用自己的“对象”分配器状态。否则它将使用（共享）主解释器的状态。

如果该值为 0 则*check_multi_interp_extensions* 必须为 1 (非零值)。如果该值为 1 则*gil* 不可为*PyInterpreterConfig_OWN_GIL*。

int **allow_fork**

如果该值为 0 则运行时将不支持在当前激活了子解释器的任何线程中 fork 进程。否则 fork 将不受限制。

请注意当 fork 被禁止时 subprocess 模块将仍然可用。

int **allow_exec**

如果该值为 0 则运行时将不支持在当前激活了子解释器的任何线程中通过 exec (例如 os.execv()) 替换当前进程。否则 exec 将不受限制。

请注意当 exec 被禁止时 subprocess 模块将仍然可用。

int **allow_threads**

如果该值为 0 则子解释器的 threading 模块将不会创建线程。否则线程将被允许。

int **allow_daemon_threads**

如果该值为 0 则子解释器的 threading 模块将不会创建守护线程。否则将允许守护线程（只要*allow_threads* 是非零值）。

int **check_multi_interp_extensions**

如果该值为 0 则所有扩展模块均可在当前子解释器被激活的任何线程中被导入，包括旧式的 (单阶段初始化) 模块。否则将只有多阶段初始化扩展模块 (参见 **PEP 489**) 可以被导入。(另请参阅*Py_mod_multiple_interpreters*。)

如果*use_main_obmalloc* 为 0 则该值必须为 1 (非零值)。

int **gil**

这将确定针对子解释器的 GIL 操作方式。它可以是以下的几种之一:

PyInterpreterConfig_DEFAULT_GIL

使用默认选择 (*PyInterpreterConfig_SHARED_GIL*)。

PyInterpreterConfig_SHARED_GIL

使用（共享）主解释器的 GIL。

PyInterpreterConfig_OWN_GIL

使用子解释器自己的 GIL。

如果该值为*PyInterpreterConfig_OWN_GIL* 则*PyInterpreterConfig.use_main_obmalloc* 必须为 0。

PyStatus **Py_NewInterpreterFromConfig**(*PyThreadState* **tstate_p, const *PyInterpreterConfig* *config)

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)。

给定的 *config* 控制着初始化解释器所使用的选项。

成功后，*tstate_p* 将被设为新的子解释器中创建的第一个线程状态。该线程状态是在当前线程状态中创建的。请注意并没有真实的线程被创建；请参阅下文有关线程状态的讨论。如果创建新的解释器没有成功，则 *tstate_p* 将被设为 `NULL`；不会设置任何异常因为异常状态是存储在当前的线程状态中而当前线程状态并不一定存在。

与所有其他 Python/C API 函数一样，在调用此函数之前必须先持有全局解释器锁并且在其返回时仍继续持有。同样地在进入函数时也必须设置当前线程状态。执行成功后，返回的线程状态将被设为当前线程状态。如果创建的子解释器具有自己的 GIL 那么调用方解释器的 GIL 将被释放。当此函数返回时，新的解释器的 GIL 将由当前线程持有而之前的解释器的 GIL 在此将保持释放状态。

Added in version 3.12.

子解释器在彼此相互隔离，并让特定功能受限的情况下是最有效率的:

```
PyInterpreterConfig config = {
    .use_main_obmalloc = 0,
    .allow_fork = 0,
    .allow_exec = 0,
    .allow_threads = 1,
    .allow_daemon_threads = 0,
    .check_multi_interp_extensions = 1,
    .gil = PyInterpreterConfig_OWN_GIL,
};
PyThreadState *tstate = Py_NewInterpreterFromConfig(&config);
```

请注意该配置只会被短暂使用而不会被修改。在初始化期间配置的值会被转换成各种*PyInterpreterState* 值。配置的只读副本可以被内部存储于*PyInterpreterState* 中。

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。

PyThreadState ***Py_NewInterpreter**(void)

🄵穩定 ABI 的一部分. 新建一个子解释器。这在本质上只是针对*Py_NewInterpreterFromConfig()* 的包装器，其配置保留了现有的行为。结果是一个未隔离的子解释器，它会共享主解释器的 GIL，允许 fork/exec，允许守护线程，也允许单阶段初始化模块。

void **Py_EndInterpreter**(*PyThreadState* *tstate)

🄵穩定 ABI 的一部分. 销毁由给定的线程状态所代表的（子）解释器。给定的线程状态必须为当前的线程状态。请参阅下文中关于线程状态的讨论。当调用返回时，当前的线程状态将为 NULL。与此解释器相关联的所有线程状态都会被销毁。在调用此函数之前必须持有目标解释器所使用的全局解释器锁。当其返回时将不再持有 GIL。

Py_FinalizeEx() 将销毁所有在当前时间点上尚未被明确销毁的子解释器。

9.6.1 解释器级 GIL

使用*Py_NewInterpreterFromConfig()* 你将可以创建一个与其他解释器完全隔离的子解释器，包括具有自己的 GIL。这种隔离带来的最大好处在于这样的解释器执行 Python 代码时不会被其他解释器所阻塞或者阻塞任何其他解释器。因此在运行 Python 代码时单个 Python 进程可以真正地利用多个 CPU 核心。这种隔离还能鼓励开发者采取不同于仅使用线程的并发方式。(参见 **PEP 554**)。

使用隔离的解释器要求谨慎地保持隔离状态。尤其是意味着不要在未确保线程安全的情况下共享任何对象或可变的状态。由于引用计数的存在即使是在其他情况下不可变的对象 (例如 None, (1, 5)) 通常也不可被共享。针对此问题的一种简单但效率较低的解决方式是在使用某些状态 (或对象) 时总是使用一个全局锁。或者，对于实际上不可变的对象 (如整数或字符串) 可以通过将其设为" 永久" 对象而无视其引用计数来确保其安全。事实上，对于内置单例、小整数和其他一些内置对象都是这样做的。

如果你能保持隔离状态那么你将能获得真正的多核计算能力而不会遇到自由线程所带来的复杂性。如果未能保持隔离状态那么你将面对自由线程所带来的全部后果，包括线程竞争和难以调试的崩溃。

除此之外，使用多个相互隔离的解释器的一个主要挑战是如何在它们之间安全 (不破坏隔离状态)、高效地进行通信。运行时和标准库还没有为此提供任何标准方式。未来的标准库模块将会帮助减少保持隔离状态所需的工作量并为解释器之间的数据通信（和共享）公开有效的工具。

Added in version 3.12.

9.6.2 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题。

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)

🄵稳定 ABI 的一部分. 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*。

Added in version 3.1.

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：

what 的值	*arg* 的含义
PyTrace_CALL	总是*Py_None*.
PyTrace_EXCEPTION	`sys.exc_info()` 返回的异常信息。
PyTrace_LINE	总是*Py_None*.
PyTrace_RETURN	返回给调用方的值，或者如果是由异常导致的则返回 `NULL`。
PyTrace_C_CALL	正在调用函数对象。
PyTrace_C_EXCEPTION	正在调用函数对象。
PyTrace_C_RETURN	正在调用函数对象。
PyTrace_OPCODE	总是*Py_None*.

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_SetProfileAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

类似于*PyEval_SetProfile()* 但会在属于当前解释器的所有在运行线程中设置性能分析函数而不是仅在当前线程上设置。

呼叫者必须持有*GIL*。

与*PyEval_SetProfile()* 一样，该函数会忽略任何被引发的异常同时在所有线程中设置性能分析函数。

Added in version 3.12.

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*。

void **PyEval_SetTraceAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

类似于`PyEval_SetTrace()` 但会在属于当前解释器的所有在运行线程中设置跟踪函数而不是仅在当前线程上设置。

呼叫者必须持有*GIL*。

与`PyEval_SetTrace()` 一样，该函数会忽略任何被引发的异常同时在所有线程中设置跟踪函数。

Added in version 3.12.

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)

从由属于同一个`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 `pthread.h` 来使用线程本地存储。

備🄴： 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。

Added in version 3.7.

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**()

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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🄵: 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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🄵: 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**()

🄵穩定 ABI 的一部分.

void **PyThread_delete_key**(int key)

🄵穩定 ABI 的一部分.

int **PyThread_set_key_value**(int key, void *value)

🄵穩定 ABI 的一部分.

void ***PyThread_get_key_value**(int key)

🄵穩定 ABI 的一部分.

void **PyThread_delete_key_value**(int key)

🄵穩定 ABI 的一部分.

void **PyThread_ReInitTLS**()

🄵穩定 ABI 的一部分.

CHAPTER 10

Python 初始化配置

Added in version 3.8.

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句柄。

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)

(繼續上一頁)

```
    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🄵 `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🄴: 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.allocator*)
- 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🄵 `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🄵 `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🄵 `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.allocator* 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再更新直到*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**

命令列引數：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。

預設值：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。

Added in version 3.11.

wchar_t ***base_exec_prefix**

sys.base_exec_prefix。

預設值：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 __PYVENV_LAUNCHER__ 环境变量设置。

如为 NULL 则从*PyConfig.executable* 设置。

預設值：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。

Added in version 3.10.

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。

Added in version 3.11.

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。

預設：`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`。

預設值：`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`。

預設值：`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**

文件系统错误处理句柄: 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"。

支持的错误处理句柄：

- "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。

預設值：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。

預設：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。

預設：0。

int **install_signal_handlers**

安装 Python 信号处理句柄?

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。

預設：0。

int **int_max_str_digits**

配置 整数字符串转换长度限制。初始值为 -1 表示该值将从命令行或环境获取否则默认为 4300 (sys.int_info.default_max_str_digits)。值为 0 表示禁用限制。大于 0 但小于 640 (sys.int_info.str_digits_check_threshold) 的值将不被支持并会产生错误。

通过 -X int_max_str_digits 命令行旗标或 PYTHONINTMAXSTRDIGITS 环境变量配置。

默认值: 在 Python 模式下为 -1。在孤立模式下为 4300 (sys.int_info.default_max_str_digits)。

Added in version 3.12.

int **isolated**

如果大于 0，则启用隔离模式:

- 将*safe_path* 设为 1: 在 Python 启动时将不在 sys.path 前添加有潜在不安全性的路径，如当前目录、脚本所在目录或空字符串。
- 將*use_environment* 設定🄵 0：忽略 PYTHON 環境變數。
- 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。

另请参阅隔离配置 和*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。

Added in version 3.9.

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。

預設：`0`。

PyWideStringList **orig_argv**

传给 Python 可执行程序的原始命令行参数列表: `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。

Added in version 3.10.

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 版的變更: 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。

需要 Python 调试编译版 (必须定义 `Py_DEBUG` 宏)。

預設：`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 版的變更: 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 __PYVENV_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**

在退出时显示总引用计数（不包括永生对象）?

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" 错误处理句柄）。

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*。

默认错误处理句柄：

- 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 **perf_profiling**

启用与 perf 性能分析器兼容的模式?

如果为非零值，则初始化 perf trampoline。更多信息参见 perf_profiling。

通过 -X perf 命令行选项和 PYTHONPERFSUPPORT 环境变量设置。

預設值：1。

Added in version 3.12.

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）。

如果大于等于 2，则为搜索模块时每个被检查的文件打印一条消息。还在退出时提供关于模块清理的信息。

由 -v 命令行选项执行递增。

通过 PYTHONVERBOSE 环境变量值设置。

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);
    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 隔离配置

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。信号处理句柄将不会被安装。

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*)
  - `__PYVENV_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。

__PYVENV_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；
  - 安装信号处理句柄；
  - 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。

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。

在核心阶段之后主要阶段之前，将有可能在 Python 中计算路径配置，这是 **PEP 432** 的动机之一。

“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 ... */
```

(繼續下一頁)

(繼續上一頁)

```
    status = Py_InitializeFromConfig(&config);
    PyConfig_Clear(&config);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }

    /* Use sys.stderr because sys.stdout is only created
       by _Py_InitializeMain() */
    int res = PyRun_SimpleString(
        "import sys; "
        "print('Run Python code before _Py_InitializeMain', "
               "file=sys.stderr)");
    if (res < 0) {
        exit(1);
    }

    /* ... put more configuration code here ... */

    status = _Py_InitializeMain();
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
}
```

CHAPTER 11

記憶體管理

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就必须这样做。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)`)。

Added in version 3.4.

