The Python/C API

發行 3.11.8

Guido van Rossum and the Python development team

4 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[F]記[F]了可使用的 API（應用程式介面）。在 extending-index 中也有相關的[F]容，它描述了編寫擴充的一般原則，但[F][F]有詳細[F]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准標頭：`<stdio.h>`、`<string.h>`、`<errno.h>`、`<limits.h>`、`<assert.h>` 和 `<stdlib.h>`（如果可用）。

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

建議在引入 Python.h 之前都要定義 PY_SSIZE_T_CLEAN。有關此巨集的□明，請參□剖析引數與建置數值。

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

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

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

要引入標頭，請將兩個（如果不同）目□放在編譯器的引入搜索路徑 (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

Declare an extension module PyInit initialization function. The function return type is *PyObject**. The macro declares any special linkage declarations required by the platform, and for C++ declares the function as extern "C".

The initialization function must be named PyInit_*name*, where *name* is the name of the module, and should be the only non-static item defined in the module file. Example:

```
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。

在 3.3 版新加入.

Py_ALWAYS_INLINE

要求編譯器總是嵌入□態行□函式 (static inline function)，編譯器可以忽略它□□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。

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

它必須在函式回傳型別之前被指定。用法：

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

在 3.11 版新加入.

Py_CHARMASK (c)

引數必須是 [-128, 127] 或 [0, 255] 範圍內的字元或整數。這個巨集會將 c 轉換為 unsigned char 並回傳。

Py_DEPRECATED (version)

將其用於已棄用的聲明。巨集必須放在符號名稱之前。

範例：

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

在 3.8 版的變更: 新增了 MSVC 支援。

Py_GETENV (s)

類似於 getenv(s)，但如果在命令列上傳遞了 -E（即如果設定了 Py_IgnoreEnvironmentFlag）則回傳 NULL。

Py_MAX (x, y)

回傳 x 和 y 之間的最大值。

在 3.3 版新加入.

Py_MEMBER_SIZE (type, member)

以位元組為單位回傳結構 (type) member 的大小。

在 3.6 版新加入.

Py_MIN (x, y)

回傳 x 和 y 之間的最小值。

在 3.3 版新加入.

Py_NO_INLINE

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

用法：

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

在 3.11 版新加入.

Py_STRINGIFY (x)

將 x 轉換為 C 字串。例如 Py_STRINGIFY(123) 會回傳 "123"。

在 3.4 版新加入.

Py_UNREACHABLE ()

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

在發布模式 (release mode) 下，巨集幫助編譯器最佳化程式碼，並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並未聲明為 _Py_NO_RETURN 的函式之呼叫後使用。

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

在 3.7 版新加入.

Py_UNUSED (arg)

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

在 3.4 版新加入.

PyDoc_STRVAR (name, str)

建立一個名□ `name` 的變數，可以在文件字串中使用。如果 Python 是在□有文件字串的情□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

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

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

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

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

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

參照計數詳細資訊

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

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，或者沒有。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，並且你不再對它負責。

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！）。因為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");
```

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

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

    n = PyObject_Length(target);
    if (n < 0)
        return -1;
    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，說明如何編寫函式來計算一個整數 list 中項目的總和；一次使用*PyList_GetItem()*，一次使用*PySequence_GetItem()*：

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

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

```
long
sum_sequence(PyObject *sequence)
{
    Py_ssize_t i, n;
    long total = 0, value;
    PyObject *item;
    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 型別

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

type **Py_ssize_t**

Part of the Stable ABI. 一個帶符號的整數型別，使得 sizeof(Py_ssize_t) == sizeof(size_t)。C99 沒有直接定義這樣的東西（size_t 是無符號整數型別）。有關詳細資訊，請參閱 **PEP 353**。PY_SSIZE_T_MAX 是*Py_ssize_t* 型別的最大正值。

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，在那裡它們透過堆疊回溯 (stack trace) 回報給使用者。

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

例外的狀態會在個別執行緒的存儲空間 (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，它回傳對例外型別物件的借用參照，否則回傳 NULL。設定例外狀態的函式有很多：*PyErr_SetString()* 是最常見的（儘管不是最通用的）設定例外狀態的函式，而*PyErr_Clear()* 是用來清除例外狀態。

完整的例外狀態由三個（都可以為 NULL 的）物件組成：例外型別、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；但是它們並不相同：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迴圈，該迴圈負責將它傳遞給 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 程式碼的個別執行緒例外狀態。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；它還替回溯中被堆疊幀 (stack frame) 參照的物件減少了通常不需要的生命週期延長。

作為一般原則，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。它應該丟棄它擁有的任何物件參照，並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，並且丟失關於錯誤確切原因的重要資訊。

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

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

這是相應的 C 程式碼：

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

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

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

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

    if (PyObject_SetItem(dict, key, incremented_item) < 0)
        goto error;
    rv = 0; /* Success */
    /* Continue with cleanup code */

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

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

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

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

1.6 嵌入式 Python

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

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

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

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

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

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

有時會希望能夠「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 的使用並想要釋放 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啟用對直譯器和擴充模組的額外檢查，這些檢查往往會在執行環境 (runtime) 增加大量開銷 (overhead)，因此預設情況下不啟用它們。

Python 原始碼發布版本中的 Misc/SpecialBuilds.txt 檔案有一份包含多種除錯構置的完整列表，為支援追蹤參照計數、為記憶體分配器除錯或對主直譯器迴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

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

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

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

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

CHAPTER 2

C API 穩定性

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

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

帶有底線前綴的名稱是私有 API (private API)，像是 `_Py_InternalState`，即使在補丁版本 (patch release) 中也可能被更改，不會另行通知。

2.1 穩定的應用程式二進位介面

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

2.1.1 Limited C API

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

Py_LIMITED_API

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

Define `Py_LIMITED_API` to the value of *`PY_VERSION_HEX`* corresponding to the lowest Python version your extension supports. The extension will work without recompilation with all Python 3 releases from the specified one onward, and can use Limited API introduced up to that version.

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

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

2.1.2 Stable ABI

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

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

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

在某些平台上，Python 將查找並加載以 `abi3` 標籤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 建置的擴充不會被較低版本的 Python 所安裝。

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

2.1.3 受限 API 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()* 則不可用。巨集運行可以更快，因為它可以依賴 list 物件的特定版本實作細節。

如果沒有定義 `Py_LIMITED_API`，一些 C API 函式將被嵌入或被替換為巨集。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` 編譯將產生一個擴充，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)。

2.1.4 受限 API 注意事項

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

`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` 作為引數的函式。在 Python 3.9 中，`NULL` 現在代表選擇預設行為，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，並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為技術原因（或者甚至是無意地，例如臭蟲）而被公開出來。

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

2.2 平台注意事項

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

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 版本就是這種情況。

2.3 受限 API 的內容

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

- *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_ExceptionMatches()*
- *PyErr_Fetch()*
- *PyErr_Format()*
- *PyErr_FormatV()*
- *PyErr_GetExcInfo()*
- *PyErr_GetHandledException()*
- *PyErr_GivenExceptionMatches()*
- *PyErr_NewException()*
- *PyErr_NewExceptionWithDoc()*
- *PyErr_NoMemory()*
- *PyErr_NormalizeException()*
- *PyErr_Occurred()*

- *PyErr_Print()*
- *PyErr_PrintEx()*
- PyErr_ProgramText()
- *PyErr_ResourceWarning()*
- *PyErr_Restore()*
- *PyErr_SetExcFromWindowsErr()*
- *PyErr_SetExcFromWindowsErrWithFilename()*
- *PyErr_SetExcFromWindowsErrWithFilenameObject()*
- *PyErr_SetExcFromWindowsErrWithFilenameObjects()*
- *PyErr_SetExcInfo()*
- *PyErr_SetFromErrno()*
- *PyErr_SetFromErrnoWithFilename()*
- *PyErr_SetFromErrnoWithFilenameObject()*
- *PyErr_SetFromErrnoWithFilenameObjects()*
- *PyErr_SetFromWindowsErr()*
- *PyErr_SetFromWindowsErrWithFilename()*
- *PyErr_SetHandledException()*
- *PyErr_SetImportError()*
- *PyErr_SetImportErrorSubclass()*
- *PyErr_SetInterrupt()*
- *PyErr_SetInterruptEx()*
- *PyErr_SetNone()*
- *PyErr_SetObject()*
- *PyErr_SetString()*
- *PyErr_SyntaxLocation()*
- *PyErr_SyntaxLocationEx()*
- *PyErr_WarnEx()*
- *PyErr_WarnExplicit()*
- *PyErr_WarnFormat()*
- *PyErr_WriteUnraisable()*
- *PyEval_AcquireLock()*
- *PyEval_AcquireThread()*
- PyEval_CallFunction()
- PyEval_CallMethod()
- PyEval_CallObjectWithKeywords()
- *PyEval_EvalCode()*
- *PyEval_EvalCodeEx()*
- *PyEval_EvalFrame()*
- *PyEval_EvalFrameEx()*

- *PyEval_GetBuiltins()*
- *PyEval_GetFrame()*
- *PyEval_GetFuncDesc()*
- *PyEval_GetFuncName()*
- *PyEval_GetGlobals()*
- *PyEval_GetLocals()*
- *PyEval_InitThreads()*
- *PyEval_ReleaseLock()*
- *PyEval_ReleaseThread()*
- *PyEval_RestoreThread()*
- *PyEval_SaveThread()*
- *PyEval_ThreadsInitialized()*
- PyExc_ArithmeticError
- PyExc_AssertionError
- PyExc_AttributeError
- PyExc_BaseException
- PyExc_BaseExceptionGroup
- PyExc_BlockingIOError
- PyExc_BrokenPipeError
- PyExc_BufferError
- PyExc_BytesWarning
- PyExc_ChildProcessError
- PyExc_ConnectionAbortedError
- PyExc_ConnectionError
- PyExc_ConnectionRefusedError
- PyExc_ConnectionResetError
- PyExc_DeprecationWarning
- PyExc_EOFError
- PyExc_EncodingWarning
- PyExc_EnvironmentError
- PyExc_Exception
- PyExc_FileExistsError
- PyExc_FileNotFoundError
- PyExc_FloatingPointError
- PyExc_FutureWarning
- PyExc_GeneratorExit
- PyExc_IOError
- PyExc_ImportError
- PyExc_ImportWarning

- PyExc_IndentationError
- PyExc_IndexError
- PyExc_InterruptedError
- PyExc_IsADirectoryError
- PyExc_KeyError
- PyExc_KeyboardInterrupt
- PyExc_LookupError
- PyExc_MemoryError
- PyExc_ModuleNotFoundError
- PyExc_NameError
- PyExc_NotADirectoryError
- PyExc_NotImplementedError
- PyExc_OSError
- PyExc_OverflowError
- PyExc_PendingDeprecationWarning
- PyExc_PermissionError
- PyExc_ProcessLookupError
- PyExc_RecursionError
- PyExc_ReferenceError
- PyExc_ResourceWarning
- PyExc_RuntimeError
- PyExc_RuntimeWarning
- PyExc_StopAsyncIteration
- PyExc_StopIteration
- PyExc_SyntaxError
- PyExc_SyntaxWarning
- PyExc_SystemError
- PyExc_SystemExit
- PyExc_TabError
- PyExc_TimeoutError
- PyExc_TypeError
- PyExc_UnboundLocalError
- PyExc_UnicodeDecodeError
- PyExc_UnicodeEncodeError
- PyExc_UnicodeError
- PyExc_UnicodeTranslateError
- PyExc_UnicodeWarning
- PyExc_UserWarning
- PyExc_ValueError

- PyExc_Warning
- PyExc_WindowsError
- PyExc_ZeroDivisionError
- PyExceptionClass_Name()
- *PyException_GetCause()*
- *PyException_GetContext()*
- *PyException_GetTraceback()*
- *PyException_SetCause()*
- *PyException_SetContext()*
- *PyException_SetTraceback()*
- *PyFile_FromFd()*
- *PyFile_GetLine()*
- *PyFile_WriteObject()*
- *PyFile_WriteString()*
- PyFilter_Type
- *PyFloat_AsDouble()*
- *PyFloat_FromDouble()*
- *PyFloat_FromString()*
- *PyFloat_GetInfo()*
- *PyFloat_GetMax()*
- *PyFloat_GetMin()*
- *PyFloat_Type*
- *PyFrameObject*
- *PyFrame_GetCode()*
- *PyFrame_GetLineNumber()*
- *PyFrozenSet_New()*
- *PyFrozenSet_Type*
- *PyGC_Collect()*
- *PyGC_Disable()*
- *PyGC_Enable()*
- *PyGC_IsEnabled()*
- *PyGILState_Ensure()*
- *PyGILState_GetThisThreadState()*
- *PyGILState_Release()*
- PyGILState_STATE
- *PyGetSetDef*
- PyGetSetDescr_Type
- *PyImport_AddModule()*
- *PyImport_AddModuleObject()*

- *PyImport_AppendInittab()*
- *PyImport_ExecCodeModule()*
- *PyImport_ExecCodeModuleEx()*
- *PyImport_ExecCodeModuleObject()*
- *PyImport_ExecCodeModuleWithPathnames()*
- *PyImport_GetImporter()*
- *PyImport_GetMagicNumber()*
- *PyImport_GetMagicTag()*
- *PyImport_GetModule()*
- *PyImport_GetModuleDict()*
- *PyImport_Import()*
- *PyImport_ImportFrozenModule()*
- *PyImport_ImportFrozenModuleObject()*
- *PyImport_ImportModule()*
- *PyImport_ImportModuleLevel()*
- *PyImport_ImportModuleLevelObject()*
- *PyImport_ImportModuleNoBlock()*
- *PyImport_ReloadModule()*
- *PyIndex_Check()*
- *PyInterpreterState*
- *PyInterpreterState_Clear()*
- *PyInterpreterState_Delete()*
- *PyInterpreterState_Get()*
- *PyInterpreterState_GetDict()*
- *PyInterpreterState_GetID()*
- *PyInterpreterState_New()*
- *PyIter_Check()*
- *PyIter_Next()*
- *PyIter_Send()*
- PyListIter_Type
- PyListRevIter_Type
- *PyList_Append()*
- *PyList_AsTuple()*
- *PyList_GetItem()*
- *PyList_GetSlice()*
- *PyList_Insert()*
- *PyList_New()*
- *PyList_Reverse()*
- *PyList_SetItem()*

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

- *PyMapping_Values()*
- *PyMem_Calloc()*
- *PyMem_Free()*
- *PyMem_Malloc()*
- *PyMem_Realloc()*
- *PyMemberDef*
- PyMemberDescr_Type
- *PyMemoryView_FromBuffer()*
- *PyMemoryView_FromMemory()*
- *PyMemoryView_FromObject()*
- *PyMemoryView_GetContiguous()*
- PyMemoryView_Type
- *PyMethodDef*
- PyMethodDescr_Type
- *PyModuleDef*
- PyModuleDef_Base
- *PyModuleDef_Init()*
- PyModuleDef_Type
- *PyModule_AddFunctions()*
- *PyModule_AddIntConstant()*
- *PyModule_AddObject()*
- *PyModule_AddObjectRef()*
- *PyModule_AddStringConstant()*
- *PyModule_AddType()*
- *PyModule_Create2()*
- *PyModule_ExecDef()*
- *PyModule_FromDefAndSpec2()*
- *PyModule_GetDef()*
- *PyModule_GetDict()*
- *PyModule_GetFilename()*
- *PyModule_GetFilenameObject()*
- *PyModule_GetName()*
- *PyModule_GetNameObject()*
- *PyModule_GetState()*
- *PyModule_New()*
- *PyModule_NewObject()*
- *PyModule_SetDocString()*
- *PyModule_Type*
- *PyNumber_Absolute()*

- *PyNumber_Add()*
- *PyNumber_And()*
- *PyNumber_AsSsize_t()*
- *PyNumber_Check()*
- *PyNumber_Divmod()*
- *PyNumber_Float()*
- *PyNumber_FloorDivide()*
- *PyNumber_InPlaceAdd()*
- *PyNumber_InPlaceAnd()*
- *PyNumber_InPlaceFloorDivide()*
- *PyNumber_InPlaceLshift()*
- *PyNumber_InPlaceMatrixMultiply()*
- *PyNumber_InPlaceMultiply()*
- *PyNumber_InPlaceOr()*
- *PyNumber_InPlacePower()*
- *PyNumber_InPlaceRemainder()*
- *PyNumber_InPlaceRshift()*
- *PyNumber_InPlaceSubtract()*
- *PyNumber_InPlaceTrueDivide()*
- *PyNumber_InPlaceXor()*
- *PyNumber_Index()*
- *PyNumber_Invert()*
- *PyNumber_Long()*
- *PyNumber_Lshift()*
- *PyNumber_MatrixMultiply()*
- *PyNumber_Multiply()*
- *PyNumber_Negative()*
- *PyNumber_Or()*
- *PyNumber_Positive()*
- *PyNumber_Power()*
- *PyNumber_Remainder()*
- *PyNumber_Rshift()*
- *PyNumber_Subtract()*
- *PyNumber_ToBase()*
- *PyNumber_TrueDivide()*
- *PyNumber_Xor()*
- *PyOS_AfterFork()*
- *PyOS_AfterFork_Child()*
- *PyOS_AfterFork_Parent()*

- *PyOS_BeforeFork()*
- *PyOS_CheckStack()*
- *PyOS_FSPath()*
- *PyOS_InputHook*
- PyOS_InterruptOccurred()
- *PyOS_double_to_string()*
- *PyOS_getsig()*
- PyOS_mystricmp()
- PyOS_mystrnicmp()
- *PyOS_setsig()*
- *PyOS_sighandler_t*
- *PyOS_snprintf()*
- *PyOS_string_to_double()*
- *PyOS_strtol()*
- *PyOS_strtoul()*
- *PyOS_vsnprintf()*
- *PyObject*
- *PyObject.ob_refcnt*
- *PyObject.ob_type*
- *PyObject_ASCII()*
- *PyObject_AsCharBuffer()*
- *PyObject_AsFileDescriptor()*
- *PyObject_AsReadBuffer()*
- *PyObject_AsWriteBuffer()*
- *PyObject_Bytes()*
- *PyObject_Call()*
- *PyObject_CallFunction()*
- *PyObject_CallFunctionObjArgs()*
- *PyObject_CallMethod()*
- *PyObject_CallMethodObjArgs()*
- *PyObject_CallNoArgs()*
- *PyObject_CallObject()*
- *PyObject_Calloc()*
- *PyObject_CheckBuffer()*
- *PyObject_CheckReadBuffer()*
- *PyObject_ClearWeakRefs()*
- *PyObject_CopyData()*
- *PyObject_DelItem()*
- PyObject_DelItemString()

- *PyObject_Dir()*
- *PyObject_Format()*
- *PyObject_Free()*
- *PyObject_GC_Del()*
- *PyObject_GC_IsFinalized()*
- *PyObject_GC_IsTracked()*
- *PyObject_GC_Track()*
- *PyObject_GC_UnTrack()*
- *PyObject_GenericGetAttr()*
- *PyObject_GenericGetDict()*
- *PyObject_GenericSetAttr()*
- *PyObject_GenericSetDict()*
- *PyObject_GetAIter()*
- *PyObject_GetAttr()*
- *PyObject_GetAttrString()*
- *PyObject_GetBuffer()*
- *PyObject_GetItem()*
- *PyObject_GetIter()*
- *PyObject_HasAttr()*
- *PyObject_HasAttrString()*
- *PyObject_Hash()*
- *PyObject_HashNotImplemented()*
- *PyObject_Init()*
- *PyObject_InitVar()*
- *PyObject_IsInstance()*
- *PyObject_IsSubclass()*
- *PyObject_IsTrue()*
- *PyObject_Length()*
- *PyObject_Malloc()*
- *PyObject_Not()*
- *PyObject_Realloc()*
- *PyObject_Repr()*
- *PyObject_RichCompare()*
- *PyObject_RichCompareBool()*
- PyObject_SelfIter()
- *PyObject_SetAttr()*
- *PyObject_SetAttrString()*
- *PyObject_SetItem()*
- *PyObject_Size()*

- *PyObject_Str()*
- *PyObject_Type()*
- *PyProperty_Type*
- PyRangeIter_Type
- PyRange_Type
- PyReversed_Type
- *PySeqIter_New()*
- *PySeqIter_Type*
- *PySequence_Check()*
- *PySequence_Concat()*
- *PySequence_Contains()*
- *PySequence_Count()*
- *PySequence_DelItem()*
- *PySequence_DelSlice()*
- *PySequence_Fast()*
- *PySequence_GetItem()*
- *PySequence_GetSlice()*
- PySequence_In()
- *PySequence_InPlaceConcat()*
- *PySequence_InPlaceRepeat()*
- *PySequence_Index()*
- *PySequence_Length()*
- *PySequence_List()*
- *PySequence_Repeat()*
- *PySequence_SetItem()*
- *PySequence_SetSlice()*
- *PySequence_Size()*
- *PySequence_Tuple()*
- PySetIter_Type
- *PySet_Add()*
- *PySet_Clear()*
- *PySet_Contains()*
- *PySet_Discard()*
- *PySet_New()*
- *PySet_Pop()*
- *PySet_Size()*
- *PySet_Type*
- *PySlice_AdjustIndices()*
- *PySlice_GetIndices()*

- *PySlice_GetIndicesEx()*
- *PySlice_New()*
- *PySlice_Type*
- *PySlice_Unpack()*
- *PyState_AddModule()*
- *PyState_FindModule()*
- *PyState_RemoveModule()*
- *PyStructSequence_Desc*
- *PyStructSequence_Field*
- *PyStructSequence_GetItem()*
- *PyStructSequence_New()*
- *PyStructSequence_NewType()*
- *PyStructSequence_SetItem()*
- *PyStructSequence_UnnamedField*
- PySuper_Type
- *PySys_AddWarnOption()*
- *PySys_AddWarnOptionUnicode()*
- *PySys_AddXOption()*
- *PySys_FormatStderr()*
- *PySys_FormatStdout()*
- *PySys_GetObject()*
- *PySys_GetXOptions()*
- PySys_HasWarnOptions()
- *PySys_ResetWarnOptions()*
- *PySys_SetArgv()*
- *PySys_SetArgvEx()*
- *PySys_SetObject()*
- *PySys_SetPath()*
- *PySys_WriteStderr()*
- *PySys_WriteStdout()*
- *PyThreadState*
- *PyThreadState_Clear()*
- *PyThreadState_Delete()*
- *PyThreadState_Get()*
- *PyThreadState_GetDict()*
- *PyThreadState_GetFrame()*
- *PyThreadState_GetID()*
- *PyThreadState_GetInterpreter()*
- *PyThreadState_New()*

- *PyThreadState_SetAsyncExc()*
- *PyThreadState_Swap()*
- PyThread_GetInfo()
- *PyThread_ReInitTLS()*
- PyThread_acquire_lock()
- PyThread_acquire_lock_timed()
- PyThread_allocate_lock()
- *PyThread_create_key()*
- *PyThread_delete_key()*
- *PyThread_delete_key_value()*
- PyThread_exit_thread()
- PyThread_free_lock()
- *PyThread_get_key_value()*
- PyThread_get_stacksize()
- PyThread_get_thread_ident()
- PyThread_get_thread_native_id()
- PyThread_init_thread()
- PyThread_release_lock()
- *PyThread_set_key_value()*
- PyThread_set_stacksize()
- PyThread_start_new_thread()
- *PyThread_tss_alloc()*
- *PyThread_tss_create()*
- *PyThread_tss_delete()*
- *PyThread_tss_free()*
- *PyThread_tss_get()*
- *PyThread_tss_is_created()*
- *PyThread_tss_set()*
- PyTraceBack_Here()
- PyTraceBack_Print()
- PyTraceBack_Type
- PyTupleIter_Type
- *PyTuple_GetItem()*
- *PyTuple_GetSlice()*
- *PyTuple_New()*
- *PyTuple_Pack()*
- *PyTuple_SetItem()*
- *PyTuple_Size()*
- *PyTuple_Type*

- *PyTypeObject*
- *PyType_ClearCache()*
- *PyType_FromModuleAndSpec()*
- *PyType_FromSpec()*
- *PyType_FromSpecWithBases()*
- *PyType_GenericAlloc()*
- *PyType_GenericNew()*
- *PyType_GetFlags()*
- *PyType_GetModule()*
- *PyType_GetModuleState()*
- *PyType_GetName()*
- *PyType_GetQualName()*
- *PyType_GetSlot()*
- *PyType_IsSubtype()*
- *PyType_Modified()*
- *PyType_Ready()*
- *PyType_Slot*
- *PyType_Spec*
- *PyType_Type*
- *PyUnicodeDecodeError_Create()*
- *PyUnicodeDecodeError_GetEncoding()*
- *PyUnicodeDecodeError_GetEnd()*
- *PyUnicodeDecodeError_GetObject()*
- *PyUnicodeDecodeError_GetReason()*
- *PyUnicodeDecodeError_GetStart()*
- *PyUnicodeDecodeError_SetEnd()*
- *PyUnicodeDecodeError_SetReason()*
- *PyUnicodeDecodeError_SetStart()*
- *PyUnicodeEncodeError_GetEncoding()*
- *PyUnicodeEncodeError_GetEnd()*
- *PyUnicodeEncodeError_GetObject()*
- *PyUnicodeEncodeError_GetReason()*
- *PyUnicodeEncodeError_GetStart()*
- *PyUnicodeEncodeError_SetEnd()*
- *PyUnicodeEncodeError_SetReason()*
- *PyUnicodeEncodeError_SetStart()*
- PyUnicodeIter_Type
- *PyUnicodeTranslateError_GetEnd()*
- *PyUnicodeTranslateError_GetObject()*

- *PyUnicodeTranslateError_GetReason()*
- *PyUnicodeTranslateError_GetStart()*
- *PyUnicodeTranslateError_SetEnd()*
- *PyUnicodeTranslateError_SetReason()*
- *PyUnicodeTranslateError_SetStart()*
- PyUnicode_Append()
- PyUnicode_AppendAndDel()
- *PyUnicode_AsASCIIString()*
- *PyUnicode_AsCharmapString()*
- PyUnicode_AsDecodedObject()
- PyUnicode_AsDecodedUnicode()
- PyUnicode_AsEncodedObject()
- *PyUnicode_AsEncodedString()*
- PyUnicode_AsEncodedUnicode()
- *PyUnicode_AsLatin1String()*
- *PyUnicode_AsMBCSString()*
- *PyUnicode_AsRawUnicodeEscapeString()*
- *PyUnicode_AsUCS4()*
- *PyUnicode_AsUCS4Copy()*
- *PyUnicode_AsUTF16String()*
- *PyUnicode_AsUTF32String()*
- *PyUnicode_AsUTF8AndSize()*
- *PyUnicode_AsUTF8String()*
- *PyUnicode_AsUnicodeEscapeString()*
- *PyUnicode_AsWideChar()*
- *PyUnicode_AsWideCharString()*
- PyUnicode_BuildEncodingMap()
- *PyUnicode_Compare()*
- *PyUnicode_CompareWithASCIIString()*
- *PyUnicode_Concat()*
- *PyUnicode_Contains()*
- *PyUnicode_Count()*
- *PyUnicode_Decode()*
- *PyUnicode_DecodeASCII()*
- *PyUnicode_DecodeCharmap()*
- PyUnicode_DecodeCodePageStateful()
- *PyUnicode_DecodeFSDefault()*
- *PyUnicode_DecodeFSDefaultAndSize()*
- *PyUnicode_DecodeLatin1()*

- *PyUnicode_DecodeLocale()*
- *PyUnicode_DecodeLocaleAndSize()*
- *PyUnicode_DecodeMBCS()*
- *PyUnicode_DecodeMBCSStateful()*
- *PyUnicode_DecodeRawUnicodeEscape()*
- *PyUnicode_DecodeUTF16()*
- *PyUnicode_DecodeUTF16Stateful()*
- *PyUnicode_DecodeUTF32()*
- *PyUnicode_DecodeUTF32Stateful()*
- *PyUnicode_DecodeUTF7()*
- *PyUnicode_DecodeUTF7Stateful()*
- *PyUnicode_DecodeUTF8()*
- *PyUnicode_DecodeUTF8Stateful()*
- *PyUnicode_DecodeUnicodeEscape()*
- *PyUnicode_EncodeCodePage()*
- *PyUnicode_EncodeFSDefault()*
- *PyUnicode_EncodeLocale()*
- *PyUnicode_FSConverter()*
- *PyUnicode_FSDecoder()*
- *PyUnicode_Find()*
- *PyUnicode_FindChar()*
- *PyUnicode_Format()*
- *PyUnicode_FromEncodedObject()*
- *PyUnicode_FromFormat()*
- *PyUnicode_FromFormatV()*
- *PyUnicode_FromObject()*
- PyUnicode_FromOrdinal()
- *PyUnicode_FromString()*
- *PyUnicode_FromStringAndSize()*
- *PyUnicode_FromWideChar()*
- PyUnicode_GetDefaultEncoding()
- *PyUnicode_GetLength()*
- *PyUnicode_GetSize()*
- *PyUnicode_InternFromString()*
- PyUnicode_InternImmortal()
- *PyUnicode_InternInPlace()*
- *PyUnicode_IsIdentifier()*
- *PyUnicode_Join()*
- PyUnicode_Partition()

- PyUnicode_RPartition()
- PyUnicode_RSplit()
- *PyUnicode_ReadChar()*
- *PyUnicode_Replace()*
- PyUnicode_Resize()
- *PyUnicode_RichCompare()*
- *PyUnicode_Split()*
- *PyUnicode_Splitlines()*
- *PyUnicode_Substring()*
- *PyUnicode_Tailmatch()*
- *PyUnicode_Translate()*
- *PyUnicode_Type*
- *PyUnicode_WriteChar()*
- *PyVarObject*
- PyVarObject.ob_base
- *PyVarObject.ob_size*
- PyWeakReference
- *PyWeakref_GetObject()*
- *PyWeakref_NewProxy()*
- *PyWeakref_NewRef()*
- PyWrapperDescr_Type
- *PyWrapper_New()*
- PyZip_Type
- *Py_AddPendingCall()*
- *Py_AtExit()*
- *Py_BEGIN_ALLOW_THREADS*
- *Py_BLOCK_THREADS*
- *Py_BuildValue()*
- *Py_BytesMain()*
- *Py_CompileString()*
- *Py_DecRef()*
- *Py_DecodeLocale()*
- *Py_END_ALLOW_THREADS*
- *Py_EncodeLocale()*
- *Py_EndInterpreter()*
- *Py_EnterRecursiveCall()*
- *Py_Exit()*
- *Py_FatalError()*
- Py_FileSystemDefaultEncodeErrors

- Py_FileSystemDefaultEncoding
- *Py_Finalize()*
- *Py_FinalizeEx()*
- *Py_GenericAlias()*
- *Py_GenericAliasType*
- *Py_GetBuildInfo()*
- *Py_GetCompiler()*
- *Py_GetCopyright()*
- *Py_GetExecPrefix()*
- *Py_GetPath()*
- *Py_GetPlatform()*
- *Py_GetPrefix()*
- *Py_GetProgramFullPath()*
- *Py_GetProgramName()*
- *Py_GetPythonHome()*
- Py_GetRecursionLimit()
- *Py_GetVersion()*
- Py_HasFileSystemDefaultEncoding
- *Py_IncRef()*
- *Py_Initialize()*
- *Py_InitializeEx()*
- *Py_Is()*
- *Py_IsFalse()*
- *Py_IsInitialized()*
- *Py_IsNone()*
- *Py_IsTrue()*
- *Py_LeaveRecursiveCall()*
- *Py_Main()*
- Py_MakePendingCalls()
- *Py_NewInterpreter()*
- *Py_NewRef()*
- *Py_ReprEnter()*
- *Py_ReprLeave()*
- *Py_SetPath()*
- *Py_SetProgramName()*
- *Py_SetPythonHome()*
- Py_SetRecursionLimit()
- *Py_UCS4*
- *Py_UNBLOCK_THREADS*

- Py_UTF8Mode
- *Py_VaBuildValue()*
- *Py_Version*
- *Py_XNewRef()*
- *Py_buffer*
- Py_intptr_t
- *Py_ssize_t*
- Py_uintptr_t
- *allocfunc*
- *binaryfunc*
- *descrgetfunc*
- *descrsetfunc*
- *destructor*
- *getattrfunc*
- *getattrofunc*
- *getiterfunc*
- getter
- *hashfunc*
- *initproc*
- *inquiry*
- *iternextfunc*
- *lenfunc*
- *newfunc*
- *objobjargproc*
- *objobjproc*
- *reprfunc*
- *richcmpfunc*
- *setattrfunc*
- *setattrofunc*
- setter
- *ssizeargfunc*
- *ssizeobjargproc*
- ssizessizeargfunc
- ssizessizeobjargproc
- symtable
- *ternaryfunc*
- *traverseproc*
- *unaryfunc*
- *visitproc*

CHAPTER 3

The Very High Level Layer

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

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

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

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

Part of the Stable ABI. The main program for the standard interpreter. This is made available for programs which embed Python. The *argc* and *argv* parameters should be prepared exactly as those which are passed to a C program's `main()` function (converted to wchar_t according to the user's locale). It is important to note that the argument list may be modified (but the contents of the strings pointed to by the argument list are not). The return value will be `0` if the interpreter exits normally (i.e., without an exception), `1` if the interpreter exits due to an exception, or `2` if the parameter list does not represent a valid Python command line.

Note that if an otherwise unhandled `SystemExit` is raised, this function will not return `1`, but exit the process, as long as `Py_InspectFlag` is not set.

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

Part of the Stable ABI *since version 3.8.* Similar to `Py_Main()` but *argv* is an array of bytes strings.

在 3.8 版新加入.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

---

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

回傳值：新的參照。*Part of the* Stable ABI. This is a simplified interface to `Py_CompileStringFlags()` below, leaving *flags* set to `NULL`.

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

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

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

回傳值：新的參照。 Parse and compile the Python source code in *str*, returning the resulting code object. The start token is given by *start*; this can be used to constrain the code which can be compiled and should be `Py_eval_input`, `Py_file_input`, or `Py_single_input`. The filename specified by *filename* is used to construct the code object and may appear in tracebacks or `SyntaxError` exception messages. This returns `NULL` if the code cannot be parsed or compiled.

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

在 3.4 版新加入.

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

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

在 3.2 版新加入.

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

回傳值：新的參照。 *Part of the* Stable ABI. This is a simplified interface to `PyEval_EvalCodeEx()`, with just the code object, and global and local variables. The other arguments are set to `NULL`.

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

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

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

回傳值：新的參照。 *Part of the* Stable ABI. Evaluate an execution frame. This is a simplified interface to `PyEval_EvalFrameEx()`, for backward compatibility.

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

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

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

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

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

int **Py_eval_input**

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

int **Py_file_input**

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

int **Py_single_input**

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

struct **PyCompilerFlags**

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

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

int **cf_flags**

Compiler flags.

int **cf_feature_version**

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

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

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

int **CO_FUTURE_DIVISION**

This bit can be set in *flags* to cause division operator `/` to be interpreted as "true division" according to **PEP 238**.

CHAPTER 4

參照計數

本節中的巨集用於管理 Python 物件的參照計數。

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

Indicate taking a new *strong reference* to object *o*, indicating it is in use and should not be destroyed.

此函式通常用於將借用參照原地 (in-place) 轉換為強參照。*Py_NewRef()* 函式可用於建立新的強參照。

When done using the object, release it by calling *Py_DECREF()*.

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

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

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

Similar to *Py_INCREF()*, but the object *o* can be NULL, in which case this has no effect.

另請見*Py_XNewRef()*。

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

Part of the Stable ABI *since version 3.10.* Create a new *strong reference* to an object: call *Py_INCREF()* on *o* and return the object *o*.

When the *strong reference* is no longer needed, *Py_DECREF()* should be called on it to release the reference.

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

舉例來說：

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

可以寫成：

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

另請參閱*Py_INCREF()*。

在 3.10 版新加入.

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

Part of the Stable ABI *since version 3.10.* 與 *Py_NewRef()* 類似，但物件 *o* 可以䈀 NULL。

如果物件 *o* 䈀 NULL，則該函式僅回傳 NULL。

在 3.10 版新加入.

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

Release a *strong reference* to object *o*, indicating the reference is no longer used.

Once the last *strong reference* is released (i.e. the object's reference count reaches 0), the object's type's deallocation function (which must not be NULL) is invoked.

此函式通常用於在退出作用域之前䈀除䈀参照。

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

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

> **警告：** The deallocation function can cause arbitrary Python code to be invoked (e.g. when a class instance with a `__del__()` method is deallocated). While exceptions in such code are not propagated, the executed code has free access to all Python global variables. This means that any object that is reachable from a global variable should be in a consistent state before *Py_DECREF()* is invoked. For example, code to delete an object from a list should copy a reference to the deleted object in a temporary variable, update the list data structure, and then call *Py_DECREF()* for the temporary variable.

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

Similar to *Py_DECREF()*, but the object *o* can be NULL, in which case this has no effect. The same warning from *Py_DECREF()* applies here as well.

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

Release a *strong reference* for object *o*. The object may be NULL, in which case the macro has no effect; otherwise the effect is the same as for *Py_DECREF()*, except that the argument is also set to NULL. The warning for *Py_DECREF()* does not apply with respect to the object passed because the macro carefully uses a temporary variable and sets the argument to NULL before releasing the reference.

It is a good idea to use this macro whenever releasing a reference to an object that might be traversed during garbage collection.

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

Part of the Stable ABI. Indicate taking a new *strong reference* to object *o*. A function version of *Py_XINCREF()*. It can be used for runtime dynamic embedding of Python.

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

Part of the Stable ABI. Release a *strong reference* to object *o*. A function version of *Py_XDECREF()*. It can be used for runtime dynamic embedding of Python.

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

CHAPTER 5

例外處理

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

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

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

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

5.1 Printing and clearing

void **`PyErr_Clear`**()

Part of the Stable ABI. Clear the error indicator. If the error indicator is not set, there is no effect.

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

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

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

If *set_sys_last_vars* is nonzero, the variables `sys.last_type`, `sys.last_value` and `sys.last_traceback` will be set to the type, value and traceback of the printed exception, respectively.

void **PyErr_Print**()

Part of the Stable ABI. `PyErr_PrintEx(1)` 的F名。

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

Part of the Stable ABI. Call `sys.unraisablehook()` using the current exception and *obj* argument.

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

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

An exception must be set when calling this function.

5.2 Raising exceptions

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

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

Part of the Stable ABI. This is the most common way to set the error indicator. The first argument specifies the exception type; it is normally one of the standard exceptions, e.g. `PyExc_RuntimeError`. You need not create a new *strong reference* to it (e.g. with *Py_INCREF()*). The second argument is an error message; it is decoded from `'utf-8'`.

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

Part of the Stable ABI. This function is similar to *PyErr_SetString()* but lets you specify an arbitrary Python object for the "value" of the exception.

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

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

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

回傳值：總是F *NULL*。 *Part of the* Stable ABI *since version 3.5.* Same as *PyErr_Format()*, but taking a `va_list` argument rather than a variable number of arguments.

在 3.5 版新加入.

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

Part of the Stable ABI. This is a shorthand for `PyErr_SetObject(type, Py_None)`.

int **PyErr_BadArgument**()

Part of the Stable ABI. This is a shorthand for `PyErr_SetString(PyExc_TypeError, message)`, where *message* indicates that a built-in operation was invoked with an illegal argument. It is mostly for internal use.

PyObject ***PyErr_NoMemory**()

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

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

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

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

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

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

回傳值：總是為 *NULL*。*Part of the* Stable ABI *since version 3.7.* Similar to `PyErr_SetFromErrnoWithFilenameObject()`, but takes a second filename object, for raising errors when a function that takes two filenames fails.

在 3.4 版新加入.

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

回傳值：總是為 *NULL*。*Part of the* Stable ABI. Similar to `PyErr_SetFromErrnoWithFilenameObject()`, but the filename is given as a C string. *filename* is decoded from the *filesystem encoding and error handler*.

PyObject ***PyErr_SetFromWindowsErr***(int ierr)

回傳值：總是為 *NULL*。*Part of the* Stable ABI *on Windows since version 3.7.* This is a convenience function to raise `WindowsError`. If called with *ierr* of `0`, the error code returned by a call to `GetLastError()` is used instead. It calls the Win32 function `FormatMessage()` to retrieve the Windows description of error code given by *ierr* or `GetLastError()`, then it constructs a tuple object whose first item is the *ierr* value and whose second item is the corresponding error message (gotten from `FormatMessage()`), and then calls `PyErr_SetObject(PyExc_WindowsError, object)`. This function always returns `NULL`.

適用：Windows。

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

回傳值：總是為 *NULL*。*Part of the* Stable ABI *on Windows since version 3.7.* Similar to `PyErr_SetFromWindowsErr()`, with an additional parameter specifying the exception type to be raised.

適用：Windows。

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

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

適用：Windows。

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

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

適用：Windows。

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

回傳值：總是為 *NULL*。 *Part of the* Stable ABI *on Windows since version 3.7.* Similar to *PyErr_SetExcFromWindowsErrWithFilenameObject()*, but accepts a second filename object.

適用：Windows。

在 3.4 版新加入.

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

回傳值：總是為 *NULL*。 *Part of the* Stable ABI *on Windows since version 3.7.* Similar to *PyErr_SetFromWindowsErrWithFilename()*, with an additional parameter specifying the exception type to be raised.

適用：Windows。

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

回傳值：總是為 *NULL*。 *Part of the* Stable ABI *since version 3.7.* This is a convenience function to raise `ImportError`. *msg* will be set as the exception's message string. *name* and *path*, both of which can be `NULL`, will be set as the `ImportError`'s respective `name` and `path` attributes.

在 3.3 版新加入.

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

回傳值：總是為 *NULL*。*Part of the* Stable ABI *since version 3.6.* Much like *PyErr_SetImportError()* but this function allows for specifying a subclass of `ImportError` to raise.

在 3.6 版新加入.

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

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

在 3.4 版新加入.

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

Part of the Stable ABI *since version 3.7.* Like *PyErr_SyntaxLocationObject()*, but *filename* is a byte string decoded from the *filesystem encoding and error handler*.

在 3.2 版新加入.

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

Part of the Stable ABI. Like *PyErr_SyntaxLocationEx()*, but the *col_offset* parameter is omitted.

void **PyErr_BadInternalCall**()

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

5.3 Issuing warnings

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

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

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

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

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

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

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

在 3.4 版新加入.

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

Part of the Stable ABI. Similar to `PyErr_WarnExplicitObject()` except that *message* and *module* are UTF-8 encoded strings, and *filename* is decoded from the *filesystem encoding and error handler*.

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

Part of the Stable ABI. Function similar to `PyErr_WarnEx()`, but use `PyUnicode_FromFormat()` to format the warning message. *format* is an ASCII-encoded string.

在 3.2 版新加入.

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

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

在 3.6 版新加入.

5.4 Querying the error indicator

PyObject ***PyErr_Occurred**()

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

The caller must hold the GIL.

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

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

Part of the Stable ABI. Equivalent to `PyErr_GivenExceptionMatches(PyErr_Occurred(), exc)`. This should only be called when an exception is actually set; a memory access violation will occur if no exception has been raised.

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

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

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

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

備F: This function is normally only used by code that needs to catch exceptions or by code that needs to save and restore the error indicator temporarily, e.g.:

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

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

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

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

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

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

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

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

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

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

PyObject ***PyErr_GetHandledException** (void)

Part of the Stable ABI *since version 3.11.* Retrieve the active exception instance, as would be returned by `sys.exception()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. Returns a new reference to the exception or `NULL`. Does not modify the interpreter's exception state.

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

to restore or clear the exception state.

在 3.11 版新加入.

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

Part of the Stable ABI *since version 3.11.* Set the active exception, as known from `sys.exception()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. To clear the exception state, pass `NULL`.

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

在 3.11 版新加入.

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

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

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

在 3.3 版新加入.

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

Part of the Stable ABI *since version 3.7.* Set the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. This function steals the references of the arguments. To clear the exception state, pass `NULL` for all three arguments. This function is kept for backwards compatibility. Prefer using `PyErr_SetHandledException()`.

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

在 3.3 版新加入.

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

5.5 Signal Handling

int **PyErr_CheckSignals**()

Part of the Stable ABI. This function interacts with Python's signal handling.

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

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

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

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

備註: The default Python signal handler for `SIGINT` raises the `KeyboardInterrupt` exception.

void **PyErr_SetInterrupt**()

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

備註: This function is async-signal-safe. It can be called without the *GIL* and from a C signal handler.

int **PyErr_SetInterruptEx**(int signum)

Part of the Stable ABI *since version 3.10.* Simulate the effect of a signal arriving. The next time `PyErr_CheckSignals()` is called, the Python signal handler for the given signal number will be called.

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

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

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

備註: This function is async-signal-safe. It can be called without the *GIL* and from a C signal handler.

在 3.10 版新加入.

int **PySignal_SetWakeupFd**(int fd)

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

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

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

5.6 例外類別

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

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

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

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

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

在 3.2 版新加入.

5.7 例外物件

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

回傳值：新的參照。*Part of the* Stable ABI. Return the traceback associated with the exception as a new reference, as accessible from Python through the `__traceback__` attribute. If there is no traceback associated, this returns NULL.

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

Part of the Stable ABI. Set the traceback associated with the exception to *tb*. Use `Py_None` to clear it.

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

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

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

Part of the Stable ABI. Set the context associated with the exception to *ctx*. Use NULL to clear it. There is no type check to make sure that *ctx* is an exception instance. This steals a reference to *ctx*.

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

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

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

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

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

5.8 Unicode Exception Objects

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

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

回傳值：新的參照。 *Part of the* Stable ABI. Create a `UnicodeDecodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.

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

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

回傳值：新的參照。 *Part of the* Stable ABI. Return the *encoding* attribute of the given exception object.

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

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

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

回傳值：新的參照。 *Part of the* Stable ABI. Return the *object* attribute of the given exception object.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

回傳值：新的參照。 *Part of the* Stable ABI. Return the *reason* attribute of the given exception object.

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

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

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

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

5.9 Recursion Control

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

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

Part of the Stable ABI *since version 3.9.* Marks a point where a recursive C-level call is about to be performed.

If `USE_STACKCHECK` is defined, this function checks if the OS stack overflowed using *`PyOS_CheckStack()`*. If this is the case, it sets a `MemoryError` and returns a nonzero value.

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

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

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

void **Py_LeaveRecursiveCall**(void)

Part of the Stable ABI *since version 3.9.* Ends a `Py_EnterRecursiveCall()`. Must be called once for each *successful* invocation of `Py_EnterRecursiveCall()`.

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

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

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

Part of the Stable ABI. Called at the beginning of the `tp_repr` implementation to detect cycles.

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

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

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

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

Part of the Stable ABI. Ends a `Py_ReprEnter()`. Must be called once for each invocation of `Py_ReprEnter()` that returns zero.

5.10 Standard Exceptions

All standard Python exceptions are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type `PyObject*`; they are all class objects. For completeness, here are all the variables:

C Name	Python Name	F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	
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	

繼續下一頁

表格 1 – 繼續上一頁

C Name	Python Name	□解
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	

在 3.3 版新加入: PyExc_BlockingIOError, PyExc_BrokenPipeError, PyExc_ChildProcessError, PyExc_ConnectionError, PyExc_ConnectionAbortedError, PyExc_ConnectionRefusedError, PyExc_ConnectionResetError, PyExc_FileExistsError, PyExc_FileNotFoundError, PyExc_InterruptedError, PyExc_IsADirectoryError, PyExc_NotADirectoryError, PyExc_PermissionError, PyExc_ProcessLookupError 和 PyExc_TimeoutError 是在 **PEP 3151** 被引入。

在 3.5 版新加入: PyExc_StopAsyncIteration 和 PyExc_RecursionError。

在 3.6 版新加入: PyExc_ModuleNotFoundError。

These are compatibility aliases to PyExc_OSError:

C Name	□解
PyExc_EnvironmentError	
PyExc_IOError	
PyExc_WindowsError	2

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

□解:

[1] This is a base class for other standard exceptions.

[2] Only defined on Windows; protect code that uses this by testing that the preprocessor macro MS_WINDOWS is defined.

5.11 Standard Warning Categories

All standard Python warning categories are available as global variables whose names are PyExc_ followed by the Python exception name. These have the type *PyObject**; they are all class objects. For completeness, here are all the variables:

C Name	Python Name	F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	

在 3.2 版新加入: PyExc_ResourceWarning.

F解:

[3] This is a base class for other standard warning categories.

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

在 3.6 版新加入.

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

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

void **PyOS_BeforeFork**()

Part of the Stable ABI *on platforms with fork() since version 3.7.* Function to prepare some internal state before a process fork. This should be called before calling `fork()` or any similar function that clones the current process. Only available on systems where `fork()` is defined.

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

在 3.7 版新加入.

void **PyOS_AfterFork_Parent**()

Part of the Stable ABI *on platforms with fork() since version 3.7.* Function to update some internal state after a process fork. This should be called from the parent process after calling `fork()` or any similar function that clones the current process, regardless of whether process cloning was successful. Only available on systems where `fork()` is defined.

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

在 3.7 版新加入.

void **PyOS_AfterFork_Child**()

Part of the Stable ABI *on platforms with fork() since version 3.7.* Function to update internal interpreter state after a process fork. This must be called from the child process after calling `fork()`, or any similar function that clones the current process, if there is any chance the process will call back into the Python interpreter. Only available on systems where `fork()` is defined.

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

在 3.7 版新加入.

也參考:

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

void **PyOS_AfterFork**()

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

在 3.7 版之後被棄用: This function is superseded by `PyOS_AfterFork_Child()`.

int **PyOS_CheckStack**()

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

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

Part of the Stable ABI.

PyOS_sighandler_t **PyOS_getsig**(int i)

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

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

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

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

Part of the Stable ABI *since version 3.7.*

> **警告:** This function should not be called directly: use the `PyConfig` API with the `PyConfig_SetBytesString()` function which ensures that *Python is preinitialized*.
>
> This function must not be called before *Python is preinitialized* and so that the LC_CTYPE locale is properly configured: see the `Py_PreInitialize()` function.

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

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

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

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

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

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

也參考:

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

在 3.5 版新加入.

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

在 3.8 版的變更: The function now uses the UTF-8 encoding on Windows if `Py_LegacyWindowsFSEncodingFlag` is zero;

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

Part of the Stable ABI *since version 3.7.* Encode a wide character string to the *filesystem encoding and error handler*. If the error handler is surrogateescape error handler, surrogate characters in the range U+DC80..U+DCFF are converted to bytes 0x80..0xFF.

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

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

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

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

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

也參考:

The `PyUnicode_EncodeFSDefault()` and `PyUnicode_EncodeLocale()` functions.

在 3.5 版新加入.

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

在 3.8 版的變更: The function now uses the UTF-8 encoding on Windows if `Py_LegacyWindowsFSEncodingFlag` is zero.

6.2 系統函式

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

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

回傳值：借用參照。 *Part of the* Stable ABI. Return the object *name* from the `sys` module or `NULL` if it does not exist, without setting an exception.

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

Part of the Stable ABI. Set *name* in the `sys` module to *v* unless *v* is `NULL`, in which case *name* is deleted from the sys module. Returns `0` on success, `-1` on error.

void **PySys_ResetWarnOptions**()

Part of the Stable ABI. Reset `sys.warnoptions` to an empty list. This function may be called prior to *`Py_Initialize()`*.

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

Part of the Stable ABI. This API is kept for backward compatibility: setting *`PyConfig.warnoptions`* should be used instead, see *Python Initialization Configuration*.

Append *s* to `sys.warnoptions`. This function must be called prior to *`Py_Initialize()`* in order to affect the warnings filter list.

在 3.11 版之後被F用.

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

Part of the Stable ABI. This API is kept for backward compatibility: setting *`PyConfig.warnoptions`* should be used instead, see *Python Initialization Configuration*.

Append *unicode* to `sys.warnoptions`.

Note: this function is not currently usable from outside the CPython implementation, as it must be called prior to the implicit import of `warnings` in *`Py_Initialize()`* to be effective, but can't be called until enough of the runtime has been initialized to permit the creation of Unicode objects.

在 3.11 版之後被F用.

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

Part of the Stable ABI. This API is kept for backward compatibility: setting *`PyConfig.module_search_paths`* and *`PyConfig.module_search_paths_set`* should be used instead, see *Python Initialization Configuration*.

Set `sys.path` to a list object of paths found in *path* which should be a list of paths separated with the platform's search path delimiter (`:` on Unix, `;` on Windows).

在 3.11 版之後被F用.

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

Part of the Stable ABI. Write the output string described by *format* to `sys.stdout`. No exceptions are raised, even if truncation occurs (see below).

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

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

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

Part of the Stable ABI. As *`PySys_WriteStdout()`*, but write to `sys.stderr` or *stderr* instead.

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

Part of the Stable ABI. Function similar to PySys_WriteStdout() but format the message using `PyUnicode_FromFormatV()` and don't truncate the message to an arbitrary length.

在 3.2 版新加入.

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

Part of the Stable ABI. As `PySys_FormatStdout()`, but write to `sys.stderr` or *stderr* instead.

在 3.2 版新加入.

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

Part of the Stable ABI *since version 3.7.* This API is kept for backward compatibility: setting `PyConfig.xoptions` should be used instead, see *Python Initialization Configuration*.

Parse *s* as a set of `-X` options and add them to the current options mapping as returned by `PySys_GetXOptions()`. This function may be called prior to `Py_Initialize()`.

在 3.2 版新加入.

在 3.11 版之後被棄用.

PyObject ***PySys_GetXOptions** ()

回傳值：借用參照。*Part of the* Stable ABI *since version 3.7.* Return the current dictionary of `-X` options, similarly to `sys._xoptions`. On error, `NULL` is returned and an exception is set.

在 3.2 版新加入.

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

Raise an auditing event with any active hooks. Return zero for success and non-zero with an exception set on failure.

If any hooks have been added, *format* and other arguments will be used to construct a tuple to pass. Apart from `N`, the same format characters as used in `Py_BuildValue()` are available. If the built value is not a tuple, it will be added into a single-element tuple. (The `N` format option consumes a reference, but since there is no way to know whether arguments to this function will be consumed, using it may cause reference leaks.)

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

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

在 3.8 版新加入.

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

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

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

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

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

The hook function is always called with the GIL held by the Python interpreter that raised the event.

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

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)

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

在 3.8 版新加入.

6.3 行程(Process)控制

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

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

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

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

void **Py_Exit**(int status)

Part of the Stable ABI. Exit the current process. This calls `Py_FinalizeEx()` and then calls the standard C library function `exit(status)`. If `Py_FinalizeEx()` indicates an error, the exit status is set to 120.

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

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

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

6.4 引入模組

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

回傳值:新的參照。*Part of the* Stable ABI. This is a wrapper around `PyImport_Import()` which takes a `const char*` as an argument instead of a `PyObject*`.

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

回傳值:新的參照。*Part of the* Stable ABI. This function is a deprecated alias of `PyImport_ImportModule()`.

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

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

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

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

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

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

回傳值：新的參照。 *Part of the* Stable ABI *since version 3.7.* Import a module. This is best described by referring to the built-in Python function `__import__()`, as the standard `__import__()` function calls this function directly.

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

在 3.3 版新加入.

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

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

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

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

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

This function always uses absolute imports.

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

回傳值：新的參照。*Part of the* Stable ABI. Reload a module. Return a new reference to the reloaded module, or `NULL` with an exception set on failure (the module still exists in this case).

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

回傳值：借用參照。 *Part of the* Stable ABI *since version 3.7.* Return the module object corresponding to a module name. The *name* argument may be of the form `package.module`. First check the modules dictionary if there's one there, and if not, create a new one and insert it in the modules dictionary. Return `NULL` with an exception set on failure.

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

在 3.3 版新加入.

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

回傳值：借用參照。*Part of the* Stable ABI. Similar to *PyImport_AddModuleObject()*, but the name is a UTF-8 encoded string instead of a Unicode object.

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

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

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

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

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

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

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

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

回傳值：新的參照。*Part of the* Stable ABI. Like *PyImport_ExecCodeModule()*, but the __file__ attribute of the module object is set to *pathname* if it is non-NULL.

也請見*PyImport_ExecCodeModuleWithPathnames()*。

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

回傳值：新的參照。*Part of the* Stable ABI *since version 3.7.* Like *PyImport_ExecCodeModuleEx()*, but the __cached__ attribute of the module object is set to *cpathname* if it is non-NULL. Of the three functions, this is the preferred one to use.

在 3.3 版新加入.

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

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

在 3.2 版新加入.

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

long **PyImport_GetMagicNumber**()

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

在 3.3 版的變更: 當失敗時回傳 -1。

const char ***PyImport_GetMagicTag**()

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

在 3.2 版新加入.

PyObject ***PyImport_GetModuleDict**()

回傳值：借用參照。*Part of the* Stable ABI. Return the dictionary used for the module administration (a.k.a. sys.modules). Note that this is a per-interpreter variable.

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

回傳值：新的參照。*Part of the* Stable ABI *since version 3.8.* Return the already imported module with the given name. If the module has not been imported yet then returns NULL but does not set an error. Returns NULL and sets an error if the lookup failed.

在 3.7 版新加入.

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

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

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

Part of the Stable ABI *since version 3.7.* Load a frozen module named *name*. Return `1` for success, `0` if the module is not found, and `-1` with an exception set if the initialization failed. To access the imported module on a successful load, use `PyImport_ImportModule()`. (Note the misnomer --- this function would reload the module if it was already imported.)

在 3.3 版新加入.

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

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

Part of the Stable ABI. Similar to `PyImport_ImportFrozenModuleObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

struct **_frozen**

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

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

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

const struct *_frozen* ***PyImport_FrozenModules**

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

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

Part of the Stable ABI. Add a single module to the existing table of built-in modules. This is a convenience wrapper around `PyImport_ExtendInittab()`, returning `-1` if the table could not be extended. The new module can be imported by the name *name*, and uses the function *initfunc* as the initialization function called on the first attempted import. This should be called before `Py_Initialize()`.

struct **_inittab**

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

const char ***name**

The module name, as an ASCII encoded string.

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

Initialization function for a module built into the interpreter.

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

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

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

6.5 Data marshalling support

These routines allow C code to work with serialized objects using the same data format as the `marshal` module. There are functions to write data into the serialization format, and additional functions that can be used to read the data back. Files used to store marshalled data must be opened in binary mode.

Numeric values are stored with the least significant byte first.

The module supports two versions of the data format: version 0 is the historical version, version 1 shares interned strings in the file, and upon unmarshalling. Version 2 uses a binary format for floating point numbers. `Py_MARSHAL_VERSION` indicates the current file format (currently 2).

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

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

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

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

Marshal a Python object, *value*, to *file*. *version* indicates the file format.

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

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

回傳值：新的參照。Return a bytes object containing the marshalled representation of *value*. *version* indicates the file format.

The following functions allow marshalled values to be read back in.

long **PyMarshal_ReadLongFromFile**(FILE *file)

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

On error, sets the appropriate exception (`EOFError`) and returns `-1`.

int **PyMarshal_ReadShortFromFile**(FILE *file)

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

On error, sets the appropriate exception (`EOFError`) and returns `-1`.

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

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

On error, sets the appropriate exception (`EOFError`, `ValueError` or `TypeError`) and returns `NULL`.

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

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

On error, sets the appropriate exception (`EOFError`, `ValueError` or `TypeError`) and returns `NULL`.

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

回傳值：新的參照。Return a Python object from the data stream in a byte buffer containing *len* bytes pointed to by *data*.

On error, sets the appropriate exception (`EOFError`, `ValueError` or `TypeError`) and returns `NULL`.

6.6 剖析引數與建置數值

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

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

6.6.1 Parsing arguments

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

Strings and buffers

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

Unless otherwise stated, buffers are not NUL-terminated.

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

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

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

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

備F: For all `#` variants of formats (`s#`, `y#`, etc.), the macro `PY_SSIZE_T_CLEAN` must be defined before including `Python.h`. On Python 3.9 and older, the type of the length argument is *Py_ssize_t* if the `PY_SSIZE_T_CLEAN` macro is defined, or int otherwise.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`u` (`str`) [const Py_UNICODE *]
Convert a Python Unicode object to a C pointer to a NUL-terminated buffer of Unicode characters. You must pass the address of a `Py_UNICODE` pointer variable, which will be filled with the pointer to an existing Unicode buffer. Please note that the width of a `Py_UNICODE` character depends on compilation options (it is either 16 or 32 bits). The Python string must not contain embedded null code points; if it does, a `ValueError` exception is raised.

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

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsWideCharString()`.

`u#` (`str`) [const Py_UNICODE *, `Py_ssize_t`]
This variant on `u` stores into two C variables, the first one a pointer to a Unicode data buffer, the second one its length. This variant allows null code points.

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsWideCharString()`.

Z (**str** 或 **None**) **[const Py_UNICODE *]**
Like `u`, but the Python object may also be `None`, in which case the `Py_UNICODE` pointer is set to `NULL`.

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsWideCharString()`.

Z# (**str** or **None**) **[const Py_UNICODE *, Py_ssize_t]**
Like `u#`, but the Python object may also be `None`, in which case the `Py_UNICODE` pointer is set to `NULL`.

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsWideCharString()`.

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

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

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

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

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

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

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

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

There are two modes of operation:

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

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

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

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

數字

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

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

h (int) [short int]
將一個 Python 整數轉換成 C 的 short int。

H (int) [unsigned short int]
將一個 Python 整數轉換成 C 的 unsigned short int，轉換過程無溢位檢查。

i (int) [int]
將一個 Python 整數轉換成 C 的 int。

I (int) [unsigned int]
將一個 Python 整數轉換成 C 的 unsigned int，轉換過程無溢位檢查。

l (int) [long int]
將一個 Python 整數轉換成 C 的 long int。

k (int) [unsigned long]
將一個 Python 整數轉換成 C 的 unsigned long，轉換過程無溢位檢查。

L (int) [long long]
將一個 Python 整數轉換成 C 的 long long。

K (int) [unsigned long long]
將一個 Python 整數轉換成 C 的 unsigned long long，轉換過程無溢位檢查。

n (int) [*Py_ssize_t*]
將一個 Python 整數轉換成 C 的*Py_ssize_t*。

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

在 3.3 版的變更: 允許 bytearray 物件。

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

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

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

D (complex) [Py_complex]
將一個 Python 複數轉換成 C 的*Py_complex* 結構。

其他物件

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

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

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

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

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

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

在 3.1 版的變更: 加入 Py_CLEANUP_SUPPORTED。

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

在 3.3 版新加入.

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

It is possible to pass "long" integers (integers whose value exceeds the platform's LONG_MAX) however no proper range checking is done --- the most significant bits are silently truncated when the receiving field is too small to receive the value (actually, the semantics are inherited from downcasts in C --- your mileage may vary).

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

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

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

在 3.3 版新加入.

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

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

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

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

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

API 函式

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

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

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

Part of the Stable ABI. Identical to `PyArg_ParseTuple()`, except that it accepts a va_list rather than a variable number of arguments.

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

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

在 3.6 版的變更: Added support for *positional-only parameters*.

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

Part of the Stable ABI. Identical to `PyArg_ParseTupleAndKeywords()`, except that it accepts a va_list rather than a variable number of arguments.

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

Part of the Stable ABI. Ensure that the keys in the keywords argument dictionary are strings. This is only needed if `PyArg_ParseTupleAndKeywords()` is not used, since the latter already does this check.

在 3.2 版新加入.

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

Part of the Stable ABI. Function used to deconstruct the argument lists of "old-style" functions --- these are functions which use the `METH_OLDARGS` parameter parsing method, which has been removed in Python 3. This is not recommended for use in parameter parsing in new code, and most code in the standard interpreter has been modified to no longer use this for that purpose. It does remain a convenient way to decompose other tuples, however, and may continue to be used for that purpose.

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

Part of the Stable ABI. A simpler form of parameter retrieval which does not use a format string to specify the types of the arguments. Functions which use this method to retrieve their parameters should be declared as `METH_VARARGS` in function or method tables. The tuple containing the actual parameters should be passed as *args*; it must actually be a tuple. The length of the tuple must be at least *min* and no more than *max*; *min* and *max* may be equal. Additional arguments must be passed to the function, each of which should be a pointer to a `PyObject*` variable; these will be filled in with the values from *args*; they will contain *borrowed references*.

The variables which correspond to optional parameters not given by *args* will not be filled in; these should be initialized by the caller. This function returns true on success and false if *args* is not a tuple or contains the wrong number of elements; an exception will be set if there was a failure.

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

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

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

The call to *PyArg_UnpackTuple()* in this example is entirely equivalent to this call to *PyArg_ParseTuple()*:

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

6.6.2 Building values

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

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

Py_BuildValue() does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

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

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

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

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

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

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

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

z (str 或 None) [const char *]
和 s 相同。

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

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

u# (str) [const wchar_t *, *Py_ssize_t*]
Convert a Unicode (UTF-16 or UCS-4) data buffer and its length to a Python Unicode object. If the Unicode buffer pointer is NULL, the length is ignored and None is returned.

U (str 或 None) [const char *]
和 s 相同。

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

i (int) [int]
將一個 C 的 int 轉換成 Python 整數物件。

b (int) [char]
將一個 C 的 char 轉換成 Python 整數物件。

h (int) [short int]
將一個 C 的 short int 轉換成 Python 整數物件。

l (int) [long int]
將一個 C 的 long int 轉換成 Python 整數物件。

B (int) [unsigned char]
將一個 C 的 unsigned char 轉換成 Python 整數物件。

H (int) [unsigned short int]
將一個 C 的 unsigned short int 轉換成 Python 整數物件。

I (int) [unsigned int]
將一個 C 的 unsigned int 轉換成 Python 整數物件。

k (int) [unsigned long]
將一個 C 的 unsigned long 轉換成 Python 整數物件。

L (int) [long long]
將一個 C 的 long long 轉換成 Python 整數物件。

K (int) [unsigned long long]
將一個 C 的 unsigned long long 轉換成 Python 整數物件。

n (int) [*Py_ssize_t*]
將一個 C 的*Py_ssize_t* 轉換成 Python 整數。

c (長度為 1 的 bytes) [char]
將一個 C 中代表一個位元組的 int 轉換成 Python 中長度為一的 bytes。

C (長度為 1 的 str) [int]
將一個 C 中代表一個字元的 int 轉換成 Python 中長度為一的 str。

d (float) [double]
將一個 C 的 double 轉換成 Python 浮點數。

f (float) [float]
將一個 C 的 float 轉換成 Python 浮點數。

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

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

S(物件)[PyObject *]
和 O 相同。

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

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

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

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

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

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

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

6.7 字串轉F與格式化

數字轉F函式和被格式化的字串輸出。

int **PyOS_snprintf**(char *str, size_t size, const char *format, ...)
Part of the Stable ABI. Output not more than *size* bytes to *str* according to the format string *format* and the extra arguments. See the Unix man page *snprintf(3)*.

int **PyOS_vsnprintf**(char *str, size_t size, const char *format, va_list va)
Part of the Stable ABI. Output not more than *size* bytes to *str* according to the format string *format* and the variable argument list *va*. Unix man page *vsnprintf(3)*.

PyOS_snprintf() and *PyOS_vsnprintf()* wrap the Standard C library functions snprintf() and vsnprintf(). Their purpose is to guarantee consistent behavior in corner cases, which the Standard C functions do not.

The wrappers ensure that str[size-1] is always '\0' upon return. They never write more than *size* bytes (including the trailing '\0') into str. Both functions require that str != NULL, size > 0, format != NULL and size < INT_MAX. Note that this means there is no equivalent to the C99 n = snprintf(NULL, 0, ...) which would determine the necessary buffer size.

當回傳值 (*rv*) 給這些函式應該被編譯如下:

- When 0 <= rv < size, the output conversion was successful and *rv* characters were written to *str* (excluding the trailing '\0' byte at str[rv]).

- When `rv >= size`, the output conversion was truncated and a buffer with `rv + 1` bytes would have been needed to succeed. `str[size-1]` is `'\0'` in this case.
- When `rv < 0`, "something bad happened." `str[size-1]` is `'\0'` in this case too, but the rest of *str* is undefined. The exact cause of the error depends on the underlying platform.

The following functions provide locale-independent string to number conversions.

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

Part of the Stable ABI. Convert the initial part of the string in `str` to an `unsigned long` value according to the given `base`, which must be between `2` and `36` inclusive, or be the special value `0`.

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

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

See also the Unix man page *strtoul(3)*.

在 3.2 版新加入.

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

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

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

See also the Unix man page *strtol(3)*.

在 3.2 版新加入.

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

Part of the Stable ABI. Convert a string `s` to a `double`, raising a Python exception on failure. The set of accepted strings corresponds to the set of strings accepted by Python's `float()` constructor, except that `s` must not have leading or trailing whitespace. The conversion is independent of the current locale.

If `endptr` is `NULL`, convert the whole string. Raise `ValueError` and return `-1.0` if the string is not a valid representation of a floating-point number.

If endptr is not `NULL`, convert as much of the string as possible and set `*endptr` to point to the first unconverted character. If no initial segment of the string is the valid representation of a floating-point number, set `*endptr` to point to the beginning of the string, raise ValueError, and return `-1.0`.

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

If any other error occurs during the conversion (for example an out-of-memory error), set the appropriate Python exception and return `-1.0`.

在 3.1 版新加入.

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

Part of the Stable ABI. Convert a `double` *val* to a string using supplied *format_code*, *precision*, and *flags*.

format_code must be one of `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` or `'r'`. For `'r'`, the supplied *precision* must be 0 and is ignored. The `'r'` format code specifies the standard `repr()` format.

flags can be zero or more of the values `Py_DTSF_SIGN`, `Py_DTSF_ADD_DOT_0`, or `Py_DTSF_ALT`, or-ed together:

- `Py_DTSF_SIGN` means to always precede the returned string with a sign character, even if *val* is non-negative.

- `Py_DTSF_ADD_DOT_0` means to ensure that the returned string will not look like an integer.
- `Py_DTSF_ALT` means to apply "alternate" formatting rules. See the documentation for the *PyOS_snprintf()* `'#'` specifier for details.

If *ptype* is non-NULL, then the value it points to will be set to one of `Py_DTST_FINITE`, `Py_DTST_INFINITE`, or `Py_DTST_NAN`, signifying that *val* is a finite number, an infinite number, or not a number, respectively.

The return value is a pointer to *buffer* with the converted string or `NULL` if the conversion failed. The caller is responsible for freeing the returned string by calling *PyMem_Free()*.

在 3.1 版新加入.

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

Case insensitive comparison of strings. The function works almost identically to `strcmp()` except that it ignores the case.

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

Case insensitive comparison of strings. The function works almost identically to `strncmp()` except that it ignores the case.

6.8 PyHash API

See also the *PyTypeObject.tp_hash* member.

type **Py_hash_t**

Hash value type: signed integer.

在 3.2 版新加入.

type **Py_uhash_t**

Hash value type: unsigned integer.

在 3.2 版新加入.

type **PyHash_FuncDef**

Hash function definition used by *PyHash_GetFuncDef()*.

const char ***name**

Hash function name (UTF-8 encoded string).

const int **hash_bits**

Internal size of the hash value in bits.

const int **seed_bits**

Size of seed input in bits.

在 3.4 版新加入.

PyHash_FuncDef ***PyHash_GetFuncDef**(void)

Get the hash function definition.

也參考:

PEP 456 "Secure and interchangeable hash algorithm".

在 3.4 版新加入.

6.9 Reflection

PyObject ***PyEval_GetBuiltins**(void)

回傳值：借用參照。*Part of the* Stable ABI. Return a dictionary of the builtins in the current execution frame, or the interpreter of the thread state if no frame is currently executing.

PyObject ***PyEval_GetLocals**(void)

回傳值：借用參照。*Part of the* Stable ABI. Return a dictionary of the local variables in the current execution frame, or `NULL` if no frame is currently executing.

PyObject ***PyEval_GetGlobals**(void)

回傳值：借用參照。*Part of the* Stable ABI. Return a dictionary of the global variables in the current execution frame, or `NULL` if no frame is currently executing.

PyFrameObject ***PyEval_GetFrame**(void)

回傳值：借用參照。 *Part of the* Stable ABI. Return the current thread state's frame, which is `NULL` if no frame is currently executing.

另請見*PyThreadState_GetFrame()*。

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

Part of the Stable ABI. Return the name of *func* if it is a function, class or instance object, else the name of *func*s type.

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

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

6.10 Codec registry and support functions

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

Part of the Stable ABI. Register a new codec search function.

As side effect, this tries to load the `encodings` package, if not yet done, to make sure that it is always first in the list of search functions.

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

Part of the Stable ABI *since version 3.10.* Unregister a codec search function and clear the registry's cache. If the search function is not registered, do nothing. Return 0 on success. Raise an exception and return -1 on error.

在 3.10 版新加入.

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

Part of the Stable ABI. Return `1` or `0` depending on whether there is a registered codec for the given *encoding*. This function always succeeds.

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

回傳值：新的參照。 *Part of the* Stable ABI. Generic codec based encoding API.

object is passed through the encoder function found for the given *encoding* using the error handling method defined by *errors*. *errors* may be `NULL` to use the default method defined for the codec. Raises a `LookupError` if no encoder can be found.

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

回傳值：新的參照。 *Part of the* Stable ABI. Generic codec based decoding API.

object is passed through the decoder function found for the given *encoding* using the error handling method defined by *errors*. *errors* may be `NULL` to use the default method defined for the codec. Raises a `LookupError` if no encoder can be found.

6.10.1 Codec lookup API

In the following functions, the *encoding* string is looked up converted to all lower-case characters, which makes encodings looked up through this mechanism effectively case-insensitive. If no codec is found, a `KeyError` is set and `NULL` returned.

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

回傳值：新的參照。*Part of the* Stable ABI. Get an encoder function for the given *encoding*.

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

回傳值：新的參照。*Part of the* Stable ABI. Get a decoder function for the given *encoding*.

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

回傳值：新的參照。*Part of the* Stable ABI. Get an `IncrementalEncoder` object for the given *encoding*.

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

回傳值：新的參照。*Part of the* Stable ABI. Get an `IncrementalDecoder` object for the given *encoding*.

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

回傳值：新的參照。*Part of the* Stable ABI. Get a `StreamReader` factory function for the given *encoding*.

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

回傳值：新的參照。*Part of the* Stable ABI. Get a `StreamWriter` factory function for the given *encoding*.

6.10.2 Registry API for Unicode encoding error handlers

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

Part of the Stable ABI. Register the error handling callback function *error* under the given *name*. This callback function will be called by a codec when it encounters unencodable characters/undecodable bytes and *name* is specified as the error parameter in the call to the encode/decode function.

The callback gets a single argument, an instance of `UnicodeEncodeError`, `UnicodeDecodeError` or `UnicodeTranslateError` that holds information about the problematic sequence of characters or bytes and their offset in the original string (see *Unicode Exception Objects* for functions to extract this information). The callback must either raise the given exception, or return a two-item tuple containing the replacement for the problematic sequence, and an integer giving the offset in the original string at which encoding/decoding should be resumed.

Return `0` on success, `-1` on error.

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

回傳值：新的參照。*Part of the* Stable ABI. Lookup the error handling callback function registered under *name*. As a special case `NULL` can be passed, in which case the error handling callback for "strict" will be returned.

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

回傳值：總是為 *NULL*。*Part of the* Stable ABI. Raise *exc* as an exception.

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

回傳值：新的參照。*Part of the* Stable ABI. Ignore the unicode error, skipping the faulty input.

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

回傳值：新的參照。*Part of the* Stable ABI. Replace the unicode encode error with `?` or `U+FFFD`.

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

回傳值：新的參照。*Part of the* Stable ABI. Replace the unicode encode error with XML character references.

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

回傳值：新的參照。*Part of the* Stable ABI. Replace the unicode encode error with backslash escapes (`\x`, `\u` and `\U`).

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

回傳值：新的參照。*Part of the* Stable ABI *since version 3.7.* Replace the unicode encode error with `\N{...}` escapes.

在 3.5 版新加入.

CHAPTER 7

抽象物件層 (Abstract Objects Layer)

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

7.1 Object Protocol

PyObject ***Py_NotImplemented**

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

Py_RETURN_NOTIMPLEMENTED

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

Py_PRINT_RAW

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

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

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

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

Part of the Stable ABI. Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This is equivalent to the Python expression `hasattr(o, attr_name)`. This function always succeeds.

備註： Exceptions that occur when this calls `__getattr__()` and `__getattribute__()` methods are silently ignored. For proper error handling, use `PyObject_GetAttr()` instead.

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

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

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

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

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

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

回傳值：新的參照。*Part of the* Stable ABI. This is the same as `PyObject_GetAttr()`, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

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

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

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

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

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

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

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

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

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

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

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

Delete attribute named *attr_name*, for object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o.attr_name`.

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

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

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

回傳值：新的參照。*Part of the* Stable ABI *since version 3.10.* A generic implementation for the getter of a `__dict__` descriptor. It creates the dictionary if necessary.

This function may also be called to get the `__dict__` of the object *o*. Pass `NULL` for *context* when calling it. Since this function may need to allocate memory for the dictionary, it may be more efficient to call `PyObject_GetAttr()` when accessing an attribute on the object.

On failure, returns NULL with an exception set.

在 3.3 版新加入.

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

Part of the Stable ABI *since version 3.7.* A generic implementation for the setter of a `__dict__` descriptor. This implementation does not allow the dictionary to be deleted.

在 3.3 版新加入.

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

Return a pointer to `__dict__` of the object *obj*. If there is no `__dict__`, return NULL without setting an exception.

This function may need to allocate memory for the dictionary, so it may be more efficient to call `PyObject_GetAttr()` when accessing an attribute on the object.

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

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

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

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

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

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

Part of the Stable ABI. Format *obj* using *format_spec*. This is equivalent to the Python expression `format(obj, format_spec)`.

format_spec may be NULL. In this case the call is equivalent to `format(obj)`. Returns the formatted string on success, NULL on failure.

PyObject ***PyObject_Repr** (*PyObject* *o)

回傳值：新的參照。 *Part of the* Stable ABI. Compute a string representation of object *o*. Returns the string representation on success, NULL on failure. This is the equivalent of the Python expression `repr(o)`. Called by the `repr()` built-in function.

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

PyObject ***PyObject_ASCII** (*PyObject* *o)

回傳值：新的參照。 *Part of the* Stable ABI. As `PyObject_Repr()`, compute a string representation of object *o*, but escape the non-ASCII characters in the string returned by `PyObject_Repr()` with `\x`, `\u` or `\U` escapes. This generates a string similar to that returned by `PyObject_Repr()` in Python 2. Called by the `ascii()` built-in function.

PyObject ***PyObject_Str** (*PyObject* *o)

回傳值：新的參照。 *Part of the* Stable ABI. Compute a string representation of object *o*. Returns the string representation on success, NULL on failure. This is the equivalent of the Python expression `str(o)`. Called by the `str()` built-in function and, therefore, by the `print()` function.

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

PyObject ***PyObject_Bytes** (*PyObject* *o)

回傳值：新的參照。 *Part of the* Stable ABI. Compute a bytes representation of object *o*. NULL is returned on failure and a bytes object on success. This is equivalent to the Python expression `bytes(o)`, when *o* is not an integer. Unlike `bytes(o)`, a TypeError is raised when *o* is an integer instead of a zero-initialized bytes object.

int **PyObject_IsSubclass**(*PyObject* *derived, *PyObject* *cls)

Part of the Stable ABI. Return `1` if the class *derived* is identical to or derived from the class *cls*, otherwise return `0`. In case of an error, return `-1`.

If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `0`.

If *cls* has a `__subclasscheck__()` method, it will be called to determine the subclass status as described in **PEP 3119**. Otherwise, *derived* is a subclass of *cls* if it is a direct or indirect subclass, i.e. contained in `cls.__mro__`.

Normally only class objects, i.e. instances of `type` or a derived class, are considered classes. However, objects can override this by having a `__bases__` attribute (which must be a tuple of base classes).

int **PyObject_IsInstance**(*PyObject* *inst, *PyObject* *cls)

Part of the Stable ABI. Return `1` if *inst* is an instance of the class *cls* or a subclass of *cls*, or `0` if not. On error, returns `-1` and sets an exception.

If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `0`.

If *cls* has a `__instancecheck__()` method, it will be called to determine the subclass status as described in **PEP 3119**. Otherwise, *inst* is an instance of *cls* if its class is a subclass of *cls*.

An instance *inst* can override what is considered its class by having a `__class__` attribute.

An object *cls* can override if it is considered a class, and what its base classes are, by having a `__bases__` attribute (which must be a tuple of base classes).

Py_hash_t **PyObject_Hash**(*PyObject* *o)

Part of the Stable ABI. Compute and return the hash value of an object *o*. On failure, return `-1`. This is the equivalent of the Python expression `hash(o)`.

在 3.2 版的變更: The return type is now Py_hash_t. This is a signed integer the same size as `Py_ssize_t`.

Py_hash_t **PyObject_HashNotImplemented**(*PyObject* *o)

Part of the Stable ABI. Set a `TypeError` indicating that `type(o)` is not *hashable* and return `-1`. This function receives special treatment when stored in a `tp_hash` slot, allowing a type to explicitly indicate to the interpreter that it is not hashable.

int **PyObject_IsTrue**(*PyObject* *o)

Part of the Stable ABI. Returns `1` if the object *o* is considered to be true, and `0` otherwise. This is equivalent to the Python expression `not not o`. On failure, return `-1`.

int **PyObject_Not**(*PyObject* *o)

Part of the Stable ABI. Returns `0` if the object *o* is considered to be true, and `1` otherwise. This is equivalent to the Python expression `not o`. On failure, return `-1`.

PyObject ***PyObject_Type**(*PyObject* *o)

回傳值：新的參照。*Part of the* Stable ABI. When *o* is non-`NULL`, returns a type object corresponding to the object type of object *o*. On failure, raises `SystemError` and returns `NULL`. This is equivalent to the Python expression `type(o)`. This function creates a new *strong reference* to the return value. There's really no reason to use this function instead of the `Py_TYPE()` function, which returns a pointer of type `PyTypeObject*`, except when a new *strong reference* is needed.

int **PyObject_TypeCheck**(*PyObject* *o, *PyTypeObject* *type)

Return non-zero if the object *o* is of type *type* or a subtype of *type*, and `0` otherwise. Both parameters must be non-`NULL`.

Py_ssize_t **PyObject_Size**(*PyObject* *o)

Py_ssize_t **PyObject_Length**(*PyObject* *o)

Part of the Stable ABI. Return the length of object *o*. If the object *o* provides either the sequence and mapping protocols, the sequence length is returned. On error, -1 is returned. This is the equivalent to the Python expression `len(o)`.

Py_ssize_t **PyObject_LengthHint**(*PyObject* *o, *Py_ssize_t* defaultvalue)

Return an estimated length for the object *o*. First try to return its actual length, then an estimate using `__length_hint__()`, and finally return the default value. On error return -1. This is the equivalent to the Python expression `operator.length_hint(o, defaultvalue)`.

在 3.4 版新加入.

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

回傳值：新的參照。 *Part of the* Stable ABI. Return element of *o* corresponding to the object *key* or NULL on failure. This is the equivalent of the Python expression `o[key]`.

int **PyObject_SetItem**(*PyObject* *o, *PyObject* *key, *PyObject* *v)

Part of the Stable ABI. Map the object *key* to the value *v*. Raise an exception and return -1 on failure; return 0 on success. This is the equivalent of the Python statement `o[key] = v`. This function *does not* steal a reference to *v*.

int **PyObject_DelItem**(*PyObject* *o, *PyObject* *key)

Part of the Stable ABI. Remove the mapping for the object *key* from the object *o*. Return -1 on failure. This is equivalent to the Python statement `del o[key]`.

PyObject ***PyObject_Dir**(*PyObject* *o)

回傳值：新的參照。 *Part of the* Stable ABI. This is equivalent to the Python expression `dir(o)`, returning a (possibly empty) list of strings appropriate for the object argument, or NULL if there was an error. If the argument is NULL, this is like the Python `dir()`, returning the names of the current locals; in this case, if no execution frame is active then NULL is returned but `PyErr_Occurred()` will return false.

PyObject ***PyObject_GetIter**(*PyObject* *o)

回傳值：新的參照。 *Part of the* Stable ABI. This is equivalent to the Python expression `iter(o)`. It returns a new iterator for the object argument, or the object itself if the object is already an iterator. Raises `TypeError` and returns NULL if the object cannot be iterated.

PyObject ***PyObject_GetAIter**(*PyObject* *o)

回傳值：新的參照。*Part of the* Stable ABI *since version 3.10.* This is the equivalent to the Python expression `aiter(o)`. Takes an `AsyncIterable` object and returns an `AsyncIterator` for it. This is typically a new iterator but if the argument is an `AsyncIterator`, this returns itself. Raises `TypeError` and returns NULL if the object cannot be iterated.

在 3.10 版新加入.

7.2 呼叫協定 (Call Protocol)

CPython 支援兩種不同的呼叫協定：*tp_call* 和 vectorcall（向量呼叫）。

7.2.1 *tp_call* 協定

設定`tp_call` 的類□之實例都是可呼叫的。該擴充槽 (slot) 的簽章□:

```
PyObject *tp_call(PyObject *callable, PyObject *args, PyObject *kwargs);
```

要達成一個呼叫會使用一個 tuple（元組）表示位置引數、一個 dict 表示關鍵字引數，類似於 Python 程式碼中的 `callable(*args, **kwargs)`。*args* 必須不□ NULL（如果□有引數，會使用一個空 tuple），但如果□有關鍵字引數，*kwargs* 可以是 *NULL*。

這個慣例不僅會被 *tp_call* 使用，`tp_new` 和`tp_init` 也這樣傳遞引數。

使用`PyObject_Call()` 或其他呼叫 *API* 來呼叫一個物件。

7.2.2 Vectorcall 協定

在 3.9 版新加入.

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

經驗法則上，如果可呼叫物件有支援，CPython 於□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`。

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。

Classes can implement the vectorcall protocol by enabling the `Py_TPFLAGS_HAVE_VECTORCALL` flag and setting `tp_vectorcall_offset` to the offset inside the object structure where a *vectorcallfunc* appears. This is a pointer to a function with the following signature:

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

- *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` flag. To get the actual number of positional arguments from *nargsf*, use `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

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。

Whenever they can do so cheaply (without additional allocation), callers are encouraged to use `PY_VECTORCALL_ARGUMENTS_OFFSET`. Doing so will allow callables such as bound methods to make their onward calls (which include a prepended *self* argument) very efficiently.

在 3.8 版新加入.

要呼叫一個實作了 vectorcall 的物件，請就像其他可呼叫物件一樣使用呼叫 *API* 中的函式。`PyObject_Vectorcall()` 通常是最有效率的。

備註: 在 CPython 3.8 中，vectorcall API 和相關函式暫定以帶開頭底線的名稱提供：`_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，做為不帶底線的新名稱的別名。

遞迴控制

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

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

Vectorcall 支援 API

Py_ssize_t **PyVectorcall_NARGS** (size_t nargsf)

給定一個 vectorcall *nargsf* 引數，回傳引數的實際數量。目前等同於：

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

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

在 3.8 版新加入.

vectorcallfunc **PyVectorcall_Function** (*PyObject* *op)

如果 *op* 不支援 vectorcall 協定（因為型別不支援或特定實例不支援），就回傳 *NULL*。否則，回傳儲存在 *op* 中的 vectorcall 函式指標。這個函式不會引發例外。

這大多在檢查 *op* 是否支援 vectorcall 時能派上用場，可以透過檢查 `PyVectorcall_Function(op) != NULL` 來達成。

在 3.9 版新加入.

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

呼叫 *callable* 的`vectorcallfunc`，其位置引數和關鍵字引數分別以 tuple 和 dict 格式給定。

This is a specialized function, intended to be put in the `tp_call` slot or be used in an implementation of `tp_call`. It does not check the `Py_TPFLAGS_HAVE_VECTORCALL` flag and it does not fall back to `tp_call`.

在 3.8 版新加入.

7.2.3 物件呼叫 API

有多個函式可被用來呼叫 Python 物件。各個函式會將其引數轉換為被呼叫物件所支援的慣用形式–可以是 *tp_call* 或 vectorcall。為了盡可能減少轉換的進行，請選擇一個適合你所擁有資料格式的函式。

下表總結了可用的函式；請參閱各個說明文件以瞭解詳情。

函式	callable	args	kwargs
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。 *Part of the* Stable ABI. 呼叫一個可呼叫的 Python 物件 *callable*，附帶由 tuple *args* 所給定的引數及由字典 *kwargs* 所給定的關鍵字引數。

args 必須不□ *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。 *Part of the* Stable ABI *since version 3.10.* 呼叫一個可呼叫的 Python 物件 *callable* □不附帶任何引數。這是不帶引數呼叫 Python 可呼叫物件的最有效方式。

成功時回傳結果，或在失敗時引發一個例外□回傳 *NULL*。

在 3.9 版新加入.

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

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

成功時回傳結果，或在失敗時引發一個例外□回傳 *NULL*。

在 3.9 版新加入.

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

回傳值：新的參照。 *Part of the* Stable ABI. 呼叫一個可呼叫的 Python 物件 *callable*，附帶由 tuple *args* 所給定的引數。如果不需要傳入引數，則 *args* 可以□ *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。 *Part of the* Stable ABI. 呼叫一個可呼叫的 Python 物件 *callable*，附帶數量可變的 C 引數。這些 C 引數使用 *Py_BuildValue()* 風格的格式字串來描述。格式可以□ *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。*Part of the* Stable ABI. 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。*Part of the* Stable ABI. 呼叫一個可呼叫的 Python 物件 *callable*，附帶數量可變的 `PyObject*` 引數。這些引數是以位置在 *NULL* 後面、數量可變的參數來提供。

成功時回傳結果，或在失敗時引發一個例外□回傳 *NULL*。

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

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

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

成功時回傳結果，或在失敗時引發一個例外□回傳 *NULL*。

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

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

成功時回傳結果，或在失敗時引發一個例外□回傳 *NULL*。

在 3.9 版新加入.

PyObject ***PyObject_CallMethodOneArg**(*PyObject* *obj, *PyObject* *name, *PyObject* *arg)

附帶一個位置引數 *arg* 地呼叫 Python 物件 *obj* 中的一個 method，其中 method 名稱由 *name* 中的 Python 字串物件給定。

成功時回傳結果，或在失敗時引發一個例外□回傳 *NULL*。

在 3.9 版新加入.

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

呼叫一個可呼叫的 Python 物件 *callable*。附帶引數與`vectorcallfunc` 的相同。如果 *callable* 支援*vectorcall*，則它會直接呼叫存放在 *callable* 中的 vectorcall 函式。

成功時回傳結果，或在失敗時引發一個例外□回傳 *NULL*。

在 3.9 版新加入.

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

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

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

在 3.9 版新加入.

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

Call a method using the vectorcall calling convention. The name of the method is given as a Python string *name*. The object whose method is called is *args[0]*, and the *args* array starting at *args[1]* represents the arguments of the call. There must be at least one positional argument. *nargsf* is the number of positional arguments including *args[0]*, plus `PY_VECTORCALL_ARGUMENTS_OFFSET` if the value of `args[0]` may temporarily be changed. Keyword arguments can be passed just like in `PyObject_Vectorcall()`.

If the object has the `Py_TPFLAGS_METHOD_DESCRIPTOR` feature, this will call the unbound method object with the full *args* vector as arguments.

成功時回傳結果，或在失敗時引發一個例外並回傳 *NULL*。

在 3.9 版新加入.

7.2.4 呼叫支援 API

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

Part of the Stable ABI. 判定物件 *o* 是否為可呼叫的。如果物件是可呼叫物件則回傳 `1`，其他情況回傳 `0`。這個函式不會呼叫失敗。

7.3 Number Protocol

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

Part of the Stable ABI. Returns `1` if the object *o* provides numeric protocols, and false otherwise. This function always succeeds.

在 3.8 版的變更: Returns `1` if *o* is an index integer.

PyObject ***PyNumber_Add**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。*Part of the* Stable ABI. Returns the result of adding *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 + o2`.

PyObject ***PyNumber_Subtract**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。*Part of the* Stable ABI. Returns the result of subtracting *o2* from *o1*, or `NULL` on failure. This is the equivalent of the Python expression `o1 - o2`.

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

回傳值：新的參照。*Part of the* Stable ABI. Returns the result of multiplying *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 * o2`.

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

回傳值：新的參照。*Part of the* Stable ABI *since version 3.7.* Returns the result of matrix multiplication on *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 @ o2`.

在 3.5 版新加入.

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

回傳值：新的參照。*Part of the* Stable ABI. Return the floor of *o1* divided by *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 // o2`.

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

回傳值：新的參照。*Part of the* Stable ABI. Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or `NULL` on failure. The return value is "approximate" because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers. This is the equivalent of the Python expression `o1 / o2`.

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

回傳值：新的參照。*Part of the* Stable ABI. Returns the remainder of dividing *o1* by *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 % o2`.

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

回傳值：新的參照。*Part of the* Stable ABI. See the built-in function `divmod()`. Returns `NULL` on failure. This is the equivalent of the Python expression `divmod(o1, o2)`.

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

回傳值：新的參照。*Part of the* Stable ABI. See the built-in function `pow()`. Returns `NULL` on failure. This is the equivalent of the Python expression `pow(o1, o2, o3)`, where *o3* is optional. If *o3* is to be ignored, pass `Py_None` in its place (passing `NULL` for *o3* would cause an illegal memory access).

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

回傳值：新的參照。*Part of the* Stable ABI. Returns the negation of *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `-o`.

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

回傳值：新的參照。*Part of the* Stable ABI. Returns *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `+o`.

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

回傳值：新的參照。*Part of the* Stable ABI. Returns the absolute value of *o*, or `NULL` on failure. This is the equivalent of the Python expression `abs(o)`.

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

回傳值：新的參照。*Part of the* Stable ABI. Returns the bitwise negation of *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `~o`.

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

回傳值：新的參照。*Part of the* Stable ABI. Returns the result of left shifting *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 << o2`.

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

回傳值：新的參照。*Part of the* Stable ABI. Returns the result of right shifting *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 >> o2`.

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

回傳值：新的參照。*Part of the* Stable ABI. Returns the "bitwise and" of *o1* and *o2* on success and `NULL` on failure. This is the equivalent of the Python expression `o1 & o2`.

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

回傳值：新的參照。*Part of the* Stable ABI. Returns the "bitwise exclusive or" of *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 ^ o2`.

PyObject ***PyNumber_Or**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。*Part of the* Stable ABI. Returns the "bitwise or" of *o1* and *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 | o2`.

PyObject ***PyNumber_InPlaceAdd**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。*Part of the* Stable ABI. Returns the result of adding *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 += o2`.

PyObject ***PyNumber_InPlaceSubtract**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。*Part of the* Stable ABI. Returns the result of subtracting *o2* from *o1*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 -= o2`.

PyObject ***PyNumber_InPlaceMultiply**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。*Part of the* Stable ABI. Returns the result of multiplying *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 *= o2`.

PyObject ***PyNumber_InPlaceMatrixMultiply**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。*Part of the* Stable ABI *since version 3.7.* Returns the result of matrix multiplication on *o1* and *o2*, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 @= o2`.

在 3.5 版新加入.

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

回傳值：新的參照。*Part of the* Stable ABI. Returns the mathematical floor of dividing *o1* by *o2*, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 //= o2`.

PyObject ***PyNumber_InPlaceTrueDivide**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。*Part of the* Stable ABI. Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or NULL on failure. The return value is "approximate" because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 /= o2`.

PyObject ***PyNumber_InPlaceRemainder**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。*Part of the* Stable ABI. Returns the remainder of dividing *o1* by *o2*, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 %= o2`.

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

回傳值：新的參照。*Part of the* Stable ABI. See the built-in function `pow()`. Returns NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 **= o2` when o3 is *Py_None*, or an in-place variant of `pow(o1, o2, o3)` otherwise. If *o3* is to be ignored, pass *Py_None* in its place (passing NULL for *o3* would cause an illegal memory access).

PyObject ***PyNumber_InPlaceLshift**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。*Part of the* Stable ABI. Returns the result of left shifting *o1* by *o2* on success, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 <<= o2`.

PyObject ***PyNumber_InPlaceRshift**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。*Part of the* Stable ABI. Returns the result of right shifting *o1* by *o2* on success, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 >>= o2`.

PyObject ***PyNumber_InPlaceAnd**(*PyObject* *o1, *PyObject* *o2)

回傳值：新的參照。*Part of the* Stable ABI. Returns the "bitwise and" of *o1* and *o2* on success and NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 &= o2`.

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

回傳值：新的參照。*Part of the* Stable ABI. Returns the "bitwise exclusive or" of *o1* by *o2* on success, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 ^= o2`.

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

回傳值：新的參照。*Part of the* Stable ABI. Returns the "bitwise or" of *o1* and *o2* on success, or NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 |= o2`.

PyObject ***PyNumber_Long**(*PyObject* *o)

回傳值：新的參照。*Part of the* Stable ABI. Returns the *o* converted to an integer object on success, or NULL on failure. This is the equivalent of the Python expression `int(o)`.

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

回傳值：新的參照。*Part of the* Stable ABI. Returns the *o* converted to a float object on success, or NULL on failure. This is the equivalent of the Python expression `float(o)`.

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

回傳值：新的參照。 *Part of the* Stable ABI. Returns the *o* converted to a Python int on success or NULL with a TypeError exception raised on failure.

在 3.10 版的變更: The result always has exact type int. Previously, the result could have been an instance of a subclass of int.

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

回傳值：新的參照。*Part of the* Stable ABI. Returns the integer *n* converted to base *base* as a string. The *base* argument must be one of 2, 8, 10, or 16. For base 2, 8, or 16, the returned string is prefixed with a base marker of '0b', '0o', or '0x', respectively. If *n* is not a Python int, it is converted with *PyNumber_Index()* first.

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

Part of the Stable ABI. Returns *o* converted to a *Py_ssize_t* value if *o* can be interpreted as an integer. If the call fails, an exception is raised and -1 is returned.

If *o* can be converted to a Python int but the attempt to convert to a *Py_ssize_t* value would raise an OverflowError, then the *exc* argument is the type of exception that will be raised (usually IndexError or OverflowError). If *exc* is NULL, then the exception is cleared and the value is clipped to PY_SSIZE_T_MIN for a negative integer or PY_SSIZE_T_MAX for a positive integer.

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

Part of the Stable ABI *since version 3.8.* Returns 1 if *o* is an index integer (has the nb_index slot of the tp_as_number structure filled in), and 0 otherwise. This function always succeeds.

7.4 Sequence Protocol

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

Part of the Stable ABI. Return 1 if the object provides the sequence protocol, and 0 otherwise. Note that it returns 1 for Python classes with a __getitem__() method, unless they are dict subclasses, since in general it is impossible to determine what type of keys the class supports. This function always succeeds.

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

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

Part of the Stable ABI. Returns the number of objects in sequence *o* on success, and -1 on failure. This is equivalent to the Python expression len(o).

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

回傳值：新的參照。 *Part of the* Stable ABI. Return the concatenation of *o1* and *o2* on success, and NULL on failure. This is the equivalent of the Python expression o1 + o2.

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

回傳值：新的參照。 *Part of the* Stable ABI. Return the result of repeating sequence object *o* *count* times, or NULL on failure. This is the equivalent of the Python expression o * count.

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

回傳值：新的參照。 *Part of the* Stable ABI. Return the concatenation of *o1* and *o2* on success, and NULL on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python expression o1 += o2.

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

回傳值：新的參照。 *Part of the* Stable ABI. Return the result of repeating sequence object *o* *count* times, or NULL on failure. The operation is done *in-place* when *o* supports it. This is the equivalent of the Python expression o *= count.

PyObject ***PySequence_GetItem**(*PyObject* *o, *Py_ssize_t* i)

回傳值：新的參照。 *Part of the* Stable ABI. Return the *i*th element of *o*, or NULL on failure. This is the equivalent of the Python expression o[i].

PyObject ***PySequence_GetSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

回傳值：新的參照。*Part of the* Stable ABI. Return the slice of sequence object *o* between *i1* and *i2*, or `NULL` on failure. This is the equivalent of the Python expression `o[i1:i2]`.

int **PySequence_SetItem**(*PyObject* *o, *Py_ssize_t* i, *PyObject* *v)

Part of the Stable ABI. Assign object *v* to the *i*th element of *o*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o[i] = v`. This function *does not* steal a reference to *v*.

If *v* is `NULL`, the element is deleted, but this feature is deprecated in favour of using `PySequence_DelItem()`.

int **PySequence_DelItem**(*PyObject* *o, *Py_ssize_t* i)

Part of the Stable ABI. Delete the *i*th element of object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o[i]`.

int **PySequence_SetSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2, *PyObject* *v)

Part of the Stable ABI. Assign the sequence object *v* to the slice in sequence object *o* from *i1* to *i2*. This is the equivalent of the Python statement `o[i1:i2] = v`.

int **PySequence_DelSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

Part of the Stable ABI. Delete the slice in sequence object *o* from *i1* to *i2*. Returns `-1` on failure. This is the equivalent of the Python statement `del o[i1:i2]`.

Py_ssize_t **PySequence_Count**(*PyObject* *o, *PyObject* *value)

Part of the Stable ABI. Return the number of occurrences of *value* in *o*, that is, return the number of keys for which `o[key] == value`. On failure, return `-1`. This is equivalent to the Python expression `o.count(value)`.

int **PySequence_Contains**(*PyObject* *o, *PyObject* *value)

Part of the Stable ABI. Determine if *o* contains *value*. If an item in *o* is equal to *value*, return `1`, otherwise return `0`. On error, return `-1`. This is equivalent to the Python expression `value in o`.

Py_ssize_t **PySequence_Index**(*PyObject* *o, *PyObject* *value)

Part of the Stable ABI. Return the first index *i* for which `o[i] == value`. On error, return `-1`. This is equivalent to the Python expression `o.index(value)`.

PyObject ***PySequence_List**(*PyObject* *o)

回傳值：新的參照。 *Part of the* Stable ABI. Return a list object with the same contents as the sequence or iterable *o*, or `NULL` on failure. The returned list is guaranteed to be new. This is equivalent to the Python expression `list(o)`.

PyObject ***PySequence_Tuple**(*PyObject* *o)

回傳值：新的參照。 *Part of the* Stable ABI. Return a tuple object with the same contents as the sequence or iterable *o*, or `NULL` on failure. If *o* is a tuple, a new reference will be returned, otherwise a tuple will be constructed with the appropriate contents. This is equivalent to the Python expression `tuple(o)`.

PyObject ***PySequence_Fast**(*PyObject* *o, const char *m)

回傳值：新的參照。*Part of the* Stable ABI. Return the sequence or iterable *o* as an object usable by the other `PySequence_Fast*` family of functions. If the object is not a sequence or iterable, raises `TypeError` with *m* as the message text. Returns `NULL` on failure.

The `PySequence_Fast*` functions are thus named because they assume *o* is a `PyTupleObject` or a `PyListObject` and access the data fields of *o* directly.

As a CPython implementation detail, if *o* is already a sequence or list, it will be returned.

Py_ssize_t **PySequence_Fast_GET_SIZE**(*PyObject* *o)

Returns the length of *o*, assuming that *o* was returned by `PySequence_Fast()` and that *o* is not `NULL`. The size can also be retrieved by calling `PySequence_Size()` on *o*, but `PySequence_Fast_GET_SIZE()` is faster because it can assume *o* is a list or tuple.

PyObject ***PySequence_Fast_GET_ITEM**(*PyObject* *o, *Py_ssize_t* i)

回傳值：借用參照。Return the *i*th element of *o*, assuming that *o* was returned by `PySequence_Fast()`, *o* is not `NULL`, and that *i* is within bounds.

PyObject ****PySequence_Fast_ITEMS**(*PyObject* *o)

Return the underlying array of PyObject pointers. Assumes that *o* was returned by `PySequence_Fast()` and *o* is not `NULL`.

Note, if a list gets resized, the reallocation may relocate the items array. So, only use the underlying array pointer in contexts where the sequence cannot change.

PyObject ***PySequence_ITEM**(*PyObject* *o, *Py_ssize_t* i)

回傳值：新的參照。 Return the *i*th element of *o* or `NULL` on failure. Faster form of `PySequence_GetItem()` but without checking that `PySequence_Check()` on *o* is true and without adjustment for negative indices.

7.5 Mapping Protocol

See also `PyObject_GetItem()`, `PyObject_SetItem()` and `PyObject_DelItem()`.

int **PyMapping_Check**(*PyObject* *o)

Part of the Stable ABI. Return `1` if the object provides the mapping protocol or supports slicing, and `0` otherwise. Note that it returns `1` for Python classes with a `__getitem__()` method, since in general it is impossible to determine what type of keys the class supports. This function always succeeds.

Py_ssize_t **PyMapping_Size**(*PyObject* *o)

Py_ssize_t **PyMapping_Length**(*PyObject* *o)

Part of the Stable ABI. Returns the number of keys in object *o* on success, and `-1` on failure. This is equivalent to the Python expression `len(o)`.

PyObject ***PyMapping_GetItemString**(*PyObject* *o, const char *key)

回傳值：新的參照。 *Part of the* Stable ABI. This is the same as `PyObject_GetItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

int **PyMapping_SetItemString**(*PyObject* *o, const char *key, *PyObject* *v)

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

int **PyMapping_DelItem**(*PyObject* *o, *PyObject* *key)

This is an alias of `PyObject_DelItem()`.

int **PyMapping_DelItemString**(*PyObject* *o, const char *key)

This is the same as `PyObject_DelItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

int **PyMapping_HasKey**(*PyObject* *o, *PyObject* *key)

Part of the Stable ABI. Return `1` if the mapping object has the key *key* and `0` otherwise. This is equivalent to the Python expression `key in o`. This function always succeeds.

備註： Exceptions which occur when this calls `__getitem__()` method are silently ignored. For proper error handling, use `PyObject_GetItem()` instead.

int **PyMapping_HasKeyString**(*PyObject* *o, const char *key)

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

備F: Exceptions that occur when this calls `__getitem__()` method or while creating the temporary `str` object are silently ignored. For proper error handling, use `PyMapping_GetItemString()` instead.

PyObject ***PyMapping_Keys**(*PyObject* *o)

回傳值：新的參照。*Part of the* Stable ABI. On success, return a list of the keys in object *o*. On failure, return `NULL`.

在 3.7 版的變更: Previously, the function returned a list or a tuple.

PyObject ***PyMapping_Values**(*PyObject* *o)

回傳值：新的參照。*Part of the* Stable ABI. On success, return a list of the values in object *o*. On failure, return `NULL`.

在 3.7 版的變更: Previously, the function returned a list or a tuple.

PyObject ***PyMapping_Items**(*PyObject* *o)

回傳值：新的參照。*Part of the* Stable ABI. On success, return a list of the items in object *o*, where each item is a tuple containing a key-value pair. On failure, return `NULL`.

在 3.7 版的變更: Previously, the function returned a list or a tuple.

7.6 F代器協議

有兩個專門用於F代器的函式。

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

Part of the Stable ABI *since version 3.8.* 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)

Part of the Stable ABI *since version 3.10.* 如果物件 *o* 有提供 `AsyncIterator` 協議，則回傳非零，否則回傳 0。這個函式一定會執行成功。

在 3.10 版新加入.

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

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

要編寫一個F代於F代器的F圈，C 程式碼應該會像這樣：

```
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;

if (iterator == NULL) {
    /* propagate error */
}

while ((item = PyIter_Next(iterator))) {
    /* do something with item */
    ...
    /* release reference when done */
    Py_DECREF(item);
}

Py_DECREF(iterator);

if (PyErr_Occurred()) {
```

(繼續下一頁)

(繼續上一頁)

```
    /* propagate error */
}
else {
    /* continue doing useful work */
}
```

type **PySendResult**

用於表示*PyIter_Send()* 不同結果的列舉 (enum) 值。

在 3.10 版新加入.

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

Part of the Stable ABI *since version 3.10.* 將 *arg* 值發送到迭代器 *iter* 中。回傳：

- 如果迭代器有回傳則為 PYGEN_RETURN。回傳值透過 *presult* 回傳。
- 如果迭代器有產生 (yield) 則為 PYGEN_NEXT。產生值透過 *presult* 回傳。
- 如果迭代器引發例外則為 PYGEN_ERROR。*presult* 被設定為 NULL。

在 3.10 版新加入.

7.7 緩沖協定 (Buffer Protocol)

Certain objects available in Python wrap access to an underlying memory array or *buffer*. Such objects include the built-in bytes and bytearray, and some extension types like array.array. Third-party libraries may define their own types for special purposes, such as image processing or numeric analysis.

While each of these types have their own semantics, they share the common characteristic of being backed by a possibly large memory buffer. It is then desirable, in some situations, to access that buffer directly and without intermediate copying.

Python provides such a facility at the C level in the form of the *buffer protocol*. This protocol has two sides:

- on the producer side, a type can export a "buffer interface" which allows objects of that type to expose information about their underlying buffer. This interface is described in the section *Buffer Object Structures*;
- on the consumer side, several means are available to obtain a pointer to the raw underlying data of an object (for example a method parameter).

Simple objects such as bytes and bytearray expose their underlying buffer in byte-oriented form. Other forms are possible; for example, the elements exposed by an array.array can be multi-byte values.

An example consumer of the buffer interface is the write() method of file objects: any object that can export a series of bytes through the buffer interface can be written to a file. While write() only needs read-only access to the internal contents of the object passed to it, other methods such as readinto() need write access to the contents of their argument. The buffer interface allows objects to selectively allow or reject exporting of read-write and read-only buffers.

There are two ways for a consumer of the buffer interface to acquire a buffer over a target object:

- call *PyObject_GetBuffer()* with the right parameters;
- call *PyArg_ParseTuple()* (or one of its siblings) with one of the y*, w* or s* *format codes*.

In both cases, *PyBuffer_Release()* must be called when the buffer isn't needed anymore. Failure to do so could lead to various issues such as resource leaks.

7.7.1 Buffer structure

Buffer structures (or simply "buffers") are useful as a way to expose the binary data from another object to the Python programmer. They can also be used as a zero-copy slicing mechanism. Using their ability to reference a block of memory, it is possible to expose any data to the Python programmer quite easily. The memory could be a large, constant array in a C extension, it could be a raw block of memory for manipulation before passing to an operating system library, or it could be used to pass around structured data in its native, in-memory format.

Contrary to most data types exposed by the Python interpreter, buffers are not `PyObject` pointers but rather simple C structures. This allows them to be created and copied very simply. When a generic wrapper around a buffer is needed, a *memoryview* object can be created.

For short instructions how to write an exporting object, see *Buffer Object Structures*. For obtaining a buffer, see `PyObject_GetBuffer()`.

type **`Py_buffer`**

Part of the Stable ABI *(including all members) since version 3.11.*

void ***`buf`**

A pointer to the start of the logical structure described by the buffer fields. This can be any location within the underlying physical memory block of the exporter. For example, with negative `strides` the value may point to the end of the memory block.

For *contiguous* arrays, the value points to the beginning of the memory block.

PyObject ***`obj`**

A new reference to the exporting object. The reference is owned by the consumer and automatically released (i.e. reference count decremented) and set to `NULL` by `PyBuffer_Release()`. The field is the equivalent of the return value of any standard C-API function.

As a special case, for *temporary* buffers that are wrapped by `PyMemoryView_FromBuffer()` or `PyBuffer_FillInfo()` this field is `NULL`. In general, exporting objects MUST NOT use this scheme.

Py_ssize_t **`len`**

`product(shape) * itemsize`. For contiguous arrays, this is the length of the underlying memory block. For non-contiguous arrays, it is the length that the logical structure would have if it were copied to a contiguous representation.

Accessing `((char *)buf)[0] up to ((char *)buf)[len-1]` is only valid if the buffer has been obtained by a request that guarantees contiguity. In most cases such a request will be `PyBUF_SIMPLE` or `PyBUF_WRITABLE`.

int **`readonly`**

An indicator of whether the buffer is read-only. This field is controlled by the `PyBUF_WRITABLE` flag.

Py_ssize_t **`itemsize`**

Item size in bytes of a single element. Same as the value of `struct.calcsize()` called on non-`NULL` `format` values.

Important exception: If a consumer requests a buffer without the `PyBUF_FORMAT` flag, `format` will be set to `NULL`, but `itemsize` still has the value for the original format.

If `shape` is present, the equality `product(shape) * itemsize == len` still holds and the consumer can use `itemsize` to navigate the buffer.

If `shape` is `NULL` as a result of a `PyBUF_SIMPLE` or a `PyBUF_WRITABLE` request, the consumer must disregard `itemsize` and assume `itemsize == 1`.

const char ***`format`**

A *NUL* terminated string in `struct` module style syntax describing the contents of a single item. If this is `NULL`, `"B"` (unsigned bytes) is assumed.

This field is controlled by the `PyBUF_FORMAT` flag.

int **ndim**

The number of dimensions the memory represents as an n-dimensional array. If it is `0`, `buf` points to a single item representing a scalar. In this case, `shape`, `strides` and `suboffsets` MUST be `NULL`. The maximum number of dimensions is given by `PyBUF_MAX_NDIM`.

Py_ssize_t ***shape**

An array of `Py_ssize_t` of length `ndim` indicating the shape of the memory as an n-dimensional array. Note that `shape[0] * ... * shape[ndim-1] * itemsize` MUST be equal to `len`.

Shape values are restricted to `shape[n] >= 0`. The case `shape[n] == 0` requires special attention. See *complex arrays* for further information.

The shape array is read-only for the consumer.

Py_ssize_t ***strides**

An array of `Py_ssize_t` of length `ndim` giving the number of bytes to skip to get to a new element in each dimension.

Stride values can be any integer. For regular arrays, strides are usually positive, but a consumer MUST be able to handle the case `strides[n] <= 0`. See *complex arrays* for further information.

The strides array is read-only for the consumer.

Py_ssize_t ***suboffsets**

An array of `Py_ssize_t` of length `ndim`. If `suboffsets[n] >= 0`, the values stored along the nth dimension are pointers and the suboffset value dictates how many bytes to add to each pointer after de-referencing. A suboffset value that is negative indicates that no de-referencing should occur (striding in a contiguous memory block).

If all suboffsets are negative (i.e. no de-referencing is needed), then this field must be `NULL` (the default value).

This type of array representation is used by the Python Imaging Library (PIL). See *complex arrays* for further information how to access elements of such an array.

The suboffsets array is read-only for the consumer.

void ***internal**

This is for use internally by the exporting object. For example, this might be re-cast as an integer by the exporter and used to store flags about whether or not the shape, strides, and suboffsets arrays must be freed when the buffer is released. The consumer MUST NOT alter this value.

Constants:

PyBUF_MAX_NDIM

The maximum number of dimensions the memory represents. Exporters MUST respect this limit, consumers of multi-dimensional buffers SHOULD be able to handle up to `PyBUF_MAX_NDIM` dimensions. Currently set to 64.

7.7.2 Buffer request types

Buffers are usually obtained by sending a buffer request to an exporting object via `PyObject_GetBuffer()`. Since the complexity of the logical structure of the memory can vary drastically, the consumer uses the *flags* argument to specify the exact buffer type it can handle.

All `Py_buffer` fields are unambiguously defined by the request type.

request-independent fields

The following fields are not influenced by *flags* and must always be filled in with the correct values: `obj`, `buf`, `len`, `itemsize`, `ndim`.

readonly, format

PyBUF_WRITABLE

Controls the `readonly` field. If set, the exporter MUST provide a writable buffer or else report failure. Otherwise, the exporter MAY provide either a read-only or writable buffer, but the choice MUST be consistent for all consumers.

PyBUF_FORMAT

Controls the `format` field. If set, this field MUST be filled in correctly. Otherwise, this field MUST be `NULL`.

`PyBUF_WRITABLE` can be |'d to any of the flags in the next section. Since `PyBUF_SIMPLE` is defined as 0, `PyBUF_WRITABLE` can be used as a stand-alone flag to request a simple writable buffer.

`PyBUF_FORMAT` can be |'d to any of the flags except `PyBUF_SIMPLE`. The latter already implies format `B` (unsigned bytes).

shape, strides, suboffsets

The flags that control the logical structure of the memory are listed in decreasing order of complexity. Note that each flag contains all bits of the flags below it.

Request	shape	strides	suboffsets
PyBUF_INDIRECT	yes	yes	if needed
PyBUF_STRIDES	yes	yes	NULL
PyBUF_ND	yes	NULL	NULL
PyBUF_SIMPLE	NULL	NULL	NULL

contiguity requests

C or Fortran *contiguity* can be explicitly requested, with and without stride information. Without stride information, the buffer must be C-contiguous.

Request	shape	strides	suboffsets	contig
PyBUF_C_CONTIGUOUS	yes	yes	NULL	C
PyBUF_F_CONTIGUOUS	yes	yes	NULL	F
PyBUF_ANY_CONTIGUOUS	yes	yes	NULL	C 或 F
PyBUF_ND	yes	NULL	NULL	C

compound requests

All possible requests are fully defined by some combination of the flags in the previous section. For convenience, the buffer protocol provides frequently used combinations as single flags.

In the following table *U* stands for undefined contiguity. The consumer would have to call *PyBuffer_IsContiguous()* to determine contiguity.

Request	shape	strides	suboffsets	contig	readonly	format
PyBUF_FULL	yes	yes	if needed	U	0	yes
PyBUF_FULL_RO	yes	yes	if needed	U	1 或 0	yes
PyBUF_RECORDS	yes	yes	NULL	U	0	yes
PyBUF_RECORDS_RO	yes	yes	NULL	U	1 或 0	yes
PyBUF_STRIDED	yes	yes	NULL	U	0	NULL
PyBUF_STRIDED_RO	yes	yes	NULL	U	1 或 0	NULL
PyBUF_CONTIG	yes	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	yes	NULL	NULL	C	1 或 0	NULL

7.7.3 Complex arrays

NumPy-style: shape and strides

The logical structure of NumPy-style arrays is defined by *itemsize*, *ndim*, *shape* and *strides*.

If `ndim == 0`, the memory location pointed to by *buf* is interpreted as a scalar of size *itemsize*. In that case, both *shape* and *strides* are `NULL`.

If *strides* is `NULL`, the array is interpreted as a standard n-dimensional C-array. Otherwise, the consumer must access an n-dimensional array as follows:

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

As noted above, *buf* can point to any location within the actual memory block. An exporter can check the validity of a buffer with this function:

```
def verify_structure(memlen, itemsize, ndim, shape, strides, offset):
    """Verify that the parameters represent a valid array within
       the bounds of the allocated memory:
           char *mem: start of the physical memory block
           memlen: length of the physical memory block
           offset: (char *)buf - mem
    """
    if offset % itemsize:
        return False
    if offset < 0 or offset+itemsize > memlen:
        return False
    if any(v % itemsize for v in strides):
        return False

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

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

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

PIL-style: shape, strides and suboffsets

In addition to the regular items, PIL-style arrays can contain pointers that must be followed in order to get to the next element in a dimension. For example, the regular three-dimensional C-array `char v[2][2][3]` can also be viewed as an array of 2 pointers to 2 two-dimensional arrays: `char (*v[2])[2][3]`. In suboffsets representation, those two pointers can be embedded at the start of *buf*, pointing to two `char x[2][3]` arrays that can be located anywhere in memory.

Here is a function that returns a pointer to the element in an N-D array pointed to by an N-dimensional index when there are both non-`NULL` strides and suboffsets:

```
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
                       Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
```

(繼續下一頁)

(繼續上一頁)

```
        if (suboffsets[i] >=0 ) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

7.7.4 Buffer-related functions

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

Part of the Stable ABI *since version 3.11.* Return `1` if *obj* supports the buffer interface otherwise `0`. When `1` is returned, it doesn't guarantee that `PyObject_GetBuffer()` will succeed. This function always succeeds.

int **PyObject_GetBuffer** (*PyObject* *exporter, *Py_buffer* *view, int flags)

Part of the Stable ABI *since version 3.11.* Send a request to *exporter* to fill in *view* as specified by *flags*. If the exporter cannot provide a buffer of the exact type, it MUST raise `BufferError`, set `view->obj` to `NULL` and return `-1`.

On success, fill in *view*, set `view->obj` to a new reference to *exporter* and return 0. In the case of chained buffer providers that redirect requests to a single object, `view->obj` MAY refer to this object instead of *exporter* (See *Buffer Object Structures*).

Successful calls to `PyObject_GetBuffer()` must be paired with calls to `PyBuffer_Release()`, similar to `malloc()` and `free()`. Thus, after the consumer is done with the buffer, `PyBuffer_Release()` must be called exactly once.

void **PyBuffer_Release** (*Py_buffer* *view)

Part of the Stable ABI *since version 3.11.* Release the buffer *view* and release the *strong reference* (i.e. decrement the reference count) to the view's supporting object, `view->obj`. This function MUST be called when the buffer is no longer being used, otherwise reference leaks may occur.

It is an error to call this function on a buffer that was not obtained via `PyObject_GetBuffer()`.

Py_ssize_t **PyBuffer_SizeFromFormat** (const char *format)

Part of the Stable ABI *since version 3.11.* Return the implied `itemsize` from `format`. On error, raise an exception and return -1.

在 3.9 版新加入.

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

Part of the Stable ABI *since version 3.11.* Return `1` if the memory defined by the *view* is C-style (*order* is `'C'`) or Fortran-style (*order* is `'F'`) *contiguous* or either one (*order* is `'A'`). Return `0` otherwise. This function always succeeds.

void ***PyBuffer_GetPointer** (const *Py_buffer* *view, const *Py_ssize_t* *indices)

Part of the Stable ABI *since version 3.11.* Get the memory area pointed to by the *indices* inside the given *view*. *indices* must point to an array of `view->ndim` indices.

int **PyBuffer_FromContiguous** (const *Py_buffer* *view, const void *buf, *Py_ssize_t* len, char fort)

Part of the Stable ABI *since version 3.11.* Copy contiguous *len* bytes from *buf* to *view*. *fort* can be `'C'` or `'F'` (for C-style or Fortran-style ordering). `0` is returned on success, `-1` on error.

int **PyBuffer_ToContiguous** (void *buf, const *Py_buffer* *src, *Py_ssize_t* len, char order)

Part of the Stable ABI *since version 3.11.* Copy *len* bytes from *src* to its contiguous representation in *buf*. *order* can be `'C'` or `'F'` or `'A'` (for C-style or Fortran-style ordering or either one). `0` is returned on success, `-1` on error.

This function fails if *len* != *src->len*.

int **PyObject_CopyData** (*PyObject* *dest, *PyObject* *src)

Part of the Stable ABI *since version 3.11.* Copy data from *src* to *dest* buffer. Can convert between C-style and or Fortran-style buffers.

`0` is returned on success, `-1` on error.

void **PyBuffer_FillContiguousStrides** (int ndims, *Py_ssize_t* *shape, *Py_ssize_t* *strides, int itemsize, char order)

Part of the Stable ABI *since version 3.11.* Fill the *strides* array with byte-strides of a *contiguous* (C-style if *order* is `'C'` or Fortran-style if *order* is `'F'`) array of the given shape with the given number of bytes per element.

int **PyBuffer_FillInfo** (*Py_buffer* *view, *PyObject* *exporter, void *buf, *Py_ssize_t* len, int readonly, int flags)

Part of the Stable ABI *since version 3.11.* Handle buffer requests for an exporter that wants to expose *buf* of size *len* with writability set according to *readonly*. *buf* is interpreted as a sequence of unsigned bytes.

The *flags* argument indicates the request type. This function always fills in *view* as specified by flags, unless *buf* has been designated as read-only and *PyBUF_WRITABLE* is set in *flags*.

On success, set `view->obj` to a new reference to *exporter* and return 0. Otherwise, raise `BufferError`, set `view->obj` to `NULL` and return `-1`;

If this function is used as part of a *getbufferproc*, *exporter* MUST be set to the exporting object and *flags* must be passed unmodified. Otherwise, *exporter* MUST be `NULL`.

7.8 舊式緩沖協定 (Buffer Protocol)

在 3.0 版之後被棄用.

這些函式是 Python 2 中「舊式緩衝區協定」API 的一部分。在 Python 3 中，該協議已經不存在，但這些函式仍有公開以供移植 2.x 程式碼。它們充當新式緩衝區協定的相容性包裝器，但它們無法讓你控制匯出 (export) 緩衝區時所獲取資源的生命週期。

因此，建議你呼叫*PyObject_GetBuffer()*（或是以 `y*` 或 `w*` *格式碼 (format code)* 呼叫*PyArg_ParseTuple()* 系列函式）獲取物件的緩衝區視圖 (buffer view)，以及緩衝區視圖可被釋放時呼叫*PyBuffer_Release()*。

int **PyObject_AsCharBuffer** (*PyObject* *obj, const char **buffer, *Py_ssize_t* *buffer_len)

Part of the Stable ABI. 回傳一個指向可用作基於字元輸入之唯讀記憶體位置的指標。*obj* 引數必須支援單一片段 (single-segment) 字元緩衝區介面。成功時回傳 `0`，並將 *buffer* 設定為記憶體位置、將 *buffer_len* 設定為緩衝區長度。回傳 `-1` 並在錯誤時設定 `TypeError`。

int **PyObject_AsReadBuffer** (*PyObject* *obj, const void **buffer, *Py_ssize_t* *buffer_len)

Part of the Stable ABI. 回傳一個指向包含任意資料之唯讀記憶體位置的指標。*obj* 引數必須支援單一片段可讀緩衝區介面。成功時回傳 `0`，並將 *buffer* 設定為記憶體位置、將 *buffer_len* 設定為緩衝區長度。回傳 `-1` 並在錯誤時設定 `TypeError`。

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

Part of the Stable ABI. 如果 *o* 支援單一片段可讀緩衝區介面，則回傳 `1`，否則回傳 `0`。這個函式一定會執行成功的。

請注意，該函式嘗試獲取和釋放緩衝區，並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)

Part of the Stable ABI. 回傳指向可寫記憶體位置的指標。*obj* 引數必須支援單一片段字元緩衝區介面。成功時回傳 `0`，並將 *buffer* 設定為記憶體位置，且將 *buffer_len* 設定為緩衝區長度。回傳 `-1` 並在錯誤時設定 `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 Type Objects

type **PyTypeObject**

Part of the Limited API *(as an opaque struct).* The C structure of the objects used to describe built-in types.

PyTypeObject **PyType_Type**

Part of the Stable ABI. This is the type object for type objects; it is the same object as `type` in the Python layer.

int **PyType_Check**(*PyObject* *o)

Return non-zero if the object *o* is a type object, including instances of types derived from the standard type object. Return 0 in all other cases. This function always succeeds.

int **PyType_CheckExact**(*PyObject* *o)

Return non-zero if the object *o* is a type object, but not a subtype of the standard type object. Return 0 in all other cases. This function always succeeds.

unsigned int **PyType_ClearCache**()

Part of the Stable ABI. Clear the internal lookup cache. Return the current version tag.

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

Part of the Stable ABI. Return the *tp_flags* member of *type*. This function is primarily meant for use with `Py_LIMITED_API`; the individual flag bits are guaranteed to be stable across Python releases, but access to *tp_flags* itself is not part of the *limited API*.

在 3.2 版新加入.

在 3.4 版的變更: The return type is now `unsigned long` rather than `long`.

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

Part of the Stable ABI. Invalidate the internal lookup cache for the type and all of its subtypes. This function must be called after any manual modification of the attributes or base classes of the type.

int **PyType_HasFeature** (*PyTypeObject* *o, int feature)

Return non-zero if the type object *o* sets the feature *feature*. Type features are denoted by single bit flags.

int **PyType_IS_GC** (*PyTypeObject* *o)

Return true if the type object includes support for the cycle detector; this tests the type flag *Py_TPFLAGS_HAVE_GC*.

int **PyType_IsSubtype** (*PyTypeObject* *a, *PyTypeObject* *b)

Part of the Stable ABI. Return true if *a* is a subtype of *b*.

This function only checks for actual subtypes, which means that `__subclasscheck__()` is not called on *b*. Call *PyObject_IsSubclass()* to do the same check that `issubclass()` would do.

PyObject ***PyType_GenericAlloc** (*PyTypeObject* *type, *Py_ssize_t* nitems)

回傳值：新的參照。 *Part of the* Stable ABI. Generic handler for the *tp_alloc* slot of a type object. Use Python's default memory allocation mechanism to allocate a new instance and initialize all its contents to `NULL`.

PyObject ***PyType_GenericNew** (*PyTypeObject* *type, *PyObject* *args, *PyObject* *kwds)

回傳值：新的參照。 *Part of the* Stable ABI. Generic handler for the *tp_new* slot of a type object. Create a new instance using the type's *tp_alloc* slot.

int **PyType_Ready** (*PyTypeObject* *type)

Part of the Stable ABI. Finalize a type object. This should be called on all type objects to finish their initialization. This function is responsible for adding inherited slots from a type's base class. Return `0` on success, or return `-1` and sets an exception on error.

備註: If some of the base classes implements the GC protocol and the provided type does not include the *Py_TPFLAGS_HAVE_GC* in its flags, then the GC protocol will be automatically implemented from its parents. On the contrary, if the type being created does include *Py_TPFLAGS_HAVE_GC* in its flags then it **must** implement the GC protocol itself by at least implementing the *tp_traverse* handle.

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

回傳值：新的參照。*Part of the* Stable ABI *since version 3.11.* Return the type's name. Equivalent to getting the type's `__name__` attribute.

在 3.11 版新加入.

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

回傳值：新的參照。 *Part of the* Stable ABI *since version 3.11.* Return the type's qualified name. Equivalent to getting the type's `__qualname__` attribute.

在 3.11 版新加入.

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

Part of the Stable ABI *since version 3.4.* Return the function pointer stored in the given slot. If the result is `NULL`, this indicates that either the slot is `NULL`, or that the function was called with invalid parameters. Callers will typically cast the result pointer into the appropriate function type.

See `PyType_Slot.slot` for possible values of the *slot* argument.

在 3.4 版新加入.

在 3.10 版的變更: *PyType_GetSlot()* can now accept all types. Previously, it was limited to *heap types*.

PyObject ***PyType_GetModule**(*PyTypeObject* *type)

Part of the Stable ABI *since version 3.10.* Return the module object associated with the given type when the type was created using *PyType_FromModuleAndSpec()*.

If no module is associated with the given type, sets TypeError and returns NULL.

This function is usually used to get the module in which a method is defined. Note that in such a method, PyType_GetModule(Py_TYPE(self)) may not return the intended result. Py_TYPE(self) may be a *subclass* of the intended class, and subclasses are not necessarily defined in the same module as their superclass. See *PyCMethod* to get the class that defines the method. See *PyType_GetModuleByDef()* for cases when PyCMethod cannot be used.

在 3.9 版新加入.

void ***PyType_GetModuleState**(*PyTypeObject* *type)

Part of the Stable ABI *since version 3.10.* Return the state of the module object associated with the given type. This is a shortcut for calling *PyModule_GetState()* on the result of *PyType_GetModule()*.

If no module is associated with the given type, sets TypeError and returns NULL.

If the *type* has an associated module but its state is NULL, returns NULL without setting an exception.

在 3.9 版新加入.

PyObject ***PyType_GetModuleByDef**(*PyTypeObject* *type, struct *PyModuleDef* *def)

Find the first superclass whose module was created from the given *PyModuleDef* *def*, and return that module.

If no module is found, raises a TypeError and returns NULL.

This function is intended to be used together with *PyModule_GetState()* to get module state from slot methods (such as *tp_init* or *nb_add*) and other places where a method's defining class cannot be passed using the *PyCMethod* calling convention.

在 3.11 版新加入.

Creating Heap-Allocated Types

The following functions and structs are used to create *heap types*.

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

回傳值: 新的參照。*Part of the* Stable ABI *since version 3.10.* Creates and returns a *heap type* from the *spec* (*Py_TPFLAGS_HEAPTYPE*).

The *bases* argument can be used to specify base classes; it can either be only one class or a tuple of classes. If *bases* is NULL, the *Py_tp_bases* slot is used instead. If that also is NULL, the *Py_tp_base* slot is used instead. If that also is NULL, the new type derives from object.

The *module* argument can be used to record the module in which the new class is defined. It must be a module object or NULL. If not NULL, the module is associated with the new type and can later be retrieved with *PyType_GetModule()*. The associated module is not inherited by subclasses; it must be specified for each class individually.

This function calls *PyType_Ready()* on the new type.

在 3.9 版新加入.

在 3.10 版的變更: The function now accepts a single class as the *bases* argument and NULL as the tp_doc slot.

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

回傳值：新的參照。*Part of the* Stable ABI *since version 3.3.* 等價於 `PyType_FromModuleAndSpec(NULL, spec, bases)`。

在 3.3 版新加入.

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

回傳值：新的參照。*Part of the* Stable ABI. 等價於 `PyType_FromSpecWithBases(spec, NULL)`。

type **PyType_Spec**

Part of the Stable ABI *(including all members).* Structure defining a type's behavior.

const char *PyType_Spec.**name**

Name of the type, used to set *PyTypeObject.tp_name*.

int PyType_Spec.**basicsize**

int PyType_Spec.**itemsize**

Size of the instance in bytes, used to set *PyTypeObject.tp_basicsize* and *PyTypeObject.tp_itemsize*.

int PyType_Spec.**flags**

Type flags, used to set *PyTypeObject.tp_flags*.

If the `Py_TPFLAGS_HEAPTYPE` flag is not set, *PyType_FromSpecWithBases()* sets it automatically.

PyType_Slot *PyType_Spec.**slots**

Array of *PyType_Slot* structures. Terminated by the special slot value `{0, NULL}`.

type **PyType_Slot**

Part of the Stable ABI *(including all members).* Structure defining optional functionality of a type, containing a slot ID and a value pointer.

int PyType_Slot.**slot**

A slot ID.

Slot IDs are named like the field names of the structures *PyTypeObject*, *PyNumberMethods*, *PySequenceMethods*, *PyMappingMethods* and *PyAsyncMethods* with an added `Py_` prefix. For example, use:

- `Py_tp_dealloc` to set *PyTypeObject.tp_dealloc*
- `Py_nb_add` to set *PyNumberMethods.nb_add*
- `Py_sq_length` to set *PySequenceMethods.sq_length*

The following fields cannot be set at all using *PyType_Spec* and `PyType_Slot`:

- *tp_dict*
- *tp_mro*
- *tp_cache*
- *tp_subclasses*
- *tp_weaklist*
- *tp_vectorcall*
- *tp_weaklistoffset* (see *PyMemberDef*)
- *tp_dictoffset* (see *PyMemberDef*)
- *tp_vectorcall_offset* (see *PyMemberDef*)

Setting Py_tp_bases or Py_tp_base may be problematic on some platforms. To avoid issues, use the *bases* argument of PyType_FromSpecWithBases() instead.

在 3.9 版的變更: Slots in PyBufferProcs may be set in the unlimited API.

在 3.11 版的變更: bf_getbuffer and bf_releasebuffer are now available under the *limited API*.

void *PyType_Slot.**pfunc**

The desired value of the slot. In most cases, this is a pointer to a function.

Slots other than Py_tp_doc may not be NULL.

8.1.2 None 物件

Note that the PyTypeObject for None is not directly exposed in the Python/C API. Since None is a singleton, testing for object identity (using == in C) is sufficient. There is no PyNone_Check() function for the same reason.

PyObject ***Py_None**

The Python None object, denoting lack of value. This object has no methods. It needs to be treated just like any other object with respect to reference counts.

Py_RETURN_NONE

Properly handle returning Py_None from within a C function (that is, increment the reference count of None and return it.)

8.2 數值物件

8.2.1 整數物件

All integers are implemented as "long" integer objects of arbitrary size.

On error, most PyLong_As* APIs return (return type)-1 which cannot be distinguished from a number. Use PyErr_Occurred() to disambiguate.

type **PyLongObject**

Part of the Limited API *(as an opaque struct).* This subtype of PyObject represents a Python integer object.

PyTypeObject **PyLong_Type**

Part of the Stable ABI. This instance of PyTypeObject represents the Python integer type. This is the same object as int in the Python layer.

int **PyLong_Check**(*PyObject* *p)

Return true if its argument is a PyLongObject or a subtype of PyLongObject. This function always succeeds.

int **PyLong_CheckExact**(*PyObject* *p)

Return true if its argument is a PyLongObject, but not a subtype of PyLongObject. This function always succeeds.

PyObject ***PyLong_FromLong**(long v)

回傳值：新的參照。 *Part of the* Stable ABI. Return a new PyLongObject object from *v*, or NULL on failure.

The current implementation keeps an array of integer objects for all integers between -5 and 256. When you create an int in that range you actually just get back a reference to the existing object.

PyObject ***PyLong_FromUnsignedLong*** (unsigned long v)

回傳值：新的參照。*Part of the* Stable ABI. Return a new `PyLongObject` object from a C `unsigned long`, or `NULL` on failure.

PyObject ***PyLong_FromSsize_t*** (*Py_ssize_t* v)

回傳值：新的參照。*Part of the* Stable ABI. Return a new `PyLongObject` object from a C `Py_ssize_t`, or `NULL` on failure.

PyObject ***PyLong_FromSize_t*** (size_t v)

回傳值：新的參照。*Part of the* Stable ABI. Return a new `PyLongObject` object from a C `size_t`, or `NULL` on failure.

PyObject ***PyLong_FromLongLong*** (long long v)

回傳值：新的參照。*Part of the* Stable ABI. Return a new `PyLongObject` object from a C `long long`, or `NULL` on failure.

PyObject ***PyLong_FromUnsignedLongLong*** (unsigned long long v)

回傳值：新的參照。*Part of the* Stable ABI. Return a new `PyLongObject` object from a C `unsigned long long`, or `NULL` on failure.

PyObject ***PyLong_FromDouble*** (double v)

回傳值：新的參照。*Part of the* Stable ABI. Return a new `PyLongObject` object from the integer part of *v*, or `NULL` on failure.

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

回傳值：新的參照。*Part of the* Stable ABI. Return a new `PyLongObject` based on the string value in *str*, which is interpreted according to the radix in *base*. If *pend* is non-`NULL`, **pend* will point to the first character in *str* which follows the representation of the number. If *base* is `0`, *str* is interpreted using the integers definition; in this case, leading zeros in a non-zero decimal number raises a `ValueError`. If *base* is not `0`, it must be between `2` and `36`, inclusive. Leading spaces and single underscores after a base specifier and between digits are ignored. If there are no digits, `ValueError` will be raised.

也參考：

Python methods `int.to_bytes()` and `int.from_bytes()` to convert a `PyLongObject` to/from an array of bytes in base `256`. You can call those from C using `PyObject_CallMethod()`.

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

回傳值：新的參照。 Convert a sequence of Unicode digits in the string *u* to a Python integer value.

在 3.3 版新加入.

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

回傳值：新的參照。*Part of the* Stable ABI. Create a Python integer from the pointer *p*. The pointer value can be retrieved from the resulting value using `PyLong_AsVoidPtr()`.

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

Part of the Stable ABI. Return a C `long` representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` method (if present) to convert it to a `PyLongObject`.

Raise `OverflowError` if the value of *obj* is out of range for a `long`.

Returns `-1` on error. Use `PyErr_Occurred()` to disambiguate.

在 3.8 版的變更: Use `__index__()` if available.

在 3.10 版的變更: This function will no longer use `__int__()`.

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

Part of the Stable ABI. Return a C `long` representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` method (if present) to convert it to a `PyLongObject`.

If the value of *obj* is greater than `LONG_MAX` or less than `LONG_MIN`, set **overflow* to `1` or `-1`, respectively, and return `-1`; otherwise, set **overflow* to `0`. If any other exception occurs set **overflow* to `0` and return `-1` as usual.

Returns -1 on error. Use *PyErr_Occurred()* to disambiguate.

在 3.8 版的變更: Use `__index__()` if available.

在 3.10 版的變更: This function will no longer use `__int__()`.

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

Part of the Stable ABI. Return a C `long long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

Raise `OverflowError` if the value of *obj* is out of range for a `long long`.

Returns -1 on error. Use *PyErr_Occurred()* to disambiguate.

在 3.8 版的變更: Use `__index__()` if available.

在 3.10 版的變更: This function will no longer use `__int__()`.

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

Part of the Stable ABI. Return a C `long long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

If the value of *obj* is greater than `LLONG_MAX` or less than `LLONG_MIN`, set **overflow* to `1` or `-1`, respectively, and return `-1`; otherwise, set **overflow* to `0`. If any other exception occurs set **overflow* to `0` and return `-1` as usual.

Returns -1 on error. Use *PyErr_Occurred()* to disambiguate.

在 3.2 版新加入.

在 3.8 版的變更: Use `__index__()` if available.

在 3.10 版的變更: This function will no longer use `__int__()`.

Py_ssize_t **PyLong_AsSsize_t**(*PyObject* *pylong)

Part of the Stable ABI. Return a C *Py_ssize_t* representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a *Py_ssize_t*.

Returns -1 on error. Use *PyErr_Occurred()* to disambiguate.

unsigned long **PyLong_AsUnsignedLong**(*PyObject* *pylong)

Part of the Stable ABI. Return a C `unsigned long` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a `unsigned long`.

Returns `(unsigned long)-1` on error. Use *PyErr_Occurred()* to disambiguate.

size_t **PyLong_AsSize_t**(*PyObject* *pylong)

Part of the Stable ABI. Return a C `size_t` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a `size_t`.

Returns `(size_t)-1` on error. Use *PyErr_Occurred()* to disambiguate.

unsigned long long **PyLong_AsUnsignedLongLong**(*PyObject* *pylong)

Part of the Stable ABI. Return a C `unsigned long long` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for an `unsigned long long`.

Returns `(unsigned long long)-1` on error. Use *PyErr_Occurred()* to disambiguate.

在 3.1 版的變更: A negative *pylong* now raises `OverflowError`, not `TypeError`.

unsigned long **PyLong_AsUnsignedLongMask**(*PyObject* *obj)

Part of the Stable ABI. Return a C `unsigned long` representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` method (if present) to convert it to a `PyLongObject`.

If the value of *obj* is out of range for an `unsigned long`, return the reduction of that value modulo `ULONG_MAX + 1`.

Returns `(unsigned long)-1` on error. Use `PyErr_Occurred()` to disambiguate.

在 3.8 版的變更: Use `__index__()` if available.

在 3.10 版的變更: This function will no longer use `__int__()`.

unsigned long long **PyLong_AsUnsignedLongLongMask**(*PyObject* *obj)

Part of the Stable ABI. Return a C `unsigned long long` representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its `__index__()` method (if present) to convert it to a `PyLongObject`.

If the value of *obj* is out of range for an `unsigned long long`, return the reduction of that value modulo `ULLONG_MAX + 1`.

Returns `(unsigned long long)-1` on error. Use `PyErr_Occurred()` to disambiguate.

在 3.8 版的變更: Use `__index__()` if available.

在 3.10 版的變更: This function will no longer use `__int__()`.

double **PyLong_AsDouble**(*PyObject* *pylong)

Part of the Stable ABI. Return a C `double` representation of *pylong*. *pylong* must be an instance of `PyLongObject`.

Raise `OverflowError` if the value of *pylong* is out of range for a `double`.

Returns `-1.0` on error. Use `PyErr_Occurred()` to disambiguate.

void ***PyLong_AsVoidPtr**(*PyObject* *pylong)

Part of the Stable ABI. Convert a Python integer *pylong* to a C `void` pointer. If *pylong* cannot be converted, an `OverflowError` will be raised. This is only assured to produce a usable `void` pointer for values created with `PyLong_FromVoidPtr()`.

Returns `NULL` on error. Use `PyErr_Occurred()` to disambiguate.

8.2.2 Boolean（布林）物件

Python 中的 boolean 是以整數子類別化來實現的。只有 `Py_False` 和 `Py_True` 兩個 boolean。因此一般的建立和刪除函式並不適用於 boolean。但下列巨集 (macro) 是可用的。

PyTypeObject **PyBool_Type**

Part of the Stable ABI. This instance of `PyTypeObject` represents the Python boolean type; it is the same object as `bool` in the Python layer.

int **PyBool_Check**(*PyObject* *o)

如果 *o* 的型別為 `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。此物件沒有任何方法。在參照 (reference) 計數上必須有著和其他物件一樣的處理方式。

PyObject ***Py_True**

Python 的 `True` 物件。此物件沒有任何方法。在參照計數上必須有著和其他物件一樣的處理方式。

Py_RETURN_FALSE

從函式回傳 `Py_False`，並適當的增加它的參照計數。

Py_RETURN_TRUE

從函式回傳 `Py_True`，並適當的增加它的參照計數。

PyObject ***PyBool_FromLong**(long v)

回傳值：新的參照。*Part of the* Stable ABI. 根據 *v* 的實際值來回傳一個 `Py_True` 或者 `Py_False` 的新參照。

8.2.3 浮點數（Floating Point）物件

type **PyFloatObject**

This subtype of *PyObject* represents a Python floating point object.

PyTypeObject **PyFloat_Type**

Part of the Stable ABI. This instance of *PyTypeObject* represents the Python floating point type. This is the same object as `float` in the Python layer.

int **PyFloat_Check**(*PyObject* *p)

Return true if its argument is a *PyFloatObject* or a subtype of *PyFloatObject*. This function always succeeds.

int **PyFloat_CheckExact**(*PyObject* *p)

Return true if its argument is a *PyFloatObject*, but not a subtype of *PyFloatObject*. This function always succeeds.

PyObject ***PyFloat_FromString**(*PyObject* *str)

回傳值：新的參照。*Part of the* Stable ABI. Create a *PyFloatObject* object based on the string value in *str*, or `NULL` on failure.

PyObject ***PyFloat_FromDouble**(double v)

回傳值：新的參照。*Part of the* Stable ABI. Create a *PyFloatObject* object from *v*, or `NULL` on failure.

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

Part of the Stable ABI. Return a C `double` representation of the contents of *pyfloat*. If *pyfloat* is not a Python floating point object but has a `__float__()` method, this method will first be called to convert *pyfloat* into a float. If `__float__()` is not defined then it falls back to `__index__()`. This method returns `-1.0` upon failure, so one should call *PyErr_Occurred()* to check for errors.

在 3.8 版的變更: Use `__index__()` if available.

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

Return a C `double` representation of the contents of *pyfloat*, but without error checking.

PyObject ***PyFloat_GetInfo**(void)

回傳值：新的參照。*Part of the* Stable ABI. Return a structseq instance which contains information about the precision, minimum and maximum values of a float. It's a thin wrapper around the header file `float.h`.

double **PyFloat_GetMax**()

Part of the Stable ABI. Return the maximum representable finite float *DBL_MAX* as C `double`.

double **PyFloat_GetMin**()

Part of the Stable ABI. Return the minimum normalized positive float *DBL_MIN* as C `double`.

Pack and Unpack functions

The pack and unpack functions provide an efficient platform-independent way to store floating-point values as byte strings. The Pack routines produce a bytes string from a C `double`, and the Unpack routines produce a C `double` from such a bytes string. The suffix (2, 4 or 8) specifies the number of bytes in the bytes string.

On platforms that appear to use IEEE 754 formats these functions work by copying bits. On other platforms, the 2-byte format is identical to the IEEE 754 binary16 half-precision format, the 4-byte format (32-bit) is identical to the IEEE 754 binary32 single precision format, and the 8-byte format to the IEEE 754 binary64 double precision format, although the packing of INFs and NaNs (if such things exist on the platform) isn't handled correctly, and attempting to unpack a bytes string containing an IEEE INF or NaN will raise an exception.

On non-IEEE platforms with more precision, or larger dynamic range, than IEEE 754 supports, not all values can be packed; on non-IEEE platforms with less precision, or smaller dynamic range, not all values can be unpacked. What happens in such cases is partly accidental (alas).

在 3.11 版新加入.

Pack functions

The pack routines write 2, 4 or 8 bytes, starting at *p*. *le* is an `int` argument, non-zero if you want the bytes string in little-endian format (exponent last, at `p+1`, `p+3`, or `p+6` `p+7`), zero if you want big-endian format (exponent first, at *p*). The `PY_BIG_ENDIAN` constant can be used to use the native endian: it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value: `0` if all is OK, `-1` if error (and an exception is set, most likely `OverflowError`).

There are two problems on non-IEEE platforms:

- What this does is undefined if *x* is a NaN or infinity.
- `-0.0` and `+0.0` produce the same bytes string.

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

Pack a C double as the IEEE 754 binary16 half-precision format.

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

Pack a C double as the IEEE 754 binary32 single precision format.

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

Pack a C double as the IEEE 754 binary64 double precision format.

Unpack functions

The unpack routines read 2, 4 or 8 bytes, starting at *p*. *le* is an `int` argument, non-zero if the bytes string is in little-endian format (exponent last, at `p+1`, `p+3` or `p+6` and `p+7`), zero if big-endian (exponent first, at *p*). The `PY_BIG_ENDIAN` constant can be used to use the native endian: it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value: The unpacked double. On error, this is `-1.0` and *`PyErr_Occurred()`* is true (and an exception is set, most likely `OverflowError`).

Note that on a non-IEEE platform this will refuse to unpack a bytes string that represents a NaN or infinity.

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

Unpack the IEEE 754 binary16 half-precision format as a C double.

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

Unpack the IEEE 754 binary32 single precision format as a C double.

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

Unpack the IEEE 754 binary64 double precision format as a C double.

8.2.4 Complex Number Objects

Python's complex number objects are implemented as two distinct types when viewed from the C API: one is the Python object exposed to Python programs, and the other is a C structure which represents the actual complex number value. The API provides functions for working with both.

Complex Numbers as C Structures

Note that the functions which accept these structures as parameters and return them as results do so *by value* rather than dereferencing them through pointers. This is consistent throughout the API.

type **Py_complex**

The C structure which corresponds to the value portion of a Python complex number object. Most of the functions for dealing with complex number objects use structures of this type as input or output values, as appropriate. It is defined as:

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

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

Return the sum of two complex numbers, using the C `Py_complex` representation.

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

Return the difference between two complex numbers, using the C `Py_complex` representation.

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

Return the negation of the complex number *num*, using the C `Py_complex` representation.

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

Return the product of two complex numbers, using the C `Py_complex` representation.

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

Return the quotient of two complex numbers, using the C `Py_complex` representation.

If *divisor* is null, this method returns zero and sets `errno` to `EDOM`.

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

Return the exponentiation of *num* by *exp*, using the C `Py_complex` representation.

If *num* is null and *exp* is not a positive real number, this method returns zero and sets `errno` to `EDOM`.

Complex Numbers as Python Objects

type **PyComplexObject**

This subtype of `PyObject` represents a Python complex number object.

PyTypeObject **PyComplex_Type**

Part of the Stable ABI. This instance of `PyTypeObject` represents the Python complex number type. It is the same object as `complex` in the Python layer.

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

Return true if its argument is a `PyComplexObject` or a subtype of `PyComplexObject`. This function always succeeds.

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

Return true if its argument is a `PyComplexObject`, but not a subtype of `PyComplexObject`. This function always succeeds.

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

回傳值：新的參照。 Create a new Python complex number object from a C `Py_complex` value.

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

回傳值：新的參照。*Part of the* Stable ABI. Return a new `PyComplexObject` object from *real* and *imag*.

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

Part of the Stable ABI. Return the real part of *op* as a C `double`.

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

Part of the Stable ABI. Return the imaginary part of *op* as a C `double`.

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

Return the `Py_complex` value of the complex number *op*.

If *op* is not a Python complex number object but has a `__complex__()` method, this method will first be called to convert *op* to a Python complex number object. If `__complex__()` is not defined then it falls back to `__float__()`. If `__float__()` is not defined then it falls back to `__index__()`. Upon failure, this method returns `-1.0` as a real value.

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F序列物件。

8.3.1 位元組物件 (Bytes Objects)

These functions raise `TypeError` when expecting a bytes parameter and called with a non-bytes parameter.

type **PyBytesObject**

This subtype of `PyObject` represents a Python bytes object.

PyTypeObject **PyBytes_Type**

Part of the Stable ABI. This instance of `PyTypeObject` represents the Python bytes type; it is the same object as `bytes` in the Python layer.

int **PyBytes_Check**(*PyObject* *o)

Return true if the object *o* is a bytes object or an instance of a subtype of the bytes type. This function always succeeds.

int **PyBytes_CheckExact**(*PyObject* *o)

Return true if the object *o* is a bytes object, but not an instance of a subtype of the bytes type. This function always succeeds.

PyObject ***PyBytes_FromString**(const char *v)

回傳值：新的參照。 *Part of the* Stable ABI. Return a new bytes object with a copy of the string *v* as value on success, and `NULL` on failure. The parameter *v* must not be `NULL`; it will not be checked.

PyObject ***PyBytes_FromStringAndSize**(const char *v, *Py_ssize_t* len)

回傳值：新的參照。 *Part of the* Stable ABI. Return a new bytes object with a copy of the string *v* as value and length *len* on success, and `NULL` on failure. If *v* is `NULL`, the contents of the bytes object are uninitialized.

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

回傳值：新的參照。*Part of the* Stable ABI. Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python bytes object and return a bytes object with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* string. The following format characters are allowed:

Format Characters	Type	Comment
%%	*n/a*	The literal % character.
%c	int	A single byte, represented as a C int.
%d	int	等價於 printf("%d").[1]
%u	unsigned int	等價於 printf("%u").[Page 121, 1]
%ld	long	等價於 printf("%ld").[1]
%lu	unsigned long	等價於 printf("%lu").[1]
%zd	Py_ssize_t	等價於 printf("%zd").[1]
%zu	size_t	等價於 printf("%zu").[1]
%i	int	等價於 printf("%i").[1]
%x	int	等價於 printf("%x").[1]
%s	const char*	A null-terminated C character array.
%p	const void*	The hex representation of a C pointer. Mostly equivalent to printf("%p") except that it is guaranteed to start with the literal 0x regardless of what the platform's printf yields.

An unrecognized format character causes all the rest of the format string to be copied as-is to the result object, and any extra arguments discarded.

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

回傳值：新的參照。 *Part of the* Stable ABI. Identical to PyBytes_FromFormat() except that it takes exactly two arguments.

PyObject ***PyBytes_FromObject**(*PyObject* *o)

回傳值：新的參照。 *Part of the* Stable ABI. Return the bytes representation of object *o* that implements the buffer protocol.

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

Part of the Stable ABI. Return the length of the bytes in bytes object *o*.

Py_ssize_t **PyBytes_GET_SIZE**(*PyObject* *o)

Similar to PyBytes_Size(), but without error checking.

char ***PyBytes_AsString**(*PyObject* *o)

Part of the Stable ABI. Return a pointer to the contents of *o*. The pointer refers to the internal buffer of *o*, which consists of len(o) + 1 bytes. The last byte in the buffer is always null, regardless of whether there are any other null bytes. The data must not be modified in any way, unless the object was just created using PyBytes_FromStringAndSize(NULL, size). It must not be deallocated. If *o* is not a bytes object at all, PyBytes_AsString() returns NULL and raises TypeError.

char ***PyBytes_AS_STRING**(*PyObject* *string)

Similar to PyBytes_AsString(), but without error checking.

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

Part of the Stable ABI. Return the null-terminated contents of the object *obj* through the output variables *buffer* and *length*. Returns 0 on success.

If *length* is NULL, the bytes object may not contain embedded null bytes; if it does, the function returns -1 and a ValueError is raised.

The buffer refers to an internal buffer of *obj*, which includes an additional null byte at the end (not counted in *length*). The data must not be modified in any way, unless the object was just created using PyBytes_FromStringAndSize(NULL, size). It must not be deallocated. If *obj* is not a bytes object at all, PyBytes_AsStringAndSize() returns -1 and raises TypeError.

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

[1] For integer specifiers (d, u, ld, lu, zd, zu, i, x): the 0-conversion flag has effect even when a precision is given.

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

Part of the Stable ABI. Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*; the caller will own the new reference. The reference to the old value of *bytes* will be stolen. If the new object cannot be created, the old reference to *bytes* will still be discarded and the value of **bytes* will be set to `NULL`; the appropriate exception will be set.

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

Part of the Stable ABI. Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*. This version releases the *strong reference* to *newpart* (i.e. decrements its reference count).

int **_PyBytes_Resize** (*PyObject* **bytes, *Py_ssize_t* newsize)

A way to resize a bytes object even though it is "immutable". Only use this to build up a brand new bytes object; don't use this if the bytes may already be known in other parts of the code. It is an error to call this function if the refcount on the input bytes object is not one. Pass the address of an existing bytes object as an lvalue (it may be written into), and the new size desired. On success, **bytes* holds the resized bytes object and `0` is returned; the address in **bytes* may differ from its input value. If the reallocation fails, the original bytes object at **bytes* is deallocated, **bytes* is set to `NULL`, `MemoryError` is set, and `-1` is returned.

8.3.2 位元組串列物件 (Byte Array Objects)

type **PyByteArrayObject**

This subtype of `PyObject` represents a Python bytearray object.

PyTypeObject **PyByteArray_Type**

Part of the Stable ABI. This instance of `PyTypeObject` represents the Python bytearray type; it is the same object as `bytearray` in the Python layer.

Type check macros

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

Return true if the object *o* is a bytearray object or an instance of a subtype of the bytearray type. This function always succeeds.

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

Return true if the object *o* is a bytearray object, but not an instance of a subtype of the bytearray type. This function always succeeds.

Direct API functions

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

回傳值：新的參照。*Part of the* Stable ABI. Return a new bytearray object from any object, *o*, that implements the *buffer protocol*.

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

回傳值：新的參照。*Part of the* Stable ABI. Create a new bytearray object from *string* and its length, *len*. On failure, `NULL` is returned.

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

回傳值：新的參照。 *Part of the* Stable ABI. Concat bytearrays *a* and *b* and return a new bytearray with the result.

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

Part of the Stable ABI. Return the size of *bytearray* after checking for a `NULL` pointer.

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

Part of the Stable ABI. Return the contents of *bytearray* as a char array after checking for a `NULL` pointer. The returned array always has an extra null byte appended.

int **PyByteArray_Resize** (*PyObject* *bytearray, *Py_ssize_t* len)

Part of the Stable ABI. Resize the internal buffer of *bytearray* to *len*.

巨集

These macros trade safety for speed and they don't check pointers.

char ***PyByteArray_AS_STRING** (*PyObject* *bytearray)

Similar to `PyByteArray_AsString()`, but without error checking.

Py_ssize_t **PyByteArray_GET_SIZE** (*PyObject* *bytearray)

Similar to `PyByteArray_Size()`, but without error checking.

8.3.3 Unicode 物件與編碼

Unicode Objects

Since the implementation of **PEP 393** in Python 3.3, Unicode objects internally use a variety of representations, in order to allow handling the complete range of Unicode characters while staying memory efficient. There are special cases for strings where all code points are below 128, 256, or 65536; otherwise, code points must be below 1114112 (which is the full Unicode range).

`Py_UNICODE*` and UTF-8 representations are created on demand and cached in the Unicode object. The `Py_UNICODE*` representation is deprecated and inefficient.

Due to the transition between the old APIs and the new APIs, Unicode objects can internally be in two states depending on how they were created:

- "canonical" Unicode objects are all objects created by a non-deprecated Unicode API. They use the most efficient representation allowed by the implementation.
- "legacy" Unicode objects have been created through one of the deprecated APIs (typically `PyUnicode_FromUnicode()`) and only bear the `Py_UNICODE*` representation; you will have to call `PyUnicode_READY()` on them before calling any other API.

備註: The "legacy" Unicode object will be removed in Python 3.12 with deprecated APIs. All Unicode objects will be "canonical" since then. See **PEP 623** for more information.

Unicode Type

These are the basic Unicode object types used for the Unicode implementation in Python:

type **Py_UCS4**

type **Py_UCS2**

type **Py_UCS1**

Part of the Stable ABI. These types are typedefs for unsigned integer types wide enough to contain characters of 32 bits, 16 bits and 8 bits, respectively. When dealing with single Unicode characters, use `Py_UCS4`.

在 3.3 版新加入.

type **Py_UNICODE**

This is a typedef of `wchar_t`, which is a 16-bit type or 32-bit type depending on the platform.

在 3.3 版的變更: In previous versions, this was a 16-bit type or a 32-bit type depending on whether you selected a "narrow" or "wide" Unicode version of Python at build time.

type **PyASCIIObject**

type **PyCompactUnicodeObject**

type **PyUnicodeObject**

These subtypes of *PyObject* represent a Python Unicode object. In almost all cases, they shouldn't be used directly, since all API functions that deal with Unicode objects take and return *PyObject* pointers.

在 3.3 版新加入.

PyTypeObject **PyUnicode_Type**

Part of the Stable ABI. This instance of *PyTypeObject* represents the Python Unicode type. It is exposed to Python code as `str`.

The following APIs are C macros and static inlined functions for fast checks and access to internal read-only data of Unicode objects:

int **PyUnicode_Check**(*PyObject* *obj)

Return true if the object *obj* is a Unicode object or an instance of a Unicode subtype. This function always succeeds.

int **PyUnicode_CheckExact**(*PyObject* *obj)

Return true if the object *obj* is a Unicode object, but not an instance of a subtype. This function always succeeds.

int **PyUnicode_READY**(*PyObject* *unicode)

Ensure the string object *o* is in the "canonical" representation. This is required before using any of the access macros described below.

Returns `0` on success and `-1` with an exception set on failure, which in particular happens if memory allocation fails.

在 3.3 版新加入.

自從版本 3.10 後不推薦使用，將會自版本 3.12 中移除。: This API will be removed with *PyUnicode_FromUnicode()*.

Py_ssize_t **PyUnicode_GET_LENGTH**(*PyObject* *unicode)

Return the length of the Unicode string, in code points. *unicode* has to be a Unicode object in the "canonical" representation (not checked).

在 3.3 版新加入.

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

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

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

Return a pointer to the canonical representation cast to UCS1, UCS2 or UCS4 integer types for direct character access. No checks are performed if the canonical representation has the correct character size; use *PyUnicode_KIND()* to select the right macro. Make sure *PyUnicode_READY()* has been called before accessing this.

在 3.3 版新加入.

PyUnicode_WCHAR_KIND

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND

PyUnicode_4BYTE_KIND

Return values of the *PyUnicode_KIND()* macro.

在 3.3 版新加入.

自從版本 3.10 後不推薦使用，將會自版本 3.12 中移除。: `PyUnicode_WCHAR_KIND` 已F用。

int **PyUnicode_KIND**(*PyObject* *unicode)

Return one of the PyUnicode kind constants (see above) that indicate how many bytes per character this Unicode object uses to store its data. *unicode* has to be a Unicode object in the "canonical" representation (not checked).

在 3.3 版新加入.

void *`PyUnicode_DATA`(*PyObject* *unicode)

Return a void pointer to the raw Unicode buffer. *unicode* has to be a Unicode object in the "canonical" representation (not checked).

在 3.3 版新加入.

void `PyUnicode_WRITE`(int kind, void *data, *Py_ssize_t* index, *Py_UCS4* value)

Write into a canonical representation *data* (as obtained with `PyUnicode_DATA()`). This function performs no sanity checks, and is intended for usage in loops. The caller should cache the *kind* value and *data* pointer as obtained from other calls. *index* is the index in the string (starts at 0) and *value* is the new code point value which should be written to that location.

在 3.3 版新加入.

Py_UCS4 `PyUnicode_READ`(int kind, void *data, *Py_ssize_t* index)

Read a code point from a canonical representation *data* (as obtained with `PyUnicode_DATA()`). No checks or ready calls are performed.

在 3.3 版新加入.

Py_UCS4 `PyUnicode_READ_CHAR`(*PyObject* *unicode, *Py_ssize_t* index)

Read a character from a Unicode object *unicode*, which must be in the "canonical" representation. This is less efficient than `PyUnicode_READ()` if you do multiple consecutive reads.

在 3.3 版新加入.

Py_UCS4 `PyUnicode_MAX_CHAR_VALUE`(*PyObject* *unicode)

Return the maximum code point that is suitable for creating another string based on *unicode*, which must be in the "canonical" representation. This is always an approximation but more efficient than iterating over the string.

在 3.3 版新加入.

Py_ssize_t `PyUnicode_GET_SIZE`(*PyObject* *unicode)

Return the size of the deprecated `Py_UNICODE` representation, in code units (this includes surrogate pairs as 2 units). *unicode* has to be a Unicode object (not checked).

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: Part of the old-style Unicode API, please migrate to using `PyUnicode_GET_LENGTH()`.

Py_ssize_t `PyUnicode_GET_DATA_SIZE`(*PyObject* *unicode)

Return the size of the deprecated `Py_UNICODE` representation in bytes. *unicode* has to be a Unicode object (not checked).

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: Part of the old-style Unicode API, please migrate to using `PyUnicode_GET_LENGTH()`.

Py_UNICODE *`PyUnicode_AS_UNICODE`(*PyObject* *unicode)

const char *`PyUnicode_AS_DATA`(*PyObject* *unicode)

Return a pointer to a `Py_UNICODE` representation of the object. The returned buffer is always terminated with an extra null code point. It may also contain embedded null code points, which would cause the string to be truncated when used in most C functions. The `AS_DATA` form casts the pointer to `const char*`. The *unicode* argument has to be a Unicode object (not checked).

在 3.3 版的變更: This function is now inefficient -- because in many cases the `Py_UNICODE` representation does not exist and needs to be created -- and can fail (return `NULL` with an exception set). Try to port the code to use the new `PyUnicode_nBYTE_DATA()` macros or use `PyUnicode_WRITE()` or `PyUnicode_READ()`.

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: Part of the old-style Unicode API, please migrate to using the `PyUnicode_nBYTE_DATA()` family of macros.

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

Part of the Stable ABI. Return 1 if the string is a valid identifier according to the language definition, section identifiers. Return 0 otherwise.

在 3.9 版的變更: The function does not call `Py_FatalError()` anymore if the string is not ready.

Unicode Character Properties

Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration.

int **Py_UNICODE_ISSPACE**(*Py_UCS4* ch)

Return 1 or 0 depending on whether *ch* is a whitespace character.

int **Py_UNICODE_ISLOWER**(*Py_UCS4* ch)

Return 1 or 0 depending on whether *ch* is a lowercase character.

int **Py_UNICODE_ISUPPER**(*Py_UCS4* ch)

Return 1 or 0 depending on whether *ch* is an uppercase character.

int **Py_UNICODE_ISTITLE**(*Py_UCS4* ch)

Return 1 or 0 depending on whether *ch* is a titlecase character.

int **Py_UNICODE_ISLINEBREAK**(*Py_UCS4* ch)

Return 1 or 0 depending on whether *ch* is a linebreak character.

int **Py_UNICODE_ISDECIMAL**(*Py_UCS4* ch)

Return 1 or 0 depending on whether *ch* is a decimal character.

int **Py_UNICODE_ISDIGIT**(*Py_UCS4* ch)

Return 1 or 0 depending on whether *ch* is a digit character.

int **Py_UNICODE_ISNUMERIC**(*Py_UCS4* ch)

Return 1 or 0 depending on whether *ch* is a numeric character.

int **Py_UNICODE_ISALPHA**(*Py_UCS4* ch)

Return 1 or 0 depending on whether *ch* is an alphabetic character.

int **Py_UNICODE_ISALNUM**(*Py_UCS4* ch)

Return 1 or 0 depending on whether *ch* is an alphanumeric character.

int **Py_UNICODE_ISPRINTABLE**(*Py_UCS4* ch)

Return 1 or 0 depending on whether *ch* is a printable character. Nonprintable characters are those characters defined in the Unicode character database as "Other" or "Separator", excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when `repr()` is invoked on a string. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.)

These APIs can be used for fast direct character conversions:

Py_UCS4 **Py_UNICODE_TOLOWER**(*Py_UCS4* ch)

Return the character *ch* converted to lower case.

在 3.3 版之後被棄用: This function uses simple case mappings.

Py_UCS4 **Py_UNICODE_TOUPPER**(*Py_UCS4* ch)

Return the character *ch* converted to upper case.

在 3.3 版之後被棄用: This function uses simple case mappings.

Py_UCS4 **Py_UNICODE_TOTITLE**(*Py_UCS4* ch)

Return the character *ch* converted to title case.

在 3.3 版之後被F用: This function uses simple case mappings.

int **Py_UNICODE_TODECIMAL**(*Py_UCS4* ch)

Return the character *ch* converted to a decimal positive integer. Return `-1` if this is not possible. This macro does not raise exceptions.

int **Py_UNICODE_TODIGIT**(*Py_UCS4* ch)

Return the character *ch* converted to a single digit integer. Return `-1` if this is not possible. This macro does not raise exceptions.

double **Py_UNICODE_TONUMERIC**(*Py_UCS4* ch)

Return the character *ch* converted to a double. Return `-1.0` if this is not possible. This macro does not raise exceptions.

These APIs can be used to work with surrogates:

Py_UNICODE_IS_SURROGATE(ch)

Check if *ch* is a surrogate (`0xD800 <= ch <= 0xDFFF`).

Py_UNICODE_IS_HIGH_SURROGATE(ch)

Check if *ch* is a high surrogate (`0xD800 <= ch <= 0xDBFF`).

Py_UNICODE_IS_LOW_SURROGATE(ch)

Check if *ch* is a low surrogate (`0xDC00 <= ch <= 0xDFFF`).

Py_UNICODE_JOIN_SURROGATES(high, low)

Join two surrogate characters and return a single Py_UCS4 value. *high* and *low* are respectively the leading and trailing surrogates in a surrogate pair.

Creating and accessing Unicode strings

To create Unicode objects and access their basic sequence properties, use these APIs:

PyObject ***PyUnicode_New**(*Py_ssize_t* size, *Py_UCS4* maxchar)

回傳值：新的參照。 Create a new Unicode object. *maxchar* should be the true maximum code point to be placed in the string. As an approximation, it can be rounded up to the nearest value in the sequence 127, 255, 65535, 1114111.

This is the recommended way to allocate a new Unicode object. Objects created using this function are not resizable.

在 3.3 版新加入.

PyObject ***PyUnicode_FromKindAndData**(int kind, const void *buffer, *Py_ssize_t* size)

回傳值：新的參照。 Create a new Unicode object with the given *kind* (possible values are *PyUnicode_1BYTE_KIND* etc., as returned by *PyUnicode_KIND()*). The *buffer* must point to an array of *size* units of 1, 2 or 4 bytes per character, as given by the kind.

If necessary, the input *buffer* is copied and transformed into the canonical representation. For example, if the *buffer* is a UCS4 string (*PyUnicode_4BYTE_KIND*) and it consists only of codepoints in the UCS1 range, it will be transformed into UCS1 (*PyUnicode_1BYTE_KIND*).

在 3.3 版新加入.

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

回傳值：新的參照。 *Part of the* Stable ABI. Create a Unicode object from the char buffer *str*. The bytes will be interpreted as being UTF-8 encoded. The buffer is copied into the new object. If the buffer is not `NULL`, the return value might be a shared object, i.e. modification of the data is not allowed.

If *str* is `NULL`, this function behaves like *PyUnicode_FromUnicode()* with the buffer set to `NULL`. This usage is deprecated in favor of *PyUnicode_New()*, and will be removed in Python 3.12.

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

回傳值：新的參照。*Part of the* Stable ABI. Create a Unicode object from a UTF-8 encoded null-terminated char buffer *str*.

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

回傳值：新的參照。*Part of the* Stable ABI. Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python Unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* ASCII-encoded string. The following format characters are allowed:

Format Characters	Type	Comment
`%%`	*n/a*	The literal % character.
`%c`	int	A single character, represented as a C int.
`%d`	int	等價於 `printf("%d")`.[1]
`%u`	unsigned int	等價於 `printf("%u")`.[1]
`%ld`	long	等價於 `printf("%ld")`.[1]
`%li`	long	等價於 `printf("%li")`.[1]
`%lu`	unsigned long	等價於 `printf("%lu")`.[1]
`%lld`	long long	等價於 `printf("%lld")`.[1]
`%lli`	long long	等價於 `printf("%lli")`.[1]
`%llu`	unsigned long long	等價於 `printf("%llu")`.[1]
`%zd`	`Py_ssize_t`	等價於 `printf("%zd")`.[1]
`%zi`	`Py_ssize_t`	等價於 `printf("%zi")`.[1]
`%zu`	size_t	等價於 `printf("%zu")`.[1]
`%i`	int	等價於 `printf("%i")`.[1]
`%x`	int	等價於 `printf("%x")`.[1]
`%s`	const char*	A null-terminated C character array.
`%p`	const void*	The hex representation of a C pointer. Mostly equivalent to `printf("%p")` except that it is guaranteed to start with the literal `0x` regardless of what the platform's `printf` yields.
`%A`	PyObject*	The result of calling `ascii()`.
`%U`	PyObject*	一 Unicode 物件。
`%V`	PyObject*, const char*	A Unicode object (which may be `NULL`) and a null-terminated C character array as a second parameter (which will be used, if the first parameter is `NULL`).
`%S`	PyObject*	The result of calling `PyObject_Str()`.
`%R`	PyObject*	The result of calling `PyObject_Repr()`.

An unrecognized format character causes all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

備註： The width formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes for `"%s"` and `"%V"` (if the `PyObject*` argument is `NULL`), and a number of characters for `"%A"`, `"%U"`, `"%S"`, `"%R"` and `"%V"` (if the `PyObject*` argument is not `NULL`).

在 3.2 版的變更: Support for `"%lld"` and `"%llu"` added.

在 3.3 版的變更: Support for `"%li"`, `"%lli"` and `"%zi"` added.

在 3.4 版的變更: Support width and precision formatter for `"%s"`, `"%A"`, `"%U"`, `"%V"`, `"%S"`, `"%R"` added.

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

回傳值：新的參照。*Part of the* Stable ABI. Identical to `PyUnicode_FromFormat()` except that it takes exactly two arguments.

[1] For integer specifiers (d, u, ld, li, lu, lld, lli, llu, zd, zi, zu, i, x): the 0-conversion flag has effect even when a precision is given.

PyObject ***PyUnicode_FromObject**(*PyObject* *obj)

回傳值：新的參照。 *Part of the* Stable ABI. Copy an instance of a Unicode subtype to a new true Unicode object if necessary. If *obj* is already a true Unicode object (not a subtype), return a new *strong reference* to the object.

Objects other than Unicode or its subtypes will cause a `TypeError`.

PyObject ***PyUnicode_FromEncodedObject**(*PyObject* *obj, const char *encoding, const char *errors)

回傳值：新的參照。 *Part of the* Stable ABI. Decode an encoded object *obj* to a Unicode object.

`bytes`, `bytearray` and other *bytes-like objects* are decoded according to the given *encoding* and using the error handling defined by *errors*. Both can be `NULL` to have the interface use the default values (see *Built-in Codecs* for details).

All other objects, including Unicode objects, cause a `TypeError` to be set.

The API returns `NULL` if there was an error. The caller is responsible for decref'ing the returned objects.

Py_ssize_t **PyUnicode_GetLength**(*PyObject* *unicode)

Part of the Stable ABI *since version 3.7.* Return the length of the Unicode object, in code points.

在 3.3 版新加入.

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

Copy characters from one Unicode object into another. This function performs character conversion when necessary and falls back to `memcpy()` if possible. Returns `-1` and sets an exception on error, otherwise returns the number of copied characters.

在 3.3 版新加入.

Py_ssize_t **PyUnicode_Fill**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* length, *Py_UCS4* fill_char)

Fill a string with a character: write *fill_char* into `unicode[start:start+length]`.

Fail if *fill_char* is bigger than the string maximum character, or if the string has more than 1 reference.

Return the number of written character, or return `-1` and raise an exception on error.

在 3.3 版新加入.

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

Part of the Stable ABI *since version 3.7.* Write a character to a string. The string must have been created through *`PyUnicode_New()`*. Since Unicode strings are supposed to be immutable, the string must not be shared, or have been hashed yet.

This function checks that *unicode* is a Unicode object, that the index is not out of bounds, and that the object can be modified safely (i.e. that it its reference count is one).

在 3.3 版新加入.

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

Part of the Stable ABI *since version 3.7.* Read a character from a string. This function checks that *unicode* is a Unicode object and the index is not out of bounds, in contrast to *`PyUnicode_READ_CHAR()`*, which performs no error checking.

在 3.3 版新加入.

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

回傳值：新的參照。 *Part of the* Stable ABI *since version 3.7.* Return a substring of *unicode*, from character index *start* (included) to character index *end* (excluded). Negative indices are not supported.

在 3.3 版新加入.

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

Part of the Stable ABI *since version 3.7.* Copy the string *unicode* into a UCS4 buffer, including a null character, if *copy_null* is set. Returns `NULL` and sets an exception on error (in particular, a `SystemError` if *buflen* is smaller than the length of *unicode*). *buffer* is returned on success.

在 3.3 版新加入.

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

Part of the Stable ABI *since version 3.7.* Copy the string *unicode* into a new UCS4 buffer that is allocated using `PyMem_Malloc()`. If this fails, `NULL` is returned with a `MemoryError` set. The returned buffer always has an extra null code point appended.

在 3.3 版新加入.

Deprecated Py_UNICODE APIs

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。.

These API functions are deprecated with the implementation of **PEP 393**. Extension modules can continue using them, as they will not be removed in Python 3.x, but need to be aware that their use can now cause performance and memory hits.

PyObject ***PyUnicode_FromUnicode** (const *Py_UNICODE* *u, *Py_ssize_t* size)

回傳值：新的參照。 Create a Unicode object from the Py_UNICODE buffer *u* of the given size. *u* may be `NULL` which causes the contents to be undefined. It is the user's responsibility to fill in the needed data. The buffer is copied into the new object.

If the buffer is not `NULL`, the return value might be a shared object. Therefore, modification of the resulting Unicode object is only allowed when *u* is `NULL`.

If the buffer is `NULL`, `PyUnicode_READY()` must be called once the string content has been filled before using any of the access macros such as `PyUnicode_KIND()`.

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: Part of the old-style Unicode API, please migrate to using `PyUnicode_FromKindAndData()`, `PyUnicode_FromWideChar()`, or `PyUnicode_New()`.

Py_UNICODE ***PyUnicode_AsUnicode** (*PyObject* *unicode)

Return a read-only pointer to the Unicode object's internal `Py_UNICODE` buffer, or `NULL` on error. This will create the `Py_UNICODE*` representation of the object if it is not yet available. The buffer is always terminated with an extra null code point. Note that the resulting `Py_UNICODE` string may also contain embedded null code points, which would cause the string to be truncated when used in most C functions.

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: Part of the old-style Unicode API, please migrate to using `PyUnicode_AsUCS4()`, `PyUnicode_AsWideChar()`, `PyUnicode_ReadChar()` or similar new APIs.

Py_UNICODE ***PyUnicode_AsUnicodeAndSize** (*PyObject* *unicode, *Py_ssize_t* *size)

Like `PyUnicode_AsUnicode()`, but also saves the `Py_UNICODE()` array length (excluding the extra null terminator) in *size*. Note that the resulting `Py_UNICODE*` string may contain embedded null code points, which would cause the string to be truncated when used in most C functions.

在 3.3 版新加入.

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: Part of the old-style Unicode API, please migrate to using `PyUnicode_AsUCS4()`, `PyUnicode_AsWideChar()`, `PyUnicode_ReadChar()` or similar new APIs.

Py_ssize_t **PyUnicode_GetSize** (*PyObject* *unicode)

Part of the Stable ABI. Return the size of the deprecated `Py_UNICODE` representation, in code units (this includes surrogate pairs as 2 units).

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: Part of the old-style Unicode API, please migrate to using `PyUnicode_GET_LENGTH()`.

Locale Encoding

The current locale encoding can be used to decode text from the operating system.

PyObject ***PyUnicode_DecodeLocaleAndSize**(const char *str, *Py_ssize_t* length, const char *errors)

回傳值：新的參照。 *Part of the* Stable ABI *since version 3.7.* Decode a string from UTF-8 on Android and VxWorks, or from the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The decoder uses `"strict"` error handler if *errors* is `NULL`. *str* must end with a null character but cannot contain embedded null characters.

Use `PyUnicode_DecodeFSDefaultAndSize()` to decode a string from `Py_FileSystemDefaultEncoding` (the locale encoding read at Python startup).

This function ignores the Python UTF-8 Mode.

也參考:

`Py_DecodeLocale()` 函式。

在 3.3 版新加入.

在 3.7 版的變更: The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, `Py_DecodeLocale()` was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

PyObject ***PyUnicode_DecodeLocale**(const char *str, const char *errors)

回傳值：新的參照。 *Part of the* Stable ABI *since version 3.7.* Similar to `PyUnicode_DecodeLocaleAndSize()`, but compute the string length using `strlen()`.

在 3.3 版新加入.

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

回傳值：新的參照。 *Part of the* Stable ABI *since version 3.7.* Encode a Unicode object to UTF-8 on Android and VxWorks, or to the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The encoder uses `"strict"` error handler if *errors* is `NULL`. Return a `bytes` object. *unicode* cannot contain embedded null characters.

Use `PyUnicode_EncodeFSDefault()` to encode a string to `Py_FileSystemDefaultEncoding` (the locale encoding read at Python startup).

This function ignores the Python UTF-8 Mode.

也參考:

`Py_EncodeLocale()` 函式。

在 3.3 版新加入.

在 3.7 版的變更: The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, `Py_EncodeLocale()` was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

File System Encoding

To encode and decode file names and other environment strings, `Py_FileSystemDefaultEncoding` should be used as the encoding, and `Py_FileSystemDefaultEncodeErrors` should be used as the error handler (**PEP 383** and **PEP 529**). To encode file names to `bytes` during argument parsing, the `"O&"` converter should be used, passing *`PyUnicode_FSConverter()`* as the conversion function:

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

Part of the Stable ABI. ParseTuple converter: encode `str` objects -- obtained directly or through the `os.PathLike` interface -- to `bytes` using *`PyUnicode_EncodeFSDefault()`*; `bytes` objects are output as-is. *result* must be a *`PyBytesObject`*`*` which must be released when it is no longer used.

在 3.1 版新加入.

在 3.6 版的變更: Accepts a *path-like object*.

To decode file names to `str` during argument parsing, the `"O&"` converter should be used, passing *`PyUnicode_FSDecoder()`* as the conversion function:

int **`PyUnicode_FSDecoder`**(*PyObject* *obj, void *result)

Part of the Stable ABI. ParseTuple converter: decode `bytes` objects -- obtained either directly or indirectly through the `os.PathLike` interface -- to `str` using *`PyUnicode_DecodeFSDefaultAndSize()`*; `str` objects are output as-is. *result* must be a *`PyUnicodeObject`*`*` which must be released when it is no longer used.

在 3.2 版新加入.

在 3.6 版的變更: Accepts a *path-like object*.

PyObject ***`PyUnicode_DecodeFSDefaultAndSize`**(const char *str, *Py_ssize_t* size)

回傳值：新的參照。*Part of the* Stable ABI. Decode a string from the *filesystem encoding and error handler*.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

`Py_FileSystemDefaultEncoding` is initialized at startup from the locale encoding and cannot be modified later. If you need to decode a string from the current locale encoding, use *`PyUnicode_DecodeLocaleAndSize()`*.

也參考:

`Py_DecodeLocale()` 函式。

在 3.6 版的變更: Use `Py_FileSystemDefaultEncodeErrors` error handler.

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

回傳值：新的參照。*Part of the* Stable ABI. Decode a null-terminated string from the *filesystem encoding and error handler*.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

Use *`PyUnicode_DecodeFSDefaultAndSize()`* if you know the string length.

在 3.6 版的變更: Use `Py_FileSystemDefaultEncodeErrors` error handler.

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

回傳值：新的參照。*Part of the* Stable ABI. Encode a Unicode object to `Py_FileSystemDefaultEncoding` with the `Py_FileSystemDefaultEncodeErrors` error handler, and return `bytes`. Note that the resulting `bytes` object may contain null bytes.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

`Py_FileSystemDefaultEncoding` is initialized at startup from the locale encoding and cannot be modified later. If you need to encode a string to the current locale encoding, use *`PyUnicode_EncodeLocale()`*.

也參考:

`Py_EncodeLocale()` 函式。

在 3.2 版新加入.

在 3.6 版的變更: Use `Py_FileSystemDefaultEncodeErrors` error handler.

wchar_t Support

`wchar_t` support for platforms which support it:

PyObject ***PyUnicode_FromWideChar***(const wchar_t *wstr, *Py_ssize_t* size)

回傳值：新的參照。*Part of the* Stable ABI. Create a Unicode object from the `wchar_t` buffer *wstr* of the given *size*. Passing `-1` as the *size* indicates that the function must itself compute the length, using `wcslen()`. Return `NULL` on failure.

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

Part of the Stable ABI. Copy the Unicode object contents into the `wchar_t` buffer *wstr*. At most *size* `wchar_t` characters are copied (excluding a possibly trailing null termination character). Return the number of `wchar_t` characters copied or `-1` in case of an error.

When *wstr* is `NULL`, instead return the *size* that would be required to store all of *unicode* including a terminating null.

Note that the resulting `wchar_t*` string may or may not be null-terminated. It is the responsibility of the caller to make sure that the `wchar_t*` string is null-terminated in case this is required by the application. Also, note that the `wchar_t*` string might contain null characters, which would cause the string to be truncated when used with most C functions.

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

Part of the Stable ABI *since version 3.7*. Convert the Unicode object to a wide character string. The output string always ends with a null character. If *size* is not `NULL`, write the number of wide characters (excluding the trailing null termination character) into **size*. Note that the resulting `wchar_t` string might contain null characters, which would cause the string to be truncated when used with most C functions. If *size* is `NULL` and the `wchar_t*` string contains null characters a `ValueError` is raised.

Returns a buffer allocated by `PyMem_New` (use `PyMem_Free()` to free it) on success. On error, returns `NULL` and **size* is undefined. Raises a `MemoryError` if memory allocation is failed.

在 3.2 版新加入.

在 3.7 版的變更: Raises a `ValueError` if *size* is `NULL` and the `wchar_t*` string contains null characters.

Built-in Codecs

Python provides a set of built-in codecs which are written in C for speed. All of these codecs are directly usable via the following functions.

Many of the following APIs take two arguments encoding and errors, and they have the same semantics as the ones of the built-in `str()` string object constructor.

Setting encoding to `NULL` causes the default encoding to be used which is UTF-8. The file system calls should use `PyUnicode_FSConverter()` for encoding file names. This uses the variable `Py_FileSystemDefaultEncoding` internally. This variable should be treated as read-only: on some systems, it will be a pointer to a static string, on others, it will change at run-time (such as when the application invokes setlocale).

Error handling is set by errors which may also be set to `NULL` meaning to use the default handling defined for the codec. Default error handling for all built-in codecs is "strict" (`ValueError` is raised).

The codecs all use a similar interface. Only deviations from the following generic ones are documented for simplicity.

Generic Codecs

These are the generic codec APIs:

PyObject ***PyUnicode_Decode**(const char *str, *Py_ssize_t* size, const char *encoding, const char *errors)

回傳值：新的參照。 *Part of the* Stable ABI. Create a Unicode object by decoding *size* bytes of the encoded string *str*. *encoding* and *errors* have the same meaning as the parameters of the same name in the `str()` built-in function. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsEncodedString**(*PyObject* *unicode, const char *encoding, const char *errors)

回傳值：新的參照。 *Part of the* Stable ABI. Encode a Unicode object and return the result as Python bytes object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode `encode()` method. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

UTF-8 Codecs

These are the UTF-8 codec APIs:

PyObject ***PyUnicode_DecodeUTF8**(const char *str, *Py_ssize_t* size, const char *errors)

回傳值：新的參照。 *Part of the* Stable ABI. Create a Unicode object by decoding *size* bytes of the UTF-8 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeUTF8Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

回傳值：新的參照。 *Part of the* Stable ABI. If *consumed* is `NULL`, behave like *`PyUnicode_DecodeUTF8()`*. If *consumed* is not `NULL`, trailing incomplete UTF-8 byte sequences will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF8String**(*PyObject* *unicode)

回傳值：新的參照。 *Part of the* Stable ABI. Encode a Unicode object using UTF-8 and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

const char ***PyUnicode_AsUTF8AndSize**(*PyObject* *unicode, *Py_ssize_t* *size)

Part of the Stable ABI *since version 3.10.* Return a pointer to the UTF-8 encoding of the Unicode object, and store the size of the encoded representation (in bytes) in *size*. The *size* argument can be `NULL`; in this case no size will be stored. The returned buffer always has an extra null byte appended (not included in *size*), regardless of whether there are any other null code points.

In the case of an error, `NULL` is returned with an exception set and no *size* is stored.

This caches the UTF-8 representation of the string in the Unicode object, and subsequent calls will return a pointer to the same buffer. The caller is not responsible for deallocating the buffer. The buffer is deallocated and pointers to it become invalid when the Unicode object is garbage collected.

在 3.3 版新加入.

在 3.7 版的變更: The return type is now `const char *` rather of `char *`.

在 3.10 版的變更: This function is a part of the *limited API*.

const char ***PyUnicode_AsUTF8**(*PyObject* *unicode)

As *`PyUnicode_AsUTF8AndSize()`*, but does not store the size.

在 3.3 版新加入.

在 3.7 版的變更: The return type is now `const char *` rather of `char *`.

UTF-32 Codecs

These are the UTF-32 codec APIs:

PyObject ***PyUnicode_DecodeUTF32**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

回傳值：新的參照。*Part of the* Stable ABI. Decode *size* bytes from a UTF-32 encoded buffer string and return the corresponding Unicode object. *errors* (if non-NULL) defines the error handling. It defaults to "strict".

If *byteorder* is non-NULL, the decoder starts decoding using the given byte order:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

If `*byteorder` is zero, and the first four bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output.

After completion, **byteorder* is set to the current byte order at the end of input data.

If *byteorder* is NULL, the codec starts in native order mode.

Return NULL if an exception was raised by the codec.

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。*Part of the* Stable ABI. If *consumed* is NULL, behave like *PyUnicode_DecodeUTF32()*. If *consumed* is not NULL, *PyUnicode_DecodeUTF32Stateful()* will not treat trailing incomplete UTF-32 byte sequences (such as a number of bytes not divisible by four) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF32String**(*PyObject* *unicode)

回傳值：新的參照。*Part of the* Stable ABI. Return a Python byte string using the UTF-32 encoding in native byte order. The string always starts with a BOM mark. Error handling is "strict". Return NULL if an exception was raised by the codec.

UTF-16 Codecs

These are the UTF-16 codec APIs:

PyObject ***PyUnicode_DecodeUTF16**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

回傳值：新的參照。*Part of the* Stable ABI. Decode *size* bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. *errors* (if non-NULL) defines the error handling. It defaults to "strict".

If *byteorder* is non-NULL, the decoder starts decoding using the given byte order:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

If `*byteorder` is zero, and the first two bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output (where it will result in either a `\ufeff` or a `\ufffe` character).

After completion, `*byteorder` is set to the current byte order at the end of input data.

If *byteorder* is NULL, the codec starts in native order mode.

Return NULL if an exception was raised by the codec.

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。*Part of the* Stable ABI. If *consumed* is NULL, behave like `PyUnicode_DecodeUTF16()`. If *consumed* is not NULL, `PyUnicode_DecodeUTF16Stateful()` will not treat trailing incomplete UTF-16 byte sequences (such as an odd number of bytes or a split surrogate pair) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF16String**(*PyObject* *unicode)

回傳值：新的參照。*Part of the* Stable ABI. Return a Python byte string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is "strict". Return NULL if an exception was raised by the codec.

UTF-7 Codecs

These are the UTF-7 codec APIs:

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

回傳值：新的參照。*Part of the* Stable ABI. Create a Unicode object by decoding *size* bytes of the UTF-7 encoded string *str*. Return NULL if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeUTF7Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

回傳值：新的參照。*Part of the* Stable ABI. If *consumed* is NULL, behave like `PyUnicode_DecodeUTF7()`. If *consumed* is not NULL, trailing incomplete UTF-7 base-64 sections will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

Unicode-Escape Codecs

These are the "Unicode Escape" codec APIs:

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

回傳值：新的參照。*Part of the* Stable ABI. Create a Unicode object by decoding *size* bytes of the Unicode-Escape encoded string *str*. Return NULL if an exception was raised by the codec.

PyObject ***PyUnicode_AsUnicodeEscapeString**(*PyObject* *unicode)

回傳值：新的參照。*Part of the* Stable ABI. Encode a Unicode object using Unicode-Escape and return the result as a bytes object. Error handling is "strict". Return NULL if an exception was raised by the codec.

Raw-Unicode-Escape Codecs

These are the "Raw Unicode Escape" codec APIs:

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

回傳值：新的參照。*Part of the* Stable ABI. Create a Unicode object by decoding *size* bytes of the Raw-Unicode-Escape encoded string *str*. Return NULL if an exception was raised by the codec.

PyObject ***PyUnicode_AsRawUnicodeEscapeString**(*PyObject* *unicode)

回傳值：新的參照。*Part of the* Stable ABI. Encode a Unicode object using Raw-Unicode-Escape and return the result as a bytes object. Error handling is "strict". Return NULL if an exception was raised by the codec.

Latin-1 Codecs

These are the Latin-1 codec APIs: Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

PyObject ***PyUnicode_DecodeLatin1**(const char *str, *Py_ssize_t* size, const char *errors)

回傳值：新的參照。 *Part of the* Stable ABI. Create a Unicode object by decoding *size* bytes of the Latin-1 encoded string *str*. Return NULL if an exception was raised by the codec.

PyObject ***PyUnicode_AsLatin1String**(*PyObject* *unicode)

回傳值：新的參照。 *Part of the* Stable ABI. Encode a Unicode object using Latin-1 and return the result as Python bytes object. Error handling is "strict". Return NULL if an exception was raised by the codec.

ASCII Codecs

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

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

回傳值：新的參照。 *Part of the* Stable ABI. Create a Unicode object by decoding *size* bytes of the ASCII encoded string *str*. Return NULL if an exception was raised by the codec.

PyObject ***PyUnicode_AsASCIIString**(*PyObject* *unicode)

回傳值：新的參照。 *Part of the* Stable ABI. Encode a Unicode object using ASCII and return the result as Python bytes object. Error handling is "strict". Return NULL if an exception was raised by the codec.

Character Map Codecs

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the encodings package). The codec uses mappings to encode and decode characters. The mapping objects provided must support the `__getitem__()` mapping interface; dictionaries and sequences work well.

These are the mapping codec APIs:

PyObject ***PyUnicode_DecodeCharmap**(const char *str, *Py_ssize_t* length, *PyObject* *mapping, const char *errors)

回傳值：新的參照。 *Part of the* Stable ABI. Create a Unicode object by decoding *size* bytes of the encoded string *str* using the given *mapping* object. Return NULL if an exception was raised by the codec.

If *mapping* is NULL, Latin-1 decoding will be applied. Else *mapping* must map bytes ordinals (integers in the range from 0 to 255) to Unicode strings, integers (which are then interpreted as Unicode ordinals) or None. Unmapped data bytes -- ones which cause a LookupError, as well as ones which get mapped to None, 0xFFFE or '\ufffe', are treated as undefined mappings and cause an error.

PyObject ***PyUnicode_AsCharmapString**(*PyObject* *unicode, *PyObject* *mapping)

回傳值：新的參照。 *Part of the* Stable ABI. Encode a Unicode object using the given *mapping* object and return the result as a bytes object. Error handling is "strict". Return NULL if an exception was raised by the codec.

The *mapping* object must map Unicode ordinal integers to bytes objects, integers in the range from 0 to 255 or None. Unmapped character ordinals (ones which cause a LookupError) as well as mapped to None are treated as "undefined mapping" and cause an error.

The following codec API is special in that maps Unicode to Unicode.

PyObject ***PyUnicode_Translate**(*PyObject* *unicode, *PyObject* *table, const char *errors)

回傳值：新的參照。 *Part of the* Stable ABI. Translate a string by applying a character mapping table to it and return the resulting Unicode object. Return NULL if an exception was raised by the codec.

The mapping table must map Unicode ordinal integers to Unicode ordinal integers or `None` (causing deletion of the character).

Mapping tables need only provide the `__getitem__()` interface; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a `LookupError`) are left untouched and are copied as-is.

errors has the usual meaning for codecs. It may be `NULL` which indicates to use the default error handling.

MBCS codecs for Windows

These are the MBCS codec APIs. They are currently only available on Windows and use the Win32 MBCS converters to implement the conversions. Note that MBCS (or DBCS) is a class of encodings, not just one. The target encoding is defined by the user settings on the machine running the codec.

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

回傳值：新的參照。 *Part of the* Stable ABI *on Windows since version 3.7.* Create a Unicode object by decoding *size* bytes of the MBCS encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeMBCSStateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

回傳值：新的參照。*Part of the* Stable ABI *on Windows since version 3.7.* If *consumed* is `NULL`, behave like *PyUnicode_DecodeMBCS()*. If *consumed* is not `NULL`, *PyUnicode_DecodeMBCSStateful()* will not decode trailing lead byte and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsMBCSString**(*PyObject* *unicode)

回傳值：新的參照。 *Part of the* Stable ABI *on Windows since version 3.7.* Encode a Unicode object using MBCS and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_EncodeCodePage**(int code_page, *PyObject* *unicode, const char *errors)

回傳值：新的參照。 *Part of the* Stable ABI *on Windows since version 3.7.* Encode the Unicode object using the specified code page and return a Python bytes object. Return `NULL` if an exception was raised by the codec. Use `CP_ACP` code page to get the MBCS encoder.

在 3.3 版新加入.

Methods & Slots

Methods and Slot Functions

The following APIs are capable of handling Unicode objects and strings on input (we refer to them as strings in the descriptions) and return Unicode objects or integers as appropriate.

They all return `NULL` or `-1` if an exception occurs.

PyObject ***PyUnicode_Concat**(*PyObject* *left, *PyObject* *right)

回傳值：新的參照。 *Part of the* Stable ABI. Concat two strings giving a new Unicode string.

PyObject ***PyUnicode_Split**(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

回傳值：新的參照。 *Part of the* Stable ABI. Split a string giving a list of Unicode strings. If *sep* is `NULL`, splitting will be done at all whitespace substrings. Otherwise, splits occur at the given separator. At most *maxsplit* splits will be done. If negative, no limit is set. Separators are not included in the resulting list.

PyObject ***PyUnicode_Splitlines**(*PyObject* *unicode, int keepends)

回傳值：新的參照。*Part of the* Stable ABI. Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If *keepends* is `0`, the Line break characters are not included in the resulting strings.

PyObject ***PyUnicode_Join***(*PyObject* *separator, *PyObject* *seq)

回傳值：新的參照。 *Part of the* Stable ABI. Join a sequence of strings using the given *separator* and return the resulting Unicode string.

Py_ssize_t **PyUnicode_Tailmatch**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Part of the Stable ABI. Return `1` if *substr* matches `unicode[start:end]` at the given tail end (*direction* == `-1` means to do a prefix match, *direction* == `1` a suffix match), `0` otherwise. Return `-1` if an error occurred.

Py_ssize_t **PyUnicode_Find**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Part of the Stable ABI. Return the first position of *substr* in `unicode[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Py_ssize_t **PyUnicode_FindChar**(*PyObject* *unicode, *Py_UCS4* ch, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Part of the Stable ABI *since version 3.7.* Return the first position of the character *ch* in `unicode[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

在 3.3 版新加入.

在 3.7 版的變更: *start* and *end* are now adjusted to behave like `unicode[start:end]`.

Py_ssize_t **PyUnicode_Count**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end)

Part of the Stable ABI. Return the number of non-overlapping occurrences of *substr* in `unicode[start:end]`. Return `-1` if an error occurred.

PyObject ***PyUnicode_Replace***(*PyObject* *unicode, *PyObject* *substr, *PyObject* *replstr, *Py_ssize_t* maxcount)

回傳值：新的參照。 *Part of the* Stable ABI. Replace at most *maxcount* occurrences of *substr* in *unicode* with *replstr* and return the resulting Unicode object. *maxcount* == `-1` means replace all occurrences.

int **PyUnicode_Compare**(*PyObject* *left, *PyObject* *right)

Part of the Stable ABI. Compare two strings and return `-1`, `0`, `1` for less than, equal, and greater than, respectively.

This function returns `-1` upon failure, so one should call `PyErr_Occurred()` to check for errors.

int **PyUnicode_CompareWithASCIIString**(*PyObject* *unicode, const char *string)

Part of the Stable ABI. Compare a Unicode object, *unicode*, with *string* and return `-1`, `0`, `1` for less than, equal, and greater than, respectively. It is best to pass only ASCII-encoded strings, but the function interprets the input string as ISO-8859-1 if it contains non-ASCII characters.

This function does not raise exceptions.

PyObject ***PyUnicode_RichCompare***(*PyObject* *left, *PyObject* *right, int op)

回傳值：新的參照。*Part of the* Stable ABI. Rich compare two Unicode strings and return one of the following:

- `NULL` in case an exception was raised
- `Py_True` or `Py_False` for successful comparisons
- `Py_NotImplemented` in case the type combination is unknown

Possible values for *op* are `Py_GT`, `Py_GE`, `Py_EQ`, `Py_NE`, `Py_LT`, and `Py_LE`.

PyObject ***PyUnicode_Format***(*PyObject* *format, *PyObject* *args)

回傳值：新的參照。*Part of the* Stable ABI. Return a new string object from *format* and *args*; this is analogous to `format % args`.

int **PyUnicode_Contains**(*PyObject* *unicode, *PyObject* *substr)

Part of the Stable ABI. Check whether *substr* is contained in *unicode* and return true or false accordingly.

substr has to coerce to a one element Unicode string. `-1` is returned if there was an error.

void **PyUnicode_InternInPlace**(*PyObject* **p_unicode)

Part of the Stable ABI. Intern the argument `*p_unicode` in place. The argument must be the address of a pointer variable pointing to a Python Unicode string object. If there is an existing interned string that is the same as `*p_unicode`, it sets `*p_unicode` to it (releasing the reference to the old string object and creating a new *strong reference* to the interned string object), otherwise it leaves `*p_unicode` alone and interns it (creating a new *strong reference*). (Clarification: even though there is a lot of talk about references, think of this function as reference-neutral; you own the object after the call if and only if you owned it before the call.)

PyObject ***PyUnicode_InternFromString**(const char *str)

回傳值：新的參照。 *Part of the* Stable ABI. A combination of `PyUnicode_FromString()` and `PyUnicode_InternInPlace()`, returning either a new Unicode string object that has been interned, or a new ("owned") reference to an earlier interned string object with the same value.

8.3.4 元組（Tuple）物件

type **PyTupleObject**

This subtype of `PyObject` represents a Python tuple object.

PyTypeObject **PyTuple_Type**

Part of the Stable ABI. This instance of `PyTypeObject` represents the Python tuple type; it is the same object as `tuple` in the Python layer.

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

Return true if *p* is a tuple object or an instance of a subtype of the tuple type. This function always succeeds.

int **PyTuple_CheckExact**(*PyObject* *p)

Return true if *p* is a tuple object, but not an instance of a subtype of the tuple type. This function always succeeds.

PyObject ***PyTuple_New**(*Py_ssize_t* len)

回傳值：新的參照。 *Part of the* Stable ABI. Return a new tuple object of size *len*, or `NULL` on failure.

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

回傳值：新的參照。 *Part of the* Stable ABI. Return a new tuple object of size *n*, or `NULL` on failure. The tuple values are initialized to the subsequent *n* C arguments pointing to Python objects. `PyTuple_Pack(2, a, b)` is equivalent to `Py_BuildValue("(OO)", a, b)`.

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

Part of the Stable ABI. Take a pointer to a tuple object, and return the size of that tuple.

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

Return the size of the tuple *p*, which must be non-`NULL` and point to a tuple; no error checking is performed.

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

回傳值：借用參照。 *Part of the* Stable ABI. Return the object at position *pos* in the tuple pointed to by *p*. If *pos* is negative or out of bounds, return `NULL` and set an `IndexError` exception.

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

回傳值：借用參照。 Like `PyTuple_GetItem()`, but does no checking of its arguments.

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

回傳值：新的參照。 *Part of the* Stable ABI. Return the slice of the tuple pointed to by *p* between *low* and *high*, or `NULL` on failure. This is the equivalent of the Python expression `p[low:high]`. Indexing from the end of the list is not supported.

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

Part of the Stable ABI. Insert a reference to object *o* at position *pos* of the tuple pointed to by *p*. Return 0 on success. If *pos* is out of bounds, return -1 and set an IndexError exception.

備𝐹: This function "steals" a reference to *o* and discards a reference to an item already in the tuple at the affected position.

void **PyTuple_SET_ITEM**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Like *PyTuple_SetItem()*, but does no error checking, and should *only* be used to fill in brand new tuples.

備𝐹: This function "steals" a reference to *o*, and, unlike *PyTuple_SetItem()*, does *not* discard a reference to any item that is being replaced; any reference in the tuple at position *pos* will be leaked.

int **_PyTuple_Resize**(*PyObject* **p, *Py_ssize_t* newsize)

Can be used to resize a tuple. *newsize* will be the new length of the tuple. Because tuples are *supposed* to be immutable, this should only be used if there is only one reference to the object. Do *not* use this if the tuple may already be known to some other part of the code. The tuple will always grow or shrink at the end. Think of this as destroying the old tuple and creating a new one, only more efficiently. Returns 0 on success. Client code should never assume that the resulting value of *p will be the same as before calling this function. If the object referenced by *p is replaced, the original *p is destroyed. On failure, returns -1 and sets *p to NULL, and raises MemoryError or SystemError.

8.3.5 Struct Sequence Objects

Struct sequence objects are the C equivalent of namedtuple() objects, i.e. a sequence whose items can also be accessed through attributes. To create a struct sequence, you first have to create a specific struct sequence type.

PyTypeObject ***PyStructSequence_NewType**(*PyStructSequence_Desc* *desc)

回傳值：新的參照。*Part of the* Stable ABI. Create a new struct sequence type from the data in *desc*, described below. Instances of the resulting type can be created with *PyStructSequence_New()*.

void **PyStructSequence_InitType**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

Initializes a struct sequence type *type* from *desc* in place.

int **PyStructSequence_InitType2**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

The same as PyStructSequence_InitType, but returns 0 on success and -1 on failure.

在 3.4 版新加入.

type **PyStructSequence_Desc**

Part of the Stable ABI *(including all members)*. Contains the meta information of a struct sequence type to create.

const char ***name**

Name of the struct sequence type.

const char ***doc**

Pointer to docstring for the type or NULL to omit.

PyStructSequence_Field ***fields**

Pointer to NULL-terminated array with field names of the new type.

int **n_in_sequence**

Number of fields visible to the Python side (if used as tuple).

type **PyStructSequence_Field**

Part of the Stable ABI *(including all members).* Describes a field of a struct sequence. As a struct sequence is modeled as a tuple, all fields are typed as `PyObject*`. The index in the `fields` array of the `PyStructSequence_Desc` determines which field of the struct sequence is described.

const char ***name**

Name for the field or `NULL` to end the list of named fields, set to `PyStructSequence_UnnamedField` to leave unnamed.

const char ***doc**

Field docstring or `NULL` to omit.

const char *const **PyStructSequence_UnnamedField**

Part of the Stable ABI *since version 3.11.* Special value for a field name to leave it unnamed.

在 3.9 版的變更: The type was changed from `char *`.

PyObject ***PyStructSequence_New**(*PyTypeObject* *type)

回傳值：新的參照。 *Part of the* Stable ABI. Creates an instance of *type*, which must have been created with `PyStructSequence_NewType()`.

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

回傳值：借用參照。 *Part of the* Stable ABI. Return the object at position *pos* in the struct sequence pointed to by *p*. No bounds checking is performed.

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

回傳值：借用參照。 Macro equivalent of `PyStructSequence_GetItem()`.

void **PyStructSequence_SetItem**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Part of the Stable ABI. Sets the field at index *pos* of the struct sequence *p* to value *o*. Like `PyTuple_SET_ITEM()`, this should only be used to fill in brand new instances.

備F: This function "steals" a reference to *o*.

void **PyStructSequence_SET_ITEM**(*PyObject* *p, *Py_ssize_t* *pos, *PyObject* *o)

Similar to `PyStructSequence_SetItem()`, but implemented as a static inlined function.

備F: This function "steals" a reference to *o*.

8.3.6 List（串列）物件

type **PyListObject**

This subtype of `PyObject` represents a Python list object.

PyTypeObject **PyList_Type**

Part of the Stable ABI. This instance of `PyTypeObject` represents the Python list type. This is the same object as `list` in the Python layer.

int **PyList_Check**(*PyObject* *p)

如果 *p* 是一個 list 物件或者是 list 型F的子型F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F的子型F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。 *Part of the* Stable ABI. Return a new list of length *len* on success, or NULL on failure.

> **備F:** If *len* is greater than zero, the returned list object's items are set to NULL. Thus you cannot use abstract API functions such as `PySequence_SetItem()` or expose the object to Python code before setting all items to a real object with `PyList_SetItem()`.

Py_ssize_t **PyList_Size**(*PyObject* *list)

Part of the Stable ABI. Return the length of the list object in *list*; this is equivalent to `len(list)` on a list object.

Py_ssize_t **PyList_GET_SIZE**(*PyObject* *list)

與`PyList_Size()`類似，但F有錯誤檢查。

PyObject ***PyList_GetItem**(*PyObject* *list, *Py_ssize_t* index)

回傳值：借用參照。 *Part of the* Stable ABI. Return the object at position *index* in the list pointed to by *list*. The position must be non-negative; indexing from the end of the list is not supported. If *index* is out of bounds (<0 or >=len(list)), return NULL and set an `IndexError` exception.

PyObject ***PyList_GET_ITEM**(*PyObject* *list, *Py_ssize_t* i)

回傳值：借用參照。 Similar to `PyList_GetItem()`, but without error checking.

int **PyList_SetItem**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

Part of the Stable ABI. Set the item at index *index* in list to *item*. Return 0 on success. If *index* is out of bounds, return -1 and set an `IndexError` exception.

> **備F:** This function "steals" a reference to *item* and discards a reference to an item already in the list at the affected position.

void **PyList_SET_ITEM**(*PyObject* *list, *Py_ssize_t* i, *PyObject* *o)

Macro form of `PyList_SetItem()` without error checking. This is normally only used to fill in new lists where there is no previous content.

> **備F:** This macro "steals" a reference to *item*, and, unlike `PyList_SetItem()`, does *not* discard a reference to any item that is being replaced; any reference in *list* at position *i* will be leaked.

int **PyList_Insert**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

Part of the Stable ABI. Insert the item *item* into list *list* in front of index *index*. Return 0 if successful; return -1 and set an exception if unsuccessful. Analogous to `list.insert(index, item)`.

int **PyList_Append**(*PyObject* *list, *PyObject* *item)

Part of the Stable ABI. Append the object *item* at the end of list *list*. Return 0 if successful; return -1 and set an exception if unsuccessful. Analogous to `list.append(item)`.

PyObject ***PyList_GetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high)

回傳值：新的參照。 *Part of the* Stable ABI. Return a list of the objects in *list* containing the objects *between low* and *high*. Return NULL and set an exception if unsuccessful. Analogous to `list[low:high]`. Indexing from the end of the list is not supported.

int **PyList_SetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high, *PyObject* *itemlist)

Part of the Stable ABI. Set the slice of *list* between *low* and *high* to the contents of *itemlist*. Analogous to `list[low:high] = itemlist`. The *itemlist* may be NULL, indicating the assignment of an empty list (slice deletion). Return 0 on success, -1 on failure. Indexing from the end of the list is not supported.

int **PyList_Sort** (*PyObject* *list)

Part of the Stable ABI. Sort the items of *list* in place. Return `0` on success, `-1` on failure. This is equivalent to `list.sort()`.

int **PyList_Reverse** (*PyObject* *list)

Part of the Stable ABI. Reverse the items of *list* in place. Return `0` on success, `-1` on failure. This is the equivalent of `list.reverse()`.

PyObject ***PyList_AsTuple** (*PyObject* *list)

回傳值:新的參照。*Part of the* Stable ABI. Return a new tuple object containing the contents of *list*; equivalent to `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**

Part of the Stable ABI. `PyTypeObject` 實例代表一個 Python 字典型態。此與 Python 層中的 `dict` 為同一個物件。

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* 是一個字典物件但並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。 *Part of the* Stable ABI. Return a new empty dictionary, or `NULL` on failure.

PyObject ***PyDictProxy_New** (*PyObject* *mapping)

回傳值:新的參照。*Part of the* Stable ABI. Return a `types.MappingProxyType` object for a mapping which enforces read-only behavior. This is normally used to create a view to prevent modification of the dictionary for non-dynamic class types.

void **PyDict_Clear** (*PyObject* *p)

Part of the Stable ABI. Empty an existing dictionary of all key-value pairs.

int **PyDict_Contains** (*PyObject* *p, *PyObject* *key)

Part of the Stable ABI. Determine if dictionary *p* contains *key*. If an item in *p* is matches *key*, return `1`, otherwise return `0`. On error, return `-1`. This is equivalent to the Python expression `key in p`.

PyObject ***PyDict_Copy** (*PyObject* *p)

回傳值：新的參照。 *Part of the* Stable ABI. Return a new dictionary that contains the same key-value pairs as *p*.

int **PyDict_SetItem** (*PyObject* *p, *PyObject* *key, *PyObject* *val)

Part of the Stable ABI. Insert *val* into the dictionary *p* with a key of *key*. *key* must be *hashable*; if it isn't, `TypeError` will be raised. Return `0` on success or `-1` on failure. This function *does not* steal a reference to *val*.

int **PyDict_SetItemString** (*PyObject* *p, const char *key, *PyObject* *val)

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

int **PyDict_DelItem**(*PyObject* *p, *PyObject* *key)

Part of the Stable ABI. Remove the entry in dictionary *p* with key *key*. *key* must be *hashable*; if it isn't, `TypeError` is raised. If *key* is not in the dictionary, `KeyError` is raised. Return `0` on success or `-1` on failure.

int **PyDict_DelItemString**(*PyObject* *p, const char *key)

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

PyObject ***PyDict_GetItem**(*PyObject* *p, *PyObject* *key)

回傳值：借用參照。*Part of the* Stable ABI. Return the object from dictionary *p* which has a key *key*. Return `NULL` if the key *key* is not present, but *without* setting an exception.

備F: Exceptions that occur while this calls `__hash__()` and `__eq__()` methods are silently ignored. Prefer the `PyDict_GetItemWithError()` function instead.

在 3.10 版的變更: Calling this API without *GIL* held had been allowed for historical reason. It is no longer allowed.

PyObject ***PyDict_GetItemWithError**(*PyObject* *p, *PyObject* *key)

回傳值：借用參照。*Part of the* Stable ABI. Variant of `PyDict_GetItem()` that does not suppress exceptions. Return `NULL` **with** an exception set if an exception occurred. Return `NULL` **without** an exception set if the key wasn't present.

PyObject ***PyDict_GetItemString**(*PyObject* *p, const char *key)

回傳值：借用參照。*Part of the* Stable ABI. This is the same as `PyDict_GetItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

備F: Exceptions that occur while this calls `__hash__()` and `__eq__()` methods or while creating the temporary `str` object are silently ignored. Prefer using the `PyDict_GetItemWithError()` function with your own `PyUnicode_FromString()` *key* instead.

PyObject ***PyDict_SetDefault**(*PyObject* *p, *PyObject* *key, *PyObject* *defaultobj)

回傳值：借用參照。This is the same as the Python-level `dict.setdefault()`. If present, it returns the value corresponding to *key* from the dictionary *p*. If the key is not in the dict, it is inserted with value *defaultobj* and *defaultobj* is returned. This function evaluates the hash function of *key* only once, instead of evaluating it independently for the lookup and the insertion.

在 3.4 版新加入.

PyObject ***PyDict_Items**(*PyObject* *p)

回傳值：新的參照。*Part of the* Stable ABI. Return a `PyListObject` containing all the items from the dictionary.

PyObject ***PyDict_Keys**(*PyObject* *p)

回傳值：新的參照。*Part of the* Stable ABI. Return a `PyListObject` containing all the keys from the dictionary.

PyObject ***PyDict_Values**(*PyObject* *p)

回傳值：新的參照。*Part of the* Stable ABI. Return a `PyListObject` containing all the values from the dictionary *p*.

Py_ssize_t **PyDict_Size**(*PyObject* *p)

Part of the Stable ABI. Return the number of items in the dictionary. This is equivalent to `len(p)` on a dictionary.

int **PyDict_Next** (*PyObject* *p, *Py_ssize_t* *ppos, *PyObject* **pkey, *PyObject* **pvalue)

Part of the Stable ABI. Iterate over all key-value pairs in the dictionary *p*. The `Py_ssize_t` referred to by *ppos* must be initialized to `0` prior to the first call to this function to start the iteration; the function returns true for each pair in the dictionary, and false once all pairs have been reported. The parameters *pkey* and *pvalue* should either point to `PyObject*` variables that will be filled in with each key and value, respectively, or may be `NULL`. Any references returned through them are borrowed. *ppos* should not be altered during iteration. Its value represents offsets within the internal dictionary structure, and since the structure is sparse, the offsets are not consecutive.

舉例來𝔽:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}
```