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。

Added in version 3.5.

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)

🄵稳定 ABI 的一部分. 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)

🄵稳定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.5.

void ***PyMem_Realloc**(void *p, size_t n)

🄵稳定 ABI 的一部分. 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)

🄵稳定 ABI 的一部分. 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 * sizeof(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 * sizeof(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🄵: 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.7* 版本開始. 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。

Added in version 3.5.

void ***PyObject_Realloc**(void *p, size_t n)

🄵穩定 ABI 的一部分. 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)

🄵稳定 ABI 的一部分. 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	PyObject_Malloc
发布版本	"pymalloc"	malloc	pymalloc	pymalloc
调试构建	"pymalloc_debug	malloc + debug	pymalloc + debug	pymalloc + debug
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

Added in version 3.4.

type **PyMemAllocatorEx**

用于描述内存块分配器的结构体。该结构体下列字段：

欄位	意義
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*。

对于其余的域，分配器也必须是线程安全的：分配器可以在不共享 GIL 的不同解释器中被调用。

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* 来了解详情。
> - 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**。

在 3.12 版的變更: 所有分配器都必须是线程安全的。

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

Runtime 檢查：

- 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* = sizeof(size_t)。将 2*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

Added in version 3.4.

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:

欄位	意義
`void *ctx`	作为第一个参数传入的用户上下文
`void* alloc(void *ctx, size_t size)`	分配一块 size 字节的区域
`void free(void *ctx, void *ptr, size_t size)`	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

Added in version 3.7.

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的相同代码:

```
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。

CHAPTER 12

对象实现支持

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。🄵穩定 ABI 的一部分. 用它的型🄵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。🄵穩定 ABI 的一部分. 它會做到`PyObject_Init()`的所有功能，🄵且會初始化一個大小可變物件的長度資訊。

PyObject_New(TYPE, typeobj)

使用 C 結構型🄵 *TYPE* 和 Python 型🄵物件 *typeobj* (`PyTypeObject*`) 分配一個新的 Python 物件。未在該 Python 物件標頭 (header) 中定義的欄位不會被初始化；呼叫者會擁有那個對於物件的唯一參照（物件的參照計數🄵一）。記憶體分配大小由 type 物件的`tp_basicsize`欄位來指定。

PyObject_NewVar(TYPE, typeobj, size)

使用 C 的結構型🄵 *TYPE* 和 Python 的型🄵物件 *typeobj* (`PyTypeObject*`) 分配一個新的 Python 物件。未在該 Python 物件標頭中定義的欄位不會被初始化。記憶體空間預留了 *TYPE* 結構大小再加上 *typeobj* 物件中`tp_itemsize`欄位提供的 *size* (`Py_ssize_t`) 欄位的值。這對於實現如 tuple 這種能🄵在建立期間🄵定自己大小的物件是很實用的。將欄位的陣列嵌入到相同的記憶體分配中可以🄵少記憶體分配的次數，這提高了記憶體管理的效率。

void **PyObject_Del**(void *op)

釋放由`PyObject_New`或者`PyObject_NewVar`分配給物件的記憶體。這通常是在物件型🄵所指定的`tp_dealloc` handler 中呼叫。呼叫這個函式以後，物件的各欄位都不可以被存取，因🄵原本分配的記憶體已不再是一個有效的 Python 物件。

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

大量的结构体被用于定义 Python 的对象类型。这一节描述了这些的结构体和它们的使用方法。

12.2.1 基本的对象类型和宏

所有的 Python 对象最终都会在对象的内存表示的开始部分共享少量的字段。这些字段由*PyObject* 和*PyVarObject* 类型来表示，相应地，这些类型又是由一些宏扩展来定义的，它们也直接或间接地被用于所有其他 Python 对象的定义。附加的宏可以在引用计数 下找到。

type **PyObject**

🄵受限 API 的一部分. （只有部分成員是穩定 *ABI* 的一部分。） 所有对象类型都是此类型的扩展。这是一个包含了 Python 将对象的指针当作对象来处理所需的信息的类型。在一个普通的“发行”编译版中，它只包含对象的引用计数和指向对应类型对象的指针。没有什么对象被实际声明为*PyObject*，但每个指向 Python 对象的指针都可以被转换为 *PyObject**。对成员的访问必须通过使用*Py_REFCNT* 和*Py_TYPE* 宏来完成。

type **PyVarObject**

🄵受限 API 的一部分. （只有部分成員是穩定 *ABI* 的一部分。） 这是一个添加了*ob_size* 字段的*PyObject* 扩展。它仅用于具有某些 长度标记的对象。此类型并不经常在 Python/C API 中出现。对成员的访问必须通过使用*Py_REFCNT*, *Py_TYPE* 和*Py_SIZE* 宏来完成。

PyObject_HEAD

这是一个在声明代表无可变长度对象的新类型时所使用的宏。PyObject_HEAD 宏被扩展为:

```
PyObject ob_base;
```

参见上面*PyObject* 的文档。

PyObject_VAR_HEAD

这是一个在声明代表每个实例具有可变长度的对象时所使用的宏。PyObject_VAR_HEAD 宏被扩展为:

```
PyVarObject ob_base;
```

請見上面*PyVarObject* 的文件。

int **Py_Is**(*PyObject* *x, *PyObject* *y)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 测试 *x* 是否为 *y* 对象，与 Python 中的 x is y 相同。

Added in version 3.10.

int **Py_IsNone**(*PyObject* *x)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 测试一个对象是否为 None 单例，与 Python 中的 x is None 相同。

Added in version 3.10.

int **Py_IsTrue**(*PyObject* *x)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 测试一个对象是否为 `True` 单例，与 Python 中的 `x is True` 相同。

Added in version 3.10.

int **Py_IsFalse**(*PyObject* *x)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 测试一个对象是否为 `False` 单例，与 Python 中的 `x is False` 相同。

Added in version 3.10.

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`。

Added in version 3.9.

void **Py_SET_TYPE**(*PyObject* *o, *PyTypeObject* *type)

將物件 *o* 的型🄵設🄵 *type*。

Added in version 3.9.

Py_ssize_t **Py_SIZE**(*PyVarObject* *o)

取得 Python 物件 *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)

將物件 *o* 的大小設🄵 *size*。

Added in version 3.9.

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 實作函式與方法

type **PyCFunction**

🄵穩定 ABI 的一部分. 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**

🄵穩定 ABI 的一部分. 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 **PyCMethod**

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 *PyCMethod(PyObject *self,
                    PyTypeObject *defining_class,
                    PyObject *const *args,
                    Py_ssize_t nargs,
                    PyObject *kwnames)
```

Added in version 3.9.

type **PyMethodDef**

🄵穩定 ABI 的一部分 （包含所有成員）. 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 ***ml_name**

方法的名称。

PyCFunction **ml_meth**

指向 C 语言实现的指针。

int **ml_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)。

Added in version 3.7.

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。

Added in version 3.7.

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。

Added in version 3.9.

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为 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 ***PyCMethod_New**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module, *PyTypeObject* *cls)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.9* 版本開始. 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`。

也參考:

`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。

Added in version 3.9.

PyObject ***PyCFunction_NewEx**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module)

回傳值：新的參照。🄵穩定 ABI 的一部分. 等價於 `PyCMethod_New(ml, self, module, NULL)`。

PyObject ***PyCFunction_New**(*PyMethodDef* *ml, *PyObject* *self)

回傳值：新的參照。🄵穩定 ABI 的一部分 自 *3.4* 版本開始. 等價於 `PyCMethod_New(ml, self, NULL, NULL)`。

12.2.3 访问扩展类型的属性

type **PyMemberDef**

🄵穩定 ABI 的一部分 （包含所有成員）. 描述某个 C 结构成员对应类型的属性的结构体。在定义类时，要把由这些结构组成的以 NULL 结尾的数组放在*tp_members* 槽位中。

其中的字段及顺序如下:

const char ***name**

成员名称。NULL 值表示 `PyMemberDef[]` 数组的结束。

字符串应当是静态的，它不会被复制。

int **type**

C 结构体中成员的类型。请参阅成员类型 了解可能的取值。

Py_ssize_t **offset**

成员在类型的对象结构体中所在位置的以字节为单位的偏移量。

int **flags**

零个或多个成员旗标，使用按位或运算进行组合。

const char ***doc**

文档字符串，或者为空。该字符串应当是静态的，它不会被拷贝。通常，它是使用*PyDoc_STR* 来定义的。

默认情况下 (当*flags* 为 0 时)，成员同时允许读取和写入访问。使用*Py_READONLY* 旗标表示只读访问。某些类型，如*Py_T_STRING*，隐含要求*Py_READONLY*。只有*Py_T_OBJECT_EX* (以及旧式的*T_OBJECT*) 成员可以删除。

对于堆分配类型（使用*PyType_FromSpec()* 或类似函数创建），PyMemberDef 可能包含特殊成员 "__vectorcalloffset__" 的定义，与类型对象中的*tp_vectorcall_offset* 相对应。它们必须用 Py_T_PYSSIZET 和 Py_READONLY 来定义，例如:

```
static PyMemberDef spam_type_members[] = {
    {"__vectorcalloffset__", Py_T_PYSSIZET,
     offsetof(Spam_object, vectorcall), Py_READONLY},
    {NULL}  /* Sentinel */
};
```

（您可能需要为 offsetof() 添加 #include <stddef.h>。）

旧式的偏移量*tp_dictoffset* 和*tp_weaklistoffset* 可使用 "__dictoffset__" 和 "__weaklistoffset__" 成员进行类似的定义，但强烈建议扩展程序改用*Py_TPFLAGS_MANAGED_DICT* 和*Py_TPFLAGS_MANAGED_WEAKREF*。

在 3.12 版的變更: PyMemberDef 将始终可用。在之前版本中，它需要包括 "structmember.h"。

PyObject ***PyMember_GetOne**(const char *obj_addr, struct *PyMemberDef* *m)

🄵穩定 ABI 的一部分. 获取属于地址 Get an attribute belonging to the object at address *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。

在 3.12 版的變更: PyMember_GetOne 将总是可用。在之前版本中，它需要包括 "structmember.h"。

int **PyMember_SetOne**(char *obj_addr, struct *PyMemberDef* *m, *PyObject* *o)

🄵穩定 ABI 的一部分. 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。

在 3.12 版的變更: PyMember_SetOne 将总是可用。在之前版本中，它需要包括 "structmember.h"。

成员旗标

以下旗标可被用于*PyMemberDef.flags*:

Py_READONLY

不可寫入。

Py_AUDIT_READ

在读取之前发出一个 object.__getattr__ 审计事件。

Py_RELATIVE_OFFSET

表示该 PyMemberDef 条目的*offset* 是指明来自子类专属数据的偏移量，而不是来自 PyObject 的偏移量。

只能在使用负的*basicsize* 创建类时被用作*Py_tp_members 槽位* 的组成部分。它在此种情况下是强制要求。

这个旗标只能在*PyType_Slot* 中使用。在类创建期间设置*tp_members* 时，Python 会清除它并将*PyMemberDef.offset* 设为相对于 PyObject 结构体的偏移量。

在 3.10 版的變更: 通过 #include "structmember.h" 提供的 RESTRICTED、READ_RESTRICTED 和 WRITE_RESTRICTED 宏已被弃用。READ_RESTRICTED 和 RESTRICTED 等同于*Py_AUDIT_READ*; WRITE_RESTRICTED 则没有任何作用。

在 3.12 版的變更: READONLY 宏被更名为*Py_READONLY*。PY_AUDIT_READ 宏被更名为 Py_ 前缀。新名称现在将始终可用。在之前的版本中，这些名称需要 #include "structmember.h"。该头文件仍然可用并提供了原有的名称。

成员类型

PyMemberDef.type 可以是下列与各种 C 类型相对应的宏之一。在 Python 中访问该成员时，它将被转换为对应的 Python 类型。当从 Python 设置成员时，它将被转换回 C 类型。如果无法转换，则会引发一个异常如 TypeError 或 ValueError。

除非标记为 (D)，否则不能使用 del 或 delattr() 删除以这种方式定义的属性。

巨集名黌	C 类型	Python 类型
Py_T_BYTE	char	int
Py_T_SHORT	short	int
Py_T_INT	int	int
Py_T_LONG	long	int
Py_T_LONGLONG	long long	int
Py_T_UBYTE	unsigned char	int
Py_T_UINT	unsigned int	int
Py_T_USHORT	unsigned short	int
Py_T_ULONG	unsigned long	int
Py_T_ULONGLONG	unsigned long long	int
Py_T_PYSSIZET	*Py_ssize_t*	int
Py_T_FLOAT	float	float
Py_T_DOUBLE	double	float
Py_T_BOOL	char （寫成 0 或 1）	bool
Py_T_STRING	const char* (*)	str (RO)
Py_T_STRING_INPLACE	const char[] (*)	str (RO)
Py_T_CHAR	char (0-127)	str (**)
Py_T_OBJECT_EX	*PyObject**	object (D)

(*): 以零结束的 UTF8 编码的 C 字符串。使用 Py_T_STRING 时的 C 表示形式是一个指针；使用 Py_T_STRING_INPLACE 时字符串将直接存储在结构体中。

(**): 长度为 1 的字符串。只接受 ASCII 字符。

(RO): 表示*Py_READONLY*。

(D): 可以删除，在这种情况下指针会被设为 NULL。读取 NULL 指针会引发 AttributeError。

Added in version 3.12: 在之前的版本中，这些宏仅通过 #include "structmember.h" 提供并且其名称不带 Py_ 前缀 (例如 T_INT)。头文件仍然可用并包含这些旧名称，以及下列已被弃用的类型：

T_OBJECT

与 Py_T_OBJECT_EX 类似，但 NULL 会被转换为 None。这将在 Python 中产生令人吃惊的行为：删除该属性实际上会将其设置为 None。

T_NONE

总是为 None。必须与*Py_READONLY* 一起使用。

定义读取器和设置器

type **PyGetSetDef**

🄵穩定 ABI 的一部分 （包含所有成員）. 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。

const char ***name**

屬性名稱

getter **get**

用于获取属性的 C 函数。

setter **set**

可选的用于设置或删除属性的 C 函数。如为 NULL，则属性将是只读的。

const char ***doc**

可選的文件字串

void ***closure**

可选的用户数据指针，为 getter 和 setter 提供附加数据。

typedef *PyObject* *(***getter**)(*PyObject**, void*)

🄵穩定 ABI 的一部分. get 函数接受一个 *PyObject** 形参 (相应的实例) 和一个用户数据指针 (关联的 closure):

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。

typedef int (***setter**)(*PyObject**, *PyObject**, void*)

🄵穩定 ABI 的一部分. set 函数接受两个 *PyObject** 形参 (相应的实例和要设置的值) 和一个用户数据指针 (关联的 closure):

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个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_ 方法槽"

PyTypeObject 槽[Page 246, 1]	*Type*	特殊方法/属性	信息[Page 246, 2] C	T	D	I
<R> *tp_name*	const char *	__name__	X	X		
tp_basicsize	*Py_ssize_t*		X	X		X
tp_itemsize	*Py_ssize_t*			X		X
tp_dealloc	*destructor*		X	X		X
tp_vectorcall_offset	*Py_ssize_t*			X		X
(*tp_getattr*)	*getattrfunc*	__getattribute__, __getattr__				G
(*tp_setattr*)	*setattrfunc*	__setattr__, __delattr__				G
tp_as_async	*PyAsyncMethods* *	子方法槽（方法域）				%
tp_repr	*reprfunc*	__repr__	X	X		X
tp_as_number	*PyNumberMethods* *	子方法槽（方法域）				%
tp_as_sequence	*PySequenceMethods* *	子方法槽（方法域）				%
tp_as_mapping	*PyMappingMethods* *	子方法槽（方法域）				%
tp_hash	*hashfunc*	__hash__	X			G
tp_call	*ternaryfunc*	__call__		X		X
tp_str	*reprfunc*	__str__	X			X
tp_getattro	*getattrofunc*	__getattribute__, __getattr__	X	X		G
tp_setattro	*setattrofunc*	__setattr__, __delattr__	X	X		G
tp_as_buffer	*PyBufferProcs* *					%
tp_flags	unsigned long		X	X		?
tp_doc	const char *	__doc__	X	X		
tp_traverse	*traverseproc*			X		G
tp_clear	*inquiry*			X		G
tp_richcompare	*richcmpfunc*	__lt__, __le__, __eq__, __ne__, __gt__, __ge__	X			G
(*tp_weaklistoffset*)	*Py_ssize_t*			X		?
tp_iter	*getiterfunc*	__iter__				X
tp_iternext	*iternextfunc*	__next__				X
tp_methods	*PyMethodDef* []		X	X		
tp_members	*PyMemberDef* []			X		
tp_getset	*PyGetSetDef* []		X	X		
tp_base	*PyTypeObject* *	__base__			X	
tp_dict	*PyObject* *	__dict__			?	
tp_descr_get	*descrgetfunc*	__get__				X
tp_descr_set	*descrsetfunc*	__set__, __delete__				X
(*tp_dictoffset*)	*Py_ssize_t*			X		?
tp_init	*initproc*	__init__	X	X		X
tp_alloc	*allocfunc*		X		?	?
tp_new	*newfunc*	__new__	X	X	?	?
tp_free	*freefunc*		X	X	?	?
tp_is_gc	*inquiry*			X		X
<*tp_bases*>	*PyObject* *	__bases__			~	
<*tp_mro*>	*PyObject* *	__mro__			~	
[*tp_cache*]	*PyObject* *					
[*tp_subclasses*]	void *	__subclasses__				
[*tp_weaklist*]	*PyObject* *					
(*tp_del*)	*destructor*					
[*tp_version_tag*]	unsigned int					
tp_finalize	*destructor*	__del__				X

繼續下一頁

表格 1 – 繼續上一頁

PyTypeObject 槽[Page 246, 1]	*Type*	特殊方法/属性	信息[2] C T D I
tp_vectorcall	*vectorcallfunc*		
[*tp_watched*]	unsigned char		

子方法槽（方法域）

方法槽	*Type*	特殊方法
am_await	*unaryfunc*	__await__
am_aiter	*unaryfunc*	__aiter__
am_anext	*unaryfunc*	__anext__
am_send	*sendfunc*	
nb_add	*binaryfunc*	__add__ __radd__
nb_inplace_add	*binaryfunc*	__iadd__
nb_subtract	*binaryfunc*	__sub__ __rsub__
nb_inplace_subtract	*binaryfunc*	__isub__
nb_multiply	*binaryfunc*	__mul__ __rmul__
nb_inplace_multiply	*binaryfunc*	__imul__
nb_remainder	*binaryfunc*	__mod__ __rmod__
nb_inplace_remainder	*binaryfunc*	__imod__
nb_divmod	*binaryfunc*	__divmod__ __rdivmod__
nb_power	*ternaryfunc*	__pow__ __rpow__
nb_inplace_power	*ternaryfunc*	__ipow__
nb_negative	*unaryfunc*	__neg__
nb_positive	*unaryfunc*	__pos__
nb_absolute	*unaryfunc*	__abs__
nb_bool	*inquiry*	__bool__
nb_invert	*unaryfunc*	__invert__
nb_lshift	*binaryfunc*	__lshift__ __rlshift__
nb_inplace_lshift	*binaryfunc*	__ilshift__
nb_rshift	*binaryfunc*	__rshift__ __rrshift__

繼續下一頁

[1] (): 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)。

[2] 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。

表格 2 – 繼續上一頁

方法槽	*Type*	特殊方法
nb_inplace_rshift	*binaryfunc*	__irshift__
nb_and	*binaryfunc*	__and__ __rand__
nb_inplace_and	*binaryfunc*	__iand__
nb_xor	*binaryfunc*	__xor__ __rxor__
nb_inplace_xor	*binaryfunc*	__ixor__
nb_or	*binaryfunc*	__or__ __ror__
nb_inplace_or	*binaryfunc*	__ior__
nb_int	*unaryfunc*	__int__
nb_reserved	void *	
nb_float	*unaryfunc*	__float__
nb_floor_divide	*binaryfunc*	__floordiv__
nb_inplace_floor_divide	*binaryfunc*	__ifloordiv__
nb_true_divide	*binaryfunc*	__truediv__
nb_inplace_true_divide	*binaryfunc*	__itruediv__
nb_index	*unaryfunc*	__index__
nb_matrix_multiply	*binaryfunc*	__matmul__ __rmatmul__
nb_inplace_matrix_multiply	*binaryfunc*	__imatmul__
mp_length	*lenfunc*	__len__
mp_subscript	*binaryfunc*	__getitem__
mp_ass_subscript	*objobjargproc*	__setitem__, __delitem__
sq_length	*lenfunc*	__len__
sq_concat	*binaryfunc*	__add__
sq_repeat	*ssizeargfunc*	__mul__
sq_item	*ssizeargfunc*	__getitem__
sq_ass_item	*ssizeobjargproc*	__setitem__ __delitem__
sq_contains	*objobjproc*	__contains__
sq_inplace_concat	*binaryfunc*	__iadd__
sq_inplace_repeat	*ssizeargfunc*	__imul__
bf_getbuffer	*getbufferproc()*	
bf_releasebuffer	*releasebufferproc()*	

槽位 typedef

typedef	参数类型	返回类型
allocfunc	*PyTypeObject* * *Py_ssize_t*	*PyObject* *
destructor	*PyObject* *	void
freefunc	void *	void
traverseproc	*PyObject* * *visitproc* void *	int
newfunc	*PyObject* * *PyObject* * *PyObject* *	*PyObject* *
initproc	*PyObject* * *PyObject* * *PyObject* *	int
reprfunc	*PyObject* *	*PyObject* *
getattrfunc	*PyObject* * const char *	*PyObject* *
setattrfunc	*PyObject* * const char * *PyObject* *	int
getattrofunc	*PyObject* * *PyObject* *	*PyObject* *
setattrofunc	*PyObject* * *PyObject* * *PyObject* *	int
descrgetfunc	*PyObject* * *PyObject* * *PyObject* *	*PyObject* *
descrsetfunc	*PyObject* * *PyObject* * *PyObject* *	int
hashfunc	*PyObject* *	Py_hash_t
richcmpfunc		*PyObject* *

更多細節請見下方的槽位类型 *typedef*。

12.3.2 PyTypeObject 定义

PyTypeObject 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;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

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

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

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

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;

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

(繼續下一頁)