The dictionary *p* should not be mutated during iteration. It is safe to modify the values of the keys as you iterate over the dictionary, but only so long as the set of keys does not change. For example:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    long i = PyLong_AsLong(value);
    if (i == -1 && PyErr_Occurred()) {
        return -1;
    }
    PyObject *o = PyLong_FromLong(i + 1);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
    Py_DECREF(o);
}
```

int **PyDict_Merge** (*PyObject* *a, *PyObject* *b, int override)

Part of the Stable ABI. Iterate over mapping object *b* adding key-value pairs to dictionary *a*. *b* may be a dictionary, or any object supporting `PyMapping_Keys()` and `PyObject_GetItem()`. If *override* is true, existing pairs in *a* will be replaced if a matching key is found in *b*, otherwise pairs will only be added if there is not a matching key in *a*. Return `0` on success or `-1` if an exception was raised.

int **PyDict_Update** (*PyObject* *a, *PyObject* *b)

Part of the Stable ABI. This is the same as `PyDict_Merge(a, b, 1)` in C, and is similar to `a.update(b)` in Python except that `PyDict_Update()` doesn't fall back to the iterating over a sequence of key value pairs if the second argument has no "keys" attribute. Return `0` on success or `-1` if an exception was raised.

int **PyDict_MergeFromSeq2** (*PyObject* *a, *PyObject* *seq2, int override)

Part of the Stable ABI. Update or merge into dictionary *a*, from the key-value pairs in *seq2*. *seq2* must be an iterable object producing iterable objects of length 2, viewed as key-value pairs. In case of duplicate keys, the last wins if *override* is true, else the first wins. Return `0` on success or `-1` if an exception was raised. Equivalent Python (except for the return value):

```
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
```

(繼續下一頁)

(繼續上一頁)

```
        if override or key not in a:
            a[key] = value
```

8.4.2 Set Objects

This section details the public API for `set` and `frozenset` objects. Any functionality not listed below is best accessed using either the abstract object protocol (including `PyObject_CallMethod()`, `PyObject_RichCompareBool()`, `PyObject_Hash()`, `PyObject_Repr()`, `PyObject_IsTrue()`, `PyObject_Print()`, and `PyObject_GetIter()`) or the abstract number protocol (including `PyNumber_And()`, `PyNumber_Subtract()`, `PyNumber_Or()`, `PyNumber_Xor()`, `PyNumber_InPlaceAnd()`, `PyNumber_InPlaceSubtract()`, `PyNumber_InPlaceOr()`, and `PyNumber_InPlaceXor()`).

type **PySetObject**

This subtype of `PyObject` is used to hold the internal data for both `set` and `frozenset` objects. It is like a `PyDictObject` in that it is a fixed size for small sets (much like tuple storage) and will point to a separate, variable sized block of memory for medium and large sized sets (much like list storage). None of the fields of this structure should be considered public and all are subject to change. All access should be done through the documented API rather than by manipulating the values in the structure.

PyTypeObject **PySet_Type**

Part of the Stable ABI. This is an instance of `PyTypeObject` representing the Python `set` type.

PyTypeObject **PyFrozenSet_Type**

Part of the Stable ABI. This is an instance of `PyTypeObject` representing the Python `frozenset` type.

The following type check macros work on pointers to any Python object. Likewise, the constructor functions work with any iterable Python object.

int **PySet_Check**(*PyObject* *p)

Return true if *p* is a `set` object or an instance of a subtype. This function always succeeds.

int **PyFrozenSet_Check**(*PyObject* *p)

Return true if *p* is a `frozenset` object or an instance of a subtype. This function always succeeds.

int **PyAnySet_Check**(*PyObject* *p)

Return true if *p* is a `set` object, a `frozenset` object, or an instance of a subtype. This function always succeeds.

int **PySet_CheckExact**(*PyObject* *p)

Return true if *p* is a `set` object but not an instance of a subtype. This function always succeeds.

在 3.10 版新加入.

int **PyAnySet_CheckExact**(*PyObject* *p)

Return true if *p* is a `set` object or a `frozenset` object but not an instance of a subtype. This function always succeeds.

int **PyFrozenSet_CheckExact**(*PyObject* *p)

Return true if *p* is a `frozenset` object but not an instance of a subtype. This function always succeeds.

PyObject ***PySet_New**(*PyObject* *iterable)

回傳值：新的參照。 *Part of the* Stable ABI. Return a new `set` containing objects returned by the *iterable*. The *iterable* may be `NULL` to create a new empty set. Return the new set on success or `NULL` on failure. Raise `TypeError` if *iterable* is not actually iterable. The constructor is also useful for copying a set (`c=set(s)`).

PyObject ***PyFrozenSet_New**(*PyObject* *iterable)

回傳值：新的參照。 *Part of the* Stable ABI. Return a new `frozenset` containing objects returned by the *iterable*. The *iterable* may be `NULL` to create a new empty frozenset. Return the new set on success or `NULL` on failure. Raise `TypeError` if *iterable* is not actually iterable.

The following functions and macros are available for instances of `set` or `frozenset` or instances of their subtypes.

Py_ssize_t **PySet_Size**(*PyObject* *anyset)

Part of the Stable ABI. Return the length of a `set` or `frozenset` object. Equivalent to `len(anyset)`. Raises a `SystemError` if *anyset* is not a `set`, `frozenset`, or an instance of a subtype.

Py_ssize_t **PySet_GET_SIZE**(*PyObject* *anyset)

Macro form of `PySet_Size()` without error checking.

int **PySet_Contains**(*PyObject* *anyset, *PyObject* *key)

Part of the Stable ABI. Return `1` if found, `0` if not found, and `-1` if an error is encountered. Unlike the Python `__contains__()` method, this function does not automatically convert unhashable sets into temporary frozensets. Raise a `TypeError` if the *key* is unhashable. Raise `SystemError` if *anyset* is not a `set`, `frozenset`, or an instance of a subtype.

int **PySet_Add**(*PyObject* *set, *PyObject* *key)

Part of the Stable ABI. Add *key* to a `set` instance. Also works with `frozenset` instances (like `PyTuple_SetItem()` it can be used to fill in the values of brand new frozensets before they are exposed to other code). Return `0` on success or `-1` on failure. Raise a `TypeError` if the *key* is unhashable. Raise a `MemoryError` if there is no room to grow. Raise a `SystemError` if *set* is not an instance of `set` or its subtype.

The following functions are available for instances of `set` or its subtypes but not for instances of `frozenset` or its subtypes.

int **PySet_Discard**(*PyObject* *set, *PyObject* *key)

Part of the Stable ABI. Return `1` if found and removed, `0` if not found (no action taken), and `-1` if an error is encountered. Does not raise `KeyError` for missing keys. Raise a `TypeError` if the *key* is unhashable. Unlike the Python `discard()` method, this function does not automatically convert unhashable sets into temporary frozensets. Raise `SystemError` if *set* is not an instance of `set` or its subtype.

PyObject ***PySet_Pop**(*PyObject* *set)

回傳值：新的參照。 *Part of the* Stable ABI. Return a new reference to an arbitrary object in the *set*, and removes the object from the *set*. Return `NULL` on failure. Raise `KeyError` if the set is empty. Raise a `SystemError` if *set* is not an instance of `set` or its subtype.

int **PySet_Clear**(*PyObject* *set)

Part of the Stable ABI. Empty an existing set of all elements. Return `0` on success. Return `-1` and raise `SystemError` if *set* is not an instance of `set` or its subtype.

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 函式型別，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` 的型別）則回傳 true。參數必須不為 `NULL`。此函式必能成功執行。

PyObject ***PyFunction_New**(*PyObject* *code, *PyObject* *globals)

回傳值：新的參照。回傳一個與程式碼物件 *code* 相關聯的函式物件。*globals* 必須是一個帶有函式能夠存取的全域變數的字典。

The function's docstring and name are retrieved from the code object. `__module__` is retrieved from *globals*. The argument defaults, annotations and closure are set to `NULL`. `__qualname__` is set to the same value as the code object's `co_qualname` field.

PyObject ***PyFunction_NewWithQualName**(*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)

回傳值：新的參照。As `PyFunction_New()`, but also allows setting the function object's `__qualname__` attribute. *qualname* should be a unicode object or `NULL`; if `NULL`, the `__qualname__` attribute is set to the same value as the code object's `co_qualname` field.

在 3.3 版新加入.

PyObject ***PyFunction_GetCode**(*PyObject* *op)

回傳值：借用參照。回傳與程式碼物件相關的函式物件 *op*。

PyObject ***PyFunction_GetGlobals**(*PyObject* *op)

回傳值：借用參照。回傳與全域函式字典相關的函式物件 *op*。

PyObject ***PyFunction_GetModule**(*PyObject* *op)

回傳值：借用參照。Return a *borrowed reference* to the `__module__` attribute of the function object *op*. It can be *NULL*.

This is normally a `string` containing the module name, but can be set to any other object by Python code.

PyObject ***PyFunction_GetDefaults**(*PyObject* *op)

回傳值：借用參照。回傳函式物件 *op* 的引數預設值，這可以是一個含有多個引數的 tuple（元組）或 `NULL`。

int **PyFunction_SetDefaults**(*PyObject* *op, *PyObject* *defaults)

設定函式物件 *op* 的引數預設值。*defaults* 必須是 `Py_None` 或一個 tuple。

引發 `SystemError` 且在失敗時回傳 `-1`。

PyObject ***PyFunction_GetClosure**(*PyObject* *op)

回傳值：借用參照。回傳與函式物件 *op* 相關聯的閉包，這可以是個 `NULL` 或是一個包含 cell 物件的 tuple。

int **PyFunction_SetClosure**(*PyObject* *op, *PyObject* *closure)

設定與函式物件 *op* 相關聯的閉包，*closure* 必須是 `Py_None` 或是一個包含 cell 物件的 tuple。

引發 `SystemError` 且在失敗時回傳 `-1`。

PyObject ***PyFunction_GetAnnotations**(*PyObject* *op)

回傳值：借用參照。回傳函式物件 *op* 的標註，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* 的標註，*annotations* 必須是一個字典或 `Py_None`。

引發 `SystemError` 且在失敗時回傳 `-1`。

8.5.2 實例方法物件 (Instance Method Objects)

An instance method is a wrapper for a *PyCFunction* and the new way to bind a *PyCFunction* to a class object. It replaces the former call `PyMethod_New(func, NULL, class)`.

PyTypeObject **PyInstanceMethod_Type**

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 物件為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 物件的物件型別。

int **PyCell_Check** (*PyObject* *ob)

如果 *ob* 是一個 cell 物件則回傳真值；*ob* 必須不為 NULL。此函式總是會成功執行。

PyObject ***PyCell_New** (*PyObject* *ob)

回傳值：新的參照。 建立並回傳一個包含 *ob* 的新 cell 物件。參數可以為 NULL。

PyObject ***PyCell_Get** (*PyObject* *cell)

回傳值：新的參照。 回傳 cell 內容中的 *cell*。

PyObject ***PyCell_GET** (*PyObject* *cell)

回傳值：借用參照。 回傳 cell 物件 *cell* 的內容，但是不檢查 *cell* 是否非 NULL 並且為一個 cell 物件。

int **PyCell_Set** (*PyObject* *cell, *PyObject* *value)

將 cell 物件 *cell* 的內容設為 *value*。這將釋放任何對 cell 物件當前內容的參照。*value* 可以為 NULL。*cell* 必須不為 NULL；如果它不是一個 cell 物件則將回傳 -1。如果設定成功則將回傳 0。

void **PyCell_SET** (*PyObject* *cell, *PyObject* *value)

將 cell 物件 *cell* 的值設為 *value*。不會調整參照計數，並且不會進行任何安全檢查；*cell* 必須為非 NULL 並且為一個 cell 物件。

8.5.5 程式碼物件

Code objects are a low-level detail of the CPython implementation. Each one represents a chunk of executable code that hasn't yet been bound into a function.

type **PyCodeObject**

The C structure of the objects used to describe code objects. The fields of this type are subject to change at any time.

PyTypeObject **PyCode_Type**

This is an instance of *PyTypeObject* representing the Python code object.

int **PyCode_Check** (*PyObject* *co)

Return true if *co* is a code object. This function always succeeds.

int **PyCode_GetNumFree** (*PyCodeObject* *co)

Return the number of free variables in *co*.

PyCodeObject ***PyCode_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)

回傳值：新的參照。 Return a new code object. If you need a dummy code object to create a frame, use *PyCode_NewEmpty()* instead. Calling *PyCode_New()* directly will bind you to a precise Python version since the definition of the bytecode changes often. The many arguments of this function are inter-dependent in complex ways, meaning that subtle changes to values are likely to result in incorrect execution or VM crashes. Use this function only with extreme care.

在 3.11 版的變更: Added `qualname` and `exceptiontable` parameters.

PyCodeObject ***PyCode_NewWithPosOnlyArgs*** (int argcount, int posonlyargcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)

回傳值：新的參照。 Similar to `PyCode_New()`, but with an extra "posonlyargcount" for positional-only arguments. The same caveats that apply to `PyCode_New` also apply to this function.

在 3.8 版新加入.

在 3.11 版的變更: Added `qualname` and `exceptiontable` parameters.

PyCodeObject ***PyCode_NewEmpty*** (const char *filename, const char *funcname, int firstlineno)

回傳值：新的參照。 Return a new empty code object with the specified filename, function name, and first line number. The resulting code object will raise an `Exception` if executed.

int **PyCode_Addr2Line** (*PyCodeObject* *co, int byte_offset)

Return the line number of the instruction that occurs on or before `byte_offset` and ends after it. If you just need the line number of a frame, use `PyFrame_GetLineNumber()` instead.

For efficiently iterating over the line numbers in a code object, use the API described in PEP 626.

int **PyCode_Addr2Location** (*PyObject* *co, int byte_offset, int *start_line, int *start_column, int *end_line, int *end_column)

Sets the passed `int` pointers to the source code line and column numbers for the instruction at `byte_offset`. Sets the value to `0` when information is not available for any particular element.

Returns `1` if the function succeeds and 0 otherwise.

在 3.11 版新加入.

PyObject ***PyCode_GetCode*** (*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_code')`. Returns a strong reference to a `PyBytesObject` representing the bytecode in a code object. On error, `NULL` is returned and an exception is raised.

This `PyBytesObject` may be created on-demand by the interpreter and does not necessarily represent the bytecode actually executed by CPython. The primary use case for this function is debuggers and profilers.

在 3.11 版新加入.

PyObject ***PyCode_GetVarnames*** (*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_varnames')`. Returns a new reference to a `PyTupleObject` containing the names of the local variables. On error, `NULL` is returned and an exception is raised.

在 3.11 版新加入.

PyObject ***PyCode_GetCellvars*** (*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_cellvars')`. Returns a new reference to a `PyTupleObject` containing the names of the local variables that are referenced by nested functions. On error, `NULL` is returned and an exception is raised.

在 3.11 版新加入.

PyObject ***PyCode_GetFreevars*** (*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_freevars')`. Returns a new reference to a `PyTupleObject` containing the names of the free variables. On error, `NULL` is returned and an exception is raised.

在 3.11 版新加入.

8.6 其他物件

8.6.1 檔案物件 (File Objects)

這些 API 是用於內建檔案物件的 Python 2 C API 的最小模擬 (minimal emulation)，它過去依賴於 C 標準函式庫對於緩衝 I/O (FILE*) 的支援。在 Python 3 中，檔案和串流使用新的 io 模組，它在操作系統的低階無緩衝 I/O 上定義了多個層級。下面描述的函式是這些新 API 的便捷 C 包裝器，主要用於直譯器中的內部錯誤報告；建議第三方程式碼改為存取 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。*Part of the* Stable ABI. Create a Python file object from the file descriptor of an already opened file *fd*. The arguments *name*, *encoding*, *errors* and *newline* can be NULL to use the defaults; *buffering* can be *-1* to use the default. *name* is ignored and kept for backward compatibility. Return NULL on failure. For a more comprehensive description of the arguments, please refer to the io.open() function documentation.

> **警告：** 由於 Python 串流有自己的緩衝層，將它們與操作系統層級檔案描述器混合使用會產生各種問題（例如資料的排序不符合預期）。

在 3.2 版的變更: 忽略 *name* 屬性。

int **PyObject_AsFileDescriptor**(*PyObject* *p)

Part of the Stable ABI. 回傳與 *p* 關聯的檔案描述器作為 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，它作為檔案描述器值回傳。設定例外並在失敗時回傳 -1。

PyObject ***PyFile_GetLine**(*PyObject* *p, int n)

回傳值：新的參照。*Part of the* Stable ABI. Equivalent to p.readline([n]), this function reads one line from the object *p*. *p* may be a file object or any object with a readline() method. If *n* is 0, exactly one line is read, regardless of the length of the line. If *n* is greater than 0, no more than *n* bytes will be read from the file; a partial line can be returned. In both cases, an empty string is returned if the end of the file is reached immediately. If *n* is less than 0, however, one line is read regardless of length, but EOFError is raised if the end of the file is reached immediately.

int **PyFile_SetOpenCodeHook**(*Py_OpenCodeHookFunction* handler)

覆蓋 io.open_code() 的正常行為以透過提供的處理程式 (handler) 傳遞其參數。

The *handler* is a function of type:

typedef *PyObject* *(***Py_OpenCodeHookFunction**)(*PyObject**, void*)

Equivalent of *PyObject* *(*)(*PyObject* *path, void *userData), where *path* is guaranteed to be *PyUnicodeObject*.

userData 指標被傳遞到掛鉤函式 (hook function) 中。由於可能會從不同的執行環境 (runtime) 呼叫掛鉤函式，因此該指標不應直接指向 Python 狀態。

由於此掛鉤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。

Once a hook has been set, it cannot be removed or replaced, and later calls to *PyFile_SetOpenCodeHook()* will fail. On failure, the function returns -1 and sets an exception if the interpreter has been initialized.

在*Py_Initialize()* 之前呼叫此函式是安全的。

不帶引數地引發一個稽核事件 (auditing event) setopencodehook。

在 3.8 版新加入.

int **PyFile_WriteObject** (*PyObject* *obj, *PyObject* *p, int flags)

Part of the Stable ABI. Write object *obj* to file object *p*. The only supported flag for *flags* is *Py_PRINT_RAW*; if given, the `str()` of the object is written instead of the `repr()`. Return `0` on success or `-1` on failure; the appropriate exception will be set.

int **PyFile_WriteString** (const char *s, *PyObject* *p)

Part of the Stable ABI. 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，並會設定合適的例外狀態。

8.6.2 模組物件模組

PyTypeObject **PyModule_Type**

Part of the Stable ABI. This instance of *PyTypeObject* represents the Python module type. This is exposed to Python programs as `types.ModuleType`.

int **PyModule_Check** (*PyObject* *p)

Return true if *p* is a module object, or a subtype of a module object. This function always succeeds.

int **PyModule_CheckExact** (*PyObject* *p)

Return true if *p* is a module object, but not a subtype of *PyModule_Type*. This function always succeeds.

PyObject ***PyModule_NewObject** (*PyObject* *name)

回傳值：新的參照。*Part of the* Stable ABI *since version 3.7.* Return a new module object with the `__name__` attribute set to *name*. The module's `__name__`, `__doc__`, `__package__`, and `__loader__` attributes are filled in (all but `__name__` are set to `None`); the caller is responsible for providing a `__file__` attribute.

在 3.3 版新加入.

在 3.4 版的變更: `__package__` 和 `__loader__` 被設為 `None`。

PyObject ***PyModule_New** (const char *name)

回傳值：新的參照。 *Part of the* Stable ABI. Similar to *PyModule_NewObject()*, but the name is a UTF-8 encoded string instead of a Unicode object.

PyObject ***PyModule_GetDict** (*PyObject* *module)

回傳值：借用參照。*Part of the* Stable ABI. Return the dictionary object that implements *module*'s namespace; this object is the same as the `__dict__` attribute of the module object. If *module* is not a module object (or a subtype of a module object), `SystemError` is raised and `NULL` is returned.

It is recommended extensions use other `PyModule_*` and `PyObject_*` functions rather than directly manipulate a module's `__dict__`.

PyObject ***PyModule_GetNameObject** (*PyObject* *module)

回傳值：新的參照。 *Part of the* Stable ABI *since version 3.7.* Return *module*'s `__name__` value. If the module does not provide one, or if it is not a string, `SystemError` is raised and `NULL` is returned.

在 3.3 版新加入.

const char ***PyModule_GetName** (*PyObject* *module)

Part of the Stable ABI. Similar to *PyModule_GetNameObject()* but return the name encoded to `'utf-8'`.

void ***PyModule_GetState** (*PyObject* *module)

Part of the Stable ABI. Return the "state" of the module, that is, a pointer to the block of memory allocated at module creation time, or `NULL`. See *PyModuleDef.m_size*.

PyModuleDef ***PyModule_GetDef** (*PyObject* *module)

Part of the Stable ABI. Return a pointer to the *PyModuleDef* struct from which the module was created, or `NULL` if the module wasn't created from a definition.

PyObject ***PyModule_GetFilenameObject** (*PyObject* *module)

回傳值：新的參照。*Part of the* Stable ABI. Return the name of the file from which *module* was loaded using *module*'s `__file__` attribute. If this is not defined, or if it is not a unicode string, raise `SystemError` and return `NULL`; otherwise return a reference to a Unicode object.

在 3.2 版新加入.

const char ***PyModule_GetFilename** (*PyObject* *module)

Part of the Stable ABI. Similar to *PyModule_GetFilenameObject()* but return the filename encoded to 'utf-8'.

在 3.2 版之後被F用: *PyModule_GetFilename()* raises `UnicodeEncodeError` on unencodable filenames, use *PyModule_GetFilenameObject()* instead.

Initializing C modules

Modules objects are usually created from extension modules (shared libraries which export an initialization function), or compiled-in modules (where the initialization function is added using *PyImport_AppendInittab()*). See building or extending-with-embedding for details.

The initialization function can either pass a module definition instance to *PyModule_Create()*, and return the resulting module object, or request "multi-phase initialization" by returning the definition struct itself.

type **PyModuleDef**

Part of the Stable ABI *(including all members).* The module definition struct, which holds all information needed to create a module object. There is usually only one statically initialized variable of this type for each module.

PyModuleDef_Base **m_base**

Always initialize this member to `PyModuleDef_HEAD_INIT`.

const char ***m_name**

Name for the new module.

const char ***m_doc**

Docstring for the module; usually a docstring variable created with *PyDoc_STRVAR* is used.

Py_ssize_t **m_size**

Module state may be kept in a per-module memory area that can be retrieved with *PyModule_GetState()*, rather than in static globals. This makes modules safe for use in multiple sub-interpreters.

This memory area is allocated based on *m_size* on module creation, and freed when the module object is deallocated, after the *m_free* function has been called, if present.

Setting `m_size` to `-1` means that the module does not support sub-interpreters, because it has global state.

Setting it to a non-negative value means that the module can be re-initialized and specifies the additional amount of memory it requires for its state. Non-negative `m_size` is required for multi-phase initialization.

更多詳情請見 **PEP 3121**。

PyMethodDef ***m_methods**

A pointer to a table of module-level functions, described by *PyMethodDef* values. Can be `NULL` if no functions are present.

PyModuleDef_Slot ***m_slots**

An array of slot definitions for multi-phase initialization, terminated by a `{0, NULL}` entry. When using single-phase initialization, *m_slots* must be `NULL`.

在 3.5 版的變更: Prior to version 3.5, this member was always set to `NULL`, and was defined as:

inquiry **m_reload**

traverseproc **m_traverse**

A traversal function to call during GC traversal of the module object, or NULL if not needed.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (*Py_mod_exec* function). More precisely, this function is not called if *m_size* is greater than 0 and the module state (as returned by *PyModule_GetState()*) is NULL.

在 3.9 版的變更: No longer called before the module state is allocated.

inquiry **m_clear**

A clear function to call during GC clearing of the module object, or NULL if not needed.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (*Py_mod_exec* function). More precisely, this function is not called if *m_size* is greater than 0 and the module state (as returned by *PyModule_GetState()*) is NULL.

Like *PyTypeObject.tp_clear*, this function is not *always* called before a module is deallocated. For example, when reference counting is enough to determine that an object is no longer used, the cyclic garbage collector is not involved and *m_free* is called directly.

在 3.9 版的變更: No longer called before the module state is allocated.

freefunc **m_free**

A function to call during deallocation of the module object, or NULL if not needed.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (*Py_mod_exec* function). More precisely, this function is not called if *m_size* is greater than 0 and the module state (as returned by *PyModule_GetState()*) is NULL.

在 3.9 版的變更: No longer called before the module state is allocated.

Single-phase initialization

The module initialization function may create and return the module object directly. This is referred to as "single-phase initialization", and uses one of the following two module creation functions:

PyObject ***PyModule_Create**(*PyModuleDef* *def)

回傳值：新的參照。 Create a new module object, given the definition in *def*. This behaves like *PyModule_Create2()* with *module_api_version* set to PYTHON_API_VERSION.

PyObject ***PyModule_Create2**(*PyModuleDef* *def, int module_api_version)

回傳值：新的參照。*Part of the* Stable ABI. Create a new module object, given the definition in *def*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a RuntimeWarning is emitted.

備F: Most uses of this function should be using *PyModule_Create()* instead; only use this if you are sure you need it.

Before it is returned from in the initialization function, the resulting module object is typically populated using functions like *PyModule_AddObjectRef()*.

Multi-phase initialization

An alternate way to specify extensions is to request "multi-phase initialization". Extension modules created this way behave more like Python modules: the initialization is split between the *creation phase*, when the module object is created, and the *execution phase*, when it is populated. The distinction is similar to the `__new__()` and `__init__()` methods of classes.

Unlike modules created using single-phase initialization, these modules are not singletons: if the *sys.modules* entry is removed and the module is re-imported, a new module object is created, and the old module is subject to normal garbage collection -- as with Python modules. By default, multiple modules created from the same definition should be independent: changes to one should not affect the others. This means that all state should be specific to the module object (using e.g. using `PyModule_GetState()`), or its contents (such as the module's `__dict__` or individual classes created with `PyType_FromSpec()`).

All modules created using multi-phase initialization are expected to support *sub-interpreters*. Making sure multiple modules are independent is typically enough to achieve this.

To request multi-phase initialization, the initialization function (PyInit_modulename) returns a `PyModuleDef` instance with non-empty `m_slots`. Before it is returned, the `PyModuleDef` instance must be initialized with the following function:

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

回傳值：借用參照。 *Part of the* Stable ABI *since version 3.5.* Ensures a module definition is a properly initialized Python object that correctly reports its type and reference count.

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

在 3.5 版新加入.

The *m_slots* member of the module definition must point to an array of `PyModuleDef_Slot` structures:

type **PyModuleDef_Slot**

int **slot**

A slot ID, chosen from the available values explained below.

void ***value**

Value of the slot, whose meaning depends on the slot ID.

在 3.5 版新加入.

The *m_slots* array must be terminated by a slot with id 0.

The available slot types are:

Py_mod_create

Specifies a function that is called to create the module object itself. The *value* pointer of this slot must point to a function of the signature:

PyObject ***create_module**(*PyObject* *spec, *PyModuleDef* *def)

The function receives a `ModuleSpec` instance, as defined in **PEP 451**, and the module definition. It should return a new module object, or set an error and return `NULL`.

This function should be kept minimal. In particular, it should not call arbitrary Python code, as trying to import the same module again may result in an infinite loop.

Multiple `Py_mod_create` slots may not be specified in one module definition.

If `Py_mod_create` is not specified, the import machinery will create a normal module object using `PyModule_New()`. The name is taken from *spec*, not the definition, to allow extension modules to dynamically adjust to their place in the module hierarchy and be imported under different names through symlinks, all while sharing a single module definition.

There is no requirement for the returned object to be an instance of `PyModule_Type`. Any type can be used, as long as it supports setting and getting import-related attributes. However, only `PyModule_Type` instances may be returned if the PyModuleDef has non-NULL m_traverse, m_clear, m_free; non-zero m_size; or slots other than Py_mod_create.

Py_mod_exec

Specifies a function that is called to *execute* the module. This is equivalent to executing the code of a Python module: typically, this function adds classes and constants to the module. The signature of the function is:

int **exec_module** (*PyObject* *module)

If multiple Py_mod_exec slots are specified, they are processed in the order they appear in the *m_slots* array.

See **PEP 489** for more details on multi-phase initialization.

Low-level module creation functions

The following functions are called under the hood when using multi-phase initialization. They can be used directly, for example when creating module objects dynamically. Note that both PyModule_FromDefAndSpec and PyModule_ExecDef must be called to fully initialize a module.

PyObject ***PyModule_FromDefAndSpec** (*PyModuleDef* *def, *PyObject* *spec)

回傳值：新的參照。 Create a new module object, given the definition in *def* and the ModuleSpec *spec*. This behaves like *PyModule_FromDefAndSpec2()* with *module_api_version* set to PYTHON_API_VERSION.

在 3.5 版新加入.

PyObject ***PyModule_FromDefAndSpec2** (*PyModuleDef* *def, *PyObject* *spec, int module_api_version)

回傳值：新的參照。*Part of the* Stable ABI *since version 3.7.* Create a new module object, given the definition in *def* and the ModuleSpec *spec*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a RuntimeWarning is emitted.

備註： Most uses of this function should be using *PyModule_FromDefAndSpec()* instead; only use this if you are sure you need it.

在 3.5 版新加入.

int **PyModule_ExecDef** (*PyObject* *module, *PyModuleDef* *def)

Part of the Stable ABI *since version 3.7.* Process any execution slots (*Py_mod_exec*) given in *def*.

在 3.5 版新加入.

int **PyModule_SetDocString** (*PyObject* *module, const char *docstring)

Part of the Stable ABI *since version 3.7.* Set the docstring for *module* to *docstring*. This function is called automatically when creating a module from PyModuleDef, using either PyModule_Create or PyModule_FromDefAndSpec.

在 3.5 版新加入.

int **PyModule_AddFunctions** (*PyObject* *module, *PyMethodDef* *functions)

Part of the Stable ABI *since version 3.7.* Add the functions from the NULL terminated *functions* array to *module*. Refer to the *PyMethodDef* documentation for details on individual entries (due to the lack of a shared module namespace, module level "functions" implemented in C typically receive the module as their first parameter, making them similar to instance methods on Python classes). This function is called automatically when creating a module from PyModuleDef, using either PyModule_Create or PyModule_FromDefAndSpec.

在 3.5 版新加入.

Support functions

The module initialization function (if using single phase initialization) or a function called from a module execution slot (if using multi-phase initialization), can use the following functions to help initialize the module state:

int **PyModule_AddObjectRef** (*PyObject* *module, const char *name, *PyObject* *value)

Part of the Stable ABI *since version 3.10.* Add an object to *module* as *name*. This is a convenience function which can be used from the module's initialization function.

On success, return `0`. On error, raise an exception and return `-1`.

Return `NULL` if *value* is `NULL`. It must be called with an exception raised in this case.

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;
}
```

The example can also be written without checking explicitly if *obj* is `NULL`:

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_XDECREF(obj);
    return res;
}
```

Note that `Py_XDECREF()` should be used instead of `Py_DECREF()` in this case, since *obj* can be `NULL`.

在 3.10 版新加入.

int **PyModule_AddObject** (*PyObject* *module, const char *name, *PyObject* *value)

Part of the Stable ABI. Similar to `PyModule_AddObjectRef()`, but steals a reference to *value* on success (if it returns `0`).

The new `PyModule_AddObjectRef()` function is recommended, since it is easy to introduce reference leaks by misusing the `PyModule_AddObject()` function.

備註: Unlike other functions that steal references, `PyModule_AddObject()` only releases the reference to *value* **on success**.

This means that its return value must be checked, and calling code must `Py_DECREF()` *value* manually on error.

用法範例:

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    if (obj == NULL) {
```

(繼續下一頁)

(繼續上一頁)

```
        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;
}
```

The example can also be written without checking explicitly if *obj* is `NULL`:

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    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;
}
```

Note that `Py_XDECREF()` should be used instead of `Py_DECREF()` in this case, since *obj* can be `NULL`.

int **PyModule_AddIntConstant**(*PyObject* *module, const char *name, long value)

Part of the Stable ABI. Add an integer constant to *module* as *name*. This convenience function can be used from the module's initialization function. Return `-1` on error, `0` on success.

int **PyModule_AddStringConstant**(*PyObject* *module, const char *name, const char *value)

Part of the Stable ABI. Add a string constant to *module* as *name*. This convenience function can be used from the module's initialization function. The string *value* must be `NULL`-terminated. Return `-1` on error, `0` on success.

PyModule_AddIntMacro(module, macro)

Add an int constant to *module*. The name and the value are taken from *macro*. For example `PyModule_AddIntMacro(module, AF_INET)` adds the int constant *AF_INET* with the value of *AF_INET* to *module*. Return `-1` on error, `0` on success.

PyModule_AddStringMacro(module, macro)

Add a string constant to *module*.

int **PyModule_AddType**(*PyObject* *module, *PyTypeObject* *type)

Part of the Stable ABI *since version 3.10.* Add a type object to *module*. The type object is finalized by calling internally `PyType_Ready()`. The name of the type object is taken from the last component of `tp_name` after dot. Return `-1` on error, `0` on success.

在 3.9 版新加入.

Module lookup

Single-phase initialization creates singleton modules that can be looked up in the context of the current interpreter. This allows the module object to be retrieved later with only a reference to the module definition.

These functions will not work on modules created using multi-phase initialization, since multiple such modules can be created from a single definition.

PyObject ***PyState_FindModule**(*PyModuleDef* *def)

回傳值：借用參照。 *Part of the* Stable ABI. Returns the module object that was created from *def* for the current interpreter. This method requires that the module object has been attached to the interpreter state with *PyState_AddModule()* beforehand. In case the corresponding module object is not found or has not been attached to the interpreter state yet, it returns `NULL`.

int **PyState_AddModule**(*PyObject* *module, *PyModuleDef* *def)

Part of the Stable ABI *since version 3.3.* Attaches the module object passed to the function to the interpreter state. This allows the module object to be accessible via *PyState_FindModule()*.

Only effective on modules created using single-phase initialization.

Python calls `PyState_AddModule` automatically after importing a module, so it is unnecessary (but harmless) to call it from module initialization code. An explicit call is needed only if the module's own init code subsequently calls `PyState_FindModule`. The function is mainly intended for implementing alternative import mechanisms (either by calling it directly, or by referring to its implementation for details of the required state updates).

The caller must hold the GIL.

Return 0 on success or -1 on failure.

在 3.3 版新加入.

int **PyState_RemoveModule**(*PyModuleDef* *def)

Part of the Stable ABI *since version 3.3.* Removes the module object created from *def* from the interpreter state. Return 0 on success or -1 on failure.

The caller must hold the GIL.

在 3.3 版新加入.

8.6.3 F代器（Iterator）物件

Python provides two general-purpose iterator objects. The first, a sequence iterator, works with an arbitrary sequence supporting the `__getitem__()` method. The second works with a callable object and a sentinel value, calling the callable for each item in the sequence, and ending the iteration when the sentinel value is returned.

PyTypeObject **PySeqIter_Type**

Part of the Stable ABI. Type object for iterator objects returned by *PySeqIter_New()* and the one-argument form of the `iter()` built-in function for built-in sequence types.

int **PySeqIter_Check**(*PyObject* *op)

Return true if the type of *op* is *PySeqIter_Type*. This function always succeeds.

PyObject ***PySeqIter_New**(*PyObject* *seq)

回傳值：新的參照。 *Part of the* Stable ABI. Return an iterator that works with a general sequence object, *seq*. The iteration ends when the sequence raises `IndexError` for the subscripting operation.

PyTypeObject **PyCallIter_Type**

Part of the Stable ABI. Type object for iterator objects returned by *PyCallIter_New()* and the two-argument form of the `iter()` built-in function.

int **PyCallIter_Check**(*PyObject* *op)

Return true if the type of *op* is *PyCallIter_Type*. This function always succeeds.

PyObject ***PyCallIter_New**(*PyObject* *callable, *PyObject* *sentinel)

回傳值：新的參照。 *Part of the* Stable ABI. Return a new iterator. The first parameter, *callable*, can be any Python callable object that can be called with no parameters; each call to it should return the next item in the iteration. When *callable* returns a value equal to *sentinel*, the iteration will be terminated.

8.6.4 Descriptor（描述器）物件

"Descriptor" 是描述物件某些屬性的物件，它們存在於型別物件的 dictionary（字典）中。

PyTypeObject **PyProperty_Type**

Part of the Stable ABI. 內建 descriptor 型別的型別物件。

PyObject ***PyDescr_NewGetSet**(*PyTypeObject* *type, struct *PyGetSetDef* *getset)

回傳值：新的參照。 *Part of the* Stable ABI.

PyObject ***PyDescr_NewMember**(*PyTypeObject* *type, struct *PyMemberDef* *meth)

回傳值：新的參照。 *Part of the* Stable ABI.

PyObject ***PyDescr_NewMethod**(*PyTypeObject* *type, struct *PyMethodDef* *meth)

回傳值：新的參照。 *Part of the* Stable ABI.

PyObject ***PyDescr_NewWrapper**(*PyTypeObject* *type, struct wrapperbase *wrapper, void *wrapped)

回傳值：新的參照。

PyObject ***PyDescr_NewClassMethod**(*PyTypeObject* *type, *PyMethodDef* *method)

回傳值：新的參照。 *Part of the* Stable ABI.

int **PyDescr_IsData**(*PyObject* *descr)

如果 descriptor 物件 *descr* 描述的是一個資料屬性則回傳非零值，或者如果它描述的是一個方法則返回 `0`。*descr* 必須為一個 descriptor 物件；沒有錯誤檢查。

PyObject ***PyWrapper_New**(*PyObject**, *PyObject**)

回傳值：新的參照。 *Part of the* Stable ABI.

8.6.5 切片物件

PyTypeObject **PySlice_Type**

Part of the Stable ABI. The type object for slice objects. This is the same as `slice` in the Python layer.

int **PySlice_Check**(*PyObject* *ob)

Return true if *ob* is a slice object; *ob* must not be `NULL`. This function always succeeds.

PyObject ***PySlice_New**(*PyObject* *start, *PyObject* *stop, *PyObject* *step)

回傳值：新的參照。 *Part of the* Stable ABI. Return a new slice object with the given values. The *start*, *stop*, and *step* parameters are used as the values of the slice object attributes of the same names. Any of the values may be `NULL`, in which case the `None` will be used for the corresponding attribute. Return `NULL` if the new object could not be allocated.

int **PySlice_GetIndices**(*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)

Part of the Stable ABI. Retrieve the start, stop and step indices from the slice object *slice*, assuming a sequence of length *length*. Treats indices greater than *length* as errors.

Returns `0` on success and `-1` on error with no exception set (unless one of the indices was not `None` and failed to be converted to an integer, in which case `-1` is returned with an exception set).

You probably do not want to use this function.

在 3.2 版的變更: The parameter type for the *slice* parameter was `PySliceObject*` before.

int **PySlice_GetIndicesEx**(*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step, *Py_ssize_t* *slicelength)

Part of the Stable ABI. Usable replacement for `PySlice_GetIndices()`. Retrieve the start, stop, and step indices from the slice object *slice* assuming a sequence of length *length*, and store the length of the slice in *slicelength*. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Returns `0` on success and `-1` on error with exception set.

備註: This function is considered not safe for resizable sequences. Its invocation should be replaced by a combination of `PySlice_Unpack()` and `PySlice_AdjustIndices()` where

```
if (PySlice_GetIndicesEx(slice, length, &start, &stop, &step, &slicelength) <
↪0) {
    // return error
}
```

is replaced by

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {
    // return error
}
slicelength = PySlice_AdjustIndices(length, &start, &stop, step);
```

在 3.2 版的變更: The parameter type for the *slice* parameter was `PySliceObject*` before.

在 3.6.1 版的變更: If `Py_LIMITED_API` is not set or set to the value between `0x03050400` and `0x03060000` (not including) or `0x03060100` or higher `PySlice_GetIndicesEx()` is implemented as a macro using `PySlice_Unpack()` and `PySlice_AdjustIndices()`. Arguments *start*, *stop* and *step* are evaluated more than once.

在 3.6.1 版之後被棄用: If `Py_LIMITED_API` is set to the value less than `0x03050400` or between `0x03060000` and `0x03060100` (not including) `PySlice_GetIndicesEx()` is a deprecated function.

int **PySlice_Unpack**(*PyObject* *slice, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)

Part of the Stable ABI *since version 3.7.* Extract the start, stop and step data members from a slice object as C integers. Silently reduce values larger than `PY_SSIZE_T_MAX` to `PY_SSIZE_T_MAX`, silently boost the start and stop values less than `PY_SSIZE_T_MIN` to `PY_SSIZE_T_MIN`, and silently boost the step values less than `-PY_SSIZE_T_MAX` to `-PY_SSIZE_T_MAX`.

Return `-1` on error, `0` on success.

在 3.6.1 版新加入.

Py_ssize_t **PySlice_AdjustIndices**(*Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* step)

Part of the Stable ABI *since version 3.7.* Adjust start/end slice indices assuming a sequence of the specified length. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Return the length of the slice. Always successful. Doesn't call Python code.

在 3.6.1 版新加入.

Ellipsis Object

PyObject ***Py_Ellipsis***

The Python `Ellipsis` object. This object has no methods. It needs to be treated just like any other object with respect to reference counts. Like `Py_None` it is a singleton object.

8.6.6 MemoryView 物件

A `memoryview` object exposes the C level *buffer interface* as a Python object which can then be passed around like any other object.

PyObject ***PyMemoryView_FromObject**(PyObject *obj)*

回傳值：新的參照。*Part of the* Stable ABI. Create a memoryview object from an object that provides the buffer interface. If *obj* supports writable buffer exports, the memoryview object will be read/write, otherwise it may be either read-only or read/write at the discretion of the exporter.

PyBUF_READ

Flag to request a readonly buffer.

PyBUF_WRITE

Flag to request a writable buffer.

PyObject ***PyMemoryView_FromMemory**(char *mem, Py_ssize_t size, int flags)*

回傳值：新的參照。*Part of the* Stable ABI *since version 3.7.* Create a memoryview object using *mem* as the underlying buffer. *flags* can be one of `PyBUF_READ` or `PyBUF_WRITE`.

在 3.3 版新加入.

PyObject ***PyMemoryView_FromBuffer**(const Py_buffer *view)*

回傳值：新的參照。*Part of the* Stable ABI *since version 3.11.* Create a memoryview object wrapping the given buffer structure *view*. For simple byte buffers, `PyMemoryView_FromMemory()` is the preferred function.

PyObject ***PyMemoryView_GetContiguous**(PyObject *obj, int buffertype, char order)*

回傳值：新的參照。*Part of the* Stable ABI. Create a memoryview object to a *contiguous* chunk of memory (in either 'C' or 'F'ortran *order*) from an object that defines the buffer interface. If memory is contiguous, the memoryview object points to the original memory. Otherwise, a copy is made and the memoryview points to a new bytes object.

buffertype can be one of `PyBUF_READ` or `PyBUF_WRITE`.

int **PyMemoryView_Check**(*PyObject* *obj)

Return true if the object *obj* is a memoryview object. It is not currently allowed to create subclasses of `memoryview`. This function always succeeds.

Py_buffer ***PyMemoryView_GET_BUFFER**(PyObject *mview)*

Return a pointer to the memoryview's private copy of the exporter's buffer. *mview* **must** be a memoryview instance; this macro doesn't check its type, you must do it yourself or you will risk crashes.

PyObject ***PyMemoryView_GET_BASE**(PyObject *mview)*

Return either a pointer to the exporting object that the memoryview is based on or `NULL` if the memoryview has been created by one of the functions `PyMemoryView_FromMemory()` or `PyMemoryView_FromBuffer()`. *mview* **must** be a memoryview instance.

8.6.7 弱参照物件

Python supports *weak references* as first-class objects. There are two specific object types which directly implement weak references. The first is a simple reference object, and the second acts as a proxy for the original object as much as it can.

int **PyWeakref_Check** (*PyObject* *ob)

Return true if *ob* is either a reference or proxy object. This function always succeeds.

int **PyWeakref_CheckRef** (*PyObject* *ob)

Return true if *ob* is a reference object. This function always succeeds.

int **PyWeakref_CheckProxy** (*PyObject* *ob)

Return true if *ob* is a proxy object. This function always succeeds.

PyObject ***PyWeakref_NewRef** (*PyObject* *ob, *PyObject* *callback)

回傳值：新的參照。 *Part of the* Stable ABI. Return a weak reference object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object; an existing reference object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be `None` or `NULL`. If *ob* is not a weakly referencable object, or if *callback* is not callable, `None`, or `NULL`, this will return `NULL` and raise `TypeError`.

PyObject ***PyWeakref_NewProxy** (*PyObject* *ob, *PyObject* *callback)

回傳值：新的參照。*Part of the* Stable ABI. Return a weak reference proxy object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object; an existing proxy object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be `None` or `NULL`. If *ob* is not a weakly referencable object, or if *callback* is not callable, `None`, or `NULL`, this will return `NULL` and raise `TypeError`.

PyObject ***PyWeakref_GetObject** (*PyObject* *ref)

回傳值：借用參照。 *Part of the* Stable ABI. Return the referenced object from a weak reference, *ref*. If the referent is no longer live, returns `Py_None`.

備F: This function returns a *borrowed reference* to the referenced object. This means that you should always call `Py_INCREF()` on the object except when it cannot be destroyed before the last usage of the borrowed reference.

PyObject ***PyWeakref_GET_OBJECT** (*PyObject* *ref)

回傳值：借用參照。 Similar to `PyWeakref_GetObject()`, but does no error checking.

void **PyObject_ClearWeakRefs** (*PyObject* *object)

Part of the Stable ABI. This function is called by the `tp_dealloc` handler to clear weak references.

This iterates through the weak references for *object* and calls callbacks for those references which have one. It returns when all callbacks have been attempted.

8.6.8 Capsules

Refer to using-capsules for more information on using these objects.

在 3.1 版新加入.

type **PyCapsule**

This subtype of `PyObject` represents an opaque value, useful for C extension modules who need to pass an opaque value (as a `void*` pointer) through Python code to other C code. It is often used to make a C function pointer defined in one module available to other modules, so the regular import mechanism can be used to access C APIs defined in dynamically loaded modules.

type **PyCapsule_Destructor**

Part of the Stable ABI. The type of a destructor callback for a capsule. Defined as:

```
typedef void (*PyCapsule_Destructor)(PyObject *);
```

See `PyCapsule_New()` for the semantics of PyCapsule_Destructor callbacks.

int **PyCapsule_CheckExact**(*PyObject* *p)

Return true if its argument is a `PyCapsule`. This function always succeeds.

PyObject ***PyCapsule_New**(void *pointer, const char *name, *PyCapsule_Destructor* destructor)

回傳值：新的參照。 *Part of the* Stable ABI. Create a `PyCapsule` encapsulating the *pointer*. The *pointer* argument may not be `NULL`.

On failure, set an exception and return `NULL`.

The *name* string may either be `NULL` or a pointer to a valid C string. If non-`NULL`, this string must outlive the capsule. (Though it is permitted to free it inside the *destructor*.)

If the *destructor* argument is not `NULL`, it will be called with the capsule as its argument when it is destroyed.

If this capsule will be stored as an attribute of a module, the *name* should be specified as `modulename.attributename`. This will enable other modules to import the capsule using `PyCapsule_Import()`.

void ***PyCapsule_GetPointer**(*PyObject* *capsule, const char *name)

Part of the Stable ABI. Retrieve the *pointer* stored in the capsule. On failure, set an exception and return `NULL`.

The *name* parameter must compare exactly to the name stored in the capsule. If the name stored in the capsule is `NULL`, the *name* passed in must also be `NULL`. Python uses the C function `strcmp()` to compare capsule names.

PyCapsule_Destructor **PyCapsule_GetDestructor**(*PyObject* *capsule)

Part of the Stable ABI. Return the current destructor stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` destructor. This makes a `NULL` return code somewhat ambiguous; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

void ***PyCapsule_GetContext**(*PyObject* *capsule)

Part of the Stable ABI. Return the current context stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` context. This makes a `NULL` return code somewhat ambiguous; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

const char ***PyCapsule_GetName**(*PyObject* *capsule)

Part of the Stable ABI. Return the current name stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` name. This makes a `NULL` return code somewhat ambiguous; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

void ***PyCapsule_Import**(const char *name, int no_block)

Part of the Stable ABI. Import a pointer to a C object from a capsule attribute in a module. The *name* parameter should specify the full name to the attribute, as in `module.attribute`. The *name* stored in the capsule must match this string exactly.

Return the capsule's internal *pointer* on success. On failure, set an exception and return `NULL`.

在 3.3 版的變更: *no_block* has no effect anymore.

int **PyCapsule_IsValid**(*PyObject* *capsule, const char *name)

Part of the Stable ABI. Determines whether or not *capsule* is a valid capsule. A valid capsule is non-`NULL`, passes `PyCapsule_CheckExact()`, has a non-`NULL` pointer stored in it, and its internal name matches the *name* parameter. (See `PyCapsule_GetPointer()` for information on how capsule names are compared.)

In other words, if `PyCapsule_IsValid()` returns a true value, calls to any of the accessors (any function starting with `PyCapsule_Get`) are guaranteed to succeed.

Return a nonzero value if the object is valid and matches the name passed in. Return `0` otherwise. This function will not fail.

int **PyCapsule_SetContext**(*PyObject* *capsule, void *context)

Part of the Stable ABI. Set the context pointer inside *capsule* to *context*.

Return `0` on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetDestructor**(*PyObject* *capsule, *PyCapsule_Destructor* destructor)

Part of the Stable ABI. Set the destructor inside *capsule* to *destructor*.

Return `0` on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetName**(*PyObject* *capsule, const char *name)

Part of the Stable ABI. Set the name inside *capsule* to *name*. If non-`NULL`, the name must outlive the capsule. If the previous *name* stored in the capsule was not `NULL`, no attempt is made to free it.

Return `0` on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetPointer**(*PyObject* *capsule, void *pointer)

Part of the Stable ABI. Set the void pointer inside *capsule* to *pointer*. The pointer may not be `NULL`.

Return `0` on success. Return nonzero and set an exception on failure.

8.6.9 Frame Objects

type **PyFrameObject**

Part of the Limited API *(as an opaque struct)*. The C structure of the objects used to describe frame objects.

There are no public members in this structure.

在 3.11 版的變更: The members of this structure were removed from the public C API. Refer to the What's New entry for details.

The `PyEval_GetFrame()` and `PyThreadState_GetFrame()` functions can be used to get a frame object.

See also *Reflection*.

PyTypeObject **PyFrame_Type**

The type of frame objects. It is the same object as `types.FrameType` in the Python layer.

在 3.11 版的變更: Previously, this type was only available after including `<frameobject.h>`.

int **PyFrame_Check**(*PyObject* *obj)

Return non-zero if *obj* is a frame object.

在 3.11 版的變更: Previously, this function was only available after including `<frameobject.h>`.

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

Get the *frame* next outer frame.

Return a *strong reference*, or `NULL` if *frame* has no outer frame.

在 3.9 版新加入.

PyObject ***PyFrame_GetBuiltins**(*PyFrameObject* *frame)

Get the *frame*'s `f_builtins` attribute.

Return a *strong reference*. The result cannot be `NULL`.

在 3.11 版新加入.

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

Part of the Stable ABI *since version 3.10.* Get the *frame* code.

Return a *strong reference*.

The result (frame code) cannot be `NULL`.

在 3.9 版新加入.

PyObject ***PyFrame_GetGenerator** (*PyFrameObject* *frame)

Get the generator, coroutine, or async generator that owns this frame, or `NULL` if this frame is not owned by a generator. Does not raise an exception, even if the return value is `NULL`.

Return a *strong reference*, or `NULL`.

在 3.11 版新加入.

PyObject ***PyFrame_GetGlobals** (*PyFrameObject* *frame)

Get the *frame*'s `f_globals` attribute.

Return a *strong reference*. The result cannot be `NULL`.

在 3.11 版新加入.

int **PyFrame_GetLasti** (*PyFrameObject* *frame)

Get the *frame*'s `f_lasti` attribute.

Returns -1 if `frame.f_lasti` is `None`.

在 3.11 版新加入.

PyObject ***PyFrame_GetLocals** (*PyFrameObject* *frame)

Get the *frame*'s `f_locals` attribute (`dict`).

Return a *strong reference*.

在 3.11 版新加入.

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

Part of the Stable ABI *since version 3.10.* Return the line number that *frame* is currently executing.

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

在 3.5 版新加入.

Coroutine 物件是那些以 `async` 關鍵字來宣告的函式所回傳的物件。

type **PyCoroObject**

用於 coroutine 物件的 C 結構。

PyTypeObject **PyCoro_Type**

與 coroutine 物件對應的型別物件。

int **PyCoro_CheckExact** (*PyObject* *ob)

如果 *ob* 的型別是 `PyCoro_Type` 則回傳真值；*ob* 必須不為 `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* 物件來建立並回傳一個新的 coroutine 物件，其中 `__name__` 和 `__qualname__` 被設為 *name* 和 *qualname*。此函式會取得一個對 *frame* 的參照 (reference)。*frame* 引數必須不為 `NULL`。

8.6.12 Context Variables Objects

在 3.7 版新加入.

在 3.7.1 版的變更:

備註： In Python 3.7.1 the signatures of all context variables C APIs were **changed** to use `PyObject` pointers instead of `PyContext`, `PyContextVar`, and `PyContextToken`, e.g.:

```
// in 3.7.0:
PyContext *PyContext_New(void);

// in 3.7.1+:
PyObject *PyContext_New(void);
```

更多細節請見 bpo-34762。

This section details the public C API for the `contextvars` module.

type **PyContext**

The C structure used to represent a `contextvars.Context` object.

type **PyContextVar**

The C structure used to represent a `contextvars.ContextVar` object.

type **PyContextToken**

The C structure used to represent a `contextvars.Token` object.

PyTypeObject **PyContext_Type**

The type object representing the *context* type.

PyTypeObject **PyContextVar_Type**

The type object representing the *context variable* type.

PyTypeObject **PyContextToken_Type**

The type object representing the *context variable token* type.

Type-check macros:

int **PyContext_CheckExact** (*PyObject* *o)

Return true if *o* is of type `PyContext_Type`. *o* must not be `NULL`. This function always succeeds.

int **PyContextVar_CheckExact** (*PyObject* *o)

Return true if *o* is of type *PyContextVar_Type*. *o* must not be NULL. This function always succeeds.

int **PyContextToken_CheckExact** (*PyObject* *o)

Return true if *o* is of type *PyContextToken_Type*. *o* must not be NULL. This function always succeeds.

Context object management functions:

PyObject ***PyContext_New** (void)

回傳值：新的參照。 Create a new empty context object. Returns NULL if an error has occurred.

PyObject ***PyContext_Copy** (*PyObject* *ctx)

回傳值：新的參照。 Create a shallow copy of the passed *ctx* context object. Returns NULL if an error has occurred.

PyObject ***PyContext_CopyCurrent** (void)

回傳值：新的參照。 Create a shallow copy of the current thread context. Returns NULL if an error has occurred.

int **PyContext_Enter** (*PyObject* *ctx)

Set *ctx* as the current context for the current thread. Returns 0 on success, and -1 on error.

int **PyContext_Exit** (*PyObject* *ctx)

Deactivate the *ctx* context and restore the previous context as the current context for the current thread. Returns 0 on success, and -1 on error.

Context variable functions:

PyObject ***PyContextVar_New** (const char *name, *PyObject* *def)

回傳值：新的參照。Create a new ContextVar object. The *name* parameter is used for introspection and debug purposes. The *def* parameter specifies a default value for the context variable, or NULL for no default. If an error has occurred, this function returns NULL.

int **PyContextVar_Get** (*PyObject* *var, *PyObject* *default_value, *PyObject* **value)

Get the value of a context variable. Returns -1 if an error has occurred during lookup, and 0 if no error occurred, whether or not a value was found.

If the context variable was found, *value* will be a pointer to it. If the context variable was *not* found, *value* will point to:

- *default_value*, if not NULL;
- the default value of *var*, if not NULL;
- NULL

Except for NULL, the function returns a new reference.

PyObject ***PyContextVar_Set** (*PyObject* *var, *PyObject* *value)

回傳值：新的參照。 Set the value of *var* to *value* in the current context. Returns a new token object for this change, or NULL if an error has occurred.

int **PyContextVar_Reset** (*PyObject* *var, *PyObject* *token)

Reset the state of the *var* context variable to that it was in before *PyContextVar_Set()* that returned the *token* was called. This function returns 0 on success and -1 on error.

8.6.13 DateTime 物件

Various date and time objects are supplied by the datetime module. Before using any of these functions, the header file datetime.h must be included in your source (note that this is not included by Python.h), and the macro PyDateTime_IMPORT must be invoked, usually as part of the module initialisation function. The macro puts a pointer to a C structure into a static variable, PyDateTimeAPI, that is used by the following macros.

type **PyDateTime_Date**

This subtype of *PyObject* represents a Python date object.

type **PyDateTime_DateTime**

This subtype of *PyObject* represents a Python datetime object.

type **PyDateTime_Time**

This subtype of *PyObject* represents a Python time object.

type **PyDateTime_Delta**

This subtype of *PyObject* represents the difference between two datetime values.

PyTypeObject **PyDateTime_DateType**

This instance of *PyTypeObject* represents the Python date type; it is the same object as datetime.date in the Python layer.

PyTypeObject **PyDateTime_DateTimeType**

This instance of *PyTypeObject* represents the Python datetime type; it is the same object as datetime.datetime in the Python layer.

PyTypeObject **PyDateTime_TimeType**

This instance of *PyTypeObject* represents the Python time type; it is the same object as datetime.time in the Python layer.

PyTypeObject **PyDateTime_DeltaType**

This instance of *PyTypeObject* represents Python type for the difference between two datetime values; it is the same object as datetime.timedelta in the Python layer.

PyTypeObject **PyDateTime_TZInfoType**

This instance of *PyTypeObject* represents the Python time zone info type; it is the same object as datetime.tzinfo in the Python layer.

用於存取 UTC 單例 (singleton) 的巨集：

PyObject ***PyDateTime_TimeZone_UTC**

回傳表示 UTC 的時區單例，是與 datetime.timezone.utc 相同的物件。

在 3.7 版新加入.

型別檢查巨集：

int **PyDate_Check** (*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_DateType* or a subtype of PyDateTime_DateType. *ob* must not be NULL. This function always succeeds.

int **PyDate_CheckExact** (*PyObject* *ob)

如果 *ob* 的型別為 *PyDateTime_DateType*，則回傳 true。*ob* 不得為 NULL。這個函式一定會執行成功。

int **PyDateTime_Check** (*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_DateTimeType* or a subtype of PyDateTime_DateTimeType. *ob* must not be NULL. This function always succeeds.

int **PyDateTime_CheckExact** (*PyObject* *ob)

如果 *ob* 的型別為 *PyDateTime_DateTimeType*，則回傳 true。*ob* 不得為 NULL。這個函式一定會執行成功。

int **PyTime_Check** (*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_TimeType* or a subtype of PyDateTime_TimeType. *ob* must not be NULL. This function always succeeds.

int **PyTime_CheckExact** (*PyObject* *ob)

如果 *ob* 的型態為*PyDateTime_TimeType*，則回傳 true。*ob* 不得為 NULL。這個函式一定會執行成功。

int **PyDelta_Check** (*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_DeltaType* or a subtype of PyDateTime_DeltaType. *ob* must not be NULL. This function always succeeds.

int **PyDelta_CheckExact** (*PyObject* *ob)

如果 *ob* 的型態為*PyDateTime_DeltaType*，則回傳 true。*ob* 不得為 NULL。這個函式一定會執行成功。

int **PyTZInfo_Check** (*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_TZInfoType* or a subtype of PyDateTime_TZInfoType. *ob* must not be NULL. This function always succeeds.

int **PyTZInfo_CheckExact** (*PyObject* *ob)

如果 *ob* 的型態為*PyDateTime_TZInfoType*，則回傳 true。*ob* 不得為 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（時間折疊）的物件 datetime.datetime。

在 3.6 版新加入.

PyObject ***PyTime_FromTime** (int hour, int minute, int second, int usecond)

回傳值：新的參照。回傳一個有特定時、分、秒、微秒的物件 datetime.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（時間折疊）的物件 datetime.time。

在 3.6 版新加入.

PyObject ***PyDelta_FromDSU** (int days, int seconds, int useconds)

回傳值：新的參照。回傳一個 datetime.timedelta 物件，表示給定的天數、秒數和微秒數。執行標准化 (normalization) 以便生成的微秒數和秒數位於 datetime.timedelta 物件記錄的範圍內。

PyObject ***PyTimeZone_FromOffset** (*PyObject* *offset)

回傳值：新的參照。回傳一個 datetime.timezone 物件，其未命名的固定偏移量由 *offset* 引數表示。

在 3.7 版新加入.

PyObject ***PyTimeZone_FromOffsetAndName** (*PyObject* *offset, *PyObject* *name)

回傳值：新的參照。回傳一個 datetime.timezone 物件，其固定偏移量由 *offset* 引數表示，並帶有 tzname *name*。

在 3.7 版新加入.

Macros to extract fields from date objects. The argument must be an instance of `PyDateTime_Date`, including subclasses (such as `PyDateTime_DateTime`). The argument must not be `NULL`, and the type is not checked:

int **PyDateTime_GET_YEAR** (*PyDateTime_Date* *o)

回傳年份，為正整數。

int **PyDateTime_GET_MONTH** (*PyDateTime_Date* *o)

回傳月份，為正整數，從 1 到 12。

int **PyDateTime_GET_DAY** (*PyDateTime_Date* *o)

回傳日期，為正整數，從 1 到 31。

Macros to extract fields from datetime objects. The argument must be an instance of `PyDateTime_DateTime`, including subclasses. The argument must not be `NULL`, and the type is not checked:

int **PyDateTime_DATE_GET_HOUR** (*PyDateTime_DateTime* *o)

回傳小時，為正整數，從 0 到 23。

int **PyDateTime_DATE_GET_MINUTE** (*PyDateTime_DateTime* *o)

回傳分鐘，為正整數，從 0 到 59。

int **PyDateTime_DATE_GET_SECOND** (*PyDateTime_DateTime* *o)

回傳秒，為正整數，從 0 到 59。

int **PyDateTime_DATE_GET_MICROSECOND** (*PyDateTime_DateTime* *o)

回傳微秒，為正整數，從 0 到 999999。

int **PyDateTime_DATE_GET_FOLD** (*PyDateTime_DateTime* *o)

回傳 fold，為 0 或 1 的正整數。

在 3.6 版新加入.

PyObject ***PyDateTime_DATE_GET_TZINFO** (*PyDateTime_DateTime* *o)

回傳 tzinfo（可能是 `None`）。

在 3.10 版新加入.

Macros to extract fields from time objects. The argument must be an instance of `PyDateTime_Time`, including subclasses. The argument must not be `NULL`, and the type is not checked:

int **PyDateTime_TIME_GET_HOUR** (*PyDateTime_Time* *o)

回傳小時，為正整數，從 0 到 23。

int **PyDateTime_TIME_GET_MINUTE** (*PyDateTime_Time* *o)

回傳分鐘，為正整數，從 0 到 59。

int **PyDateTime_TIME_GET_SECOND** (*PyDateTime_Time* *o)

回傳秒，為正整數，從 0 到 59。

int **PyDateTime_TIME_GET_MICROSECOND** (*PyDateTime_Time* *o)

回傳微秒，為正整數，從 0 到 999999。

int **PyDateTime_TIME_GET_FOLD** (*PyDateTime_Time* *o)

回傳 fold，為 0 或 1 的正整數。

在 3.6 版新加入.

PyObject ***PyDateTime_TIME_GET_TZINFO** (*PyDateTime_Time* *o)

回傳 tzinfo（可能是 `None`）。

在 3.10 版新加入.

Macros to extract fields from time delta objects. The argument must be an instance of `PyDateTime_Delta`, including subclasses. The argument must not be `NULL`, and the type is not checked:

int **PyDateTime_DELTA_GET_DAYS**(*PyDateTime_Delta* *o)

以 -999999999 到 999999999 之間的整數形式回傳天數。

在 3.3 版新加入.

int **PyDateTime_DELTA_GET_SECONDS**(*PyDateTime_Delta* *o)

以 0 到 86399 之間的整數形式回傳秒數。

在 3.3 版新加入.

int **PyDateTime_DELTA_GET_MICROSECONDS**(*PyDateTime_Delta* *o)

以 0 到 999999 之間的整數形式回傳微秒數。

在 3.3 版新加入.

F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F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F回傳一個新的 `datetime.date` 物件。

8.6.14 型F提示物件

提供了數個用於型F提示的F建型F。目前有兩種 -- GenericAlias 和 Union。只有 `GenericAlias` 有公開 (expose) 給 C。

PyObject ***Py_GenericAlias**(*PyObject* *origin, *PyObject* *args)

Part of the Stable ABI *since version 3.9.* 建立一個 GenericAlias 物件，等同於呼叫 Python 的 `types.GenericAlias` class。*origin* 和 *args* 引數分F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F 1 的 tuple 且 `__args__` 會被設F `(args,)`。只會進行最少的引數檢查，所以即便 *origin* 不是個型F，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F回傳 `NULL`。

以下是個讓一個擴充型F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:

The data model method `__class_getitem__()`.

在 3.9 版新加入.

PyTypeObject **Py_GenericAliasType**

Part of the Stable ABI *since version 3.9.* `Py_GenericAlias()` 所回傳該物件的 C 型F。等價於 Python 中的 `types.GenericAlias`。

在 3.9 版新加入.

CHAPTER 9

Initialization, Finalization, and Threads

See also *Python Initialization Configuration*.

9.1 Before Python Initialization

In an application embedding Python, the *Py_Initialize()* function must be called before using any other Python/C API functions; with the exception of a few functions and the *global configuration variables*.

The following functions can be safely called before Python is initialized:

- Configuration functions:
  - *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()*
- Informative functions:
  - *Py_IsInitialized()*
  - *PyMem_GetAllocator()*
  - *PyObject_GetArenaAllocator()*

  - *Py_GetBuildInfo()*
  - *Py_GetCompiler()*
  - *Py_GetCopyright()*
  - *Py_GetPlatform()*
  - *Py_GetVersion()*
- Utilities:
  - *Py_DecodeLocale()*
- Memory allocators:
  - *PyMem_RawMalloc()*
  - *PyMem_RawRealloc()*
  - *PyMem_RawCalloc()*
  - *PyMem_RawFree()*

備F: The following functions **should not be called** before *Py_Initialize()*: *Py_EncodeLocale()*, *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()*, *Py_GetProgramFullPath()*, *Py_GetPythonHome()*, *Py_GetProgramName()* and *PyEval_InitThreads()*.

9.2 Global configuration variables

Python has variables for the global configuration to control different features and options. By default, these flags are controlled by command line options.

When a flag is set by an option, the value of the flag is the number of times that the option was set. For example, -b sets *Py_BytesWarningFlag* to 1 and -bb sets *Py_BytesWarningFlag* to 2.

int **Py_BytesWarningFlag**

Issue a warning when comparing bytes or bytearray with str or bytes with int. Issue an error if greater or equal to 2.

由 -b 選項設定。

int **Py_DebugFlag**

Turn on parser debugging output (for expert only, depending on compilation options).

由 -d 選項與 PYTHONDEBUG 環境變數設定。

int **Py_DontWriteBytecodeFlag**

If set to non-zero, Python won't try to write .pyc files on the import of source modules.

由 -B 選項與 PYTHONDONTWRITEBYTECODE 環境變數設定。

int **Py_FrozenFlag**

Suppress error messages when calculating the module search path in *Py_GetPath()*.