(繼續上一頁)

```
    /* Attribute descriptor and subclassing stuff */
    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;

    /* bitset of which type-watchers care about this type */
    unsigned char tp_watched;
} 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**

🄵穩定 ABI 的一部分. 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**

🄵穩定 ABI 的一部分. 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__

🄵稳定 ABI 的一部分. 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。

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。

在 3.12 版的變更: 在 3.12 版之前，不推荐可变堆类型 实现 vectorcall 协议。当用户在 Python 代码中设置 __call__ 时，只有 *tp_call* 会被更新，很可能使它与 vectorcall 函数不一致。自 3.12 起，设置 __call__ 将通过清除*Py_TPFLAGS_HAVE_VECTORCALL* 旗标来禁用 vectorcall 优化。

继承：

该字段总是会被继承。但是，*Py_TPFLAGS_HAVE_VECTORCALL* 旗标并不总是会被继承。如果它未被设置，则子类不会使用*vectorcall*，除非显式地调用了*PyVectorcall_Call()*。

getattrfunc *PyTypeObject*.**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：

群組：*tp_getattr*、*tp_getattro*

该字段会被子类和*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 *PyTypeObject*.**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：

群組：*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 **PyTypeObject*.**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。

Added in version 3.5: 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 *PyTypeObject*.**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。

预設：

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：

群組：*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。

預設：

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。

getattrofunc *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：

群組：*tp_getattr*、*tp_getattro*

该字段会被子类同*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*。

預設：

PyBaseObject_Type 使用*PyObject_GenericGetAttr()*。

setattrofunc *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：

群組：*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*。

預設：

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：

这个字段的继承很复杂。大多数旗标位都是单独继承的，也就是说，如果基类型设置了一个旗标位，则子类型将继承该旗标位。从属于扩展结构体的旗标位仅在扩展结构体被继承时才会被继承，也就是说，基类型的旗标位值会与指向扩展结构体的指针一起被拷贝到子类型中。*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。.. XXX 那么大多数旗标位 真的都是单独继承的吗?

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比特位，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 和 DECREF）。堆类型应当也支持垃圾回收 因为它们会形成对它们自己的模块对象的循环引用。

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, **kwds) (其中 obj 不为 None) 必须等价于 meth(obj, *args, **kwds)。
- meth.__get__(None, cls)(*args, **kwds) 必须等价于 meth(*args, **kwds)。

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。

Added in version 3.8.

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_MANAGED_DICT

该比特位表示类的实例具有 __dict__ 属性，并且该字典的空间是由 VM 管理的。

如果设置了该旗标，则*Py_TPFLAGS_HAVE_GC* 也应当被设置。

Added in version 3.12.

继承：

此旗标将被继承，除非某个超类设置了*tp_dictoffset* 字段。

Py_TPFLAGS_MANAGED_WEAKREF

该比特位表示类的实例应当是可被弱引用的。

Added in version 3.12.

继承：

此旗标将被继承，除非某个超类设置了*tp_weaklistoffset* 字段。

Py_TPFLAGS_ITEMS_AT_END

仅适用于可变大小的类型，也就是说，具有非零*tp_itemsize* 值的类型。

表示此类型的实例的可变大小部分位于该实例内存区的末尾，其偏移量为 Py_TYPE(obj)->tp_basicsize (每个子类可能不一样)。

当设置此旗标时，请确保所有子类要么使用此内存布局，要么不是可变大小。Python 不会检查这一点。

Added in version 3.12.

继承：

这个旗标会被继承。

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。

Added in version 3.4.

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：

如果继承了*tp_call* 则也会继承这个比特位。

Added in version 3.9.

在 3.12 版的變更: 现在当类的 `__call__()` 方法被重新赋值时该旗标将从类中移除。

现在该旗标能被可变类所继承。

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。

Added in version 3.10.

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。

Added in version 3.10.

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

Added in version 3.10.

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🄴： *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

Added in version 3.10.

Py_TPFLAGS_VALID_VERSION_TAG

内部使用。请不要设置或取消设置此旗标。用于指明一个类具有被修改的调用*PyType_Modified()*

警告： 这个旗标存在于头文件中，但是属于内部特性而不应直接使用。它将在未来的 CPython 版本中被移除

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* 所定义的类型的实例。

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:

常數	对照
Py_LT	<
Py_LE	<=
Py_EQ	==
Py_NE	!=
Py_GT	>
Py_GE	>=

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`。

Added in version 3.7.

继承：

群組：*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**

虽然此字段仍然受到支持，但是如果可能就应当改用*Py_TPFLAGS_MANAGED_WEAKREF*。

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。

同时设置*Py_TPFLAGS_MANAGED_WEAKREF* 位和*tp_weaklist* 将导致报错。

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。

預設：

如果在*tp_dict* 字段中设置了*Py_TPFLAGS_MANAGED_WEAKREF* 位，则*tp_weaklistoffset* 将被设为负值，用以表明使用此字段是不安全的。

getiterfunc *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🄴： 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。

该字段通常应当在 PyType_Ready 被调用之前初始化为 NULL；它也可以初始化为一个包含类型初始属性的字典。一旦*PyType_Ready()* 完成类型的初始化，该类型的额外属性只有在它们不与被重载的操作 (如 __add__()) 相对应的情况下才会被添加到该字典中。一旦类型的初始化结束，该字段就应被视为是只读的。

某些类型不会将它们的字典存储在该槽位中。请使用*PyType_GetDict()* 来获取任意类型对应的字典。

在 3.12 版的變更: 内部细节：对于静态内置类型，该值总是为 NULL。这种类型的字典是存储在 PyInterpreterState 中。请使用*PyType_GetDict()* 来获取任意类型的字典。

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**

虽然此字段仍然受到支持，但是如果可能就应当改用*Py_TPFLAGS_MANAGED_DICT*。

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。

该值指定字典相对实例结构体开始位置的偏移量。

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。

同时设置*Py_TPFLAGS_MANAGED_WEAKREF* 位和*tp_dictoffset* 将导致报错。

继承：

该字段会被子类型所继承。子类型不应重写这个偏移量；这样做是不安全的，如果 C 代码试图在之前的偏移量上访问字典的话。要正确地支持继承，请使用*Py_TPFLAGS_MANAGED_DICT*。

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__。

如果在*tp_dict* 字段中设置了*Py_TPFLAGS_MANAGED_DICT*，那么*tp_dictoffset* 将被设为 -1，以表示使用该字段是不安全的。

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。

预設：

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 *kwds);
```

subtype 参数是被创建的对象的类型；*args* 和 *kwds* 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。

预設：

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

预設：

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。

void ***PyTypeObject*.**tp_subclasses**

一组子类。仅限内部使用的。可能为无效的指针。

要获取子类的列表。请调用 Python 方法 __subclasses__()。

在 3.12 版的變更: 对于某些类型，该字段将不带有效的 *PyObject**。类型已被改为 void* 以指明这一点。

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。

在 3.12 版的變更: 内部细节: 对于静态内置类型这将总是为 NULL，即使添加了弱引用也是如此。每个弱引用都转而保存在 PyInterpreterState 上。请使用公共 C-API 或内部 _PyObject_GET_WEAKREFS_LISTPTR() 宏来避免此差异。

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。

Added in version 3.4.

在 3.8 版的變更: 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。

也參考:

” 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。

Added in version 3.9: （这个字段从 3.8 起即存在，但是从 3.9 开始投入使用）

unsigned char *PyTypeObject*.**tp_watched**

内部对象。请勿使用。

Added in version 3.12.

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。

这是通过填充*PyType_Spec* 结构体并调用*PyType_FromSpec()*，*PyType_FromSpecWithBases()*，*PyType_FromModuleAndSpec()* 或*PyType_FromMetaclass()* 来实现的。

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;
```

(繼續下一頁)

(繼續上一頁)

```
    binaryfunc nb_inplace_add;
    binaryfunc nb_inplace_subtract;
    binaryfunc nb_inplace_multiply;
    binaryfunc nb_inplace_remainder;
    ternaryfunc nb_inplace_power;
    binaryfunc nb_inplace_lshift;
    binaryfunc nb_inplace_rshift;
    binaryfunc nb_inplace_and;
    binaryfunc nb_inplace_xor;
    binaryfunc nb_inplace_or;

    binaryfunc nb_floor_divide;
    binaryfunc nb_true_divide;
    binaryfunc nb_inplace_floor_divide;
    binaryfunc nb_inplace_true_divide;

    unaryfunc nb_index;

    binaryfunc nb_matrix_multiply;
    binaryfunc nb_inplace_matrix_multiply;
} PyNumberMethods;
```

備🄴: 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🄴: *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) 回傳 `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

Added in version 3.5.

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。

Added in version 3.10.

12.9 槽位类型 typedef

typedef *PyObject* *(***allocfunc**)(*PyTypeObject* *cls, *Py_ssize_t* nitems)

🄵稳定 ABI 的一部分. 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**)

🄵稳定 ABI 的一部分.

typedef void (***freefunc**)(void*)

請見*tp_free*。

typedef *PyObject* *(***newfunc**)(*PyObject**, *PyObject**, *PyObject**)

🄵稳定 ABI 的一部分. 請見*tp_new*。

typedef int (***initproc**)(*PyObject**, *PyObject**, *PyObject**)

🄵穩定 ABI 的一部分. 請見*tp_init*。

typedef *PyObject* *(***reprfunc**)(*PyObject**)

🄵穩定 ABI 的一部分. 請見*tp_repr*。

typedef *PyObject* *(***getattrfunc**)(*PyObject* *self, char *attr)

🄵穩定 ABI 的一部分. 返回对象的指定属性的值。

typedef int (***setattrfunc**)(*PyObject* *self, char *attr, *PyObject* *value)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 返回对象的指定属性的值。

請見*tp_getattro*。

typedef int (***setattrofunc**)(*PyObject* *self, *PyObject* *attr, *PyObject* *value)

🄵穩定 ABI 的一部分. 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**)

🄵穩定 ABI 的一部分. 請見*tp_descr_get*。

typedef int (***descrsetfunc**)(*PyObject**, *PyObject**, *PyObject**)

🄵穩定 ABI 的一部分. 請見*tp_descr_set*。

typedef *Py_hash_t* (***hashfunc**)(*PyObject**)

🄵穩定 ABI 的一部分. 請見*tp_hash*。

typedef *PyObject* *(***richcmpfunc**)(*PyObject**, *PyObject**, int)

🄵穩定 ABI 的一部分. 請見*tp_richcompare*。

typedef *PyObject* *(***getiterfunc**)(*PyObject**)

🄵穩定 ABI 的一部分. 請見*tp_iter*。

typedef *PyObject* *(***iternextfunc**)(*PyObject**)

🄵穩定 ABI 的一部分. 請見*tp_iternext*。

typedef *Py_ssize_t* (***lenfunc**)(*PyObject**)

🄵穩定 ABI 的一部分.

typedef int (***getbufferproc**)(*PyObject**, *Py_buffer**, int)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始.

typedef void (***releasebufferproc**)(*PyObject**, *Py_buffer**)

🄵穩定 ABI 的一部分 自 *3.12* 版本開始.

typedef *PyObject* *(***unaryfunc**)(*PyObject**)

🄵穩定 ABI 的一部分.

typedef *PyObject* *(***binaryfunc**)(*PyObject**, *PyObject**)

🄵穩定 ABI 的一部分.

typedef *PySendResult* (***sendfunc**)(*PyObject**, *PyObject**, *PyObject***)

請見*am_send*。

typedef *PyObject* *(***ternaryfunc**)(*PyObject**, *PyObject**, *PyObject**)

🄵穩定 ABI 的一部分.

typedef *PyObject* *(***ssizeargfunc**)(*PyObject**, *Py_ssize_t*)

🄵穩定 ABI 的一部分.

typedef int (***ssizeobjargproc**)(*PyObject**, *Py_ssize_t*, *PyObject**)

🄵穩定 ABI 的一部分.

typedef int (***objobjproc**)(*PyObject**, *PyObject**)

🄵穩定 ABI 的一部分.

typedef int (***objobjargproc**)(*PyObject**, *PyObject**, *PyObject**)

🄵穩定 ABI 的一部分.

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 */
```

(繼續下一頁)

(繼續上一頁)

```
    0,                                  /* tp_iternext */
    0,                                  /* tp_methods */
    0,                                  /* tp_members */
    0,                                  /* tp_getset */
    0,                                  /* tp_base */
    0,                                  /* tp_dict */
    0,                                  /* tp_descr_get */
    0,                                  /* tp_descr_set */
    0,                                  /* tp_dictoffset */
    0,                                  /* tp_init */
    0,                                  /* tp_alloc */
    myobj_new,                          /* tp_new */
};
```

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;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
        Py_TPFLAGS_HAVE_GC | Py_TPFLAGS_MANAGED_DICT |
        Py_TPFLAGS_MANAGED_WEAKREF,
    .tp_new = myobj_new,
    .tp_traverse = (traverseproc)myobj_traverse,
    .tp_clear = (inquiry)myobj_clear,
    .tp_alloc = PyType_GenericNew,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
    .tp_hash = (hashfunc)myobj_hash,
    .tp_richcompare = PyBaseObject_Type.tp_richcompare,
};
```

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
```

(繼續下一頁)

(繼續上一頁)

```
} MyObject;

static PyTypeObject 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 PyTypeObject 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 ***PyUnstable_Object_GC_NewWithExtraData**(*PyTypeObject* *type, size_t extra_size)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

与*PyObject_GC_New* 类似但会在对象的末尾分配 *extra_size* 个字节（在*tp_basicsize* 偏移量处）。除*Python 对象标头* 外，分配的内存将初始化为零。

附加数据将与对象一起被释放，但在其他情况下则不会由 Python 来管理。

警告： 此函数被标记为非稳定的因为在实例之后保留附加数据的机制尚未确定。要分配可变数量的字段，推荐改用*PyVarObject* 和*tp_itemsize*。

Added in version 3.12.

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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.9* 版本開始. 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()`。

Added in version 3.9.

int **PyObject_GC_IsFinalized**(*PyObject* *op)

🄵穩定 ABI 的一部分 自 *3.9* 版本開始. 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()`。

Added in version 3.9.

void **PyObject_GC_Del**(void *op)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分. 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)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 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。

Added in version 3.10.

int **PyGC_Disable**(void)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 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。

Added in version 3.10.

int **PyGC_IsEnabled**(void)

🄵穩定 ABI 的一部分 自 *3.10* 版本開始. 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。

Added in version 3.10.

12.11.2 查询垃圾回收器状态

该 C-API 提供了以下接口用于查询有关垃圾回收器的信息。

void **PyUnstable_GC_VisitObjects**(*gcvisitobjects_t* callback, void *arg)

這是不穩定 *API*，它可能在小版本發布中🄵有任何警告地被變更。

在全部活动的支持 GC 的对象上运行所提供的 *callback*。*arg* 会被传递给所有 *callback* 的发起调用。

警告： 如果新对象被回调（取消）分配后再被访问其行为是未定义的。

垃圾回收在运行期间被禁用。在回调中显式地运行回收可能导致未定义的行为，例如多次访问同一对象或完全不访问。

Added in version 3.12.

typedef int (***gcvisitobjects_t**)(*PyObject* *object, void *arg)

要传给`PyUnstable_GC_VisitObjects()`的访问者函数的类型。*arg* 与传给 `PyUnstable_GC_VisitObjects` 的 *arg* 相同。返回 `0` 以继续迭代，返回 `1` 以停止迭代。其他返回值目前被保留因此返回任何其他值的行为都是未定义的。

Added in version 3.12.

CHAPTER 13

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` 🄵發布候選版本、`0xF` 則🄵最終版。

PY_RELEASE_SERIAL

在 `3.4.1a2` 中的 `2`。零則🄵最終發布版本。

PY_VERSION_HEX

被編碼🄵單一整數的 Python 版本號。

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 >= ...`。

該版本也可透過符號 *`Py_Version`* 獲得。

const unsigned long **Py_Version**

🄵穩定 ABI 的一部分 自 *3.11* 版本開始. 編碼🄵單個常數整數的 Python 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。

Added in version 3.11.

所有提到的巨集都定義在 Include/patchlevel.h。

APPENDIX A

術語表

>>>
互動式 shell 的預設 Python 提示字元。常見於能在直譯器中以互動方式被執行的程式碼範例。

...
可以表示：

- 在一個被縮排的程式碼區塊、在一對匹配的左右定界符（delimiter，例如括號、方括號、花括號或三引號）🄵部，或是在指定一個裝飾器 (decorator) 之後，要輸入程式碼時，互動式 shell 顯示的預設 Python 提示字元。
- 🄵建常數 `Ellipsis`。

2to3
一個試著將 Python 2.x 程式碼轉🄵🄵 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 在可以標準函式庫中以 `lib2to3` 被使用；它提供了一個獨立的入口點，在 `Tools/scripts/2to3`。請參🄵 2to3-reference。

abstract base class（抽象基底類🄵）
抽象基底類🄵（又稱🄵 ABC）提供了一種定義介面的方法，作🄵*duck-typing*（鴨子型🄵）的補充。其他類似的技術，像是 `hasattr()`，則顯得笨拙或是帶有細微的錯誤（例如使用魔術方法 (magic method)）。ABC 🄵用🄵擬的 subclass（子類🄵），它們🄵不繼承自另一個 class（類🄵），但仍可被 `isinstance()` 及 `issubclass()` 辨識；請參🄵 `abc` 模組的🄵明文件。Python 有許多🄵建的 ABC，用於資料結構（在 `collections.abc` 模組）、數字（在 `numbers` 模組）、串流（在 `io` 模組）及 import 尋檢器和載入器（在 `importlib.abc` 模組）。你可以使用 `abc` 模組建立自己的 ABC。

annotation（🄵釋）
一個與變數、class 屬性、函式的參數或回傳值相關聯的標🄵。照慣例，它被用來作🄵*type hint*（型🄵提示）。

在執行環境 (runtime)，區域變數的🄵釋無法被存取，但全域變數、class 屬性和函式的🄵解，會分🄵被儲存在模組、class 和函式的 `__annotations__` 特殊屬性中。

請參🄵*variable annotation*、*function annotation*、**PEP 484** 和 **PEP 526**，這些章節皆有此功能的🄵明。關於🄵釋的最佳實踐方法也請參🄵 annotations-howto。

argument（引數）
呼叫函式時被傳遞給*function*（或*method*）的值。引數有兩種：

- 關鍵字引數 *(keyword argument)*：在函式呼叫中，以識🄵字（identifier，例如 `name=`）開頭的引數，或是以 `**` 後面 dictionary（字典）🄵的值被傳遞的引數。例如，`3` 和 `5` 都是以下 `complex()` 呼叫中的關鍵字引數：

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

- 位置引數 *(positional argument)*：不是關鍵字引數的引數。位置引數可在一個引數列表的起始處出現，和（或）作🄵 `*` 之後的*iterable*（可🄵代物件）中的元素被傳遞。例如，`3` 和 `5` 都是以下呼叫中的位置引數：

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

引數會被指定給函式主體中的附名區域變數。關於支配這個指定過程的規則，請參🄵 calls 章節。在語法上，任何運算式都可以被用來表示一個引數；其評估值會被指定給區域變數。

另請參🄵術語表的*parameter*（參數）條目、常見問題中的引數和參數之間的差🄵，以及 **PEP 362**。

asynchronous context manager（非同步情境管理器）
一個可以控制 `async with` 陳述式中所見環境的物件，而它是透過定義 `__aenter__()` 和 `__aexit__()` method（方法）來控制的。由 **PEP 492** 引入。

asynchronous generator（非同步🄵生器）
一個會回傳*asynchronous generator iterator*（非同步🄵生器🄵代器）的函式。它看起來像一個以 `async def` 定義的協程函式 (coroutine function)，但不同的是它包含了 `yield` 運算式，能生成一系列可用於 `async for` 🄵圈的值。

這個術語通常用來表示一個非同步🄵生器函式，但在某些情境中，也可能是表示非同步🄵生器🄵代器 *(asynchronous generator iterator)*。萬一想表達的意思不🄵清楚，那就使用完整的術語，以避免歧義。

一個非同步🄵生器函式可能包含 `await` 運算式，以及 `async for` 和 `async with` 陳述式。

asynchronous generator iterator（非同步🄵生器🄵代器）
一個由*asynchronous generator*（非同步🄵生器）函式所建立的物件。

這是一個*asynchronous iterator*（非同步🄵代器），當它以 `__anext__()` method 被呼叫時，會回傳一個可等待物件 (awaitable object)，該物件將執行非同步🄵生器的函式主體，直到遇到下一個 `yield` 運算式。

每個 `yield` 會暫停處理程序，🄵記住位置執行狀態（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（二進位檔案）
一個能🄵讀取和寫入*bytes-like objects*（類位元組串物件）的*file object*（檔案物件）。二進位檔案的例子有：以二進位模式（`'rb'`、`'wb'` 或 `'rb+'`）開🄵的檔案、`sys.stdin.buffer`、`sys.stdout.buffer`，以及 `io.BytesIO` 和 `gzip.GzipFile` 實例。

另請參🄵*text file*（文字檔案），它是一個能🄵讀取和寫入 `str` 物件的檔案物件。

borrowed reference（借用參照）
在 Python 的 C API 中，借用參照是一個對物件的參照，其中使用該物件的程式碼🄵不擁有這個參照。如果該物件被銷🄵，它會成🄵一個迷途指標 (dangling pointer)。例如，一次垃圾回收 (garbage collection) 可以移除對物件的最後一個*strong reference*（🄵參照），而將該物件銷🄵。

對*borrowed reference* 呼叫`Py_INCREF()` 以將它原地 (in-place) 轉🄵🄵*strong reference* 是被建議的做法，除非該物件不能在最後一次使用借用參照之前被銷🄵。`Py_NewRef()` 函式可用於建立一個新的*strong reference*。

bytes-like object（類位元組串物件）
一個支援*緩衝協定 (Buffer Protocol)*且能🄵匯出 C-*contiguous* 緩衝區的物件。這包括所有的 `bytes`、`bytearray` 和 `array.array` 物件，以及許多常見的 `memoryview` 物件。類位元組串物件可用於處理二進位資料的各種運算；這些運算包括壓縮、儲存至二進位檔案和透過 socket（插座）發送。

有些運算需要二進位資料是可變的。🄵明文件通常會將這些物件稱🄵「可讀寫的類位元組串物件」。可變緩衝區的物件包括 `bytearray`，以及 `bytearray` 的 `memoryview`。其他的運算需要讓二進位資料被儲存在不可變物件（「唯讀的類位元組串物件」）中；這些物件包括 `bytes`，以及 `bytes` 物件的 `memoryview`。

bytecode（位元組碼）
Python 的原始碼會被編譯成位元組碼，它是 Python 程式在 CPython 直譯器中的🄵部表示法。該位元組碼也會被暫存在 `.pyc` 檔案中，以便第二次執行同一個檔案時能🄵更快速（可以不用從原始碼重新編譯🄵位元組碼）。這種「中間語言 (intermediate language)」據🄵是運行在一個*virtual machine*（🄵擬機器）上，該🄵擬機器會執行與每個位元組碼對應的機器碼 (machine code)。要注意的是，位元組碼理論上是無法在不同的 Python 🄵擬機器之間運作的，也不能在不同版本的 Python 之間保持穩定。

位元組碼的指令列表可以在 dis 模組的🄵明文件中找到。

callable（可呼叫物件）
一個 callable 是可以被呼叫的物件，呼叫時可能以下列形式帶有一組引數（請見*argument*）：