Private flag used by _freeze_module and frozenmain programs.

int **Py_HashRandomizationFlag**

如果環境變數 PYTHONHASHSEED 被設定F一個非空字串則設F 1。

If the flag is non-zero, read the PYTHONHASHSEED environment variable to initialize the secret hash seed.

int **Py_IgnoreEnvironmentFlag**

Ignore all PYTHON* environment variables, e.g. PYTHONPATH and PYTHONHOME, that might be set.

由 -E 與 -I 選項設定。

int **Py_InspectFlag**

When a script is passed as first argument or the -c option is used, enter interactive mode after executing the script or the command, even when sys.stdin does not appear to be a terminal.

由 -i 選項與 PYTHONINSPECT 環境變數設定。

int **Py_InteractiveFlag**

由 -i 選項設定。

int **Py_IsolatedFlag**

Run Python in isolated mode. In isolated mode sys.path contains neither the script's directory nor the user's site-packages directory.

由 -i 選項設定。

在 3.4 版新加入.

int **Py_LegacyWindowsFSEncodingFlag**

If the flag is non-zero, use the mbcs encoding with replace error handler, instead of the UTF-8 encoding with surrogatepass error handler, for the *filesystem encoding and error handler*.

如果環境變數 PYTHONLEGACYWINDOWSFSENCODING 被設定為一個非空字串則設為 1。

更多詳情請見 **PEP 529**。

適用：Windows。

int **Py_LegacyWindowsStdioFlag**

If the flag is non-zero, use io.FileIO instead of io._WindowsConsoleIO for sys standard streams.

Set to 1 if the PYTHONLEGACYWINDOWSSTDIO environment variable is set to a non-empty string.

更多詳情請見 **PEP 528**。

適用：Windows。

int **Py_NoSiteFlag**

Disable the import of the module site and the site-dependent manipulations of sys.path that it entails. Also disable these manipulations if site is explicitly imported later (call site.main() if you want them to be triggered).

由 -S 選項設定。

int **Py_NoUserSiteDirectory**

Don't add the user site-packages directory to sys.path.

由 -s 選項、-I 選項與 PYTHONNOUSERSITE 環境變數設定。

int **Py_OptimizeFlag**

由 -O 選項與 PYTHONOPTIMIZE 環境變數設定。

int **Py_QuietFlag**

Don't display the copyright and version messages even in interactive mode.

由 -q 選項設定。

在 3.2 版新加入.

int **Py_UnbufferedStdioFlag**

Force the stdout and stderr streams to be unbuffered.

由 -u 選項與 PYTHONUNBUFFERED 環境變數設定。

int **Py_VerboseFlag**

Print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. If greater or equal to `2`, print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

由 `-v` 選項與 `PYTHONVERBOSE` 環境變數設定。

9.3 Initializing and finalizing the interpreter

void **Py_Initialize**()

Part of the Stable ABI. Initialize the Python interpreter. In an application embedding Python, this should be called before using any other Python/C API functions; see *Before Python Initialization* for the few exceptions.

This initializes the table of loaded modules (`sys.modules`), and creates the fundamental modules `builtins`, `__main__` and `sys`. It also initializes the module search path (`sys.path`). It does not set `sys.argv`; use `PySys_SetArgvEx()` for that. This is a no-op when called for a second time (without calling `Py_FinalizeEx()` first). There is no return value; it is a fatal error if the initialization fails.

備註: On Windows, changes the console mode from `O_TEXT` to `O_BINARY`, which will also affect non-Python uses of the console using the C Runtime.

---

void **Py_InitializeEx**(int initsigs)

Part of the Stable ABI. This function works like `Py_Initialize()` if *initsigs* is `1`. If *initsigs* is `0`, it skips initialization registration of signal handlers, which might be useful when Python is embedded.

int **Py_IsInitialized**()

Part of the Stable ABI. Return true (nonzero) when the Python interpreter has been initialized, false (zero) if not. After `Py_FinalizeEx()` is called, this returns false until `Py_Initialize()` is called again.

int **Py_FinalizeEx**()

Part of the Stable ABI *since version 3.6.* Undo all initializations made by `Py_Initialize()` and subsequent use of Python/C API functions, and destroy all sub-interpreters (see `Py_NewInterpreter()` below) that were created and not yet destroyed since the last call to `Py_Initialize()`. Ideally, this frees all memory allocated by the Python interpreter. This is a no-op when called for a second time (without calling `Py_Initialize()` again first). Normally the return value is `0`. If there were errors during finalization (flushing buffered data), `-1` is returned.

This function is provided for a number of reasons. An embedding application might want to restart Python without having to restart the application itself. An application that has loaded the Python interpreter from a dynamically loadable library (or DLL) might want to free all memory allocated by Python before unloading the DLL. During a hunt for memory leaks in an application a developer might want to free all memory allocated by Python before exiting from the application.

Bugs and caveats: The destruction of modules and objects in modules is done in random order; this may cause destructors (`__del__()` methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once; this can happen if an application calls `Py_Initialize()` and `Py_FinalizeEx()` more than once.

引發一個不附帶引數的稽核事件 `cpython._PySys_ClearAuditHooks`。

在 3.6 版新加入.

void **Py_Finalize**()

Part of the Stable ABI. This is a backwards-compatible version of `Py_FinalizeEx()` that disregards the return value.

9.4 Process-wide parameters

int **Py_SetStandardStreamEncoding**(const char *encoding, const char *errors)

This API is kept for backward compatibility: setting *PyConfig.stdio_encoding* and *PyConfig.stdio_errors* should be used instead, see *Python Initialization Configuration*.

This function should be called before *Py_Initialize()*, if it is called at all. It specifies which encoding and error handling to use with standard IO, with the same meanings as in `str.encode()`.

It overrides `PYTHONIOENCODING` values, and allows embedding code to control IO encoding when the environment variable does not work.

encoding and/or *errors* may be `NULL` to use `PYTHONIOENCODING` and/or default values (depending on other settings).

Note that `sys.stderr` always uses the "backslashreplace" error handler, regardless of this (or any other) setting.

If *Py_FinalizeEx()* is called, this function will need to be called again in order to affect subsequent calls to *Py_Initialize()*.

Returns `0` if successful, a nonzero value on error (e.g. calling after the interpreter has already been initialized).

在 3.4 版新加入.

在 3.11 版之後被F用.

void **Py_SetProgramName**(const wchar_t *name)

Part of the Stable ABI. This API is kept for backward compatibility: setting *PyConfig.program_name* should be used instead, see *Python Initialization Configuration*.

This function should be called before *Py_Initialize()* is called for the first time, if it is called at all. It tells the interpreter the value of the `argv[0]` argument to the `main()` function of the program (converted to wide characters). This is used by *Py_GetPath()* and some other functions below to find the Python run-time libraries relative to the interpreter executable. The default value is `'python'`. The argument should point to a zero-terminated wide character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_*` string.

在 3.11 版之後被F用.

wchar_t ***Py_GetProgramName**()

Part of the Stable ABI. Return the program name set with *Py_SetProgramName()*, or the default. The returned string points into static storage; the caller should not modify its value.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

在 3.10 版的變更: It now returns `NULL` if called before *Py_Initialize()*.

wchar_t ***Py_GetPrefix**()

Part of the Stable ABI. Return the *prefix* for installed platform-independent files. This is derived through a number of complicated rules from the program name set with *Py_SetProgramName()* and some environment variables; for example, if the program name is `'/usr/local/bin/python'`, the prefix is `'/usr/local'`. The returned string points into static storage; the caller should not modify its value. This corresponds to the **prefix** variable in the top-level `Makefile` and the `--prefix` argument to the **configure** script at build time. The value is available to Python code as `sys.prefix`. It is only useful on Unix. See also the next function.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

在 3.10 版的變更: It now returns `NULL` if called before *Py_Initialize()*.

wchar_t ***Py_GetExecPrefix**()

Part of the Stable ABI. Return the *exec-prefix* for installed platform-*dependent* files. This is derived through a number of complicated rules from the program name set with *Py_SetProgramName()* and some environment variables; for example, if the program name is '/usr/local/bin/python', the exec-prefix is '/usr/local'. The returned string points into static storage; the caller should not modify its value. This corresponds to the **exec_prefix** variable in the top-level Makefile and the --exec-prefix argument to the **configure** script at build time. The value is available to Python code as sys.exec_prefix. It is only useful on Unix.

Background: The exec-prefix differs from the prefix when platform dependent files (such as executables and shared libraries) are installed in a different directory tree. In a typical installation, platform dependent files may be installed in the /usr/local/plat subtree while platform independent may be installed in /usr/local.

Generally speaking, a platform is a combination of hardware and software families, e.g. Sparc machines running the Solaris 2.x operating system are considered the same platform, but Intel machines running Solaris 2.x are another platform, and Intel machines running Linux are yet another platform. Different major revisions of the same operating system generally also form different platforms. Non-Unix operating systems are a different story; the installation strategies on those systems are so different that the prefix and exec-prefix are meaningless, and set to the empty string. Note that compiled Python bytecode files are platform independent (but not independent from the Python version by which they were compiled!).

System administrators will know how to configure the **mount** or **automount** programs to share /usr/local between platforms while having /usr/local/plat be a different filesystem for each platform.

This function should not be called before *Py_Initialize()*, otherwise it returns NULL.

在 3.10 版的變更: It now returns NULL if called before *Py_Initialize()*.

wchar_t ***Py_GetProgramFullPath**()

Part of the Stable ABI. Return the full program name of the Python executable; this is computed as a side-effect of deriving the default module search path from the program name (set by *Py_SetProgramName()* above). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as sys.executable.

This function should not be called before *Py_Initialize()*, otherwise it returns NULL.

在 3.10 版的變更: It now returns NULL if called before *Py_Initialize()*.

wchar_t ***Py_GetPath**()

Part of the Stable ABI. Return the default module search path; this is computed from the program name (set by *Py_SetProgramName()* above) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is ':' on Unix and macOS, ';' on Windows. The returned string points into static storage; the caller should not modify its value. The list sys.path is initialized with this value on interpreter startup; it can be (and usually is) modified later to change the search path for loading modules.

This function should not be called before *Py_Initialize()*, otherwise it returns NULL.

在 3.10 版的變更: It now returns NULL if called before *Py_Initialize()*.

void **Py_SetPath**(const wchar_t*)

Part of the Stable ABI *since version 3.7*. This API is kept for backward compatibility: setting *PyConfig.module_search_paths* and *PyConfig.module_search_paths_set* should be used instead, see *Python Initialization Configuration*.

Set the default module search path. If this function is called before *Py_Initialize()*, then *Py_GetPath()* won't attempt to compute a default search path but uses the one provided instead. This is useful if Python is embedded by an application that has full knowledge of the location of all modules. The path components should be separated by the platform dependent delimiter character, which is ':' on Unix and macOS, ';' on Windows.

This also causes sys.executable to be set to the program full path (see *Py_GetProgramFullPath()*) and for sys.prefix and sys.exec_prefix to be empty. It is up to the caller to modify these if required after calling *Py_Initialize()*.

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_*` string.

The path argument is copied internally, so the caller may free it after the call completes.

在 3.8 版的變更: The program full path is now used for `sys.executable`, instead of the program name.

在 3.11 版之後被棄用.

const char ***Py_GetVersion**()

Part of the Stable ABI. Return the version of this Python interpreter. This is a string that looks something like

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

The first word (up to the first space character) is the current Python version; the first characters are the major and minor version separated by a period. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.version`.

See also the *Py_Version* constant.

const char ***Py_GetPlatform**()

Part of the Stable ABI. Return the platform identifier for the current platform. On Unix, this is formed from the "official" name of the operating system, converted to lower case, followed by the major revision number; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is `'sunos5'`. On macOS, it is `'darwin'`. On Windows, it is `'win'`. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.platform`.

const char ***Py_GetCopyright**()

Part of the Stable ABI. Return the official copyright string for the current Python version, for example

`'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'`

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.copyright`.

const char ***Py_GetCompiler**()

Part of the Stable ABI. Return an indication of the compiler used to build the current Python version, in square brackets, for example:

```
"[GCC 2.7.2.2]"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

const char ***Py_GetBuildInfo**()

Part of the Stable ABI. Return information about the sequence number and build date and time of the current Python interpreter instance, for example

```
"#67, Aug  1 1997, 22:34:28"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

void **PySys_SetArgvEx**(int argc, wchar_t **argv, int updatepath)

Part of the Stable ABI. This API is kept for backward compatibility: setting *PyConfig.argv*, *PyConfig.parse_argv* and *PyConfig.safe_path* should be used instead, see *Python Initialization Configuration*.

Set `sys.argv` based on *argc* and *argv*. These parameters are similar to those passed to the program's `main()` function with the difference that the first entry should refer to the script file to be executed rather than the executable hosting the Python interpreter. If there isn't a script that will be run, the first entry in *argv* can be an empty string. If this function fails to initialize `sys.argv`, a fatal condition is signalled using *Py_FatalError()*.

If *updatepath* is zero, this is all the function does. If *updatepath* is non-zero, the function also modifies `sys.path` according to the following algorithm:

- If the name of an existing script is passed in `argv[0]`, the absolute path of the directory where the script is located is prepended to `sys.path`.
- Otherwise (that is, if *argc* is `0` or `argv[0]` doesn't point to an existing file name), an empty string is prepended to `sys.path`, which is the same as prepending the current working directory (`"."`).

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_*` string.

See also `PyConfig.orig_argv` and `PyConfig.argv` members of the *Python Initialization Configuration*.

備註: It is recommended that applications embedding the Python interpreter for purposes other than executing a single script pass `0` as *updatepath*, and update `sys.path` themselves if desired. See CVE-2008-5983.

On versions before 3.1.3, you can achieve the same effect by manually popping the first `sys.path` element after having called `PySys_SetArgv()`, for example using:

```
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```

在 3.1.3 版新加入.

在 3.11 版之後被棄用.

void **PySys_SetArgv**(int argc, wchar_t **argv)

Part of the Stable ABI. This API is kept for backward compatibility: setting `PyConfig.argv` and `PyConfig.parse_argv` should be used instead, see *Python Initialization Configuration*.

This function works like `PySys_SetArgvEx()` with *updatepath* set to `1` unless the **python** interpreter was started with the `-I`.

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_*` string.

See also `PyConfig.orig_argv` and `PyConfig.argv` members of the *Python Initialization Configuration*.

在 3.4 版的變更: The *updatepath* value depends on `-I`.

在 3.11 版之後被棄用.

void **Py_SetPythonHome**(const wchar_t *home)

Part of the Stable ABI. This API is kept for backward compatibility: setting `PyConfig.home` should be used instead, see *Python Initialization Configuration*.

Set the default "home" directory, that is, the location of the standard Python libraries. See `PYTHONHOME` for the meaning of the argument string.

The argument should point to a zero-terminated character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_*` string.

在 3.11 版之後被棄用.

wchar_t ***Py_GetPythonHome**()

Part of the Stable ABI. Return the default "home", that is, the value set by a previous call to `Py_SetPythonHome()`, or the value of the `PYTHONHOME` environment variable if it is set.

This function should not be called before `Py_Initialize()`, otherwise it returns `NULL`.

在 3.10 版的變更: It now returns `NULL` if called before `Py_Initialize()`.

9.5 Thread State and the Global Interpreter Lock

The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the *global interpreter lock* or *GIL*, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

Therefore, the rule exists that only the thread that has acquired the *GIL* may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads (see `sys.setswitchinterval()`). The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.

The Python interpreter keeps some thread-specific bookkeeping information inside a data structure called `PyThreadState`. There's also one global variable pointing to the current `PyThreadState`: it can be retrieved using `PyThreadState_Get()`.

9.5.1 Releasing the GIL from extension code

Most extension code manipulating the *GIL* has the following simple structure:

```
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.
```

This is so common that a pair of macros exists to simplify it:

```
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

The `Py_BEGIN_ALLOW_THREADS` macro opens a new block and declares a hidden local variable; the `Py_END_ALLOW_THREADS` macro closes the block.

The block above expands to the following code:

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

Here is how these functions work: the global interpreter lock is used to protect the pointer to the current thread state. When releasing the lock and saving the thread state, the current thread state pointer must be retrieved before the lock is released (since another thread could immediately acquire the lock and store its own thread state in the global variable). Conversely, when acquiring the lock and restoring the thread state, the lock must be acquired before storing the thread state pointer.

備F: Calling system I/O functions is the most common use case for releasing the GIL, but it can also be useful before calling long-running computations which don't need access to Python objects, such as compression or cryptographic functions operating over memory buffers. For example, the standard `zlib` and `hashlib` modules release the GIL when compressing or hashing data.

9.5.2 Non-Python created threads

When threads are created using the dedicated Python APIs (such as the `threading` module), a thread state is automatically associated to them and the code showed above is therefore correct. However, when threads are created from C (for example by a third-party library with its own thread management), they don't hold the GIL, nor is there a thread state structure for them.

If you need to call Python code from these threads (often this will be part of a callback API provided by the aforementioned third-party library), you must first register these threads with the interpreter by creating a thread state data structure, then acquiring the GIL, and finally storing their thread state pointer, before you can start using the Python/C API. When you are done, you should reset the thread state pointer, release the GIL, and finally free the thread state data structure.

The `PyGILState_Ensure()` and `PyGILState_Release()` functions do all of the above automatically. The typical idiom for calling into Python from a C thread is:

```
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Release the thread. No Python API allowed beyond this point. */
PyGILState_Release(gstate);
```

Note that the `PyGILState_*` functions assume there is only one global interpreter (created automatically by `Py_Initialize()`). Python supports the creation of additional interpreters (using `Py_NewInterpreter()`), but mixing multiple interpreters and the `PyGILState_*` API is unsupported.

9.5.3 Cautions about fork()

Another important thing to note about threads is their behaviour in the face of the C `fork()` call. On most systems with `fork()`, after a process forks only the thread that issued the fork will exist. This has a concrete impact both on how locks must be handled and on all stored state in CPython's runtime.

The fact that only the "current" thread remains means any locks held by other threads will never be released. Python solves this for `os.fork()` by acquiring the locks it uses internally before the fork, and releasing them afterwards. In addition, it resets any lock-objects in the child. When extending or embedding Python, there is no way to inform Python of additional (non-Python) locks that need to be acquired before or reset after a fork. OS facilities such as `pthread_atfork()` would need to be used to accomplish the same thing. Additionally, when extending or embedding Python, calling `fork()` directly rather than through `os.fork()` (and returning to or calling into Python) may result in a deadlock by one of Python's internal locks being held by a thread that is defunct after the fork. `PyOS_AfterFork_Child()` tries to reset the necessary locks, but is not always able to.

The fact that all other threads go away also means that CPython's runtime state there must be cleaned up properly, which `os.fork()` does. This means finalizing all other `PyThreadState` objects belonging to the current interpreter and all other `PyInterpreterState` objects. Due to this and the special nature of the *"main" interpreter*, `fork()` should only be called in that interpreter's "main" thread, where the CPython global runtime was originally initialized. The only exception is if `exec()` will be called immediately after.

9.5.4 高階 API

These are the most commonly used types and functions when writing C extension code, or when embedding the Python interpreter:

type **PyInterpreterState**

Part of the Limited API *(as an opaque struct)*. This data structure represents the state shared by a number of cooperating threads. Threads belonging to the same interpreter share their module administration and a few other internal items. There are no public members in this structure.

Threads belonging to different interpreters initially share nothing, except process state like available memory, open file descriptors and such. The global interpreter lock is also shared by all threads, regardless of to which interpreter they belong.

type **PyThreadState**

Part of the Limited API *(as an opaque struct)*. This data structure represents the state of a single thread. The only public data member is:

PyInterpreterState ***interp**

This thread's interpreter state.

void **PyEval_InitThreads**()

Part of the Stable ABI. Deprecated function which does nothing.

In Python 3.6 and older, this function created the GIL if it didn't exist.

在 3.9 版的變更: 此函式現在不會做任何事情。

在 3.7 版的變更: This function is now called by `Py_Initialize()`, so you don't have to call it yourself anymore.

在 3.2 版的變更: This function cannot be called before `Py_Initialize()` anymore.

在 3.9 版之後被棄用.

int **PyEval_ThreadsInitialized**()

Part of the Stable ABI. Returns a non-zero value if `PyEval_InitThreads()` has been called. This function can be called without holding the GIL, and therefore can be used to avoid calls to the locking API when running single-threaded.

在 3.7 版的變更: The *GIL* is now initialized by `Py_Initialize()`.

在 3.9 版之後被棄用.

PyThreadState ***PyEval_SaveThread**()

Part of the Stable ABI. Release the global interpreter lock (if it has been created) and reset the thread state to `NULL`, returning the previous thread state (which is not `NULL`). If the lock has been created, the current thread must have acquired it.

void **PyEval_RestoreThread**(*PyThreadState* *tstate)

Part of the Stable ABI. Acquire the global interpreter lock (if it has been created) and set the thread state to *tstate*, which must not be `NULL`. If the lock has been created, the current thread must not have acquired it, otherwise deadlock ensues.

備註: Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use `_Py_IsFinalizing()` or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

PyThreadState ***PyThreadState_Get**()

Part of the Stable ABI. Return the current thread state. The global interpreter lock must be held. When the current thread state is `NULL`, this issues a fatal error (so that the caller needn't check for `NULL`).

PyThreadState ***PyThreadState_Swap***(*PyThreadState* *tstate)

Part of the Stable ABI. Swap the current thread state with the thread state given by the argument *tstate*, which may be NULL. The global interpreter lock must be held and is not released.

The following functions use thread-local storage, and are not compatible with sub-interpreters:

PyGILState_STATE **PyGILState_Ensure**()

Part of the Stable ABI. Ensure that the current thread is ready to call the Python C API regardless of the current state of Python, or of the global interpreter lock. This may be called as many times as desired by a thread as long as each call is matched with a call to *PyGILState_Release()*. In general, other thread-related APIs may be used between *PyGILState_Ensure()* and *PyGILState_Release()* calls as long as the thread state is restored to its previous state before the Release(). For example, normal usage of the *Py_BEGIN_ALLOW_THREADS* and *Py_END_ALLOW_THREADS* macros is acceptable.

The return value is an opaque "handle" to the thread state when *PyGILState_Ensure()* was called, and must be passed to *PyGILState_Release()* to ensure Python is left in the same state. Even though recursive calls are allowed, these handles *cannot* be shared - each unique call to *PyGILState_Ensure()* must save the handle for its call to *PyGILState_Release()*.

When the function returns, the current thread will hold the GIL and be able to call arbitrary Python code. Failure is a fatal error.

備F: Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use _Py_IsFinalizing() or sys.is_finalizing() to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

---

void **PyGILState_Release**(PyGILState_STATE)

Part of the Stable ABI. Release any resources previously acquired. After this call, Python's state will be the same as it was prior to the corresponding *PyGILState_Ensure()* call (but generally this state will be unknown to the caller, hence the use of the GILState API).

Every call to *PyGILState_Ensure()* must be matched by a call to *PyGILState_Release()* on the same thread.

PyThreadState ***PyGILState_GetThisThreadState***()

Part of the Stable ABI. Get the current thread state for this thread. May return NULL if no GILState API has been used on the current thread. Note that the main thread always has such a thread-state, even if no auto-thread-state call has been made on the main thread. This is mainly a helper/diagnostic function.

int **PyGILState_Check**()

Return 1 if the current thread is holding the GIL and 0 otherwise. This function can be called from any thread at any time. Only if it has had its Python thread state initialized and currently is holding the GIL will it return 1. This is mainly a helper/diagnostic function. It can be useful for example in callback contexts or memory allocation functions when knowing that the GIL is locked can allow the caller to perform sensitive actions or otherwise behave differently.

在 3.4 版新加入.

The following macros are normally used without a trailing semicolon; look for example usage in the Python source distribution.

Py_BEGIN_ALLOW_THREADS

Part of the Stable ABI. This macro expands to { PyThreadState *_save; _save = PyEval_SaveThread();. Note that it contains an opening brace; it must be matched with a following *Py_END_ALLOW_THREADS* macro. See above for further discussion of this macro.

Py_END_ALLOW_THREADS

Part of the Stable ABI. This macro expands to PyEval_RestoreThread(_save); }. Note that it contains a closing brace; it must be matched with an earlier *Py_BEGIN_ALLOW_THREADS* macro. See above for further discussion of this macro.

Py_BLOCK_THREADS

Part of the Stable ABI. This macro expands to `PyEval_RestoreThread(_save);`: it is equivalent to *Py_END_ALLOW_THREADS* without the closing brace.

Py_UNBLOCK_THREADS

Part of the Stable ABI. This macro expands to `_save = PyEval_SaveThread();`: it is equivalent to *Py_BEGIN_ALLOW_THREADS* without the opening brace and variable declaration.

9.5.5 低階 API

All of the following functions must be called after *Py_Initialize()*.

在 3.7 版的變更: *Py_Initialize()* now initializes the *GIL*.

PyInterpreterState ***PyInterpreterState_New**()

Part of the Stable ABI. Create a new interpreter state object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

引發一個不附帶引數的稽核事件 `cpython.PyInterpreterState_New`。

void **PyInterpreterState_Clear**(*PyInterpreterState* *interp)

Part of the Stable ABI. Reset all information in an interpreter state object. The global interpreter lock must be held.

引發一個不附帶引數的稽核事件 `cpython.PyInterpreterState_Clear`。

void **PyInterpreterState_Delete**(*PyInterpreterState* *interp)

Part of the Stable ABI. Destroy an interpreter state object. The global interpreter lock need not be held. The interpreter state must have been reset with a previous call to *PyInterpreterState_Clear()*.

PyThreadState ***PyThreadState_New**(*PyInterpreterState* *interp)

Part of the Stable ABI. Create a new thread state object belonging to the given interpreter object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

void **PyThreadState_Clear**(*PyThreadState* *tstate)

Part of the Stable ABI. Reset all information in a thread state object. The global interpreter lock must be held.

在 3.9 版的變更: This function now calls the `PyThreadState.on_delete` callback. Previously, that happened in *PyThreadState_Delete()*.

void **PyThreadState_Delete**(*PyThreadState* *tstate)

Part of the Stable ABI. Destroy a thread state object. The global interpreter lock need not be held. The thread state must have been reset with a previous call to *PyThreadState_Clear()*.

void **PyThreadState_DeleteCurrent**(void)

Destroy the current thread state and release the global interpreter lock. Like *PyThreadState_Delete()*, the global interpreter lock need not be held. The thread state must have been reset with a previous call to *PyThreadState_Clear()*.

PyFrameObject ***PyThreadState_GetFrame**(*PyThreadState* *tstate)

Part of the Stable ABI *since version 3.10.* Get the current frame of the Python thread state *tstate*.

Return a *strong reference*. Return `NULL` if no frame is currently executing.

也請見*PyEval_GetFrame()*。

tstate 不可[F] `NULL`。

在 3.9 版新加入.

uint64_t **PyThreadState_GetID**(*PyThreadState* *tstate)

Part of the Stable ABI *since version 3.10.* Get the unique thread state identifier of the Python thread state *tstate*.

tstate 不可F NULL。

在 3.9 版新加入.

PyInterpreterState ***PyThreadState_GetInterpreter**(*PyThreadState* *tstate)

Part of the Stable ABI *since version 3.10.* Get the interpreter of the Python thread state *tstate*.

tstate 不可F NULL。

在 3.9 版新加入.

void **PyThreadState_EnterTracing**(*PyThreadState* *tstate)

Suspend tracing and profiling in the Python thread state *tstate*.

Resume them using the *PyThreadState_LeaveTracing()* function.

在 3.11 版新加入.

void **PyThreadState_LeaveTracing**(*PyThreadState* *tstate)

Resume tracing and profiling in the Python thread state *tstate* suspended by the *PyThreadState_EnterTracing()* function.

See also *PyEval_SetTrace()* and *PyEval_SetProfile()* functions.

在 3.11 版新加入.

PyInterpreterState ***PyInterpreterState_Get**(void)

Part of the Stable ABI *since version 3.9.* Get the current interpreter.

Issue a fatal error if there no current Python thread state or no current interpreter. It cannot return NULL.

The caller must hold the GIL.

在 3.9 版新加入.

int64_t **PyInterpreterState_GetID**(*PyInterpreterState* *interp)

Part of the Stable ABI *since version 3.7.* Return the interpreter's unique ID. If there was any error in doing so then -1 is returned and an error is set.

The caller must hold the GIL.

在 3.7 版新加入.

PyObject ***PyInterpreterState_GetDict**(*PyInterpreterState* *interp)

Part of the Stable ABI *since version 3.8.* Return a dictionary in which interpreter-specific data may be stored. If this function returns NULL then no exception has been raised and the caller should assume no interpreter-specific dict is available.

This is not a replacement for *PyModule_GetState()*, which extensions should use to store interpreter-specific state information.

在 3.8 版新加入.

typedef *PyObject* *(***_PyFrameEvalFunction**)(*PyThreadState* *tstate, _PyInterpreterFrame *frame, int throwflag)

Type of a frame evaluation function.

The *throwflag* parameter is used by the throw() method of generators: if non-zero, handle the current exception.

在 3.9 版的變更: The function now takes a *tstate* parameter.

在 3.11 版的變更: The *frame* parameter changed from PyFrameObject* to _PyInterpreterFrame*.

_PyFrameEvalFunction **_PyInterpreterState_GetEvalFrameFunc**(*PyInterpreterState* *interp)

Get the frame evaluation function.

See the **PEP 523** "Adding a frame evaluation API to CPython".

在 3.9 版新加入.

void **_PyInterpreterState_SetEvalFrameFunc**(*PyInterpreterState* *interp, *_PyFrameEvalFunction* eval_frame)

Set the frame evaluation function.

See the **PEP 523** "Adding a frame evaluation API to CPython".

在 3.9 版新加入.

PyObject ***PyThreadState_GetDict**()

回傳值：借用參照。*Part of the* Stable ABI. Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no current thread state is available. If this function returns `NULL`, no exception has been raised and the caller should assume no current thread state is available.

int **PyThreadState_SetAsyncExc**(unsigned long id, *PyObject* *exc)

Part of the Stable ABI. Asynchronously raise an exception in a thread. The *id* argument is the thread id of the target thread; *exc* is the exception object to be raised. This function does not steal any references to *exc*. To prevent naive misuse, you must write your own C extension to call this. Must be called with the GIL held. Returns the number of thread states modified; this is normally one, but will be zero if the thread id isn't found. If *exc* is `NULL`, the pending exception (if any) for the thread is cleared. This raises no exceptions.

在 3.7 版的變更: The type of the *id* parameter changed from `long` to `unsigned long`.

void **PyEval_AcquireThread**(*PyThreadState* *tstate)

Part of the Stable ABI. Acquire the global interpreter lock and set the current thread state to *tstate*, which must not be `NULL`. The lock must have been created earlier. If this thread already has the lock, deadlock ensues.

備註： Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use `_Py_IsFinalizing()` or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

在 3.8 版的變更: Updated to be consistent with `PyEval_RestoreThread()`, `Py_END_ALLOW_THREADS()`, and `PyGILState_Ensure()`, and terminate the current thread if called while the interpreter is finalizing.

`PyEval_RestoreThread()` is a higher-level function which is always available (even when threads have not been initialized).

void **PyEval_ReleaseThread**(*PyThreadState* *tstate)

Part of the Stable ABI. Reset the current thread state to `NULL` and release the global interpreter lock. The lock must have been created earlier and must be held by the current thread. The *tstate* argument, which must not be `NULL`, is only used to check that it represents the current thread state --- if it isn't, a fatal error is reported.

`PyEval_SaveThread()` is a higher-level function which is always available (even when threads have not been initialized).

void **PyEval_AcquireLock**()

Part of the Stable ABI. Acquire the global interpreter lock. The lock must have been created earlier. If this thread already has the lock, a deadlock ensues.

在 3.2 版之後被棄用: This function does not update the current thread state. Please use `PyEval_RestoreThread()` or `PyEval_AcquireThread()` instead.

備註: Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use `_Py_IsFinalizing()` or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

在 3.8 版的變更: Updated to be consistent with *PyEval_RestoreThread()*, *Py_END_ALLOW_THREADS()*, and *PyGILState_Ensure()*, and terminate the current thread if called while the interpreter is finalizing.

void **PyEval_ReleaseLock**()

Part of the Stable ABI. Release the global interpreter lock. The lock must have been created earlier.

在 3.2 版之後被棄用: This function does not update the current thread state. Please use *PyEval_SaveThread()* or *PyEval_ReleaseThread()* instead.

9.6 Sub-interpreter support

While in most uses, you will only embed a single Python interpreter, there are cases where you need to create several independent interpreters in the same process and perhaps even in the same thread. Sub-interpreters allow you to do that.

The "main" interpreter is the first one created when the runtime initializes. It is usually the only Python interpreter in a process. Unlike sub-interpreters, the main interpreter has unique process-global responsibilities like signal handling. It is also responsible for execution during runtime initialization and is usually the active interpreter during runtime finalization. The *PyInterpreterState_Main()* function returns a pointer to its state.

You can switch between sub-interpreters using the *PyThreadState_Swap()* function. You can create and destroy them using the following functions:

PyThreadState ***Py_NewInterpreter**()

Part of the Stable ABI. Create a new sub-interpreter. This is an (almost) totally separate environment for the execution of Python code. In particular, the new interpreter has separate, independent versions of all imported modules, including the fundamental modules `builtins`, `__main__` and `sys`. The table of loaded modules (`sys.modules`) and the module search path (`sys.path`) are also separate. The new environment has no `sys.argv` variable. It has new standard I/O stream file objects `sys.stdin`, `sys.stdout` and `sys.stderr` (however these refer to the same underlying file descriptors).

The return value points to the first thread state created in the new sub-interpreter. This thread state is made in the current thread state. Note that no actual thread is created; see the discussion of thread states below. If creation of the new interpreter is unsuccessful, `NULL` is returned; no exception is set since the exception state is stored in the current thread state and there may not be a current thread state. (Like all other Python/C API functions, the global interpreter lock must be held before calling this function and is still held when it returns; however, unlike most other Python/C API functions, there needn't be a current thread state on entry.)

Extension modules are shared between (sub-)interpreters as follows:

- For modules using multi-phase initialization, e.g. *PyModule_FromDefAndSpec()*, a separate module object is created and initialized for each interpreter. Only C-level static and global variables are shared between these module objects.
- For modules using single-phase initialization, e.g. *PyModule_Create()*, the first time a particular extension is imported, it is initialized normally, and a (shallow) copy of its module's dictionary is squirreled away. When the same extension is imported by another (sub-)interpreter, a new module is initialized and filled with the contents of this copy; the extension's `init` function is not called. Objects in the module's dictionary thus end up shared across (sub-)interpreters, which might cause unwanted behavior (see *Bugs and caveats* below).

  Note that this is different from what happens when an extension is imported after the interpreter has been completely re-initialized by calling *Py_FinalizeEx()* and *Py_Initialize()*; in that case, the extension's `initmodule` function *is* called again. As with multi-phase initialization, this means that only C-level static and global variables are shared between these modules.

void **Py_EndInterpreter**(*PyThreadState* *tstate)

Part of the Stable ABI. Destroy the (sub-)interpreter represented by the given thread state. The given thread state must be the current thread state. See the discussion of thread states below. When the call returns, the current thread state is `NULL`. All thread states associated with this interpreter are destroyed. (The global interpreter lock must be held before calling this function and is still held when it returns.) *`Py_FinalizeEx()`* will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

9.6.1 Bugs and caveats

Because sub-interpreters (and the main interpreter) are part of the same process, the insulation between them isn't perfect --- for example, using low-level file operations like `os.close()` they can (accidentally or maliciously) affect each other's open files. Because of the way extensions are shared between (sub-)interpreters, some extensions may not work properly; this is especially likely when using single-phase initialization or (static) global variables. It is possible to insert objects created in one sub-interpreter into a namespace of another (sub-)interpreter; this should be avoided if possible.

Special care should be taken to avoid sharing user-defined functions, methods, instances or classes between sub-interpreters, since import operations executed by such objects may affect the wrong (sub-)interpreter's dictionary of loaded modules. It is equally important to avoid sharing objects from which the above are reachable.

Also note that combining this functionality with `PyGILState_*` APIs is delicate, because these APIs assume a bijection between Python thread states and OS-level threads, an assumption broken by the presence of sub-interpreters. It is highly recommended that you don't switch sub-interpreters between a pair of matching *`PyGILState_Ensure()`* and *`PyGILState_Release()`* calls. Furthermore, extensions (such as `ctypes`) using these APIs to allow calling of Python code from non-Python created threads will probably be broken when using sub-interpreters.

9.7 Asynchronous Notifications

A mechanism is provided to make asynchronous notifications to the main interpreter thread. These notifications take the form of a function pointer and a void pointer argument.

int **Py_AddPendingCall**(int (*func)(void*), void *arg)

Part of the Stable ABI. Schedule a function to be called from the main interpreter thread. On success, `0` is returned and *func* is queued for being called in the main thread. On failure, `-1` is returned without setting any exception.

When successfully queued, *func* will be *eventually* called from the main interpreter thread with the argument *arg*. It will be called asynchronously with respect to normally running Python code, but with both these conditions met:

- on a *bytecode* boundary;
- with the main thread holding the *global interpreter lock* (*func* can therefore use the full C API).

func must return `0` on success, or `-1` on failure with an exception set. *func* won't be interrupted to perform another asynchronous notification recursively, but it can still be interrupted to switch threads if the global interpreter lock is released.

This function doesn't need a current thread state to run, and it doesn't need the global interpreter lock.

To call this function in a subinterpreter, the caller must hold the GIL. Otherwise, the function *func* can be scheduled to be called from the wrong interpreter.

> **警告:** This is a low-level function, only useful for very special cases. There is no guarantee that *func* will be called as quick as possible. If the main thread is busy executing a system call, *func* won't be called before the system call returns. This function is generally **not** suitable for calling Python code from arbitrary C threads. Instead, use the *PyGILState API*.

在 3.1 版新加入.

在 3.9 版的變更: If this function is called in a subinterpreter, the function *func* is now scheduled to be called from the subinterpreter, rather than being called from the main interpreter. Each subinterpreter now has its own list of scheduled calls.

9.8 Profiling and Tracing

The Python interpreter provides some low-level support for attaching profiling and execution tracing facilities. These are used for profiling, debugging, and coverage analysis tools.

This C interface allows the profiling or tracing code to avoid the overhead of calling through Python-level callable objects, making a direct C function call instead. The essential attributes of the facility have not changed; the interface allows trace functions to be installed per-thread, and the basic events reported to the trace function are the same as had been reported to the Python-level trace functions in previous versions.

typedef int (***Py_tracefunc**)(*PyObject* *obj, *PyFrameObject* *frame, int what, *PyObject* *arg)

The type of the trace function registered using `PyEval_SetProfile()` and `PyEval_SetTrace()`. The first parameter is the object passed to the registration function as *obj*, *frame* is the frame object to which the event pertains, *what* is one of the constants `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION`, `PyTrace_C_RETURN`, or `PyTrace_OPCODE`, and *arg* depends on the value of *what*:

Value of *what*	Meaning of *arg*
`PyTrace_CALL`	Always `Py_None`.
`PyTrace_EXCEPTION`	Exception information as returned by `sys.exc_info()`.
`PyTrace_LINE`	Always `Py_None`.
`PyTrace_RETURN`	Value being returned to the caller, or `NULL` if caused by an exception.
`PyTrace_C_CALL`	Function object being called.
`PyTrace_C_EXCEPTION`	Function object being called.
`PyTrace_C_RETURN`	Function object being called.
`PyTrace_OPCODE`	Always `Py_None`.

int **PyTrace_CALL**

The value of the *what* parameter to a `Py_tracefunc` function when a new call to a function or method is being reported, or a new entry into a generator. Note that the creation of the iterator for a generator function is not reported as there is no control transfer to the Python bytecode in the corresponding frame.

int **PyTrace_EXCEPTION**

The value of the *what* parameter to a `Py_tracefunc` function when an exception has been raised. The callback function is called with this value for *what* when after any bytecode is processed after which the exception becomes set within the frame being executed. The effect of this is that as exception propagation causes the Python stack to unwind, the callback is called upon return to each frame as the exception propagates. Only trace functions receives these events; they are not needed by the profiler.

int **PyTrace_LINE**

The value passed as the *what* parameter to a `Py_tracefunc` function (but not a profiling function) when a line-number event is being reported. It may be disabled for a frame by setting `f_trace_lines` to *0* on that frame.

int **PyTrace_RETURN**

The value for the *what* parameter to *Py_tracefunc* functions when a call is about to return.

int **PyTrace_C_CALL**

The value for the *what* parameter to *Py_tracefunc* functions when a C function is about to be called.

int **PyTrace_C_EXCEPTION**

The value for the *what* parameter to *Py_tracefunc* functions when a C function has raised an exception.

int **PyTrace_C_RETURN**

The value for the *what* parameter to *Py_tracefunc* functions when a C function has returned.

int **PyTrace_OPCODE**

The value for the *what* parameter to *Py_tracefunc* functions (but not profiling functions) when a new opcode is about to be executed. This event is not emitted by default: it must be explicitly requested by setting `f_trace_opcodes` to *1* on the frame.

void **PyEval_SetProfile**(*Py_tracefunc* func, *PyObject* *obj)

Set the profiler function to *func*. The *obj* parameter is passed to the function as its first parameter, and may be any Python object, or `NULL`. If the profile function needs to maintain state, using a different value for *obj* for each thread provides a convenient and thread-safe place to store it. The profile function is called for all monitored events except *PyTrace_LINE* *PyTrace_OPCODE* and *PyTrace_EXCEPTION*.

See also the `sys.setprofile()` function.

呼叫者必須持有*GIL*。

void **PyEval_SetTrace**(*Py_tracefunc* func, *PyObject* *obj)

Set the tracing function to *func*. This is similar to *PyEval_SetProfile()*, except the tracing function does receive line-number events and per-opcode events, but does not receive any event related to C function objects being called. Any trace function registered using *PyEval_SetTrace()* will not receive *PyTrace_C_CALL*, *PyTrace_C_EXCEPTION* or *PyTrace_C_RETURN* as a value for the *what* parameter.

也請見 `sys.settrace()` 函式。

呼叫者必須持有*GIL*。

9.9 Advanced Debugger Support

These functions are only intended to be used by advanced debugging tools.

PyInterpreterState ***PyInterpreterState_Head**()

Return the interpreter state object at the head of the list of all such objects.

PyInterpreterState ***PyInterpreterState_Main**()

Return the main interpreter state object.

PyInterpreterState ***PyInterpreterState_Next**(*PyInterpreterState* *interp)

Return the next interpreter state object after *interp* from the list of all such objects.

PyThreadState ***PyInterpreterState_ThreadHead**(*PyInterpreterState* *interp)

Return the pointer to the first *PyThreadState* object in the list of threads associated with the interpreter *interp*.

PyThreadState ***PyThreadState_Next**(*PyThreadState* *tstate)

Return the next thread state object after *tstate* from the list of all such objects belonging to the same *PyInterpreterState* object.

9.10 Thread Local Storage Support

The Python interpreter provides low-level support for thread-local storage (TLS) which wraps the underlying native TLS implementation to support the Python-level thread local storage API (`threading.local`). The CPython C level APIs are similar to those offered by pthreads and Windows: use a thread key and functions to associate a `void*` value per thread.

The GIL does *not* need to be held when calling these functions; they supply their own locking.

Note that `Python.h` does not include the declaration of the TLS APIs, you need to include `pythread.h` to use thread-local storage.

備F: None of these API functions handle memory management on behalf of the `void*` values. You need to allocate and deallocate them yourself. If the `void*` values happen to be `PyObject*`, these functions don't do refcount operations on them either.

9.10.1 Thread Specific Storage (TSS) API

TSS API is introduced to supersede the use of the existing TLS API within the CPython interpreter. This API uses a new type `Py_tss_t` instead of `int` to represent thread keys.

在 3.7 版新加入.

也參考:

"A New C-API for Thread-Local Storage in CPython" (**PEP 539**)

type **Py_tss_t**

This data structure represents the state of a thread key, the definition of which may depend on the underlying TLS implementation, and it has an internal field representing the key's initialization state. There are no public members in this structure.

When *Py_LIMITED_API* is not defined, static allocation of this type by `Py_tss_NEEDS_INIT` is allowed.

Py_tss_NEEDS_INIT

This macro expands to the initializer for `Py_tss_t` variables. Note that this macro won't be defined with *Py_LIMITED_API*.

Dynamic Allocation

Dynamic allocation of the `Py_tss_t`, required in extension modules built with *Py_LIMITED_API*, where static allocation of this type is not possible due to its implementation being opaque at build time.

Py_tss_t ***PyThread_tss_alloc**()

Part of the Stable ABI *since version 3.7.* Return a value which is the same state as a value initialized with `Py_tss_NEEDS_INIT`, or `NULL` in the case of dynamic allocation failure.

void **PyThread_tss_free**(*Py_tss_t* *key)

Part of the Stable ABI *since version 3.7.* Free the given *key* allocated by `PyThread_tss_alloc()`, after first calling `PyThread_tss_delete()` to ensure any associated thread locals have been unassigned. This is a no-op if the *key* argument is `NULL`.

備F: A freed key becomes a dangling pointer. You should reset the key to `NULL`.

Methods

The parameter *key* of these functions must not be `NULL`. Moreover, the behaviors of `PyThread_tss_set()` and `PyThread_tss_get()` are undefined if the given `Py_tss_t` has not been initialized by `PyThread_tss_create()`.

int **PyThread_tss_is_created**(*Py_tss_t* *key)

Part of the Stable ABI *since version 3.7.* Return a non-zero value if the given `Py_tss_t` has been initialized by `PyThread_tss_create()`.

int **PyThread_tss_create**(*Py_tss_t* *key)

Part of the Stable ABI *since version 3.7.* Return a zero value on successful initialization of a TSS key. The behavior is undefined if the value pointed to by the *key* argument is not initialized by `Py_tss_NEEDS_INIT`. This function can be called repeatedly on the same key -- calling it on an already initialized key is a no-op and immediately returns success.

void **PyThread_tss_delete**(*Py_tss_t* *key)

Part of the Stable ABI *since version 3.7.* Destroy a TSS key to forget the values associated with the key across all threads, and change the key's initialization state to uninitialized. A destroyed key is able to be initialized again by `PyThread_tss_create()`. This function can be called repeatedly on the same key -- calling it on an already destroyed key is a no-op.

int **PyThread_tss_set**(*Py_tss_t* *key, void *value)

Part of the Stable ABI *since version 3.7.* Return a zero value to indicate successfully associating a `void*` value with a TSS key in the current thread. Each thread has a distinct mapping of the key to a `void*` value.

void ***PyThread_tss_get**(*Py_tss_t* *key)

Part of the Stable ABI *since version 3.7.* Return the `void*` value associated with a TSS key in the current thread. This returns `NULL` if no value is associated with the key in the current thread.

9.10.2 Thread Local Storage (TLS) API

在 3.7 版之後被⾏用: This API is superseded by *Thread Specific Storage (TSS) API*.

備⾏: This version of the API does not support platforms where the native TLS key is defined in a way that cannot be safely cast to `int`. On such platforms, `PyThread_create_key()` will return immediately with a failure status, and the other TLS functions will all be no-ops on such platforms.

Due to the compatibility problem noted above, this version of the API should not be used in new code.

int **PyThread_create_key**()

Part of the Stable ABI.

void **PyThread_delete_key**(int key)

Part of the Stable ABI.

int **PyThread_set_key_value**(int key, void *value)

Part of the Stable ABI.

void ***PyThread_get_key_value**(int key)

Part of the Stable ABI.

void **PyThread_delete_key_value**(int key)

Part of the Stable ABI.

void **PyThread_ReInitTLS**()

Part of the Stable ABI.

CHAPTER 10

Python Initialization Configuration

在 3.8 版新加入.

Python can be initialized with `Py_InitializeFromConfig()` and the `PyConfig` structure. It can be preinitialized with `Py_PreInitialize()` and the `PyPreConfig` structure.

There are two kinds of configuration:

- The *Python Configuration* can be used to build a customized Python which behaves as the regular Python. For example, environment variables and command line arguments are used to configure Python.
- The *Isolated Configuration* can be used to embed Python into an application. It isolates Python from the system. For example, environment variables are ignored, the LC_CTYPE locale is left unchanged and no signal handler is registered.

The `Py_RunMain()` function can be used to write a customized Python program.

See also *Initialization, Finalization, and Threads*.

也參考:

PEP 587 "Python Initialization Configuration".

10.1 范例

Example of customized Python always running in isolated mode:

```
int main(int argc, char **argv)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config.isolated = 1;

    /* Decode command line arguments.
       Implicitly preinitialize Python (in isolated mode). */
    status = PyConfig_SetBytesArgv(&config, argc, argv);
    if (PyStatus_Exception(status)) {
        goto exception;
```

(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**

List of `wchar_t*` strings.

If *length* is non-zero, *items* must be non-`NULL` and all strings must be non-`NULL`.

Methods:

PyStatus **PyWideStringList_Append**(*PyWideStringList* *list, const wchar_t *item)

Append *item* to *list*.

Python must be preinitialized to call this function.

PyStatus **PyWideStringList_Insert**(*PyWideStringList* *list, *Py_ssize_t* index, const wchar_t *item)

Insert *item* into *list* at *index*.

If *index* is greater than or equal to *list* length, append *item* to *list*.

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

Python must be preinitialized to call this function.

Structure fields:

Py_ssize_t **length**

List length.

wchar_t ****items**

List items.

10.3 PyStatus

type **PyStatus**

Structure to store an initialization function status: success, error or exit.

For an error, it can store the C function name which created the error.

Structure fields:

int **exitcode**

Exit code. Argument passed to `exit()`.

const char ***err_msg**

錯誤訊息。

const char ***func**

Name of the function which created an error, can be `NULL`.

Functions to create a status:

PyStatus **PyStatus_Ok**(void)

Success.

PyStatus **PyStatus_Error**(const char *err_msg)

Initialization error with a message.

err_msg 不可F `NULL`。

PyStatus **PyStatus_NoMemory**(void)

Memory allocation failure (out of memory).

PyStatus **PyStatus_Exit**(int exitcode)

Exit Python with the specified exit code.

Functions to handle a status:

int **PyStatus_Exception**(*PyStatus* status)

Is the status an error or an exit? If true, the exception must be handled; by calling `Py_ExitStatusException()` for example.

int **PyStatus_IsError**(*PyStatus* status)

Is the result an error?

int **PyStatus_IsExit**(*PyStatus* status)

Is the result an exit?

void **Py_ExitStatusException**(*PyStatus* status)

Call `exit(exitcode)` if *status* is an exit. Print the error message and exit with a non-zero exit code if *status* is an error. Must only be called if `PyStatus_Exception(status)` is non-zero.

備F: Internally, Python uses macros which set `PyStatus.func`, whereas functions to create a status set `func` to `NULL`.

範例：

```
PyStatus alloc(void **ptr, size_t size)
{
    *ptr = PyMem_RawMalloc(size);
    if (*ptr == NULL) {
        return PyStatus_NoMemory();
    }
```

(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**

Structure used to preinitialize Python.

Function to initialize a preconfiguration:

void **PyPreConfig_InitPythonConfig**(*PyPreConfig* *preconfig)

Initialize the preconfiguration with *Python Configuration*.

void **PyPreConfig_InitIsolatedConfig**(*PyPreConfig* *preconfig)

Initialize the preconfiguration with *Isolated Configuration*.

Structure fields:

int **allocator**

Name of the Python memory allocators:

- `PYMEM_ALLOCATOR_NOT_SET` (`0`): don't change memory allocators (use defaults).
- `PYMEM_ALLOCATOR_DEFAULT` (`1`): *default memory allocators*.
- `PYMEM_ALLOCATOR_DEBUG` (`2`): *default memory allocators* with *debug hooks*.
- `PYMEM_ALLOCATOR_MALLOC` (`3`): use `malloc()` of the C library.
- `PYMEM_ALLOCATOR_MALLOC_DEBUG` (`4`): force usage of `malloc()` with *debug hooks*.
- `PYMEM_ALLOCATOR_PYMALLOC` (`5`): *Python pymalloc memory allocator*.
- `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` (`6`): *Python pymalloc memory allocator* with *debug hooks*.

`PYMEM_ALLOCATOR_PYMALLOC` and `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` are not supported if Python is `configured using --without-pymalloc`.

請見*記憶體管理*。

預設：`PYMEM_ALLOCATOR_NOT_SET`。

int **configure_locale**

Set the LC_CTYPE locale to the user preferred locale.

If equals to `0`, set *coerce_c_locale* and *coerce_c_locale_warn* members to `0`.

請見*locale encoding*。

Default: `1` in Python config, `0` in isolated config.

int **coerce_c_locale**

If equals to 2, coerce the C locale.

If equals to 1, read the LC_CTYPE locale to decide if it should be coerced.

請見*locale encoding*。

Default: -1 in Python config, 0 in isolated config.

int **coerce_c_locale_warn**

If non-zero, emit a warning if the C locale is coerced.

Default: -1 in Python config, 0 in isolated config.

int **dev_mode**

Python Development Mode: see *PyConfig.dev_mode*.

Default: -1 in Python mode, 0 in isolated mode.

int **isolated**

Isolated mode: see *PyConfig.isolated*.

Default: 0 in Python mode, 1 in isolated mode.

int **legacy_windows_fs_encoding**

如果不F 0:

- 將*PyPreConfig.utf8_mode* 設F 0、
- 將*PyConfig.filesystem_encoding* 設F "mbcs"、
- 將*PyConfig.filesystem_errors* 設F "replace"。

Initialized the from PYTHONLEGACYWINDOWSFSENCODING environment variable value.

Only available on Windows. #ifdef MS_WINDOWS macro can be used for Windows specific code.

預設: 0。

int **parse_argv**

If non-zero, *Py_PreInitializeFromArgs()* and *Py_PreInitializeFromBytesArgs()* parse their argv argument the same way the regular Python parses command line arguments: see Command Line Arguments.

Default: 1 in Python config, 0 in isolated config.

int **use_environment**

Use environment variables? See *PyConfig.use_environment*.

Default: 1 in Python config and 0 in isolated config.

int **utf8_mode**

If non-zero, enable the Python UTF-8 Mode.

Set to 0 or 1 by the -X utf8 command line option and the PYTHONUTF8 environment variable.

Also set to 1 if the LC_CTYPE locale is C or POSIX.

Default: -1 in Python config and 0 in isolated config.

10.5 Preinitialize Python with PyPreConfig

The preinitialization of Python:

- Set the Python memory allocators (`PyPreConfig.allocator`)
- Configure the LC_CTYPE locale (*locale encoding*)
- Set the Python UTF-8 Mode (`PyPreConfig.utf8_mode`)

The current preconfiguration (`PyPreConfig` type) is stored in `_PyRuntime.preconfig`.

Functions to preinitialize Python:

PyStatus **`Py_PreInitialize`**(const *PyPreConfig* *preconfig)

Preinitialize Python from *preconfig* preconfiguration.

preconfig 不可F `NULL`。

PyStatus **`Py_PreInitializeFromBytesArgs`**(const *PyPreConfig* *preconfig, int argc, char *const *argv)

Preinitialize Python from *preconfig* preconfiguration.

Parse *argv* command line arguments (bytes strings) if `parse_argv` of *preconfig* is non-zero.

preconfig 不可F `NULL`。

PyStatus **`Py_PreInitializeFromArgs`**(const *PyPreConfig* *preconfig, int argc, wchar_t *const *argv)

Preinitialize Python from *preconfig* preconfiguration.

Parse *argv* command line arguments (wide strings) if `parse_argv` of *preconfig* is non-zero.

preconfig 不可F `NULL`。

The caller is responsible to handle exceptions (error or exit) using `PyStatus_Exception()` and `Py_ExitStatusException()`.

For *Python Configuration* (`PyPreConfig_InitPythonConfig()`), if Python is initialized with command line arguments, the command line arguments must also be passed to preinitialize Python, since they have an effect on the pre-configuration like encodings. For example, the `-X utf8` command line option enables the Python UTF-8 Mode.

`PyMem_SetAllocator()` can be called after `Py_PreInitialize()` and before `Py_InitializeFromConfig()` to install a custom memory allocator. It can be called before `Py_PreInitialize()` if `PyPreConfig.allocator` is set to `PYMEM_ALLOCATOR_NOT_SET`.

Python memory allocation functions like `PyMem_RawMalloc()` must not be used before the Python preinitialization, whereas calling directly `malloc()` and `free()` is always safe. `Py_DecodeLocale()` must not be called before the Python preinitialization.

Example using the preinitialization to enable the Python UTF-8 Mode:

```
PyStatus status;
PyPreConfig preconfig;
PyPreConfig_InitPythonConfig(&preconfig);

preconfig.utf8_mode = 1;

status = Py_PreInitialize(&preconfig);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}

/* at this point, Python speaks UTF-8 */

Py_Initialize();
/* ... use Python API here ... */
Py_Finalize();
```

10.6 PyConfig

type **PyConfig**

Structure containing most parameters to configure Python.

When done, the *PyConfig_Clear()* function must be used to release the configuration memory.

Structure methods:

void **PyConfig_InitPythonConfig**(*PyConfig* *config)

Initialize configuration with the *Python Configuration*.

void **PyConfig_InitIsolatedConfig**(*PyConfig* *config)

Initialize configuration with the *Isolated Configuration*.

PyStatus **PyConfig_SetString**(*PyConfig* *config, wchar_t *const *config_str, const wchar_t *str)

Copy the wide character string *str* into `*config_str`.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesString**(*PyConfig* *config, wchar_t *const *config_str, const char *str)

Decode *str* using *Py_DecodeLocale()* and set the result into `*config_str`.

Preinitialize Python if needed.

PyStatus **PyConfig_SetArgv**(*PyConfig* *config, int argc, wchar_t *const *argv)

Set command line arguments (*argv* member of *config*) from the *argv* list of wide character strings.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesArgv**(*PyConfig* *config, int argc, char *const *argv)

Set command line arguments (*argv* member of *config*) from the *argv* list of bytes strings. Decode bytes using *Py_DecodeLocale()*.

Preinitialize Python if needed.

PyStatus **PyConfig_SetWideStringList**(*PyConfig* *config, *PyWideStringList* *list, *Py_ssize_t* length, wchar_t **items)

Set the list of wide strings *list* to *length* and *items*.

Preinitialize Python if needed.

PyStatus **PyConfig_Read**(*PyConfig* *config)

Read all Python configuration.

Fields which are already initialized are left unchanged.

Fields for *path configuration* are no longer calculated or modified when calling this function, as of Python 3.11.

The *PyConfig_Read()* function only parses *PyConfig.argv* arguments once: *PyConfig.parse_argv* is set to `2` after arguments are parsed. Since Python arguments are strippped from *PyConfig.argv*, parsing arguments twice would parse the application options as Python options.

Preinitialize Python if needed.

在 3.10 版的變更: The *PyConfig.argv* arguments are now only parsed once, *PyConfig.parse_argv* is set to `2` after arguments are parsed, and arguments are only parsed if *PyConfig.parse_argv* equals `1`.

在 3.11 版的變更: *PyConfig_Read()* no longer calculates all paths, and so fields listed under *Python Path Configuration* may no longer be updated until *Py_InitializeFromConfig()* is called.

void **PyConfig_Clear**(*PyConfig* *config)

Release configuration memory.

Most PyConfig methods *preinitialize Python* if needed. In that case, the Python preinitialization configuration (*PyPreConfig*) in based on the *PyConfig*. If configuration fields which are in common with *PyPreConfig* are tuned, they must be set before calling a *PyConfig* method:

- *PyConfig.dev_mode*
- *PyConfig.isolated*
- *PyConfig.parse_argv*
- *PyConfig.use_environment*

Moreover, if *PyConfig_SetArgv()* or *PyConfig_SetBytesArgv()* is used, this method must be called before other methods, since the preinitialization configuration depends on command line arguments (if *parse_argv* is non-zero).

The caller of these methods is responsible to handle exceptions (error or exit) using PyStatus_Exception() and Py_ExitStatusException().

Structure fields:

PyWideStringList **argv**

Command line arguments: sys.argv.

Set *parse_argv* to 1 to parse *argv* the same way the regular Python parses Python command line arguments and then to strip Python arguments from *argv*.

If *argv* is empty, an empty string is added to ensure that sys.argv always exists and is never empty.

預設值：NULL。

See also the *orig_argv* member.

int **safe_path**

If equals to zero, Py_RunMain() prepends a potentially unsafe path to sys.path at startup:

- If *argv[0]* is equal to L"-m" (python -m module), prepend the current working directory.
- If running a script (python script.py), prepend the script's directory. If it's a symbolic link, resolve symbolic links.
- Otherwise (python -c code and python), prepend an empty string, which means the current working directory.

Set to 1 by the -P command line option and the PYTHONSAFEPATH environment variable.

Default: 0 in Python config, 1 in isolated config.

在 3.11 版新加入.

wchar_t ***base_exec_prefix**

sys.base_exec_prefix。

預設值：NULL。

Part of the *Python Path Configuration* output.

wchar_t ***base_executable**

Python base executable: sys._base_executable.

Set by the __PYVENV_LAUNCHER__ environment variable.

Set from *PyConfig.executable* if NULL.

預設值：NULL。

Part of the *Python Path Configuration* output.

wchar_t ***base_prefix**

`sys.base_prefix`。

預設值：`NULL`。

Part of the *Python Path Configuration* output.

int **buffered_stdio**

If equals to `0` and *configure_c_stdio* is non-zero, disable buffering on the C streams stdout and stderr.

Set to `0` by the `-u` command line option and the `PYTHONUNBUFFERED` environment variable.

stdin is always opened in buffered mode.

預設值：`1`。

int **bytes_warning**

If equals to `1`, issue a warning when comparing `bytes` or `bytearray` with `str`, or comparing `bytes` with `int`.

If equal or greater to `2`, raise a `BytesWarning` exception in these cases.

Incremented by the `-b` command line option.

預設：`0`。

int **warn_default_encoding**

If non-zero, emit a `EncodingWarning` warning when `io.TextIOWrapper` uses its default encoding. See io-encoding-warning for details.

預設：`0`。

在 3.10 版新加入.

int **code_debug_ranges**

If equals to `0`, disables the inclusion of the end line and column mappings in code objects. Also disables traceback printing carets to specific error locations.

Set to `0` by the `PYTHONNODEBUGRANGES` environment variable and by the `-X no_debug_ranges` command line option.

預設值：`1`。

在 3.11 版新加入.

wchar_t ***check_hash_pycs_mode**

Control the validation behavior of hash-based `.pyc` files: value of the `--check-hash-based-pycs` command line option.

Valid values:

- `L"always"`: Hash the source file for invalidation regardless of value of the 'check_source' flag.
- `L"never"`: Assume that hash-based pycs always are valid.
- `L"default"`: The 'check_source' flag in hash-based pycs determines invalidation.

預設：`L"default"`。

See also **PEP 552** "Deterministic pycs".

int **configure_c_stdio**

If non-zero, configure C standard streams:

- On Windows, set the binary mode (`O_BINARY`) on stdin, stdout and stderr.
- If *buffered_stdio* equals zero, disable buffering of stdin, stdout and stderr streams.
- If *interactive* is non-zero, enable stream buffering on stdin and stdout (only stdout on Windows).

Default: 1 in Python config, 0 in isolated config.

int **dev_mode**

If non-zero, enable the Python Development Mode.

Set to 1 by the -X dev option and the PYTHONDEVMODE environment variable.

Default: -1 in Python mode, 0 in isolated mode.

int **dump_refs**

Dump Python references?

If non-zero, dump all objects which are still alive at exit.

Set to 1 by the PYTHONDUMPREFS environment variable.

Need a special build of Python with the Py_TRACE_REFS macro defined: see the configure --with-trace-refs option.

預設：0。

wchar_t ***exec_prefix**

The site-specific directory prefix where the platform-dependent Python files are installed: sys.exec_prefix.

預設值：NULL。

Part of the *Python Path Configuration* output.

wchar_t ***executable**

The absolute path of the executable binary for the Python interpreter: sys.executable.

預設值：NULL。

Part of the *Python Path Configuration* output.

int **faulthandler**

Enable faulthandler?

If non-zero, call faulthandler.enable() at startup.

Set to 1 by -X faulthandler and the PYTHONFAULTHANDLER environment variable.

Default: -1 in Python mode, 0 in isolated mode.

wchar_t ***filesystem_encoding**

Filesystem encoding: sys.getfilesystemencoding().

On macOS, Android and VxWorks: use "utf-8" by default.

On Windows: use "utf-8" by default, or "mbcs" if *legacy_windows_fs_encoding* of *PyPreConfig* is non-zero.

Default encoding on other platforms:

- "utf-8" if *PyPreConfig.utf8_mode* is non-zero.
- "ascii" if Python detects that nl_langinfo(CODESET) announces the ASCII encoding, whereas the mbstowcs() function decodes from a different encoding (usually Latin1).
- "utf-8" if nl_langinfo(CODESET) returns an empty string.
- Otherwise, use the *locale encoding*: nl_langinfo(CODESET) result.

At Python startup, the encoding name is normalized to the Python codec name. For example, "ANSI_X3.4-1968" is replaced with "ascii".

See also the *filesystem_errors* member.

wchar_t ***filesystem_errors**

Filesystem error handler: sys.getfilesystemencodeerrors().

On Windows: use "surrogatepass" by default, or "replace" if *legacy_windows_fs_encoding* of *PyPreConfig* is non-zero.

On other platforms: use "surrogateescape" by default.

Supported error handlers:

- "strict"
- "surrogateescape"
- "surrogatepass" (only supported with the UTF-8 encoding)

See also the *filesystem_encoding* member.

unsigned long **hash_seed**

int **use_hash_seed**

Randomized hash function seed.

If *use_hash_seed* is zero, a seed is chosen randomly at Python startup, and *hash_seed* is ignored.

Set by the PYTHONHASHSEED environment variable.

Default *use_hash_seed* value: -1 in Python mode, 0 in isolated mode.

wchar_t ***home**

Python home directory.

If *Py_SetPythonHome()* has been called, use its argument if it is not NULL.

Set by the PYTHONHOME environment variable.

預設值：NULL。

Part of the *Python Path Configuration* input.

int **import_time**

If non-zero, profile import time.

Set the 1 by the -X importtime option and the PYTHONPROFILEIMPORTTIME environment variable.

預設：0。

int **inspect**

Enter interactive mode after executing a script or a command.

If greater than 0, enable inspect: when a script is passed as first argument or the -c option is used, enter interactive mode after executing the script or the command, even when sys.stdin does not appear to be a terminal.

Incremented by the -i command line option. Set to 1 if the PYTHONINSPECT environment variable is non-empty.

預設：0。

int **install_signal_handlers**

Install Python signal handlers?

Default: 1 in Python mode, 0 in isolated mode.

int **interactive**

If greater than 0, enable the interactive mode (REPL).

Incremented by the -i command line option.

預設：0。

int **isolated**

If greater than 0, enable isolated mode:

- Set *safe_path* to 1: don't prepend a potentially unsafe path to sys.path at Python startup.
- 將*use_environment* 設定為 0。
- Set *user_site_directory* to 0: don't add the user site directory to sys.path.
- Python REPL doesn't import readline nor enable default readline configuration on interactive prompts.

Set to 1 by the -I command line option.

Default: 0 in Python mode, 1 in isolated mode.

也請見*PyPreConfig.isolated*。

int **legacy_windows_stdio**

If non-zero, use io.FileIO instead of io._WindowsConsoleIO for sys.stdin, sys.stdout and sys.stderr.

Set to 1 if the PYTHONLEGACYWINDOWSSTDIO environment variable is set to a non-empty string.

Only available on Windows. #ifdef MS_WINDOWS macro can be used for Windows specific code.

預設：0。

See also the **PEP 528** (Change Windows console encoding to UTF-8).

int **malloc_stats**

If non-zero, dump statistics on *Python pymalloc memory allocator* at exit.

Set to 1 by the PYTHONMALLOCSTATS environment variable.

The option is ignored if Python is configured using the --without-pymalloc option.

預設：0。

wchar_t ***platlibdir**

Platform library directory name: sys.platlibdir.

Set by the PYTHONPLATLIBDIR environment variable.

Default: value of the PLATLIBDIR macro which is set by the configure --with-platlibdir option (default: "lib", or "DLLs" on Windows).

Part of the *Python Path Configuration* input.

在 3.9 版新加入.

在 3.11 版的變更: This macro is now used on Windows to locate the standard library extension modules, typically under DLLs. However, for compatibility, note that this value is ignored for any non-standard layouts, including in-tree builds and virtual environments.

wchar_t ***pythonpath_env**

Module search paths (sys.path) as a string separated by DELIM (os.pathsep).

Set by the PYTHONPATH environment variable.

預設值：NULL。

Part of the *Python Path Configuration* input.

PyWideStringList **module_search_paths** int **module_search_paths_set**

Module search paths: sys.path.

If *module_search_paths_set* is equal to 0, *Py_InitializeFromConfig()* will replace *module_search_paths* and sets *module_search_paths_set* to 1.

Default: empty list (module_search_paths) and 0 (module_search_paths_set).

Part of the *Python Path Configuration* output.

int **optimization_level**

Compilation optimization level:

- 0: Peephole optimizer, set __debug__ to True.
- 1: Level 0, remove assertions, set __debug__ to False.
- 2: Level 1, strip docstrings.

Incremented by the -O command line option. Set to the PYTHONOPTIMIZE environment variable value.

預設： 0。

PyWideStringList **orig_argv**

The list of the original command line arguments passed to the Python executable: sys.orig_argv.

If *orig_argv* list is empty and *argv* is not a list only containing an empty string, *PyConfig_Read()* copies *argv* into *orig_argv* before modifying *argv* (if *parse_argv* is non-zero).

See also the *argv* member and the *Py_GetArgcArgv()* function.

Default: empty list.

在 3.10 版新加入.

int **parse_argv**

Parse command line arguments?

If equals to 1, parse *argv* the same way the regular Python parses command line arguments, and strip Python arguments from *argv*.

The *PyConfig_Read()* function only parses *PyConfig.argv* arguments once: *PyConfig.parse_argv* is set to 2 after arguments are parsed. Since Python arguments are strippped from *PyConfig.argv*, parsing arguments twice would parse the application options as Python options.

Default: 1 in Python mode, 0 in isolated mode.

在 3.10 版的變更: The *PyConfig.argv* arguments are now only parsed if *PyConfig.parse_argv* equals to 1.

int **parser_debug**

Parser debug mode. If greater than 0, turn on parser debugging output (for expert only, depending on compilation options).

Incremented by the -d command line option. Set to the PYTHONDEBUG environment variable value.

預設： 0。

int **pathconfig_warnings**

If non-zero, calculation of path configuration is allowed to log warnings into stderr. If equals to 0, suppress these warnings.

Default: 1 in Python mode, 0 in isolated mode.

Part of the *Python Path Configuration* input.

在 3.11 版的變更: Now also applies on Windows.

wchar_t ***prefix**

The site-specific directory prefix where the platform independent Python files are installed: `sys.prefix`.

預設值：NULL。

Part of the *Python Path Configuration* output.

wchar_t ***program_name**

Program name used to initialize *executable* and in early error messages during Python initialization.

- If `Py_SetProgramName()` has been called, use its argument.
- On macOS, use `PYTHONEXECUTABLE` environment variable if set.
- If the `WITH_NEXT_FRAMEWORK` macro is defined, use `__PYVENV_LAUNCHER__` environment variable if set.
- Use `argv[0]` of *argv* if available and non-empty.
- Otherwise, use `L"python"` on Windows, or `L"python3"` on other platforms.

預設值：NULL。

Part of the *Python Path Configuration* input.

wchar_t ***pycache_prefix**

Directory where cached `.pyc` files are written: `sys.pycache_prefix`.

Set by the `-X pycache_prefix=PATH` command line option and the `PYTHONPYCACHEPREFIX` environment variable.

If NULL, `sys.pycache_prefix` is set to `None`.

預設值：NULL。

int **quiet**

Quiet mode. If greater than 0, don't display the copyright and version at Python startup in interactive mode.

Incremented by the `-q` command line option.

預設：0。

wchar_t ***run_command**

Value of the `-c` command line option.

Used by *Py_RunMain()*.

預設值：NULL。

wchar_t ***run_filename**

Filename passed on the command line: trailing command line argument without `-c` or `-m`. It is used by the *Py_RunMain()* function.

For example, it is set to `script.py` by the `python3 script.py arg` command line.

也請見*PyConfig.skip_source_first_line* 選項。

預設值：NULL。

wchar_t ***run_module**

Value of the `-m` command line option.

Used by *Py_RunMain()*.

預設值：NULL。

int **show_ref_count**

Show total reference count at exit?

Set to `1` by `-X showrefcount` command line option.

Need a debug build of Python (the `Py_REF_DEBUG` macro must be defined).

預設：`0`。

int **site_import**

Import the `site` module at startup?

If equal to zero, disable the import of the module site and the site-dependent manipulations of `sys.path` that it entails.

Also disable these manipulations if the `site` module is explicitly imported later (call `site.main()` if you want them to be triggered).

Set to `0` by the `-S` command line option.

`sys.flags.no_site` is set to the inverted value of *`site_import`*.

預設值：`1`。

int **skip_source_first_line**

If non-zero, skip the first line of the *`PyConfig.run_filename`* source.

It allows the usage of non-Unix forms of `#!cmd`. This is intended for a DOS specific hack only.

Set to `1` by the `-x` command line option.

預設：`0`。

wchar_t ***stdio_encoding**

wchar_t ***stdio_errors**

Encoding and encoding errors of `sys.stdin`, `sys.stdout` and `sys.stderr` (but `sys.stderr` always uses `"backslashreplace"` error handler).

If *`Py_SetStandardStreamEncoding()`* has been called, use its *error* and *errors* arguments if they are not `NULL`.

Use the `PYTHONIOENCODING` environment variable if it is non-empty.

Default encoding:

- `"UTF-8"` if *`PyPreConfig.utf8_mode`* is non-zero.
- Otherwise, use the *locale encoding*.

Default error handler:

- On Windows: use `"surrogateescape"`.
- `"surrogateescape"` if *`PyPreConfig.utf8_mode`* is non-zero, or if the LC_CTYPE locale is "C" or "POSIX".
- `"strict"` otherwise.

int **tracemalloc**

Enable tracemalloc?

If non-zero, call `tracemalloc.start()` at startup.

Set by `-X tracemalloc=N` command line option and by the `PYTHONTRACEMALLOC` environment variable.

Default: `-1` in Python mode, `0` in isolated mode.

int **use_environment**

Use environment variables?

If equals to zero, ignore the environment variables.

Set to 0 by the -E environment variable.

Default: 1 in Python config and 0 in isolated config.

int **user_site_directory**

If non-zero, add the user site directory to sys.path.

Set to 0 by the -s and -I command line options.

Set to 0 by the PYTHONNOUSERSITE environment variable.

Default: 1 in Python mode, 0 in isolated mode.

int **verbose**

Verbose mode. If greater than 0, print a message each time a module is imported, showing the place (filename or built-in module) from which it is loaded.

If greater or equal to 2, print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Incremented by the -v command line option.

Set to the PYTHONVERBOSE environment variable value.

預設：0。

PyWideStringList **warnoptions**

Options of the warnings module to build warnings filters, lowest to highest priority: sys.warnoptions.

The warnings module adds sys.warnoptions in the reverse order: the last *PyConfig.warnoptions* item becomes the first item of warnings.filters which is checked first (highest priority).

The -W command line options adds its value to *warnoptions*, it can be used multiple times.

The PYTHONWARNINGS environment variable can also be used to add warning options. Multiple options can be specified, separated by commas (,).

Default: empty list.

int **write_bytecode**

If equal to 0, Python won't try to write .pyc files on the import of source modules.

Set to 0 by the -B command line option and the PYTHONDONTWRITEBYTECODE environment variable.

sys.dont_write_bytecode is initialized to the inverted value of *write_bytecode*.

預設值：1。

PyWideStringList **xoptions**

Values of the -X command line options: sys._xoptions.

Default: empty list.

If *parse_argv* is non-zero, *argv* arguments are parsed the same way the regular Python parses command line arguments, and Python arguments are stripped from *argv*.

The *xoptions* options are parsed to set other options: see the -X command line option.

在 3.9 版的變更: The show_alloc_count field has been removed.

10.7 Initialization with PyConfig

Function to initialize Python:

PyStatus **Py_InitializeFromConfig**(const *PyConfig* *config)

Initialize Python from *config* configuration.

The caller is responsible to handle exceptions (error or exit) using *PyStatus_Exception()* and *Py_ExitStatusException()*.

If *PyImport_FrozenModules()*, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* are used, they must be set or called after Python preinitialization and before the Python initialization. If Python is initialized multiple times, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* must be called before each Python initialization.

The current configuration (PyConfig type) is stored in PyInterpreterState.config.

Example setting the program name:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name. Implicitly preinitialize Python. */
    status = PyConfig_SetString(&config, &config.program_name,
                                L"/path/to/my_program");
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);
    return;

exception:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}
```

More complete example modifying the default configuration, read the configuration, and then override some parameters. Note that since 3.11, many parameters are not calculated until initialization, and so values cannot be read from the configuration structure. Any values set before initialize is called will be left unchanged by initialization:

```
PyStatus init_python(const char *program_name)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name before reading the configuration
       (decode byte string from the locale encoding).

       Implicitly preinitialize Python. */
    status = PyConfig_SetBytesString(&config, &config.program_name,
                                     program_name);
    if (PyStatus_Exception(status)) {
```

(繼續下一頁)

(繼續上一頁)

```
        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 Isolated Configuration

PyPreConfig_InitIsolatedConfig() and *PyConfig_InitIsolatedConfig()* functions create a configuration to isolate Python from the system. For example, to embed Python into an application.

This configuration ignores global configuration variables, environment variables, command line arguments (*PyConfig.argv* is not parsed) and user site directory. The C standard streams (ex: `stdout`) and the LC_CTYPE locale are left unchanged. Signal handlers are not installed.

Configuration files are still used with this configuration to determine paths that are unspecified. Ensure *PyConfig.home* is specified to avoid computing the default path configuration.

10.9 Python Configuration

PyPreConfig_InitPythonConfig() and *PyConfig_InitPythonConfig()* functions create a configuration to build a customized Python which behaves as the regular Python.

Environments variables and command line arguments are used to configure Python, whereas global configuration variables are ignored.

This function enables C locale coercion (**PEP 538**) and Python UTF-8 Mode (**PEP 540**) depending on the LC_CTYPE locale, PYTHONUTF8 and PYTHONCOERCECLOCALE environment variables.

10.10 Python Path Configuration

PyConfig contains multiple fields for the path configuration:

- Path configuration inputs:
  - *PyConfig.home*
  - *PyConfig.platlibdir*
  - *PyConfig.pathconfig_warnings*
  - *PyConfig.program_name*
  - *PyConfig.pythonpath_env*
  - current working directory: to get absolute paths
  - PATH environment variable to get the program full path (from *PyConfig.program_name*)
  - __PYVENV_LAUNCHER__ 環境變數
  - (Windows only) Application paths in the registry under "SoftwarePythonPythonCoreX.YPythonPath" of HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE (where X.Y is the Python version).
- Path configuration output fields:
  - *PyConfig.base_exec_prefix*
  - *PyConfig.base_executable*
  - *PyConfig.base_prefix*
  - *PyConfig.exec_prefix*
  - *PyConfig.executable*
  - *PyConfig.module_search_paths_set*, *PyConfig.module_search_paths*
  - *PyConfig.prefix*

If at least one "output field" is not set, Python calculates the path configuration to fill unset fields. If *module_search_paths_set* is equal to 0, *module_search_paths* is overridden and *module_search_paths_set* is set to 1.

It is possible to completely ignore the function calculating the default path configuration by setting explicitly all path configuration output fields listed above. A string is considered as set even if it is non-empty. module_search_paths is considered as set if module_search_paths_set is set to 1. In this case, module_search_paths will be used without modification.

Set *pathconfig_warnings* to 0 to suppress warnings when calculating the path configuration (Unix only, Windows does not log any warning).

If *base_prefix* or *base_exec_prefix* fields are not set, they inherit their value from *prefix* and *exec_prefix* respectively.

Py_RunMain() and *Py_Main()* modify sys.path:

- If *run_filename* is set and is a directory which contains a `__main__.py` script, prepend *run_filename* to `sys.path`.
- If *isolated* is zero:
  - If *run_module* is set, prepend the current directory to `sys.path`. Do nothing if the current directory cannot be read.
  - If *run_filename* is set, prepend the directory of the filename to `sys.path`.
  - Otherwise, prepend an empty string to `sys.path`.

If *site_import* is non-zero, `sys.path` can be modified by the `site` module. If *user_site_directory* is non-zero and the user's site-package directory exists, the `site` module appends the user's site-package directory to `sys.path`.

The following configuration files are used by the path configuration:

- `pyvenv.cfg`
- `._pth` file (ex: `python._pth`)
- `pybuilddir.txt` (Unix only)

If a `._pth` file is present:

- 將*isolated* 設定F 1。
- 將*use_environment* 設定F 0。
- 將*site_import* 設定F 0。
- 將*safe_path* 設定F 1。

The `__PYVENV_LAUNCHER__` environment variable is used to set *PyConfig.base_executable*

10.11 Py_RunMain()

int **Py_RunMain**(void)

Execute the command (*PyConfig.run_command*), the script (*PyConfig.run_filename*) or the module (*PyConfig.run_module*) specified on the command line or in the configuration.

By default and when if `-i` option is used, run the REPL.

Finally, finalizes Python and returns an exit status that can be passed to the `exit()` function.

See *Python Configuration* for an example of customized Python always running in isolated mode using *Py_RunMain()*.

10.12 Py_GetArgcArgv()

void **Py_GetArgcArgv**(int *argc, wchar_t ***argv)

Get the original command line arguments, before Python modified them.

See also *PyConfig.orig_argv* member.

10.13 Multi-Phase Initialization Private Provisional API

This section is a private provisional API introducing multi-phase initialization, the core feature of **PEP 432**:

- "Core" initialization phase, "bare minimum Python":
  - Builtin types;
  - Builtin exceptions;
  - Builtin and frozen modules;
  - The `sys` module is only partially initialized (ex: `sys.path` doesn't exist yet).
- "Main" initialization phase, Python is fully initialized:
  - Install and configure `importlib`;
  - Apply the *Path Configuration*;
  - Install signal handlers;
  - Finish `sys` module initialization (ex: create `sys.stdout` and `sys.path`);
  - Enable optional features like `faulthandler` and `tracemalloc`;
  - Import the `site` module;
  - etc.

Private provisional API:

- `PyConfig._init_main`: if set to `0`, `Py_InitializeFromConfig()` stops at the "Core" initialization phase.
- `PyConfig._isolated_interpreter`: if non-zero, disallow threads, subprocesses and fork.

PyStatus **_Py_InitializeMain**(void)

Move to the "Main" initialization phase, finish the Python initialization.

No module is imported during the "Core" phase and the `importlib` module is not configured: the *Path Configuration* is only applied during the "Main" phase. It may allow to customize Python in Python to override or tune the *Path Configuration*, maybe install a custom `sys.meta_path` importer or an import hook, etc.

It may become possible to calculatin the *Path Configuration* in Python, after the Core phase and before the Main phase, which is one of the **PEP 432** motivation.

The "Core" phase is not properly defined: what should be and what should not be available at this phase is not specified yet. The API is marked as private and provisional: the API can be modified or even be removed anytime until a proper public API is designed.

Example running Python code between "Core" and "Main" initialization phases:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config._init_main = 0;

    /* ... customize 'config' configuration ... */

    status = Py_InitializeFromConfig(&config);
    PyConfig_Clear(&config);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
```

(繼續下一頁)

(繼續上一頁)

```
    /* Use sys.stderr because sys.stdout is only created
       by _Py_InitializeMain() */
    int res = PyRun_SimpleString(
        "import sys; "
        "print('Run Python code before _Py_InitializeMain', "
               "file=sys.stderr)");
    if (res < 0) {
        exit(1);
    }

    /* ... put more configuration code here ... */

    status = _Py_InitializeMain();
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
}
```

CHAPTER 11

記憶體管理

11.1 總覽

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the *Python memory manager*. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

To avoid memory corruption, extension writers should never try to operate on Python objects with the functions exported by the C library: `malloc()`, `calloc()`, `realloc()` and `free()`. This will result in mixed calls between the C allocator and the Python memory manager with fatal consequences, because they implement different algorithms and operate on different heaps. However, one may safely allocate and release memory blocks with the C library allocator for individual purposes, as shown in the following example:

```
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyBytes_FromString(buf);
free(buf); /* malloc'ed */
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the bytes object returned as a result.

In most situations, however, it is recommended to allocate memory from the Python heap specifically because the latter is under control of the Python memory manager. For example, this is required when the interpreter is extended with new object types written in C. Another reason for using the Python heap is the desire to *inform* the Python memory manager about the memory needs of the extension module. Even when the requested memory is used exclusively for internal, highly specific purposes, delegating all memory requests to the Python memory manager causes the interpreter to have a more accurate image of its memory footprint as a whole. Consequently, under certain circumstances, the Python memory manager may or may not trigger appropriate actions, like garbage collection, memory compaction or other preventive procedures. Note that by using the C library allocator as shown in the previous example, the allocated memory for the I/O buffer escapes completely the Python memory manager.

也參考:

The `PYTHONMALLOC` environment variable can be used to configure the memory allocators used by Python.

The `PYTHONMALLOCSTATS` environment variable can be used to print statistics of the *pymalloc memory allocator* every time a new pymalloc object arena is created, and on shutdown.

11.2 Allocator Domains

All allocating functions belong to one of three different "domains" (see also `PyMemAllocatorDomain`). These domains represent different allocation strategies and are optimized for different purposes. The specific details on how every domain allocates memory or what internal functions each domain calls is considered an implementation detail, but for debugging purposes a simplified table can be found at *here*. There is no hard requirement to use the memory returned by the allocation functions belonging to a given domain for only the purposes hinted by that domain (although this is the recommended practice). For example, one could use the memory returned by `PyMem_RawMalloc()` for allocating Python objects or the memory returned by `PyObject_Malloc()` for allocating memory for buffers.

The three allocation domains are:

- Raw domain: intended for allocating memory for general-purpose memory buffers where the allocation *must* go to the system allocator or where the allocator can operate without the *GIL*. The memory is requested directly to the system.
- "Mem" domain: intended for allocating memory for Python buffers and general-purpose memory buffers where the allocation must be performed with the *GIL* held. The memory is taken from the Python private heap.
- Object domain: intended for allocating memory belonging to Python objects. The memory is taken from the Python private heap.

When freeing memory previously allocated by the allocating functions belonging to a given domain,the matching specific deallocating functions must be used. For example, `PyMem_Free()` must be used to free memory allocated using `PyMem_Malloc()`.

11.3 Raw Memory Interface

The following function sets are wrappers to the system allocator. These functions are thread-safe, the *GIL* does not need to be held.

The *default raw memory allocator* uses the following functions: `malloc()`, `calloc()`, `realloc()` and `free()`; call `malloc(1)` (or `calloc(1, 1)`) when requesting zero bytes.

在 3.4 版新加入.

void ***PyMem_RawMalloc**(size_t n)

Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawMalloc(1)` had been called instead. The memory will not have been initialized in any way.

void *`PyMem_RawCalloc`(size_t nelem, size_t elsize)

Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawCalloc(1, 1)` had been called instead.

在 3.5 版新加入.

void *`PyMem_RawRealloc`(void *p, size_t n)

Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyMem_RawMalloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to `PyMem_RawMalloc()`, `PyMem_RawRealloc()` or `PyMem_RawCalloc()`.

If the request fails, `PyMem_RawRealloc()` returns `NULL` and *p* remains a valid pointer to the previous memory area.

void `PyMem_RawFree`(void *p)

Frees the memory block pointed to by *p*, which must have been returned by a previous call to `PyMem_RawMalloc()`, `PyMem_RawRealloc()` or `PyMem_RawCalloc()`. Otherwise, or if `PyMem_RawFree(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

11.4 Memory Interface

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

The *default memory allocator* uses the *pymalloc memory allocator*.

警告: The *GIL* must be held when using these functions.

在 3.6 版的變更: The default allocator is now pymalloc instead of system `malloc()`.

void *`PyMem_Malloc`(size_t n)

Part of the Stable ABI. Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void *`PyMem_Calloc`(size_t nelem, size_t elsize)

Part of the Stable ABI *since version 3.7*. Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_Calloc(1, 1)` had been called instead.

在 3.5 版新加入.

void *`PyMem_Realloc`(void *p, size_t n)

Part of the Stable ABI. Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is NULL, the call is equivalent to PyMem_Malloc(n); else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-NULL.

Unless *p* is NULL, it must have been returned by a previous call to *PyMem_Malloc()*, *PyMem_Realloc()* or *PyMem_Calloc()*.

If the request fails, *PyMem_Realloc()* returns NULL and *p* remains a valid pointer to the previous memory area.

void **PyMem_Free**(void *p)

Part of the Stable ABI. Frees the memory block pointed to by *p*, which must have been returned by a previous call to *PyMem_Malloc()*, *PyMem_Realloc()* or *PyMem_Calloc()*. Otherwise, or if PyMem_Free(p) has been called before, undefined behavior occurs.

If *p* is NULL, no operation is performed.

The following type-oriented macros are provided for convenience. Note that *TYPE* refers to any C type.

PyMem_New(TYPE, n)

Same as *PyMem_Malloc()*, but allocates (n * sizeof(TYPE)) bytes of memory. Returns a pointer cast to TYPE*. The memory will not have been initialized in any way.

PyMem_Resize(p, TYPE, n)

Same as *PyMem_Realloc()*, but the memory block is resized to (n * sizeof(TYPE)) bytes. Returns a pointer cast to TYPE*. On return, *p* will be a pointer to the new memory area, or NULL in the event of failure.

This is a C preprocessor macro; *p* is always reassigned. Save the original value of *p* to avoid losing memory when handling errors.

void **PyMem_Del**(void *p)

和*PyMem_Free()* 相同。

In addition, the following macro sets are provided for calling the Python memory allocator directly, without involving the C API functions listed above. However, note that their use does not preserve binary compatibility across Python versions and is therefore deprecated in extension modules.

- PyMem_MALLOC(size)
- PyMem_NEW(type, size)
- PyMem_REALLOC(ptr, size)
- PyMem_RESIZE(ptr, type, size)
- PyMem_FREE(ptr)
- PyMem_DEL(ptr)

11.5 Object allocators

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

備註: There is no guarantee that the memory returned by these allocators can be successfully cast to a Python object when intercepting the allocating functions in this domain by the methods described in the *Customize Memory Allocators* section.

The *default object allocator* uses the *pymalloc memory allocator*.

警告: The *GIL* must be held when using these functions.

void *`PyObject_Malloc`(size_t n)

Part of the Stable ABI. Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void *`PyObject_Calloc`(size_t nelem, size_t elsize)

Part of the Stable ABI *since version 3.7.* Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Calloc(1, 1)` had been called instead.

在 3.5 版新加入.

void *`PyObject_Realloc`(void *p, size_t n)

Part of the Stable ABI. Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyObject_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to `PyObject_Malloc()`, `PyObject_Realloc()` or `PyObject_Calloc()`.

If the request fails, `PyObject_Realloc()` returns `NULL` and *p* remains a valid pointer to the previous memory area.

void `PyObject_Free`(void *p)

Part of the Stable ABI. Frees the memory block pointed to by *p*, which must have been returned by a previous call to `PyObject_Malloc()`, `PyObject_Realloc()` or `PyObject_Calloc()`. Otherwise, or if `PyObject_Free(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

11.6 Default Memory Allocators

Default memory allocators:

Configuration	Name	PyMem_RawMalloc	PyMem_Malloc	PyObject_Malloc
Release build	`"pymalloc"`	`malloc`	`pymalloc`	`pymalloc`
Debug build	`"pymalloc_debug"`	`malloc` + debug	`pymalloc` + debug	`pymalloc` + debug
Release build, without pymalloc	`"malloc"`	`malloc`	`malloc`	`malloc`
Debug build, without pymalloc	`"malloc_debug"`	`malloc` + debug	`malloc` + debug	`malloc` + debug

Legend:

- Name: value for `PYTHONMALLOC` environment variable.
- `malloc`: system allocators from the standard C library, C functions: `malloc()`, `calloc()`, `realloc()` and `free()`.
- `pymalloc`: *pymalloc memory allocator*.
- "+ debug": with *debug hooks on the Python memory allocators*.

- ”Debug build”: Python build in debug mode.

11.7 Customize Memory Allocators

在 3.4 版新加入.

type **PyMemAllocatorEx**

Structure used to describe a memory block allocator. The structure has the following fields:

Field	Meaning
`void *ctx`	user context passed as first argument
`void* malloc(void *ctx, size_t size)`	allocate a memory block
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	allocate a memory block initialized with zeros
`void* realloc(void *ctx, void *ptr, size_t new_size)`	allocate or resize a memory block
`void free(void *ctx, void *ptr)`	free a memory block

在 3.5 版的變更: The `PyMemAllocator` structure was renamed to *PyMemAllocatorEx* and a new `calloc` field was added.

type **PyMemAllocatorDomain**

Enum used to identify an allocator domain. Domains:

PYMEM_DOMAIN_RAW

函式:

- *PyMem_RawMalloc()*
- *PyMem_RawRealloc()*
- *PyMem_RawCalloc()*
- *PyMem_RawFree()*

PYMEM_DOMAIN_MEM

函式:

- *PyMem_Malloc()*,
- *PyMem_Realloc()*
- *PyMem_Calloc()*
- *PyMem_Free()*

PYMEM_DOMAIN_OBJ

函式:

- *PyObject_Malloc()*
- *PyObject_Realloc()*
- *PyObject_Calloc()*
- *PyObject_Free()*

void **PyMem_GetAllocator**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

Get the memory block allocator of the specified domain.

void **PyMem_SetAllocator**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

Set the memory block allocator of the specified domain.

The new allocator must return a distinct non-NULL pointer when requesting zero bytes.

For the *PYMEM_DOMAIN_RAW* domain, the allocator must be thread-safe: the *GIL* is not held when the allocator is called.

If the new allocator is not a hook (does not call the previous allocator), the *PyMem_SetupDebugHooks()* function must be called to reinstall the debug hooks on top on the new allocator.

See also *PyPreConfig.allocator* and *Preinitialize Python with PyPreConfig*.

> **警告:** *PyMem_SetAllocator()* does have the following contract:
>
> - It can be called after *Py_PreInitialize()* and before *Py_InitializeFromConfig()* to install a custom memory allocator. There are no restrictions over the installed allocator other than the ones imposed by the domain (for instance, the Raw Domain allows the allocator to be called without the GIL held). See *the section on allocator domains* for more information.
> - If called after Python has finish initializing (after *Py_InitializeFromConfig()* has been called) the allocator **must** wrap the existing allocator. Substituting the current allocator for some other arbitrary one is **not supported**.

void **PyMem_SetupDebugHooks**(void)

Setup *debug hooks in the Python memory allocators* to detect memory errors.

11.8 Debug hooks on the Python memory allocators

When Python is built in debug mode, the *PyMem_SetupDebugHooks()* function is called at the *Python preinitialization* to setup debug hooks on Python memory allocators to detect memory errors.

The PYTHONMALLOC environment variable can be used to install debug hooks on a Python compiled in release mode (ex: PYTHONMALLOC=debug).

The *PyMem_SetupDebugHooks()* function can be used to set debug hooks after calling *PyMem_SetAllocator()*.

These debug hooks fill dynamically allocated memory blocks with special, recognizable bit patterns. Newly allocated memory is filled with the byte 0xCD (PYMEM_CLEANBYTE), freed memory is filled with the byte 0xDD (PYMEM_DEADBYTE). Memory blocks are surrounded by "forbidden bytes" filled with the byte 0xFD (PYMEM_FORBIDDENBYTE). Strings of these bytes are unlikely to be valid addresses, floats, or ASCII strings.

Runtime checks:

- Detect API violations. For example, detect if *PyObject_Free()* is called on a memory block allocated by *PyMem_Malloc()*.
- Detect write before the start of the buffer (buffer underflow).
- Detect write after the end of the buffer (buffer overflow).
- Check that the *GIL* is held when allocator functions of *PYMEM_DOMAIN_OBJ* (ex: *PyObject_Malloc()*) and *PYMEM_DOMAIN_MEM* (ex: *PyMem_Malloc()*) domains are called.

On error, the debug hooks use the tracemalloc module to get the traceback where a memory block was allocated. The traceback is only displayed if tracemalloc is tracing Python memory allocations and the memory block was traced.

Let S = sizeof(size_t). 2*S bytes are added at each end of each block of N bytes requested. The memory layout is like so, where p represents the address returned by a malloc-like or realloc-like function (p[i:j] means the slice of bytes from *(p+i) inclusive up to *(p+j) exclusive; note that the treatment of negative indices differs from a Python slice):

p[-2*S:-S]
Number of bytes originally asked for. This is a size_t, big-endian (easier to read in a memory dump).

p[-S]
API identifier (ASCII character):

- 'r' for *PYMEM_DOMAIN_RAW*.
- 'm' for *PYMEM_DOMAIN_MEM*.
- 'o' for *PYMEM_DOMAIN_OBJ*.

p[-S+1:0]
Copies of PYMEM_FORBIDDENBYTE. Used to catch under- writes and reads.

p[0:N]
The requested memory, filled with copies of PYMEM_CLEANBYTE, used to catch reference to uninitialized memory. When a realloc-like function is called requesting a larger memory block, the new excess bytes are also filled with PYMEM_CLEANBYTE. When a free-like function is called, these are overwritten with PYMEM_DEADBYTE, to catch reference to freed memory. When a realloc- like function is called requesting a smaller memory block, the excess old bytes are also filled with PYMEM_DEADBYTE.

p[N:N+S]
Copies of PYMEM_FORBIDDENBYTE. Used to catch over- writes and reads.

p[N+S:N+2*S]
Only used if the PYMEM_DEBUG_SERIALNO macro is defined (not defined by default).

A serial number, incremented by 1 on each call to a malloc-like or realloc-like function. Big-endian size_t. If "bad memory" is detected later, the serial number gives an excellent way to set a breakpoint on the next run, to capture the instant at which this block was passed out. The static function bumpserialno() in obmalloc.c is the only place the serial number is incremented, and exists so you can set such a breakpoint easily.

A realloc-like or free-like function first checks that the PYMEM_FORBIDDENBYTE bytes at each end are intact. If they've been altered, diagnostic output is written to stderr, and the program is aborted via Py_FatalError(). The other main failure mode is provoking a memory error when a program reads up one of the special bit patterns and tries to use it as an address. If you get in a debugger then and look at the object, you're likely to see that it's entirely filled with PYMEM_DEADBYTE (meaning freed memory is getting used) or PYMEM_CLEANBYTE (meaning uninitialized memory is getting used).

在 3.6 版的變更: The *PyMem_SetupDebugHooks()* function now also works on Python compiled in release mode. On error, the debug hooks now use tracemalloc to get the traceback where a memory block was allocated. The debug hooks now also check if the GIL is held when functions of *PYMEM_DOMAIN_OBJ* and *PYMEM_DOMAIN_MEM* domains are called.

在 3.8 版的變更: Byte patterns 0xCB (PYMEM_CLEANBYTE), 0xDB (PYMEM_DEADBYTE) and 0xFB (PYMEM_FORBIDDENBYTE) have been replaced with 0xCD, 0xDD and 0xFD to use the same values than Windows CRT debug malloc() and free().

11.9 The pymalloc allocator

Python has a *pymalloc* allocator optimized for small objects (smaller or equal to 512 bytes) with a short lifetime. It uses memory mappings called "arenas" with a fixed size of either 256 KiB on 32-bit platforms or 1 MiB on 64-bit platforms. It falls back to *PyMem_RawMalloc()* and *PyMem_RawRealloc()* for allocations larger than 512 bytes.

pymalloc is the *default allocator* of the *PYMEM_DOMAIN_MEM* (ex: *PyMem_Malloc()*) and *PYMEM_DOMAIN_OBJ* (ex: *PyObject_Malloc()*) domains.

The arena allocator uses the following functions:

- VirtualAlloc() and VirtualFree() on Windows,
- mmap() and munmap() if available,

- `malloc()` and `free()` otherwise.

This allocator is disabled if Python is configured with the `--without-pymalloc` option. It can also be disabled at runtime using the `PYTHONMALLOC` environment variable (ex: `PYTHONMALLOC=malloc`).

11.9.1 Customize pymalloc Arena Allocator

在 3.4 版新加入.

type **PyObjectArenaAllocator**

Structure used to describe an arena allocator. The structure has three fields:

Field	Meaning
`void *ctx`	user context passed as first argument
`void* alloc(void *ctx, size_t size)`	allocate an arena of size bytes
`void free(void *ctx, void *ptr, size_t size)`	free an arena

void **PyObject_GetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

Get the arena allocator.

void **PyObject_SetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

Set the arena allocator.

11.10 tracemalloc C API

在 3.7 版新加入.

int **PyTraceMalloc_Track**(unsigned int domain, uintptr_t ptr, size_t size)

Track an allocated memory block in the `tracemalloc` module.

Return `0` on success, return `-1` on error (failed to allocate memory to store the trace). Return `-2` if tracemalloc is disabled.

If memory block is already tracked, update the existing trace.

int **PyTraceMalloc_Untrack**(unsigned int domain, uintptr_t ptr)

Untrack an allocated memory block in the `tracemalloc` module. Do nothing if the block was not tracked.

Return `-2` if tracemalloc is disabled, otherwise return `0`.

11.11 范例

Here is the example from section 總覽, rewritten so that the I/O buffer is allocated from the Python heap by using the first function set:

```
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

The same code using the type-oriented function set:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

Note that in the two examples above, the buffer is always manipulated via functions belonging to the same set. Indeed, it is required to use the same memory API family for a given memory block, so that the risk of mixing different allocators is reduced to a minimum. The following code sequence contains two errors, one of which is labeled as *fatal* because it mixes two different allocators operating on different heaps.

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);  /* Wrong -- should be PyMem_Free() */
free(buf2);       /* Right -- allocated via malloc() */
free(buf1);       /* Fatal -- should be PyMem_Del()  */
```

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with `PyObject_New`, `PyObject_NewVar` and `PyObject_Del()`.

These will be explained in the next chapter on defining and implementing new object types in C.

CHAPTER 12

Object Implementation Support

This chapter describes the functions, types, and macros used when defining new object types.

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。*Part of the* Stable ABI. 用它的型別和初始參照來初始化新分配物件 *op*。已初始化的物件會被回傳。如果 *type* 表示了該物件參與迴圈垃圾檢查器，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。*Part of the* Stable ABI. 它會做到`PyObject_Init()` 的所有功能，並且會初始化一個大小可變物件的長度資訊。

PyObject_New(TYPE, typeobj)

Allocate a new Python object using the C structure type *TYPE* and the Python type object *typeobj* (`PyTypeObject*`). Fields not defined by the Python object header are not initialized. The caller will own the only reference to the object (i.e. its reference count will be one). The size of the memory allocation is determined from the `tp_basicsize` field of the type object.

PyObject_NewVar(TYPE, typeobj, size)

Allocate a new Python object using the C structure type *TYPE* and the Python type object *typeobj* (`PyTypeObject*`). Fields not defined by the Python object header are not initialized. The allocated memory allows for the *TYPE* structure plus *size* (`Py_ssize_t`) fields of the size given by the `tp_itemsize` field of *typeobj*. This is useful for implementing objects like tuples, which are able to determine their size at construction time. Embedding the array of fields into the same allocation decreases the number of allocations, improving the memory management efficiency.

void **PyObject_Del**(void *op)

Releases memory allocated to an object using `PyObject_New` or `PyObject_NewVar`. This is normally called from the *tp_dealloc* handler specified in the object's type. The fields of the object should not be accessed after this call as the memory is no longer a valid Python object.

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

There are a large number of structures which are used in the definition of object types for Python. This section describes these structures and how they are used.

12.2.1 Base object types and macros

All Python objects ultimately share a small number of fields at the beginning of the object's representation in memory. These are represented by the *PyObject* and *PyVarObject* types, which are defined, in turn, by the expansions of some macros also used, whether directly or indirectly, in the definition of all other Python objects.

type **PyObject**

Part of the Limited API. *(Only some members are part of the stable ABI.)* All object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In a normal "release" build, it contains only the object's reference count and a pointer to the corresponding type object. Nothing is actually declared to be a *PyObject*, but every pointer to a Python object can be cast to a *PyObject**. Access to the members must be done by using the macros *Py_REFCNT* and *Py_TYPE*.

type **PyVarObject**

Part of the Limited API. *(Only some members are part of the stable ABI.)* This is an extension of *PyObject* that adds the *ob_size* field. This is only used for objects that have some notion of *length*. This type does not often appear in the Python/C API. Access to the members must be done by using the macros *Py_REFCNT*, *Py_TYPE*, and *Py_SIZE*.

PyObject_HEAD

This is a macro used when declaring new types which represent objects without a varying length. The PyObject_HEAD macro expands to:

```
PyObject ob_base;
```

See documentation of *PyObject* above.

PyObject_VAR_HEAD

This is a macro used when declaring new types which represent objects with a length that varies from instance to instance. The PyObject_VAR_HEAD macro expands to:

```
PyVarObject ob_base;
```

See documentation of *PyVarObject* above.

int **Py_Is**(*PyObject* *x, *PyObject* *y)

Part of the Stable ABI *since version 3.10.* Test if the *x* object is the *y* object, the same as `x is y` in Python.

在 3.10 版新加入.

int **Py_IsNone**(*PyObject* *x)

Part of the Stable ABI *since version 3.10.* Test if an object is the `None` singleton, the same as `x is None` in Python.

在 3.10 版新加入.

int **Py_IsTrue**(*PyObject* *x)

Part of the Stable ABI *since version 3.10.* Test if an object is the `True` singleton, the same as `x is True` in Python.

在 3.10 版新加入.

int **Py_IsFalse**(*PyObject* *x)

Part of the Stable ABI *since version 3.10.* Test if an object is the `False` singleton, the same as `x is False` in Python.

在 3.10 版新加入.

PyTypeObject ***Py_TYPE**(*PyObject* *o)

Get the type of the Python object *o*.

Return a *borrowed reference*.

Use the `Py_SET_TYPE()` function to set an object type.

在 3.11 版的變更: `Py_TYPE()` is changed to an inline static function. The parameter type is no longer `const PyObject*`.

int **Py_IS_TYPE**(*PyObject* *o, *PyTypeObject* *type)

Return non-zero if the object *o* type is *type*. Return zero otherwise. Equivalent to: `Py_TYPE(o) == type`.

在 3.9 版新加入.

void **Py_SET_TYPE**(*PyObject* *o, *PyTypeObject* *type)

Set the object *o* type to *type*.

在 3.9 版新加入.

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

Get the reference count of the Python object *o*.

Use the `Py_SET_REFCNT()` function to set an object reference count.

在 3.11 版的變更: The parameter type is no longer `const PyObject*`.

在 3.10 版的變更: `Py_REFCNT()` is changed to the inline static function.

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

Set the object *o* reference counter to *refcnt*.

在 3.9 版新加入.

Py_ssize_t **Py_SIZE**(*PyVarObject* *o)

Get the size of the Python object *o*.

Use the `Py_SET_SIZE()` function to set an object size.

在 3.11 版的變更: `Py_SIZE()` is changed to an inline static function. The parameter type is no longer `const PyVarObject*`.

void **Py_SET_SIZE**(*PyVarObject* *o, *Py_ssize_t* size)

Set the object *o* size to *size*.

在 3.9 版新加入.

PyObject_HEAD_INIT (type)

This is a macro which expands to initialization values for a new *PyObject* type. This macro expands to:

```
_PyObject_EXTRA_INIT
1, type,
```

PyVarObject_HEAD_INIT (type, size)

This is a macro which expands to initialization values for a new *PyVarObject* type, including the *ob_size* field. This macro expands to:

```
_PyObject_EXTRA_INIT
1, type, size,
```

12.2.2 Implementing functions and methods

type **PyCFunction**

Part of the Stable ABI. Type of the functions used to implement most Python callables in C. Functions of this type take two *PyObject** parameters and return one such value. If the return value is NULL, an exception shall have been set. If not NULL, the return value is interpreted as the return value of the function as exposed in Python. The function must return a new reference.

The function signature is:

```
PyObject *PyCFunction(PyObject *self,
                      PyObject *args);
```

type **PyCFunctionWithKeywords**

Part of the Stable ABI. Type of the functions used to implement Python callables in C with signature *METH_VARARGS | METH_KEYWORDS*. The function signature is:

```
PyObject *PyCFunctionWithKeywords(PyObject *self,
                                  PyObject *args,
                                  PyObject *kwargs);
```

type **_PyCFunctionFast**

Type of the functions used to implement Python callables in C with signature *METH_FASTCALL*. The function signature is:

```
PyObject *_PyCFunctionFast(PyObject *self,
                           PyObject *const *args,
                           Py_ssize_t nargs);
```

type **_PyCFunctionFastWithKeywords**

Type of the functions used to implement Python callables in C with signature *METH_FASTCALL | METH_KEYWORDS*. The function signature is:

```
PyObject *_PyCFunctionFastWithKeywords(PyObject *self,
                                       PyObject *const *args,
                                       Py_ssize_t nargs,
                                       PyObject *kwnames);
```

type **PyCMethod**

Type of the functions used to implement Python callables in C with signature *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*. The function signature is:

```
PyObject *PyCMethod(PyObject *self,
                    PyTypeObject *defining_class,
                    PyObject *const *args,
```

(繼續下一頁)

(繼續上一頁)

```
                    Py_ssize_t nargs,
                    PyObject *kwnames)
```

在 3.9 版新加入.

type **PyMethodDef**

Part of the Stable ABI *(including all members).* Structure used to describe a method of an extension type. This structure has four fields:

const char ***ml_name**

Name of the method.

PyCFunction **ml_meth**

Pointer to the C implementation.

int **ml_flags**

Flags bits indicating how the call should be constructed.

const char ***ml_doc**

Points to the contents of the docstring.

The `ml_meth` is a C function pointer. The functions may be of different types, but they always return `PyObject*`. If the function is not of the `PyCFunction`, the compiler will require a cast in the method table. Even though `PyCFunction` defines the first parameter as `PyObject*`, it is common that the method implementation uses the specific C type of the *self* object.

The `ml_flags` field is a bitfield which can include the following flags. The individual flags indicate either a calling convention or a binding convention.

There are these calling conventions:

METH_VARARGS

This is the typical calling convention, where the methods have the type `PyCFunction`. The function expects two `PyObject*` values. The first one is the *self* object for methods; for module functions, it is the module object. The second parameter (often called *args*) is a tuple object representing all arguments. This parameter is typically processed using `PyArg_ParseTuple()` or `PyArg_UnpackTuple()`.

METH_KEYWORDS

Can only be used in certain combinations with other flags: *METH_VARARGS* | *METH_KEYWORDS*, *METH_FASTCALL* | *METH_KEYWORDS* and *METH_METHOD* | *METH_FASTCALL* | *METH_KEYWORDS*.

METH_VARARGS* | *METH_KEYWORDS

Methods with these flags must be of type `PyCFunctionWithKeywords`. The function expects three parameters: *self*, *args*, *kwargs* where *kwargs* is a dictionary of all the keyword arguments or possibly `NULL` if there are no keyword arguments. The parameters are typically processed using `PyArg_ParseTupleAndKeywords()`.

METH_FASTCALL

Fast calling convention supporting only positional arguments. The methods have the type `_PyCFunctionFast`. The first parameter is *self*, the second parameter is a C array of `PyObject*` values indicating the arguments and the third parameter is the number of arguments (the length of the array).

在 3.7 版新加入.

在 3.10 版的變更: `METH_FASTCALL` is now part of the *stable ABI*.

METH_FASTCALL* | *METH_KEYWORDS

Extension of `METH_FASTCALL` supporting also keyword arguments, with methods of type `_PyCFunctionFastWithKeywords`. Keyword arguments are passed the same way as in the *vectorcall protocol*: there is an additional fourth `PyObject*` parameter which is a tuple representing the names of the keyword arguments (which are guaranteed to be strings) or possibly `NULL` if there are no keywords. The values of the keyword arguments are stored in the *args* array, after the positional arguments.

在 3.7 版新加入.

METH_METHOD

Can only be used in the combination with other flags: *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*.

METH_METHOD* | *METH_FASTCALL* | *METH_KEYWORDS

Extension of *METH_FASTCALL | METH_KEYWORDS* supporting the *defining class*, that is, the class that contains the method in question. The defining class might be a superclass of `Py_TYPE(self)`.

The method needs to be of type `PyCMethod`, the same as for `METH_FASTCALL | METH_KEYWORDS` with `defining_class` argument added after `self`.

在 3.9 版新加入.

METH_NOARGS

Methods without parameters don't need to check whether arguments are given if they are listed with the `METH_NOARGS` flag. They need to be of type `PyCFunction`. The first parameter is typically named *self* and will hold a reference to the module or object instance. In all cases the second parameter will be `NULL`.

The function must have 2 parameters. Since the second parameter is unused, `Py_UNUSED` can be used to prevent a compiler warning.

METH_O

Methods with a single object argument can be listed with the `METH_O` flag, instead of invoking `PyArg_ParseTuple()` with a `"O"` argument. They have the type `PyCFunction`, with the *self* parameter, and a `PyObject*` parameter representing the single argument.

These two constants are not used to indicate the calling convention but the binding when use with methods of classes. These may not be used for functions defined for modules. At most one of these flags may be set for any given method.

METH_CLASS

The method will be passed the type object as the first parameter rather than an instance of the type. This is used to create *class methods*, similar to what is created when using the `classmethod()` built-in function.

METH_STATIC

The method will be passed `NULL` as the first parameter rather than an instance of the type. This is used to create *static methods*, similar to what is created when using the `staticmethod()` built-in function.

One other constant controls whether a method is loaded in place of another definition with the same method name.