```
callable(argument1, argument2, argumentN)
```

一個*function* 與其延伸的*method* 都是 callable。一個有實作 `__call__()` 方法的 class 之實例也是個 callable。

callback（回呼）
作🄵引數被傳遞的一個副程式 (subroutine) 函式，會在未來的某個時間點被執行。

class（類🄵）
一個用於建立使用者定義物件的模板。Class 的定義通常會包含 method 的定義，這些 method 可以在 class 的實例上進行操作。

class variable（類🄵變數）
一個在 class 中被定義，且應該只能在 class 層次（意即不是在 class 的實例中）被修改的變數。

complex number（🄵數）
一個我們熟悉的實數系統的擴充，在此所有數字都會被表示🄵一個實部和一個🄵部之和。🄵數就是🄵數單位（`-1` 的平方根）的實數倍，此單位通常在數學中被寫🄵 `i`，在工程學中被寫🄵 `j`。Python 🄵建了對🄵數的支援，它是用後者的記法來表示🄵數；🄵部會帶著一個後綴的 `j` 被編寫，例如 `3+1j`。若要將 `math` 模組🄵的工具等效地用於🄵數，請使用 `cmath` 模組。🄵數的使用是一個相當進階的數學功能。如果你🄵有察覺到對它們的需求，那🄵幾乎能確定你可以安全地忽略它們。

context manager（情境管理器）
一個可以控制 `with` 陳述式中所見環境的物件，而它是透過定義 `__enter__()` 和 `__exit__()` method 來控制的。請參🄵 **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🄵。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，🄵可能會包含 `await`、`async for` 和 `async with` 關鍵字。這些關鍵字由 **PEP 492** 引入。

CPython
Python 程式語言的標准實作 (canonical implementation)，被發布在 python.org 上。「CPython」這個術語在必要時被使用，以區分此實作與其它語言的實作，例如 Jython 或 IronPython。

decorator（裝飾器）
一個函式，它會回傳另一個函式，通常它會使用 `@wrapper` 語法，被應用🄵一種函式的變🄴 (function transformation)。裝飾器的常見範例是 `classmethod()` 和 `staticmethod()`。

裝飾器語法只是語法糖。以下兩個函式定義在語義上是等效的：

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

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

Class 也存在相同的概念，但在那🄵比較不常用。關於裝飾器的更多🄵容，請參🄵函式定義和 class 定義的🄵明文件。

descriptor（描述器）
任何定義了 `__get__()`、`__set__()` 或 `__delete__()` method 的物件。當一個 class 屬性是一個描述器時，它的特殊連結行🄴會在屬性查找時被觸發。通常，使用 *a.b* 來取得、設定或🄵除某個屬性時，會在 *a* 的 class 字典中查找名稱🄵 *b* 的物件，但如果 *b* 是一個描述器，則相對應的描述器 method 會被呼叫。對描述器的理解是深入理解 Python 的關鍵，因🄵它們是許多功能的基礎，這些功能包括函式、method、屬性 (property)、class method、🄵態 method，以及對 super class（父類🄵）的參照。

關於描述器 method 的更多資訊，請參🄵 descriptors 或描述器使用指南。

dictionary（字典）
一個關聯陣列 (associative array)，其中任意的鍵會被映射到值。鍵可以是任何帶有 `__hash__()` 和 `__eq__()` method 的物件。在 Perl 中被稱🄵雜🄵 (hash)。

dictionary comprehension（字典綜合運算）
一種緊密的方法，用來處理一個可🄴代物件中的全部或部分元素，🄵將處理結果以一個字典回傳。

`results = {n: n ** 2 for n in range(10)}` 會🄵生一個字典，它包含了鍵 `n` 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🄵制將字典檢視轉🄵完整的 list（串列），須使用 `list(dictview)`。請參🄵 dict-views。

docstring（🄵明字串）
一個在 class、函式或模組中，作🄵第一個運算式出現的字串文本。雖然它在套件執行時會被忽略，但它會被編譯器辨識，🄵被放入所屬 class、函式或模組的 `__doc__` 屬性中。由於🄵明字串可以透過🄵省 (introspection) 來🄵覽，因此它是物件的🄵明文件存放的標準位置。

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）
一個讓使用者透過檔案導向 (file-oriented) API（如 `read()` 或 `write()` 等 method）來操作底層資源的物件。根據檔案物件被建立的方式，它能🄵協調對真實磁碟檔案或是其他類型的儲存器或通訊裝置（例如標準輸入 / 輸出、記憶體🄴緩衝區、socket（插座）、管🄵 (pipe) 等）的存取。檔案物件也被稱🄵*類檔案物件 (file-like object)* 或*串流 (stream)*。

實際上，有三種檔案物件：原始的*二進位檔案*、緩衝的*二進位檔案*和*文字檔案*。它們的介面在 `io` 模組中被定義。建立檔案物件的標準方法是使用 `open()` 函式。

file-like object（類檔案物件）
file object（檔案物件）的同義字。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）
Python 所使用的一種編碼和錯誤處理函式，用來解碼來自作業系統的位元組，以及將 Unicode 編碼到作業系統。

檔案系統編碼必須保證能成功解碼所有小於 128 的位元組。如果檔案系統編碼無法提供此保證，則 API 函式會引發 `UnicodeError`。

`sys.getfilesystemencoding()` 和 `sys.getfilesystemencodeerrors()` 函式可用於取得檔案系統編碼和錯誤處理函式。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）會在 Python 🄵動時由*`PyConfig_Read()`* 函式來配置：請參🄵*`filesystem_encoding`*，以及*`PyConfig`* 的成員*`filesystem_errors`*。

另請參🄵*locale encoding*（區域編碼）。

finder（尋檢器）
一個物件，它會嘗試🄵正在被 import 的模組尋找*loader*（載入器）。

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🄵 **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🄵 `2`，與 float（浮點數）真除法所回傳的 `2.75` 不同。請注意，`(-11) // 4` 的結果是 `-3`，因🄵是 `-2.75` 被向下無條件舍去。請參🄵 **PEP 238**。

function（函式）
一連串的陳述式，它能🄵向呼叫者回傳一些值。它也可以被傳遞零個或多個*引數*，這些引數可被使用於函式本體的執行。另請參🄵*parameter*（參數）、*method*（方法），以及 function 章節。

function annotation（函式🄵釋）
函式參數或回傳值的一個*annotation*（🄵釋）。

函式🄵釋通常被使用於*型🄵提示*：例如，這個函式預期會得到兩個 `int` 引數，🄵會有一個 `int` 回傳值：

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

函式🄵釋的語法在 function 章節有詳細解釋。

請參🄵*variable annotation* 和 **PEP 484**，皆有此功能的描述。關於🄵釋的最佳實踐方法，另請參🄵 annotations-howto。

__future__
future 陳述式：`from __future__ import <feature>`，會指示編譯器使用那些在 Python 未來的發布版本中將成🄵標準的語法或語義，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（🄵生器）
一個會回傳*generator iterator*（🄵生器🄵代器）的函式。它看起來像一個正常的函式，但不同的是它包含了 `yield` 運算式，能🄵生一系列的值，這些值可用於 for 🄵圈，或是以 `next()` 函式，每次檢索其中的一個值。

這個術語通常用來表示一個🄵生器函式，但在某些情境中，也可能是表示🄵生器🄵代器。萬一想表達的意思不🄵清楚，那就使用完整的術語，以避免歧義。

generator iterator（🄵生器🄵代器）
一個由*generator*（🄵生器）函式所建立的物件。

每個 `yield` 會暫停處理程序，🄵記住位置執行狀態（包括區域變數及擱置中的 try 陳述式）。當🄵生器🄵代器回復時，它會從停止的地方繼續執行（與那些每次調用時都要重新開始的函式有所不同）。

generator expression（🄵生器運算式）
一個會回傳🄵代器的運算式。它看起來像一個正常的運算式，後面接著一個 `for` 子句，該子句定義了🄵圈變數、範圍以及一個選擇性的 `if` 子句。該組合運算式會🄵外層函式🄵生多個值：

```
>>> sum(i*i for i in range(10))         # 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F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F定。

另請參F*single dispatch*（單一調度）術語表條目、`functools.singledispatch()` 裝飾器和 **PEP 443**。

generic type（泛型型F）
一個能F被參數化 (parameterized) 的*type*（型F）；通常是一個 容器型F，像是 `list` 和 `dict`。它被用於型F提示和F釋。

詳情請參F泛型F名型F、**PEP 483**、**PEP 484**、**PEP 585** 和 `typing` 模組。

GIL
請參F*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F建型F，如 `dict`）自動地避免F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F多執行緒 (multi-threaded)，但代價是會犧牲掉多處理器的機器能F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F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F未成功，因F在一般的單一處理器情F下，效能會有所損失。一般認F，若要克服這個效能問題，會使實作變得F雜許多，進而付出更高的維護成本。

hash-based pyc（雜F架構的 pyc）
一個位元組碼 (bytecode) 暫存檔，它使用雜F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F pyc-invalidation。

hashable（可雜F的）
如果一個物件有一個雜F值，該值在其生命F期中永不改變（它需要一個 `__hash__()` method），且可與其他物件互相比較（它需要一個 `__eq__()` method），那F它就是一個可雜F物件。比較結果F相等的多個可雜F物件，它們必須擁有相同的雜F值。

可雜F性 (hashability) 使一個物件可用作 dictionary（字典）的鍵和 set（集合）的成員，因F這些資料結構都在其F部使用了雜F值。

大多數的 Python 不可變F建物件都是可雜F的；可變的容器（例如 list 或 dictionary）F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F的，它們本身才是可雜F的。若物件是使用者自定 class 的實例，則這些物件會被預設F可雜F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F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F定雜F值的地方，扮演重要的角色，例如 dictionary（字典）中的一個鍵。

import path（引入路徑）
一個位置（或路徑項目）的列表，而那些位置就是在 import 模組時，會被*path based finder*（基於路徑的尋檢器）搜尋模組的位置。在 import 期間，此位置列表通常是來自 `sys.path`，但對於子套件 (subpackage) 而言，它也可能是來自父套件的 `__path__` 屬性。

importing（引入）
一個過程。一個模組中的 Python 程式碼可以透過此過程，被另一個模組中的 Python 程式碼使用。

importer（引入器）
一個能🄵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🄵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🄵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🄵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🄵*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🄵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🄵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🄵本已經執行完成。

iterable（可🄵代物件）
一種能🄵一次回傳一個其中成員的物件。可🄵代物件的例子包括所有的序列型🄵（像是 `list`、`str` 和 `tuple`）和某些非序列型🄵，像是 `dict`、*檔案物件*，以及你所定義的任何 class 物件，只要那些 class 有 `__iter__()` method 或是實作*sequence*（序列）語意的 `__getitem__()` method，該物件就是可🄵代物件。

可🄵代物件可用於 `for` 🄵圈和許多其他需要一個序列的地方 (`zip()`、`map()`...)。當一個可🄵代物件作🄵引數被傳遞給🄵建函式 `iter()` 時，它會🄵該物件回傳一個🄵代器。此🄵代器適用於針對一組值進行一遍 (one pass) 運算。使用🄵代器時，通常不一定要呼叫 `iter()` 或自行處理🄵代器物件。`for` 陳述式會自動地🄵你處理這些事，它會建立一個暫時性的未命名變數，用於在🄵圈期間保有該🄵代器。另請參🄵*iterator*（🄵代器）、*sequence*（序列）和*generator*（🄵生器）。

iterator（🄵代器）
一個表示資料流的物件。重🄵地呼叫🄵代器的 `__next__()` method（或是將它傳遞給🄵建函式 `next()`）會依序回傳資料流中的各項目。當不再有資料時，則會引發 `StopIteration` 例外。此時，該🄵代器物件已被用盡，而任何對其 `__next__()` method 的進一步呼叫，都只會再次引發 `StopIteration`。🄵代器必須有一個 `__iter__()` method，它會回傳🄵代器物件本身，所以每個🄵代器也都是可🄵代物件，且可以用於大多數適用其他可🄵代物件的場合。一個明顯的例外，是嘗試多遍🄵代 (multiple iteration passes) 的程式碼。一個容器物件（像是 `list`）在每次你將它傳遞給 `iter()` 函式或在 `for` 🄵圈中使用它時，都會🄵生一個全新的🄵代器。使用🄵代器嘗試此事（多遍🄵代）時，只會回傳在前一遍🄵代中被用過的、同一個已被用盡的🄵代器物件，使其看起來就像一個空的容器。

在 typeiter 文中可以找到更多資訊。

CPython 實作細節： CPython 🄵不是始終如一地都會檢查「🄵代器有定義 `__iter__()`」這個規定。

key function（鍵函式）
鍵函式或理序函式 (collation function) 是一個可呼叫 (callable) 函式，它會回傳一個用於排序 (sorting) 或定序 (ordering) 的值。例如，`locale.strxfrm()` 被用來🄵生一個了解區域特定排序慣例的排序鍵。

Python 中的許多工具，都接受以鍵函式來控制元素被定序或分組的方式。它們包括 `min()`、`max()`、`sorted()`、`list.sort()`、`heapq.merge()`、`heapq.nsmallest()`、`heapq.nlargest()` 和 `itertools.groupby()`。

有幾種方法可以建立一個鍵函式。例如，`str.lower()` method 可以作🄵不分大小寫排序的鍵函式。或者，一個鍵函式也可以從 `lambda` 運算式被建造，例如 `lambda r: (r[0], r[2])`。另外，`operator.attrgetter()`、`operator.itemgetter()` 和 `operator.methodcaller()` 🄵三個鍵函式的建構函式 (constructor)。關於如何建立和使用鍵函式的範例，請參🄵如何排序。

keyword argument（關鍵字引數）
請參🄵*argument*（引數）。

lambda
由單一*expression*（運算式）所組成的一個匿名行🄵函式 (inline function)，於該函式被呼叫時求值。建立 lambda 函式的語法是 `lambda [parameters]: expression`

LBYL
Look before you leap.（三思而後行。）這種編碼風格會在進行呼叫或查找之前，明確地測試先🄵條件。這種風格與*EAFP* 方式形成對比，且它的特色是會有許多 `if` 陳述式的存在。

在一個多執行緒環境中，LBYL 方式有在「三思」和「後行」之間引入了競🄵條件 (race condition) 的風險。例如以下程式碼 `if key in mapping: return mapping[key]`，如果另一個執行緒在測試之後但在查找之前，從 *mapping* 中移除了 *key*，則該程式碼就會失效。這個問題可以用鎖 (lock) 或使用 EAFP 編碼方式來解🄵。

list（串列）
一個 Python 🄵建的*sequence*（序列）。🄵管它的名字是 list，它其實更類似其他語言中的一個陣列 (array) 而較不像一個鏈結串列 (linked list)，因🄵存取元素的時間🄵雜度是 *O*(1)。

list comprehension（串列綜合運算）
一種用來處理一個序列中的全部或部分元素，🄵將處理結果以一個 list 回傳的簡要方法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 會🄵生一個字串 list，其中包含 0 到 255 範圍🄵，所有偶數的十六進位數 (0x..)。`if` 子句是選擇性的。如果省略它，則 `range(256)` 中的所有元素都會被處理。

loader（載入器）
一個能🄵載入模組的物件。它必須定義一個名🄵 `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()` 可以用來取得區域編碼。

也請參考*filesystem encoding and error handler*。

magic method（魔術方法）
special method（特殊方法）的一個非正式同義詞。

mapping（對映）
一個容器物件，它支援任意鍵的查找，且能實作 abstract base classes（抽象基底類🄵）中，`collections.abc.Mapping` 或 `collections.abc.MutableMapping` 所指定的 method。範例包括 `dict`、`collections.defaultdict`、`collections.OrderedDict` 和 `collections.Counter`。

meta path finder（元路徑尋檢器）
一種經由搜尋 `sys.meta_path` 而回傳的*finder*（尋檢器）。元路徑尋檢器與*路徑項目尋檢器 (path entry finder)* 相關但是不同。

關於元路徑尋檢器實作的 method，請參🄵 `importlib.abc.MetaPathFinder`。

metaclass（元類🄵）
一種 class 的 class。Class 定義過程會建立一個 class 名稱、一個 class dictionary（字典），以及一個 base class（基底類🄵）的列表。Metaclass 負責接受這三個引數，🄵建立該 class。大多數的物件導向程式語言會提供一個預設的實作。Python 的特🄵之處在於它能🄵建立自訂的 metaclass。大部分的使用者從未需要此工具，但是當需要時，metaclass 可以提供🄵大且優雅的解🄵方案。它們已被用於記🄵屬性存取、增加執行緒安全性、追🄵物件建立、實作單例模式 (singleton)，以及許多其他的任務。

更多資訊可以在 metaclasses 章節中找到。

method（方法）
一個在 class 本體🄵被定義的函式。如果 method 作🄵其 class 實例的一個屬性被呼叫，則它將會得到該實例物件成🄵它的第一個*argument*（引數）（此引數通常被稱🄵 `self`）。請參🄵*function*（函式）和*nested scope*（巢狀作用域）。

method resolution order（方法解析順序）
方法解析順序是在查找某個成員的過程中，base class（基底類🄵）被搜尋的順序。關於 Python 自 2.3 版直譯器所使用的演算法細節，請參🄵 python_2.3_mro。

module（模組）
一個擔任 Python 程式碼的組織單位 (organizational unit) 的物件。模組有一個命名空間，它包含任意的 Python 物件。模組是藉由*importing* 的過程，被載入至 Python。

另請參🄵*package*（套件）。

module spec（模組規格）
一個命名空間，它包含用於載入模組的 import 相關資訊。它是 `importlib.machinery.ModuleSpec` 的一個實例。

MRO
請參🄵*method resolution order*（方法解析順序）。

mutable（可變物件）
可變物件可以改變它們的值，但維持它們的 `id()`。另請參🄵*immutable*（不可變物件）。

named tuple（附名元組）
術語「named tuple（附名元組）」是指從 tuple 繼承的任何型🄵或 class，且它的可索引 (indexable) 元素也可以用附名屬性來存取。這些型🄵或 class 也可以具有其他的特性。

有些🄵建型🄵是 named tuple，包括由 `time.localtime()` 和 `os.stat()` 回傳的值。另一個例子是 `sys.float_info`：

```
>>> sys.float_info[1]                   # indexed access
1024
>>> sys.float_info.max_exp              # named field access
1024
>>> isinstance(sys.float_info, tuple)   # kind of tuple
True
```

有些 named tuple 是🄵建型🄵（如上例）。或者，一個 named tuple 也可以從一個正規的 class 定義來建立，只要該 class 是繼承自 `tuple`，且定義了附名欄位 (named field) 即可。這類的 class 可以手工編寫、可以繼承自 `typing.NamedTuple` 來建立，也可以使用工廠函式 (factory function) `collections.namedtuple()` 來建立。後者技術也增加了一些額外的 method，這些 method 可能是在手寫或🄵建的 named tuple 中，無法找到的。

namespace（命名空間）
變數被儲存的地方。命名空間是以 dictionary（字典）被實作。有區域的、全域的及🄵建的命名空間，而在物件中（在 method 中）也有巢狀的命名空間。命名空間藉由防止命名衝突，來支援模組化。例如，函式 `builtins.open` 和 `os.open()` 是透過它們的命名空間來區分彼此。命名空間也藉由明確地區分是哪個模組在實作一個函式，來增進可讀性及可維護性。例如，寫出 `random.seed()` 或 `itertools.islice()` 明確地表示，這些函式分🄵是由 `random` 和 `itertools` 模組在實作。

namespace package（命名空間套件）
一個 **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🄵下，巢狀作用域僅適用於參照，而無法用於賦值。區域變數能在最🄵層作用域中讀取及寫入。同樣地，全域變數是在全域命名空間中讀取及寫入。`nonlocal` 容許對外層作用域進行寫入。

new-style class（新式類🄵）
一個舊名，它是指現在所有的 class 物件所使用的 class 風格。在早期的 Python 版本中，只有新式 class 才能使用 Python 較新的、多樣的功能，像是 `__slots__`、描述器 (descriptor)、屬性 (property)、`__getattribute__()`、class method（類🄵方法）和 static method（🄵態方法）。

object（物件）
具有狀態（屬性或值）及被定義的行🄵（method）的任何資料。它也是任何*new-style class*（新式類🄵）的最終 base class（基底類🄵）。

package（套件）
一個 Python 的*module*（模組），它可以包含子模組 (submodule) 或是遞🄵的子套件 (subpackage)。技術上而言，套件就是具有 `__path__` 屬性的一個 Python 模組。

另請參🄵*regular package*（正規套件）和*namespace package*（命名空間套件）。

parameter（參數）
在*function*（函式）或 method 定義中的一個命名實體 (named entity)，它指明該函式能🄵接受的一個*argument*（引數），或在某些情🄵下指示多個引數。共有有五種不同的參數類型：

- *positional-or-keyword*（位置或關鍵字）：指明一個可以按照位置或是作🄵關鍵字引數被傳遞的引數。這是參數的預設類型，例如以下的 *foo* 和 *bar*：

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

- *positional-only*（僅限位置）：指明一個只能按照位置被提供的引數。在函式定義的參數列表中包含一個 `/` 字元，就可以在該字元前面定義僅限位置參數，例如以下的 *posonly1* 和 *posonly2*：

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

- *keyword-only*（僅限關鍵字）：指明一個只能以關鍵字被提供的引數。在函式定義的參數列表中，包含一個任意數量位置參數 (var-positional parameter) 或是單純的 `*` 字元，就可以在其後方定義僅限關鍵字參數，例如以下的 *kw_only1* 和 *kw_only2*：

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

- *var-positional*（任意數量位置）：指明一串能以任意序列被提供的位置引數（在已被其他參數接受的任何位置引數之外）。這類參數是透過在其參數名稱字首加上 `*` 來定義的，例如以下的 *args*：

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

- *var-keyword*（任意數量關鍵字）：指明可被提供的任意數量關鍵字引數（在已被其他參數接受的任何關鍵字引數之外）。這類參數是透過在其參數名稱字首加上 `**` 來定義的，例如上面範例中的 *kwargs*。

參數可以指明引數是選擇性的或必需的，也可以🄵一些選擇性的引數指定預設值。

另請參🄵術語表的*argument*（引數）條目、常見問題中的引數和參數之間的差🄵、`inspect.Parameter` class、function 章節，以及 **PEP 362**。

path entry（路徑項目）
在*import path*（引入路徑）中的一個位置，而*path based finder*（基於路徑的尋檢器）會參考該位置來尋找要 import 的模組。

path entry finder（路徑項目尋檢器）
被 `sys.path_hooks` 中的一個可呼叫物件 (callable)（意即一個*path entry hook*）所回傳的一種*finder*，它知道如何以一個*path entry*定位模組。

關於路徑項目尋檢器實作的 method，請參🄵 `importlib.abc.PathEntryFinder`。

path entry hook（路徑項目🄵）
在 `sys.path_hooks` 列表中的一個可呼叫物件 (callable)，若它知道如何在一個特定的*path entry* 中尋找模組，則會回傳一個*path entry finder*（路徑項目尋檢器）。

path based finder（基於路徑的尋檢器）
預設的*元路徑尋檢器 (meta path finder)* 之一，它會在一個*import path* 中搜尋模組。

path-like object（類路徑物件）
一個表示檔案系統路徑的物件。類路徑物件可以是一個表示路徑的 `str` 或 `bytes` 物件，或是一個實作 `os.PathLike` 協定的物件。透過呼叫 `os.fspath()` 函式，一個支援 `os.PathLike` 協定的物件可以被轉🄵🄵 `str` 或 `bytes` 檔案系統路徑；而 `os.fsdecode()` 及 `os.fsencode()` 則分🄵可用於確保 `str` 及 `bytes` 的結果。由 **PEP 519** 引入。

PEP
Python Enhancement Proposal（Python 增🄴提案）。PEP 是一份設計🄵明文件，它能🄵 Python 社群提供資訊，或是描述 Python 的一個新功能或該功能的程序和環境。PEP 應該要提供簡潔的技術規範以及被提案功能的運作原理。

PEP 的存在目的，是要成🄵重大新功能的提案、社群中關於某個問題的意見收集，以及已納入 Python 的設計🄵策的記🄵，這些過程的主要機制。PEP 的作者要負責在社群🄴建立共識🄴記🄵反對意見。

請參🄵 **PEP 1**。

portion（部分）
在單一目🄵中的一組檔案（也可能儲存在一個 zip 檔中），這些檔案能對一個命名空間套件 (namespace package) 有所貢獻，如同 **PEP 420** 中的定義。

positional argument（位置引數）
請參🄵*argument*（引數）。

provisional API（暫行 API）
暫行 API 是指，從標準函式庫的向後相容性 (backwards compatibility) 保證中，故意被排除的 API。雖然此類介面，只要它們被標示🄵暫行的，理論上🄵不會有重大的變更，但如果核心開發人員認🄵有必要，也可能會出現向後不相容的變更（甚至包括移除該介面）。這種變更🄵不會無端地🄵生——只有 API 被納入之前未察覺的嚴重基本缺陷被揭露時，它們才會發生。

即使對於暫行 API，向後不相容的變更也會被視🄴「最後的解🄵方案」——對於任何被發現的問題，仍然會盡可能找出一個向後相容的解🄵方案。

這個過程使得標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__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

當用於引用模組時，完全限定名稱 *(fully qualified name)* 是表示該模組的完整點分隔路徑，包括任何的父套件，例如 `email.mime.text`：

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

reference count（參照計數）
對於一個物件的參照次數。當一個物件的參照計數下降到零時，它會被解除配置 (deallocated)。有些物件是「不滅的 (immortal)」🄵擁有不會被改變的參照計數，也因此永遠不會被解除配置。參照計數通常在 Python 程式碼中看不到，但它🄵是*CPython* 實作的一個關鍵元素。程式設計師可以呼叫 `getrefcount()` 函式來回傳一個特定物件的參照計數。

regular package（正規套件）
一個傳統的*package*（套件），例如一個包含 `__init__.py` 檔案的目🄵。

另請參🄵*namespace package*（命名空間套件）。

__slots__
在 class 🄵部的一個宣告，它藉由預先宣告實例屬性的空間，以及消除實例 dictionary（字典），來節省記憶體。雖然該技術很普遍，但它有點難以正確地使用，最好保留給那種在一個記憶體關鍵 (memory-critical) 的應用程式中存在大量實例的罕見情🄵。

sequence（序列）
一個*iterable*（可🄵代物件），它透過 `__getitem__()` special method（特殊方法），使用整數索引來支援高效率的元素存取，🄵定義了一個 `__len__()` method 來回傳該序列的長度。一些🄵建序列型🄵包括 `list`、`str`、`tuple` 和 `bytes`。請注意，雖然 `dict` 也支援 `__getitem__()` 和 `__len__()`，但它被視🄵對映 (mapping) 而不是序列，因🄵其查找方式是使用任意的*immutable* 鍵，而不是整數。

抽象基底類🄵 (abstract base class) `collections.abc.Sequence` 定義了一個更加豐富的介面，🄵不僅止於 `__getitem__()` 和 `__len__()`，還增加了 `count()`、`index()`、`__contains__()` 和 `__reversed__()`。實作此擴充介面的型🄵，可以使用 `register()` 被明確地🄵🄵。更多關於序列方法的文件，請見常見序列操作。

set comprehension（集合綜合運算）
一種緊密的方法，用來處理一個可🄵代物件中的全部或部分元素，🄵將處理結果以一個 set 回傳。`results = {c for c in 'abracadabra' if c not in 'abc'}` 會🄵生一個字串 set：`{'r', 'd'}`。請參🄵 comprehensions。

single dispatch（單一調度）
generic function（泛型函式）調度的一種形式，在此，實作的選擇是基於單一引數的型🄵。

slice（切片）
一個物件，它通常包含一段*sequence*（序列）的某一部分。建立一段切片的方法是使用下標符號 (subscript notation) `[]`，若要給出多個數字，則在數字之間使用冒號，例如 `variable_name[1:3:5]`。在括號（下標）符號的🄵部，會使用 `slice` 物件。

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（🄵態型🄵檢查器）
會讀取 Python 程式碼🄵分析的外部工具，能🄵找出錯誤，像是使用了不正確的型🄵。另請參🄵*型🄵提示 (type hints)* 以及 `typing` 模組。

strong reference（🄵參照）
在 Python 的 C API 中，🄵參照是對物件的參照，該物件🄵持有該參照的程式碼所擁有。建立參照時透過呼叫*`Py_INCREF()`* 來獲得🄵參照、🄵除參照時透過*`Py_DECREF()`* 釋放🄵參照。

`Py_NewRef()` 函式可用於建立一個對物件的🄵參照。通常，在退出🄵參照的作用域之前，必須在該🄵參照上呼叫*`Py_DECREF()`* 函式，以避免🄵漏一個參照。

另請參🄵*borrowed reference*（借用參照）。

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*（二進位檔案），它是一個能🄵讀取和寫入*類位元組串物件 (bytes-like object)* 的檔案物件。

triple-quoted string（三引號🄵字串）
由三個雙引號 (") 或單引號 (') 的作🄵邊界的一個字串。雖然它們🄵🄵有提供🄵於單引號字串的任何額外功能，但基於許多原因，它們仍是很有用的。它們讓你可以在字串中包含未跳🄵 (unescaped) 的單引號和雙引號，而且它們不需使用連續字元 (continuation character) 就可以跨越多行，這使得它們在編寫🄵明字串時特🄵有用。

type（型🄵）
一個 Python 物件的型🄵🄵定了它是什🄵類型的物件；每個物件都有一個型🄵。一個物件的型🄵可以用它的 `__class__` 屬性來存取，或以 `type(obj)` 來檢索。

type alias（型🄵🄵名）
一個型🄵的同義詞，透過將型🄵指定給一個識🄵符 (identifier) 來建立。

型🄵🄵名對於簡化*型🄵提示 (type hint)* 很有用。例如：

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

可以寫成這樣，更具有可讀性：

```
Color = tuple[int, int, int]

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

請參🄵 `typing` 和 **PEP 484**，有此功能的描述。

type hint（型🄵提示）
一種*annotation*（🄵釋），它指定一個變數、一個 class 屬性或一個函式的參數或回傳值的預期型🄵。

型🄵提示是選擇性的，而不是被 Python 🄵制的，但它們對*🄵態型🄵檢查器 (static type checkers)*很有用，🄵能協助 IDE 完成程式碼的補全 (completion) 和重構 (refactoring)。

全域變數、class 屬性和函式（不含區域變數）的型🄵提示，都可以使用 `typing.get_type_hints()` 來存取。

請參🄵 `typing` 和 **PEP 484**，有此功能的描述。

universal newlines（通用🄵行字元）
一種解譯文字流 (text stream) 的方式，會將以下所有的情🄵識🄵🄵一行的結束：Unix 行尾慣例 `'\n'`、Windows 慣例 `'\r\n'` 和舊的 Macintosh 慣例 `'\r'`。請參🄵 **PEP 278** 和 **PEP 3116**，以及用於 `bytes.splitlines()` 的附加用途。

variable annotation（變數🄵釋）
一個變數或 class 屬性的*annotation*（🄵釋）。

🄵釋變數或 class 屬性時，賦值是選擇性的：

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

變數🄵釋通常用於型🄵提示 *(type hint)*：例如，這個變數預期會取得 `int`（整數）值：

```
count: int = 0
```

變數🄵釋的語法在 annassign 章節有詳細的解釋。

請參🄵*function annotation*（函式🄵釋）、**PEP 484** 和 **PEP 526**，皆有此功能的描述。關於🄵釋的最佳實踐方法，另請參🄵 annotations-howto。

virtual environment（🄵擬環境）
一個協作隔離 (cooperatively isolated) 的執行環境，能讓 Python 的使用者和應用程式得以安裝和升級 Python 發🄵套件，而不會對同一個系統上運行的其他 Python 應用程式的行🄵🄵生干擾。

另請參🄵 `venv`。

virtual machine（🄵擬機器）
一部完全由軟體所定義的電腦 (computer)。Python 的🄵擬機器會執行由*bytecode*（位元組碼）編譯器所發出的位元組碼。

Zen of Python（Python 之🄵）
Python 設計原則與哲學的列表，其🄴容有助於理解和使用此語言。此列表可以透過在互動式提式字元後輸入「`import this`」來找到它。

APPENDIX B

關於這些🄵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 標准函式庫和 Python 🄵明文件貢獻過。Python 所發🄵的原始碼中含有部份貢獻者的清單，請見 Misc/ACKS 。

正因🄵 Python 社群的撰寫與貢獻才有這份這🄵棒的🄵明文件 -- 感謝所有貢獻過的人們！

APPENDIX C

沿革與授權

C.1 軟體沿革

Python 是由荷蘭數學和計算機科學研究學會（CWI，見 https://www.cwi.nl/）的 Guido van Rossum 於 1990 年代早期所創造，目的是作🄵一種稱🄵 ABC 語言的後繼者。🄵管 Python 包含了許多來自其他人的貢獻，Guido 仍是其主要作者。

1995 年，Guido 在維吉尼亞州雷斯頓的國家創新研究公司（CNRI，見 https://www.cnri.reston.va.us/）繼續他在 Python 的工作，🄵在那🄵發🄵了該軟體的多個版本。

2000 年五月，Guido 和 Python 核心開發團隊轉移到 BeOpen.com 🄵成立了 BeOpen PythonLabs 團隊。同年十月，PythonLabs 團隊轉移到 Digital Creations（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🄵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🄵 https://opensource.org/）。歷史上，大多數但非全部的 Python 版本，也是 GPL 相容的；以下表格總結各個版本的差🄵。

發🄵版本	源自	年份	擁有者	GPL 相容性？
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🄵： GPL 相容🄵不表示我們是在 GPL 下發🄵 Python。不像 GPL，所有的 Python 授權都可以讓您發🄵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🄵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🄵成🄵可能。

C.2 關於存取或以其他方式使用 Python 的合約條款

Python 軟體和🄵明文件的授權是基於*PSF 授權合約*。

從 Python 3.8.6 開始，🄵明文件中的範例、程式庫和其他程式碼，是被雙重授權 (dual licensed) 於 PSF 授權合約以及*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🄵被收🄵軟體的授權與致謝。

C.2.1 用於 PYTHON 3.12.3 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
   3.12.3 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 3.12.3 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 3.12.3 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 3.12.3 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
   3.12.3.

4. PSF is making Python 3.12.3 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 3.12.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.12.3
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A␣
↪RESULT OF
```

```
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.12.3, 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 3.12.3, 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
   http://www.pythonlabs.com/logos.html may be used according to the permissions
```

(繼續下一頁)

(繼續上一頁)

```
   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
   does not grant permission to use CNRI trademarks or trade name in a trademark
```

(繼續下一頁)

(繼續上一頁)

```
   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 3.12.3 🄵明文件🄵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🄵軟體的授權與致謝

本節是一個不完整但持續增加的授權與致謝清單，對象是在 Python 發🄵版本中所收🄴的第三方軟體。

C.3.1 Mersenne Twister

random 模組底下的 _random C 擴充程式包含了以 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 的下載🄵容🄴基礎的程式碼。以下是原始程式碼的完整聲明：

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

socket 模組使用 getaddrinfo() 和 getnameinfo() 函式，它們在 WIDE 專案（https://www.wide.ad.jp/）🄴，於不同的原始檔案中被編碼：

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 非同步 socket 服務

test.support.asynchat 和 test.support.asyncore 模組包含以下聲明：

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie 管理

`http.cookies` 模組包含以下聲明：

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 執行追🄵

`trace` 模組包含以下聲明：

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
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.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 遠端程序呼叫

xmlrpc.client 模組包含以下聲明：

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

`test.test_epoll` 模組包含以下聲明：

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

`select` 模組對於 kqueue 介面包含以下聲明：

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

`Python/pyhash.c` 檔案包含 Marek Majkowski' 基於 Dan Bernstein 的 SipHash24 演算法的實作。它包含以下聲明：

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 與 dtoa

`Python/dtoa.c` 檔案提供了 C 的 dtoa 和 strtod 函式，用於將 C 的雙精度浮點數和字串互相轉🄴。該檔案是衍生自 David M. Gay 建立的同名檔案，後者現在可以從 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c 下載。於 2009 年 3 月 16 日所檢索的原始檔案包含以下版權與授權聲明：

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

如果 OpenSSL 函式庫可被作業系統使用，則 hashlib、posix、ssl、crypt 模組會使用它來提升效能。此外，因🄵 Windows 和 macOS 的 Python 安裝程式可能包含 OpenSSL 函式庫的副本，所以我們也在此收🄵 OpenSSL 授權的副本。對於 OpenSSL 3.0 版本以及由此衍生的更新版本則適用 Apache 許可證 v2：

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
```

(繼續下一頁)

(繼續上一頁)

```
   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,
   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
```

(繼續下一頁)

(繼續上一頁)

```
      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.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

除非在建置 pyexpat 擴充時設定🄴 --with-system-expat，否則該擴充會用一個🄴含 expat 原始碼的副本來建置：

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

除非在建置 _ctypes 模組底下 _ctypes 擴充程式時設定🄴 --with-system-libffi，否則該擴充會用一個🄴含 libffi 原始碼的副本來建置：

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

如果在系統上找到的 zlib 版本太舊以致於無法用於建置 `zlib` 擴充，則該擴充會用一個🄴含 zlib 原始碼的副本來建置：

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

`tracemalloc` 使用的雜🄴表 (hash table) 實作，是以 cfuhash 專案🄴基礎：

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
```

(繼續下一頁)

(繼續上一頁)

```
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

除非在建置 `decimal` 模組底下 `_decimal` C 擴充程式時設定🄵 `--with-system-libmpdec`，否則該模組會用一個🄵含 libmpdec 函式庫的副本來建置：

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N 測試套件

`test` 程式包中的 C14N 2.0 測試套件 (`Lib/test/xmltestdata/c14n-20/`) 是從 W3C 網站 https://www.w3.org/TR/xml-c14n2-testcases/ 被檢索，且是基於 3-clause BSD 授權被發🄵：

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.
```

(繼續下一頁)

(繼續上一頁)

```
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 模組使用 SoX 專案的 g771.c 檔案中的程式碼。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, MERCHANTIBILITY 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` 模組的部分🄴容是從 uvloop 0.16 中收🄴過來，其基於 MIT 授權來發🄴:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
```

(繼續下一頁)

(繼續上一頁)

```
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

APPENDIX D

版權宣告

Python 和這份🄴明文件的版權:

完整的授權條款資訊請參見沿革與授權。

索引

非依字母順序

A

B

C

D

E

N

O

P

Q

R

S

T

U

V

W

Z