METH_COEXIST

The method will be loaded in place of existing definitions. Without *METH_COEXIST*, the default is to skip repeated definitions. Since slot wrappers are loaded before the method table, the existence of a *sq_contains* slot, for example, would generate a wrapped method named `__contains__()` and preclude the loading of a corresponding PyCFunction with the same name. With the flag defined, the PyCFunction will be loaded in place of the wrapper object and will co-exist with the slot. This is helpful because calls to PyCFunctions are optimized more than wrapper object calls.

PyObject ***PyCMethod_New**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module, *PyTypeObject* *cls)

回傳值：新的參照。 *Part of the* Stable ABI *since version 3.9.* Turn *ml* into a Python *callable* object. The caller must ensure that *ml* outlives the *callable*. Typically, *ml* is defined as a static variable.

The *self* parameter will be passed as the *self* argument to the C function in `ml->ml_meth` when invoked. *self* can be `NULL`.

The *callable* object's `__module__` attribute can be set from the given *module* argument. *module* should be a Python string, which will be used as name of the module the function is defined in. If unavailable, it can be set to `None` or `NULL`.

也參考:

`function.__module__`

The *cls* parameter will be passed as the *defining_class* argument to the C function. Must be set if `METH_METHOD` is set on `ml->ml_flags`.

在 3.9 版新加入.

PyObject ***PyCFunction_NewEx**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module)

回傳值: 新的參照。 *Part of the* Stable ABI. Equivalent to `PyCMethod_New(ml, self, module, NULL)`.

PyObject ***PyCFunction_New**(*PyMethodDef* *ml, *PyObject* *self)

回傳值:新的參照。*Part of the* Stable ABI *since version 3.4.* Equivalent to `PyCMethod_New(ml, self, NULL, NULL)`.

12.2.3 Accessing attributes of extension types

type **PyMemberDef**

Part of the Stable ABI *(including all members).* Structure which describes an attribute of a type which corresponds to a C struct member. Its fields are:

Field	C Type	Meaning
`name`	const char *	name of the member
`type`	int	the type of the member in the C struct
`offset`	Py_ssize_t	the offset in bytes that the member is located on the type's object struct
`flags`	int	flag bits indicating if the field should be read-only or writable
`doc`	const char *	points to the contents of the docstring

`type` can be one of many `T_` macros corresponding to various C types. When the member is accessed in Python, it will be converted to the equivalent Python type.

Macro name	C type
T_SHORT	short
T_INT	int
T_LONG	long
T_FLOAT	float
T_DOUBLE	double
T_STRING	const char *
T_OBJECT	PyObject *
T_OBJECT_EX	PyObject *
T_CHAR	char
T_BYTE	char
T_UBYTE	unsigned char
T_UINT	unsigned int
T_USHORT	unsigned short
T_ULONG	unsigned long
T_BOOL	char
T_LONGLONG	long long
T_ULONGLONG	unsigned long long
T_PYSSIZET	Py_ssize_t

`T_OBJECT` and `T_OBJECT_EX` differ in that `T_OBJECT` returns `None` if the member is `NULL` and `T_OBJECT_EX` raises an `AttributeError`. Try to use `T_OBJECT_EX` over `T_OBJECT` because `T_OBJECT_EX` handles use of the `del` statement on that attribute more correctly than `T_OBJECT`.

`flags` can be `0` for write and read access or `READONLY` for read-only access. Using `T_STRING` for `type` implies `READONLY`. `T_STRING` data is interpreted as UTF-8. Only `T_OBJECT` and `T_OBJECT_EX` members can be deleted. (They are set to `NULL`).

Heap allocated types (created using *`PyType_FromSpec()`* or similar), `PyMemberDef` may contain definitions for the special members `__dictoffset__`, `__weaklistoffset__` and __vectorcalloffset__, corresponding to *tp_dictoffset*, *tp_weaklistoffset* and *tp_vectorcall_offset* in type objects. These must be defined with T_PYSSIZET and READONLY, for example:

```
static PyMemberDef spam_type_members[] = {
    {"__dictoffset__", T_PYSSIZET, offsetof(Spam_object, dict), READONLY},
    {NULL}  /* Sentinel */
};
```

PyObject ***PyMember_GetOne** (const char *obj_addr, struct *PyMemberDef* *m)

Get an attribute belonging to the object at address *obj_addr*. The attribute is described by PyMemberDef *m*. Returns NULL on error.

int **PyMember_SetOne** (char *obj_addr, struct *PyMemberDef* *m, *PyObject* *o)

Set an attribute belonging to the object at address *obj_addr* to object *o*. The attribute to set is described by PyMemberDef *m*. Returns 0 if successful and a negative value on failure.

type **PyGetSetDef**

Part of the Stable ABI *(including all members).* Structure to define property-like access for a type. See also description of the *PyTypeObject.tp_getset* slot.

Field	C Type	Meaning
name	const char *	attribute name
get	getter	C function to get the attribute
set	setter	optional C function to set or delete the attribute, if omitted the attribute is readonly
doc	const char *	optional docstring
closure	void *	optional user data pointer, providing additional data for getter and setter

The get function takes one *PyObject** parameter (the instance) and a user data pointer (the associated closure):

```
typedef PyObject *(*getter)(PyObject *, void *);
```

It should return a new reference on success or NULL with a set exception on failure.

set functions take two *PyObject** parameters (the instance and the value to be set) and a user data pointer (the associated closure):

```
typedef int (*setter)(PyObject *, PyObject *, void *);
```

In case the attribute should be deleted the second parameter is NULL. Should return 0 on success or -1 with a set exception on failure.

12.3 型F物件

Perhaps one of the most important structures of the Python object system is the structure that defines a new type: the *PyTypeObject* structure. Type objects can be handled using any of the PyObject_* or PyType_* functions, but do not offer much that's interesting to most Python applications. These objects are fundamental to how objects behave, so they are very important to the interpreter itself and to any extension module that implements new types.

Type objects are fairly large compared to most of the standard types. The reason for the size is that each type object stores a large number of values, mostly C function pointers, each of which implements a small part of the type's functionality. The fields of the type object are examined in detail in this section. The fields will be described in the order in which they occur in the structure.

In addition to the following quick reference, the 範例 section provides at-a-glance insight into the meaning and use of *PyTypeObject*.

12.3.1 Quick Reference

"tp slots"

PyTypeObject Slot[Page 238, 1]	*Type*	special methods/attrs	Info[Page 238, 2]			
			C	T	D	I
<R> tp_name	const char *	__name__	X	X		
tp_basicsize	Py_ssize_t		X	X		X
tp_itemsize	Py_ssize_t			X		X
tp_dealloc	destructor		X	X		X
tp_vectorcall_offset	Py_ssize_t			X		X
(tp_getattr)	getattrfunc	__getattribute__, __getattr__				G
(tp_setattr)	setattrfunc	__setattr__, __delattr__				G
tp_as_async	PyAsyncMethods *	*sub-slots*				%
tp_repr	reprfunc	__repr__	X	X		X
tp_as_number	PyNumberMethods *	*sub-slots*				%
tp_as_sequence	PySequenceMethods *	*sub-slots*				%
tp_as_mapping	PyMappingMethods *	*sub-slots*				%
tp_hash	hashfunc	__hash__	X			G
tp_call	ternaryfunc	__call__		X		X
tp_str	reprfunc	__str__	X			X
tp_getattro	getattrofunc	__getattribute__, __getattr__	X	X		G
tp_setattro	setattrofunc	__setattr__, __delattr__	X	X		G
tp_as_buffer	PyBufferProcs *					%
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]	PyObject *	__subclasses__				
[tp_weaklist]	PyObject *					
(tp_del)	destructor					
[tp_version_tag]	unsigned int					
tp_finalize	destructor	__del__				X
tp_vectorcall	vectorcallfunc					

sub-slots

Slot	*Type*	special methods
am_await	*unaryfunc*	__await__
am_aiter	*unaryfunc*	__aiter__
am_anext	*unaryfunc*	__anext__
am_send	*sendfunc*	
nb_add	*binaryfunc*	__add__ __radd__
nb_inplace_add	*binaryfunc*	__iadd__
nb_subtract	*binaryfunc*	__sub__ __rsub__
nb_inplace_subtract	*binaryfunc*	__isub__
nb_multiply	*binaryfunc*	__mul__ __rmul__
nb_inplace_multiply	*binaryfunc*	__imul__
nb_remainder	*binaryfunc*	__mod__ __rmod__
nb_inplace_remainder	*binaryfunc*	__imod__
nb_divmod	*binaryfunc*	__divmod__ __rdivmod__
nb_power	*ternaryfunc*	__pow__ __rpow__
nb_inplace_power	*ternaryfunc*	__ipow__
nb_negative	*unaryfunc*	__neg__
nb_positive	*unaryfunc*	__pos__
nb_absolute	*unaryfunc*	__abs__
nb_bool	*inquiry*	__bool__
nb_invert	*unaryfunc*	__invert__
nb_lshift	*binaryfunc*	__lshift__ __rlshift__
nb_inplace_lshift	*binaryfunc*	__ilshift__
nb_rshift	*binaryfunc*	__rshift__ __rrshift__
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__

繼續下一頁

[1] (): A slot name in parentheses indicates it is (effectively) deprecated.
<>: Names in angle brackets should be initially set to NULL and treated as read-only.
[]: Names in square brackets are for internal use only.
<R> (as a prefix) means the field is required (must be non-NULL).
[2] Columns:
"O": set on PyBaseObject_Type
"T": set on *PyType_Type*
"D": default (if slot is set to NULL)

```
X - PyType_Ready sets this value if it is NULL
~ - PyType_Ready always sets this value (it should be NULL)
? - PyType_Ready may set this value depending on other slots

Also see the inheritance column ("I").
```

"I": inheritance

```
X - type slot is inherited via *PyType_Ready* if defined with a *NULL* value
% - the slots of the sub-struct are inherited individually
G - inherited, but only in combination with other slots; see the slot's description
? - it's complicated; see the slot's description
```

Note that some slots are effectively inherited through the normal attribute lookup chain.

表格 2 – 繼續上一頁

Slot	*Type*	special methods
nb_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()*	

slot typedefs

typedef	Parameter Types	Return Type
allocfunc	*PyTypeObject* * *Py_ssize_t*	*PyObject* *
destructor	*PyObject* *	void
freefunc	void *	void
traverseproc	*PyObject* * *visitproc* void *	int
newfunc	*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*Slot Type typedefs*。

12.3.2 PyTypeObject Definition

The structure definition for `PyTypeObject` can be found in `Include/object.h`. For convenience of reference, this repeats the definition found there:

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;

    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;
```

(繼續下一頁)

(繼續上一頁)

```
    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    // Strong reference on a heap type, borrowed reference on a static type
    struct _typeobject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
    freefunc tp_free; /* Low-level free-memory routine */
    inquiry tp_is_gc; /* For PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_cache;
    PyObject *tp_subclasses;
    PyObject *tp_weaklist;
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6 */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;
} PyTypeObject;
```

12.3.3 PyObject Slots

The type object structure extends the `PyVarObject` structure. The `ob_size` field is used for dynamic types (created by `type_new()`, usually called from a class statement). Note that `PyType_Type` (the metatype) initializes `tp_itemsize`, which means that its instances (i.e. type objects) *must* have the `ob_size` field.

Py_ssize_t `PyObject`.**`ob_refcnt`**

Part of the Stable ABI. This is the type object's reference count, initialized to `1` by the `PyObject_HEAD_INIT` macro. Note that for *statically allocated type objects*, the type's instances (objects whose `ob_type` points back to the type) do *not* count as references. But for *dynamically allocated type objects*, the instances *do* count as references.

Inheritance:

This field is not inherited by subtypes.

PyTypeObject *`PyObject`.**`ob_type`**

Part of the Stable ABI. This is the type's type, in other words its metatype. It is initialized by the argument to the `PyObject_HEAD_INIT` macro, and its value should normally be `&PyType_Type`. However, for dynamically loadable extension modules that must be usable on Windows (at least), the compiler complains that this is not a valid initializer. Therefore, the convention is to pass `NULL` to the `PyObject_HEAD_INIT` macro and to initialize this field explicitly at the start of the module's initialization function, before doing anything else. This is typically done like this:

```
Foo_Type.ob_type = &PyType_Type;
```

This should be done before any instances of the type are created. `PyType_Ready()` checks if `ob_type` is `NULL`, and if so, initializes it to the `ob_type` field of the base class. `PyType_Ready()` will not change this field if it is non-zero.

Inheritance:

This field is inherited by subtypes.

PyObject **PyObject._ob_next*

PyObject **PyObject._ob_prev*

These fields are only present when the macro `Py_TRACE_REFS` is defined (see the `configure --with-trace-refs option`).

Their initialization to `NULL` is taken care of by the `PyObject_HEAD_INIT` macro. For *statically allocated objects*, these fields always remain `NULL`. For *dynamically allocated objects*, these two fields are used to link the object into a doubly linked list of *all* live objects on the heap.

This could be used for various debugging purposes; currently the only uses are the `sys.getobjects()` function and to print the objects that are still alive at the end of a run when the environment variable `PYTHONDUMPREFS` is set.

Inheritance:

These fields are not inherited by subtypes.

12.3.4 PyVarObject Slots

Py_ssize_t `PyVarObject.ob_size`

Part of the Stable ABI. For *statically allocated type objects*, this should be initialized to zero. For *dynamically allocated type objects*, this field has a special internal meaning.

Inheritance:

This field is not inherited by subtypes.

12.3.5 PyTypeObject Slots

Each slot has a section describing inheritance. If `PyType_Ready()` may set a value when the field is set to `NULL` then there will also be a "Default" section. (Note that many fields set on `PyBaseObject_Type` and `PyType_Type` effectively act as defaults.)

const char *`PyTypeObject.tp_name`

Pointer to a NUL-terminated string containing the name of the type. For types that are accessible as module globals, the string should be the full module name, followed by a dot, followed by the type name; for built-in types, it should be just the type name. If the module is a submodule of a package, the full package name is part of the full module name. For example, a type named `T` defined in module `M` in subpackage `Q` in package `P` should have the `tp_name` initializer `"P.Q.M.T"`.

For *dynamically allocated type objects*, this should just be the type name, and the module name explicitly stored in the type dict as the value for key `'__module__'`.

For *statically allocated type objects*, the *tp_name* field should contain a dot. Everything before the last dot is made accessible as the `__module__` attribute, and everything after the last dot is made accessible as the `__name__` attribute.

If no dot is present, the entire `tp_name` field is made accessible as the `__name__` attribute, and the `__module__` attribute is undefined (unless explicitly set in the dictionary, as explained above). This means your type will be impossible to pickle. Additionally, it will not be listed in module documentations created with pydoc.

This field must not be `NULL`. It is the only required field in `PyTypeObject()` (other than potentially `tp_itemsize`).

Inheritance:

This field is not inherited by subtypes.

Py_ssize_t PyTypeObject.**tp_basicsize**

Py_ssize_t PyTypeObject.**tp_itemsize**

These fields allow calculating the size in bytes of instances of the type.

There are two kinds of types: types with fixed-length instances have a zero `tp_itemsize` field, types with variable-length instances have a non-zero `tp_itemsize` field. For a type with fixed-length instances, all instances have the same size, given in `tp_basicsize`.

For a type with variable-length instances, the instances must have an `ob_size` field, and the instance size is `tp_basicsize` plus N times `tp_itemsize`, where N is the "length" of the object. The value of N is typically stored in the instance's `ob_size` field. There are exceptions: for example, ints use a negative `ob_size` to indicate a negative number, and N is `abs(ob_size)` there. Also, the presence of an `ob_size` field in the instance layout doesn't mean that the instance structure is variable-length (for example, the structure for the list type has fixed-length instances, yet those instances have a meaningful `ob_size` field).

The basic size includes the fields in the instance declared by the macro `PyObject_HEAD` or `PyObject_VAR_HEAD` (whichever is used to declare the instance struct) and this in turn includes the `_ob_prev` and `_ob_next` fields if they are present. This means that the only correct way to get an initializer for the `tp_basicsize` is to use the `sizeof` operator on the struct used to declare the instance layout. The basic size does not include the GC header size.

A note about alignment: if the variable items require a particular alignment, this should be taken care of by the value of `tp_basicsize`. Example: suppose a type implements an array of `double`. `tp_itemsize` is `sizeof(double)`. It is the programmer's responsibility that `tp_basicsize` is a multiple of `sizeof(double)` (assuming this is the alignment requirement for `double`).

For any type with variable-length instances, this field must not be `NULL`.

Inheritance:

These fields are inherited separately by subtypes. If the base type has a non-zero `tp_itemsize`, it is generally not safe to set `tp_itemsize` to a different non-zero value in a subtype (though this depends on the implementation of the base type).

destructor PyTypeObject.**tp_dealloc**

A pointer to the instance destructor function. This function must be defined unless the type guarantees that its instances will never be deallocated (as is the case for the singletons `None` and `Ellipsis`). The function signature is:

```
void tp_dealloc(PyObject *self);
```

The destructor function is called by the `Py_DECREF()` and `Py_XDECREF()` macros when the new reference count is zero. At this point, the instance is still in existence, but there are no references to it. The destructor function should free all references which the instance owns, free all memory buffers owned by the instance (using the freeing function corresponding to the allocation function used to allocate the buffer), and call the type's `tp_free` function. If the type is not subtypable (doesn't have the `Py_TPFLAGS_BASETYPE` flag bit set), it is permissible to call the object deallocator directly instead of via `tp_free`. The object deallocator should be the one used to allocate the instance; this is normally `PyObject_Del()` if the instance was allocated using `PyObject_New` or `PyObject_NewVar`, or `PyObject_GC_Del()` if the instance was allocated using `PyObject_GC_New` or `PyObject_GC_NewVar`.

If the type supports garbage collection (has the `Py_TPFLAGS_HAVE_GC` flag bit set), the destructor should call `PyObject_GC_UnTrack()` before clearing any member fields.

```
static void foo_dealloc(foo_object *self) {
    PyObject_GC_UnTrack(self);
    Py_CLEAR(self->ref);
    Py_TYPE(self)->tp_free((PyObject *)self);
}
```

Finally, if the type is heap allocated (`Py_TPFLAGS_HEAPTYPE`), the deallocator should release the owned reference to its type object (via `Py_DECREF()`) after calling the type deallocator. In order to avoid dangling pointers, the recommended way to achieve this is:

```
static void foo_dealloc(foo_object *self) {
    PyTypeObject *tp = Py_TYPE(self);
    // free references and buffers here
    tp->tp_free(self);
    Py_DECREF(tp);
}
```

Inheritance:

This field is inherited by subtypes.

Py_ssize_t *PyTypeObject*.**tp_vectorcall_offset**

An optional offset to a per-instance function that implements calling the object using the *vectorcall protocol*, a more efficient alternative of the simpler *tp_call*.

This field is only used if the flag *Py_TPFLAGS_HAVE_VECTORCALL* is set. If so, this must be a positive integer containing the offset in the instance of a *vectorcallfunc* pointer.

The *vectorcallfunc* pointer may be NULL, in which case the instance behaves as if *Py_TPFLAGS_HAVE_VECTORCALL* was not set: calling the instance falls back to *tp_call*.

Any class that sets Py_TPFLAGS_HAVE_VECTORCALL must also set *tp_call* and make sure its behaviour is consistent with the *vectorcallfunc* function. This can be done by setting *tp_call* to *PyVectorcall_Call()*.

> **警告:** It is not recommended for *mutable heap types* to implement the vectorcall protocol. When a user sets __call__ in Python code, only *tp_call* is updated, likely making it inconsistent with the vectorcall function.

在 3.8 版的變更: Before version 3.8, this slot was named tp_print. In Python 2.x, it was used for printing to a file. In Python 3.0 to 3.7, it was unused.

Inheritance:

This field is always inherited. However, the *Py_TPFLAGS_HAVE_VECTORCALL* flag is not always inherited. If it's not, then the subclass won't use *vectorcall*, except when *PyVectorcall_Call()* is explicitly called. This is in particular the case for types without the *Py_TPFLAGS_IMMUTABLETYPE* flag set (including subclasses defined in Python).

getattrfunc *PyTypeObject*.**tp_getattr**

An optional pointer to the get-attribute-string function.

This field is deprecated. When it is defined, it should point to a function that acts the same as the *tp_getattro* function, but taking a C string instead of a Python string object to give the attribute name.

Inheritance:

Group: *tp_getattr*, *tp_getattro*

This field is inherited by subtypes together with *tp_getattro*: a subtype inherits both *tp_getattr* and *tp_getattro* from its base type when the subtype's *tp_getattr* and *tp_getattro* are both NULL.

setattrfunc *PyTypeObject*.**tp_setattr**

An optional pointer to the function for setting and deleting attributes.

This field is deprecated. When it is defined, it should point to a function that acts the same as the *tp_setattro* function, but taking a C string instead of a Python string object to give the attribute name.

Inheritance:

Group: *tp_setattr*, *tp_setattro*

This field is inherited by subtypes together with *tp_setattro*: a subtype inherits both *tp_setattr* and *tp_setattro* from its base type when the subtype's *tp_setattr* and *tp_setattro* are both NULL.

PyAsyncMethods *PyTypeObject.**tp_as_async**

Pointer to an additional structure that contains fields relevant only to objects which implement *awaitable* and *asynchronous iterator* protocols at the C-level. See *Async Object Structures* for details.

在 3.5 版新加入: Formerly known as tp_compare and tp_reserved.

Inheritance:

The *tp_as_async* field is not inherited, but the contained fields are inherited individually.

reprfunc PyTypeObject.**tp_repr**

An optional pointer to a function that implements the built-in function repr().

The signature is the same as for *PyObject_Repr()*:

```
PyObject *tp_repr(PyObject *self);
```

The function must return a string or a Unicode object. Ideally, this function should return a string that, when passed to eval(), given a suitable environment, returns an object with the same value. If this is not feasible, it should return a string starting with '<' and ending with '>' from which both the type and the value of the object can be deduced.

Inheritance:

This field is inherited by subtypes.

預設:

When this field is not set, a string of the form <%s object at %p> is returned, where %s is replaced by the type name, and %p by the object's memory address.

PyNumberMethods *PyTypeObject.**tp_as_number**

Pointer to an additional structure that contains fields relevant only to objects which implement the number protocol. These fields are documented in *Number Object Structures*.

Inheritance:

The *tp_as_number* field is not inherited, but the contained fields are inherited individually.

PySequenceMethods *PyTypeObject.**tp_as_sequence**

Pointer to an additional structure that contains fields relevant only to objects which implement the sequence protocol. These fields are documented in *Sequence Object Structures*.

Inheritance:

The *tp_as_sequence* field is not inherited, but the contained fields are inherited individually.

PyMappingMethods *PyTypeObject.**tp_as_mapping**

Pointer to an additional structure that contains fields relevant only to objects which implement the mapping protocol. These fields are documented in *Mapping Object Structures*.

Inheritance:

The *tp_as_mapping* field is not inherited, but the contained fields are inherited individually.

hashfunc PyTypeObject.**tp_hash**

An optional pointer to a function that implements the built-in function hash().

The signature is the same as for *PyObject_Hash()*:

```
Py_hash_t tp_hash(PyObject *);
```

The value -1 should not be returned as a normal return value; when an error occurs during the computation of the hash value, the function should set an exception and return -1.

When this field is not set (*and* *tp_richcompare* is not set), an attempt to take the hash of the object raises TypeError. This is the same as setting it to *PyObject_HashNotImplemented()*.

This field can be set explicitly to *PyObject_HashNotImplemented()* to block inheritance of the hash method from a parent type. This is interpreted as the equivalent of `__hash__ = None` at the Python level, causing `isinstance(o, collections.Hashable)` to correctly return `False`. Note that the converse is also true - setting `__hash__ = None` on a class at the Python level will result in the `tp_hash` slot being set to *PyObject_HashNotImplemented()*.

Inheritance:

Group: *tp_hash*, *tp_richcompare*

This field is inherited by subtypes together with *tp_richcompare*: a subtype inherits both of *tp_richcompare* and *tp_hash*, when the subtype's *tp_richcompare* and *tp_hash* are both `NULL`.

ternaryfunc PyTypeObject.**tp_call**

An optional pointer to a function that implements calling the object. This should be `NULL` if the object is not callable. The signature is the same as for *PyObject_Call()*:

```
PyObject *tp_call(PyObject *self, PyObject *args, PyObject *kwargs);
```

Inheritance:

This field is inherited by subtypes.

reprfunc PyTypeObject.**tp_str**

An optional pointer to a function that implements the built-in operation `str()`. (Note that `str` is a type now, and `str()` calls the constructor for that type. This constructor calls *PyObject_Str()* to do the actual work, and *PyObject_Str()* will call this handler.)

The signature is the same as for *PyObject_Str()*:

```
PyObject *tp_str(PyObject *self);
```

The function must return a string or a Unicode object. It should be a "friendly" string representation of the object, as this is the representation that will be used, among other things, by the `print()` function.

Inheritance:

This field is inherited by subtypes.

預設:

When this field is not set, *PyObject_Repr()* is called to return a string representation.

getattrofunc PyTypeObject.**tp_getattro**

An optional pointer to the get-attribute function.

The signature is the same as for *PyObject_GetAttr()*:

```
PyObject *tp_getattro(PyObject *self, PyObject *attr);
```

It is usually convenient to set this field to *PyObject_GenericGetAttr()*, which implements the normal way of looking for object attributes.

Inheritance:

Group: *tp_getattr*, *tp_getattro*

This field is inherited by subtypes together with *tp_getattr*: a subtype inherits both *tp_getattr* and *tp_getattro* from its base type when the subtype's *tp_getattr* and *tp_getattro* are both `NULL`.

預設:

`PyBaseObject_Type` uses *PyObject_GenericGetAttr()*.

setattrofunc PyTypeObject.**tp_setattro**

An optional pointer to the function for setting and deleting attributes.

The signature is the same as for *PyObject_SetAttr()*:

```
int tp_setattro(PyObject *self, PyObject *attr, PyObject *value);
```

In addition, setting *value* to NULL to delete an attribute must be supported. It is usually convenient to set this field to *PyObject_GenericSetAttr()*, which implements the normal way of setting object attributes.

Inheritance:

Group: *tp_setattr*, *tp_setattro*

This field is inherited by subtypes together with *tp_setattr*: a subtype inherits both *tp_setattr* and *tp_setattro* from its base type when the subtype's *tp_setattr* and *tp_setattro* are both NULL.

預設:

PyBaseObject_Type uses *PyObject_GenericSetAttr()*.

PyBufferProcs **PyTypeObject*.**tp_as_buffer**

Pointer to an additional structure that contains fields relevant only to objects which implement the buffer interface. These fields are documented in *Buffer Object Structures*.

Inheritance:

The *tp_as_buffer* field is not inherited, but the contained fields are inherited individually.

unsigned long *PyTypeObject*.**tp_flags**

This field is a bit mask of various flags. Some flags indicate variant semantics for certain situations; others are used to indicate that certain fields in the type object (or in the extension structures referenced via *tp_as_number*, *tp_as_sequence*, *tp_as_mapping*, and *tp_as_buffer*) that were historically not always present are valid; if such a flag bit is clear, the type fields it guards must not be accessed and must be considered to have a zero or NULL value instead.

Inheritance:

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type's value of the flag bit is copied into the subtype together with a pointer to the extension structure. The *Py_TPFLAGS_HAVE_GC* flag bit is inherited together with the *tp_traverse* and *tp_clear* fields, i.e. if the *Py_TPFLAGS_HAVE_GC* flag bit is clear in the subtype and the *tp_traverse* and *tp_clear* fields in the subtype exist and have NULL values.

預設:

PyBaseObject_Type uses Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE.

Bit Masks:

The following bit masks are currently defined; these can be ORed together using the | operator to form the value of the *tp_flags* field. The macro *PyType_HasFeature()* takes a type and a flags value, *tp* and *f*, and checks whether tp->tp_flags & f is non-zero.

Py_TPFLAGS_HEAPTYPE

This bit is set when the type object itself is allocated on the heap, for example, types created dynamically using *PyType_FromSpec()*. In this case, the *ob_type* field of its instances is considered a reference to the type, and the type object is INCREF'ed when a new instance is created, and DECREF'ed when an instance is destroyed (this does not apply to instances of subtypes; only the type referenced by the instance's ob_type gets INCREF'ed or DECREF'ed).

Inheritance:

???

Py_TPFLAGS_BASETYPE

This bit is set when the type can be used as the base type of another type. If this bit is clear, the type cannot be subtyped (similar to a "final" class in Java).

Inheritance:

???

Py_TPFLAGS_READY

This bit is set when the type object has been fully initialized by *PyType_Ready()*.

Inheritance:

???

Py_TPFLAGS_READYING

This bit is set while *PyType_Ready()* is in the process of initializing the type object.

Inheritance:

???

Py_TPFLAGS_HAVE_GC

This bit is set when the object supports garbage collection. If this bit is set, instances must be created using *PyObject_GC_New* and destroyed using *PyObject_GC_Del()*. More information in section *Supporting Cyclic Garbage Collection*. This bit also implies that the GC-related fields *tp_traverse* and *tp_clear* are present in the type object.

Inheritance:

Group: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

The *Py_TPFLAGS_HAVE_GC* flag bit is inherited together with the *tp_traverse* and *tp_clear* fields, i.e. if the *Py_TPFLAGS_HAVE_GC* flag bit is clear in the subtype and the *tp_traverse* and *tp_clear* fields in the subtype exist and have `NULL` values.

Py_TPFLAGS_DEFAULT

This is a bitmask of all the bits that pertain to the existence of certain fields in the type object and its extension structures. Currently, it includes the following bits: `Py_TPFLAGS_HAVE_STACKLESS_EXTENSION`.

Inheritance:

???

Py_TPFLAGS_METHOD_DESCRIPTOR

This bit indicates that objects behave like unbound methods.

If this flag is set for `type(meth)`, then:

- `meth.__get__(obj, cls)(*args, **kwds)` (with `obj` not None) must be equivalent to `meth(obj, *args, **kwds)`.
- `meth.__get__(None, cls)(*args, **kwds)` must be equivalent to `meth(*args, **kwds)`.

This flag enables an optimization for typical method calls like `obj.meth()`: it avoids creating a temporary "bound method" object for `obj.meth`.

在 3.8 版新加入.

Inheritance:

This flag is never inherited by types without the *Py_TPFLAGS_IMMUTABLETYPE* flag set. For extension types, it is inherited whenever *tp_descr_get* is inherited.

Py_TPFLAGS_LONG_SUBCLASS

Py_TPFLAGS_LIST_SUBCLASS

Py_TPFLAGS_TUPLE_SUBCLASS

Py_TPFLAGS_BYTES_SUBCLASS

Py_TPFLAGS_UNICODE_SUBCLASS

Py_TPFLAGS_DICT_SUBCLASS

Py_TPFLAGS_BASE_EXC_SUBCLASS

Py_TPFLAGS_TYPE_SUBCLASS

These flags are used by functions such as *PyLong_Check()* to quickly determine if a type is a subclass of a built-in type; such specific checks are faster than a generic check, like *PyObject_IsInstance()*. Custom types that inherit from built-ins should have their *tp_flags* set appropriately, or the code that interacts with such types will behave differently depending on what kind of check is used.

Py_TPFLAGS_HAVE_FINALIZE

This bit is set when the *tp_finalize* slot is present in the type structure.

在 3.4 版新加入.

在 3.8 版之後被棄用: This flag isn't necessary anymore, as the interpreter assumes the *tp_finalize* slot is always present in the type structure.

Py_TPFLAGS_HAVE_VECTORCALL

This bit is set when the class implements the *vectorcall protocol*. See *tp_vectorcall_offset* for details.

Inheritance:

This bit is inherited for types with the *Py_TPFLAGS_IMMUTABLETYPE* flag set, if *tp_call* is also inherited.

在 3.9 版新加入.

Py_TPFLAGS_IMMUTABLETYPE

This bit is set for type objects that are immutable: type attributes cannot be set nor deleted.

PyType_Ready() automatically applies this flag to *static types*.

Inheritance:

This flag is not inherited.

在 3.10 版新加入.

Py_TPFLAGS_DISALLOW_INSTANTIATION

Disallow creating instances of the type: set *tp_new* to NULL and don't create the `__new__` key in the type dictionary.

The flag must be set before creating the type, not after. For example, it must be set before *PyType_Ready()* is called on the type.

The flag is set automatically on *static types* if *tp_base* is NULL or `&PyBaseObject_Type` and *tp_new* is NULL.

Inheritance:

This flag is not inherited. However, subclasses will not be instantiable unless they provide a non-NULL *tp_new* (which is only possible via the C API).

備註: To disallow instantiating a class directly but allow instantiating its subclasses (e.g. for an *abstract base class*), do not use this flag. Instead, make *tp_new* only succeed for subclasses.

在 3.10 版新加入.

Py_TPFLAGS_MAPPING

This bit indicates that instances of the class may match mapping patterns when used as the subject of a match block. It is automatically set when registering or subclassing collections.abc.Mapping, and unset when registering collections.abc.Sequence.

備F: *Py_TPFLAGS_MAPPING* and *Py_TPFLAGS_SEQUENCE* are mutually exclusive; it is an error to enable both flags simultaneously.

Inheritance:

This flag is inherited by types that do not already set *Py_TPFLAGS_SEQUENCE*.

也參考:

PEP 634 -- Structural Pattern Matching: Specification

在 3.10 版新加入.

Py_TPFLAGS_SEQUENCE

This bit indicates that instances of the class may match sequence patterns when used as the subject of a match block. It is automatically set when registering or subclassing collections.abc.Sequence, and unset when registering collections.abc.Mapping.

備F: *Py_TPFLAGS_MAPPING* and *Py_TPFLAGS_SEQUENCE* are mutually exclusive; it is an error to enable both flags simultaneously.

Inheritance:

This flag is inherited by types that do not already set *Py_TPFLAGS_MAPPING*.

也參考:

PEP 634 -- Structural Pattern Matching: Specification

在 3.10 版新加入.

const char **PyTypeObject*.**tp_doc**

An optional pointer to a NUL-terminated C string giving the docstring for this type object. This is exposed as the __doc__ attribute on the type and instances of the type.

Inheritance:

This field is *not* inherited by subtypes.

traverseproc PyTypeObject.**tp_traverse**

An optional pointer to a traversal function for the garbage collector. This is only used if the *Py_TPFLAGS_HAVE_GC* flag bit is set. The signature is:

```
int tp_traverse(PyObject *self, visitproc visit, void *arg);
```

More information about Python's garbage collection scheme can be found in section *Supporting Cyclic Garbage Collection*.

The *tp_traverse* pointer is used by the garbage collector to detect reference cycles. A typical implementation of a *tp_traverse* function simply calls *Py_VISIT()* on each of the instance's members that are Python objects that the instance owns. For example, this is function local_traverse() from the _thread extension module:

```
static int
local_traverse(localobject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->args);
```

(繼續下一頁)

(繼續上一頁)

```
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

Note that `Py_VISIT()` is called only on those members that can participate in reference cycles. Although there is also a `self->key` member, it can only be `NULL` or a Python string and therefore cannot be part of a reference cycle.

On the other hand, even if you know a member can never be part of a cycle, as a debugging aid you may want to visit it anyway just so the `gc` module's `get_referents()` function will include it.

> **警告:** When implementing `tp_traverse`, only the members that the instance *owns* (by having *strong references* to them) must be visited. For instance, if an object supports weak references via the `tp_weaklist` slot, the pointer supporting the linked list (what *tp_weaklist* points to) must **not** be visited as the instance does not directly own the weak references to itself (the weakreference list is there to support the weak reference machinery, but the instance has no strong reference to the elements inside it, as they are allowed to be removed even if the instance is still alive).

Note that `Py_VISIT()` requires the *visit* and *arg* parameters to `local_traverse()` to have these specific names; don't name them just anything.

Instances of *heap-allocated types* hold a reference to their type. Their traversal function must therefore either visit `Py_TYPE(self)`, or delegate this responsibility by calling `tp_traverse` of another heap-allocated type (such as a heap-allocated superclass). If they do not, the type object may not be garbage-collected.

在 3.9 版的變更: Heap-allocated types are expected to visit `Py_TYPE(self)` in `tp_traverse`. In earlier versions of Python, due to bug 40217, doing this may lead to crashes in subclasses.

Inheritance:

Group: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

This field is inherited by subtypes together with `tp_clear` and the `Py_TPFLAGS_HAVE_GC` flag bit: the flag bit, `tp_traverse`, and `tp_clear` are all inherited from the base type if they are all zero in the subtype.

inquiry `PyTypeObject`.**`tp_clear`**

An optional pointer to a clear function for the garbage collector. This is only used if the `Py_TPFLAGS_HAVE_GC` flag bit is set. The signature is:

```
int tp_clear(PyObject *);
```

The `tp_clear` member function is used to break reference cycles in cyclic garbage detected by the garbage collector. Taken together, all `tp_clear` functions in the system must combine to break all reference cycles. This is subtle, and if in any doubt supply a `tp_clear` function. For example, the tuple type does not implement a `tp_clear` function, because it's possible to prove that no reference cycle can be composed entirely of tuples. Therefore the `tp_clear` functions of other types must be sufficient to break any cycle containing a tuple. This isn't immediately obvious, and there's rarely a good reason to avoid implementing `tp_clear`.

Implementations of `tp_clear` should drop the instance's references to those of its members that may be Python objects, and set its pointers to those members to `NULL`, as in the following example:

```
static int
local_clear(localobject *self)
{
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
```

(繼續下一頁)

(繼續上一頁)

```
    return 0;
}
```

The *Py_CLEAR()* macro should be used, because clearing references is delicate: the reference to the contained object must not be released (via *Py_DECREF()*) until after the pointer to the contained object is set to NULL. This is because releasing the reference may cause the contained object to become trash, triggering a chain of reclamation activity that may include invoking arbitrary Python code (due to finalizers, or weakref callbacks, associated with the contained object). If it's possible for such code to reference *self* again, it's important that the pointer to the contained object be NULL at that time, so that *self* knows the contained object can no longer be used. The *Py_CLEAR()* macro performs the operations in a safe order.

Note that *tp_clear* is not *always* called before an instance is deallocated. For example, when reference counting is enough to determine that an object is no longer used, the cyclic garbage collector is not involved and *tp_dealloc* is called directly.

Because the goal of *tp_clear* functions is to break reference cycles, it's not necessary to clear contained objects like Python strings or Python integers, which can't participate in reference cycles. On the other hand, it may be convenient to clear all contained Python objects, and write the type's *tp_dealloc* function to invoke *tp_clear*.

More information about Python's garbage collection scheme can be found in section *Supporting Cyclic Garbage Collection*.

Inheritance:

Group: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

This field is inherited by subtypes together with *tp_traverse* and the *Py_TPFLAGS_HAVE_GC* flag bit: the flag bit, *tp_traverse*, and *tp_clear* are all inherited from the base type if they are all zero in the subtype.

richcmpfunc *PyTypeObject*.**tp_richcompare**

An optional pointer to the rich comparison function, whose signature is:

```
PyObject *tp_richcompare(PyObject *self, PyObject *other, int op);
```

The first parameter is guaranteed to be an instance of the type that is defined by *PyTypeObject*.

The function should return the result of the comparison (usually Py_True or Py_False). If the comparison is undefined, it must return Py_NotImplemented, if another error occurred it must return NULL and set an exception condition.

The following constants are defined to be used as the third argument for *tp_richcompare* and for *PyObject_RichCompare()*:

常數	Comparison
Py_LT	<
Py_LE	<=
Py_EQ	==
Py_NE	!=
Py_GT	>
Py_GE	>=

The following macro is defined to ease writing rich comparison functions:

Py_RETURN_RICHCOMPARE(VAL_A, VAL_B, op)

Return `Py_True` or `Py_False` from the function, depending on the result of a comparison. VAL_A and VAL_B must be orderable by C comparison operators (for example, they may be C ints or floats). The third argument specifies the requested operation, as for `PyObject_RichCompare()`.

The returned value is a new *strong reference*.

On error, sets an exception and returns `NULL` from the function.

在 3.7 版新加入.

Inheritance:

Group: `tp_hash`, `tp_richcompare`

This field is inherited by subtypes together with `tp_hash`: a subtype inherits `tp_richcompare` and `tp_hash` when the subtype's `tp_richcompare` and `tp_hash` are both `NULL`.

預設:

`PyBaseObject_Type` provides a `tp_richcompare` implementation, which may be inherited. However, if only `tp_hash` is defined, not even the inherited function is used and instances of the type will not be able to participate in any comparisons.

Py_ssize_t `PyTypeObject`**.tp_weaklistoffset**

If the instances of this type are weakly referenceable, this field is greater than zero and contains the offset in the instance structure of the weak reference list head (ignoring the GC header, if present); this offset is used by `PyObject_ClearWeakRefs()` and the `PyWeakref_*` functions. The instance structure needs to include a field of type `PyObject*` which is initialized to `NULL`.

Do not confuse this field with `tp_weaklist`; that is the list head for weak references to the type object itself.

Inheritance:

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype uses a different weak reference list head than the base type. Since the list head is always found via `tp_weaklistoffset`, this should not be a problem.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types are weakly referenceable, the type is made weakly referenceable by adding a weak reference list head slot to the instance layout and setting the `tp_weaklistoffset` of that slot's offset.

When a type's `__slots__` declaration contains a slot named `__weakref__`, that slot becomes the weak reference list head for instances of the type, and the slot's offset is stored in the type's *tp_weaklistoffset*.

When a type's `__slots__` declaration does not contain a slot named `__weakref__`, the type inherits its *tp_weaklistoffset* from its base type.

getiterfunc *PyTypeObject*.**tp_iter**

An optional pointer to a function that returns an *iterator* for the object. Its presence normally signals that the instances of this type are *iterable* (although sequences may be iterable without this function).

This function has the same signature as *PyObject_GetIter()*:

```
PyObject *tp_iter(PyObject *self);
```

Inheritance:

This field is inherited by subtypes.

iternextfunc *PyTypeObject*.**tp_iternext**

An optional pointer to a function that returns the next item in an *iterator*. The signature is:

```
PyObject *tp_iternext(PyObject *self);
```

When the iterator is exhausted, it must return `NULL`; a `StopIteration` exception may or may not be set. When another error occurs, it must return `NULL` too. Its presence signals that the instances of this type are iterators.

Iterator types should also define the *tp_iter* function, and that function should return the iterator instance itself (not a new iterator instance).

This function has the same signature as *PyIter_Next()*.

Inheritance:

This field is inherited by subtypes.

struct *PyMethodDef* **PyTypeObject*.**tp_methods**

An optional pointer to a static `NULL`-terminated array of *PyMethodDef* structures, declaring regular methods of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a method descriptor.

Inheritance:

This field is not inherited by subtypes (methods are inherited through a different mechanism).

struct *PyMemberDef* **PyTypeObject*.**tp_members**

An optional pointer to a static `NULL`-terminated array of *PyMemberDef* structures, declaring regular data members (fields or slots) of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a member descriptor.

Inheritance:

This field is not inherited by subtypes (members are inherited through a different mechanism).

struct *PyGetSetDef* **PyTypeObject*.**tp_getset**

An optional pointer to a static `NULL`-terminated array of *PyGetSetDef* structures, declaring computed attributes of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a getset descriptor.

Inheritance:

This field is not inherited by subtypes (computed attributes are inherited through a different mechanism).

PyTypeObject *`PyTypeObject`.**tp_base**

An optional pointer to a base type from which type properties are inherited. At this level, only single inheritance is supported; multiple inheritance require dynamically creating a type object by calling the metatype.

備F: Slot initialization is subject to the rules of initializing globals. C99 requires the initializers to be "address constants". Function designators like `PyType_GenericNew()`, with implicit conversion to a pointer, are valid C99 address constants.

However, the unary '&' operator applied to a non-static variable like `PyBaseObject_Type` is not required to produce an address constant. Compilers may support this (gcc does), MSVC does not. Both compilers are strictly standard conforming in this particular behavior.

Consequently, `tp_base` should be set in the extension module's init function.

Inheritance:

This field is not inherited by subtypes (obviously).

預設:

This field defaults to `&PyBaseObject_Type` (which to Python programmers is known as the type `object`).

PyObject *`PyTypeObject`.**tp_dict**

The type's dictionary is stored here by `PyType_Ready()`.

This field should normally be initialized to `NULL` before PyType_Ready is called; it may also be initialized to a dictionary containing initial attributes for the type. Once `PyType_Ready()` has initialized the type, extra attributes for the type may be added to this dictionary only if they don't correspond to overloaded operations (like `__add__()`).

Inheritance:

This field is not inherited by subtypes (though the attributes defined in here are inherited through a different mechanism).

預設:

If this field is `NULL`, `PyType_Ready()` will assign a new dictionary to it.

警告: It is not safe to use `PyDict_SetItem()` on or otherwise modify `tp_dict` with the dictionary C-API.

descrgetfunc `PyTypeObject`.**tp_descr_get**

An optional pointer to a "descriptor get" function.

The function signature is:

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *type);
```

Inheritance:

This field is inherited by subtypes.

descrsetfunc `PyTypeObject`.**tp_descr_set**

An optional pointer to a function for setting and deleting a descriptor's value.

The function signature is:

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

The *value* argument is set to `NULL` to delete the value.

Inheritance:

This field is inherited by subtypes.

Py_ssize_t `PyTypeObject`.**`tp_dictoffset`**

If the instances of this type have a dictionary containing instance variables, this field is non-zero and contains the offset in the instances of the type of the instance variable dictionary; this offset is used by `PyObject_GenericGetAttr()`.

Do not confuse this field with `tp_dict`; that is the dictionary for attributes of the type object itself.

If the value of this field is greater than zero, it specifies the offset from the start of the instance structure. If the value is less than zero, it specifies the offset from the *end* of the instance structure. A negative offset is more expensive to use, and should only be used when the instance structure contains a variable-length part. This is used for example to add an instance variable dictionary to subtypes of `str` or `tuple`. Note that the `tp_basicsize` field should account for the dictionary added to the end in that case, even though the dictionary is not included in the basic object layout. On a system with a pointer size of 4 bytes, `tp_dictoffset` should be set to `-4` to indicate that the dictionary is at the very end of the structure.

The `tp_dictoffset` should be regarded as write-only. To get the pointer to the dictionary call `PyObject_GenericGetDict()`. Calling `PyObject_GenericGetDict()` may need to allocate memory for the dictionary, so it is may be more efficient to call `PyObject_GetAttr()` when accessing an attribute on the object.

Inheritance:

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype instances store the dictionary at a difference offset than the base type. Since the dictionary is always found via `tp_dictoffset`, this should not be a problem.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types has an instance variable dictionary, a dictionary slot is added to the instance layout and the `tp_dictoffset` is set to that slot's offset.

When a type defined by a class statement has a `__slots__` declaration, the type inherits its `tp_dictoffset` from its base type.

(Adding a slot named `__dict__` to the `__slots__` declaration does not have the expected effect, it just causes confusion. Maybe this should be added as a feature just like `__weakref__` though.)

預設：

This slot has no default. For *static types*, if the field is `NULL` then no `__dict__` gets created for instances.

initproc `PyTypeObject`.**`tp_init`**

An optional pointer to an instance initialization function.

This function corresponds to the `__init__()` method of classes. Like `__init__()`, it is possible to create an instance without calling `__init__()`, and it is possible to reinitialize an instance by calling its `__init__()` method again.

The function signature is:

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwds);
```

The self argument is the instance to be initialized; the *args* and *kwds* arguments represent positional and keyword arguments of the call to `__init__()`.

The `tp_init` function, if not `NULL`, is called when an instance is created normally by calling its type, after the type's `tp_new` function has returned an instance of the type. If the `tp_new` function returns an instance of some other type that is not a subtype of the original type, no `tp_init` function is called; if `tp_new` returns an instance of a subtype of the original type, the subtype's `tp_init` is called.

Returns `0` on success, `-1` and sets an exception on error.

Inheritance:

This field is inherited by subtypes.

預設：

For *static types* this field does not have a default.

allocfunc `PyTypeObject`.**tp_alloc**

An optional pointer to an instance allocation function.

The function signature is:

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems);
```

Inheritance:

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement).

預設：

For dynamic subtypes, this field is always set to `PyType_GenericAlloc()`, to force a standard heap allocation strategy.

For static subtypes, `PyBaseObject_Type` uses `PyType_GenericAlloc()`. That is the recommended value for all statically defined types.

newfunc `PyTypeObject`.**tp_new**

An optional pointer to an instance creation function.

The function signature is:

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds);
```

The *subtype* argument is the type of the object being created; the *args* and *kwds* arguments represent positional and keyword arguments of the call to the type. Note that *subtype* doesn't have to equal the type whose `tp_new` function is called; it may be a subtype of that type (but not an unrelated type).

The `tp_new` function should call `subtype->tp_alloc(subtype, nitems)` to allocate space for the object, and then do only as much further initialization as is absolutely necessary. Initialization that can safely be ignored or repeated should be placed in the `tp_init` handler. A good rule of thumb is that for immutable types, all initialization should take place in `tp_new`, while for mutable types, most initialization should be deferred to `tp_init`.

Set the `Py_TPFLAGS_DISALLOW_INSTANTIATION` flag to disallow creating instances of the type in Python.

Inheritance:

This field is inherited by subtypes, except it is not inherited by *static types* whose `tp_base` is `NULL` or `&PyBaseObject_Type`.

預設：

For *static types* this field has no default. This means if the slot is defined as `NULL`, the type cannot be called to create new instances; presumably there is some other way to create instances, like a factory function.

freefunc `PyTypeObject`.**tp_free**

An optional pointer to an instance deallocation function. Its signature is:

```
void tp_free(void *self);
```

An initializer that is compatible with this signature is `PyObject_Free()`.

Inheritance:

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement)

預設：

In dynamic subtypes, this field is set to a deallocator suitable to match *PyType_GenericAlloc()* and the value of the *Py_TPFLAGS_HAVE_GC* flag bit.

For static subtypes, `PyBaseObject_Type` uses *PyObject_Del()*.

inquiry *PyTypeObject*.**tp_is_gc**

An optional pointer to a function called by the garbage collector.

The garbage collector needs to know whether a particular object is collectible or not. Normally, it is sufficient to look at the object's type's *tp_flags* field, and check the *Py_TPFLAGS_HAVE_GC* flag bit. But some types have a mixture of statically and dynamically allocated instances, and the statically allocated instances are not collectible. Such types should define this function; it should return `1` for a collectible instance, and `0` for a non-collectible instance. The signature is:

```
int tp_is_gc(PyObject *self);
```

(The only example of this are types themselves. The metatype, *PyType_Type*, defines this function to distinguish between statically and *dynamically allocated types*.)

Inheritance:

This field is inherited by subtypes.

預設：

This slot has no default. If this field is `NULL`, *Py_TPFLAGS_HAVE_GC* is used as the functional equivalent.

PyObject **PyTypeObject*.**tp_bases**

Tuple of base types.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is *initialized*.

For dynamically created classes, the `Py_tp_bases` *slot* can be used instead of the *bases* argument of *PyType_FromSpecWithBases()*. The argument form is preferred.

> **警告：** Multiple inheritance does not work well for statically defined types. If you set `tp_bases` to a tuple, Python will not raise an error, but some slots will only be inherited from the first base.

Inheritance:

This field is not inherited.

PyObject **PyTypeObject*.**tp_mro**

Tuple containing the expanded set of base types, starting with the type itself and ending with `object`, in Method Resolution Order.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is *initialized*.

Inheritance:

This field is not inherited; it is calculated fresh by *PyType_Ready()*.

PyObject **PyTypeObject*.**tp_cache**

Unused. Internal use only.

Inheritance:

This field is not inherited.

PyObject **PyTypeObject*.**tp_subclasses**

List of weak references to subclasses. Internal use only.

Inheritance:

This field is not inherited.

PyObject **PyTypeObject*.**tp_weaklist**

Weak reference list head, for weak references to this type object. Not inherited. Internal use only.

Inheritance:

This field is not inherited.

destructor *PyTypeObject*.**tp_del**

This field is deprecated. Use *tp_finalize* instead.

unsigned int *PyTypeObject*.**tp_version_tag**

Used to index into the method cache. Internal use only.

Inheritance:

This field is not inherited.

destructor *PyTypeObject*.**tp_finalize**

An optional pointer to an instance finalization function. Its signature is:

```
void tp_finalize(PyObject *self);
```

If *tp_finalize* is set, the interpreter calls it once when finalizing an instance. It is called either from the garbage collector (if the instance is part of an isolated reference cycle) or just before the object is deallocated. Either way, it is guaranteed to be called before attempting to break reference cycles, ensuring that it finds the object in a sane state.

tp_finalize should not mutate the current exception status; therefore, a recommended way to write a non-trivial finalizer is:

```
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);
}
```

Also, note that, in a garbage collected Python, *tp_dealloc* may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a garbage collection on any thread). This is not a problem for Python API calls, since the thread on which tp_dealloc is called will own the Global Interpreter Lock (GIL). However, if the object being destroyed in turn destroys objects from some other C or C++ library, care should be taken to ensure that destroying those objects on the thread which called tp_dealloc will not violate any assumptions of the library.

Inheritance:

This field is inherited by subtypes.

在 3.4 版新加入.

在 3.8 版的變更: Before version 3.8 it was necessary to set the *Py_TPFLAGS_HAVE_FINALIZE* flags bit in order for this field to be used. This is no longer required.

也參考:

"Safe object finalization" (**PEP 442**) *vectorcallfunc* `PyTypeObject`.**tp_vectorcall**

Vectorcall function to use for calls of this type object. In other words, it is used to implement *vectorcall* for `type.__call__`. If `tp_vectorcall` is `NULL`, the default call implementation using `__new__()` and `__init__()` is used.

Inheritance:

This field is never inherited.

在 3.9 版新加入: (the field exists since 3.8 but it's only used since 3.9)

12.3.6 Static Types

Traditionally, types defined in C code are *static*, that is, a static `PyTypeObject` structure is defined directly in code and initialized using `PyType_Ready()`.

This results in types that are limited relative to types defined in Python:

- Static types are limited to one base, i.e. they cannot use multiple inheritance.
- Static type objects (but not necessarily their instances) are immutable. It is not possible to add or modify the type object's attributes from Python.
- Static type objects are shared across *sub-interpreters*, so they should not include any subinterpreter-specific state.

Also, since `PyTypeObject` is only part of the *Limited API* as an opaque struct, any extension modules using static types must be compiled for a specific Python minor version.

12.3.7 Heap Types

An alternative to *static types* is *heap-allocated types*, or *heap types* for short, which correspond closely to classes created by Python's `class` statement. Heap types have the `Py_TPFLAGS_HEAPTYPE` flag set.

This is done by filling a `PyType_Spec` structure and calling `PyType_FromSpec()`, `PyType_FromSpecWithBases()`, or `PyType_FromModuleAndSpec()`.

12.4 Number Object Structures

type **PyNumberMethods**

This structure holds pointers to the functions which an object uses to implement the number protocol. Each function is used by the function of similar name documented in the *Number Protocol* section.

Here is the structure definition:

```
typedef struct {
     binaryfunc nb_add;
     binaryfunc nb_subtract;
     binaryfunc nb_multiply;
     binaryfunc nb_remainder;
     binaryfunc nb_divmod;
     ternaryfunc nb_power;
     unaryfunc nb_negative;
     unaryfunc nb_positive;
     unaryfunc nb_absolute;
     inquiry nb_bool;
     unaryfunc nb_invert;
     binaryfunc nb_lshift;
     binaryfunc nb_rshift;
```

(繼續下一頁)

(繼續上一頁)

```
    binaryfunc nb_and;
    binaryfunc nb_xor;
    binaryfunc nb_or;
    unaryfunc nb_int;
    void *nb_reserved;
    unaryfunc nb_float;

    binaryfunc nb_inplace_add;
    binaryfunc nb_inplace_subtract;
    binaryfunc nb_inplace_multiply;
    binaryfunc nb_inplace_remainder;
    ternaryfunc nb_inplace_power;
    binaryfunc nb_inplace_lshift;
    binaryfunc nb_inplace_rshift;
    binaryfunc nb_inplace_and;
    binaryfunc nb_inplace_xor;
    binaryfunc nb_inplace_or;

    binaryfunc nb_floor_divide;
    binaryfunc nb_true_divide;
    binaryfunc nb_inplace_floor_divide;
    binaryfunc nb_inplace_true_divide;

    unaryfunc nb_index;

    binaryfunc nb_matrix_multiply;
    binaryfunc nb_inplace_matrix_multiply;
} PyNumberMethods;
```

備註: Binary and ternary functions must check the type of all their operands, and implement the necessary conversions (at least one of the operands is an instance of the defined type). If the operation is not defined for the given operands, binary and ternary functions must return `Py_NotImplemented`, if another error occurred they must return `NULL` and set an exception.

備註: The *nb_reserved* field should always be `NULL`. It was previously called `nb_long`, and was renamed in Python 3.0.1.

binaryfunc *PyNumberMethods*.**nb_add**

binaryfunc *PyNumberMethods*.**nb_subtract**

binaryfunc *PyNumberMethods*.**nb_multiply**

binaryfunc *PyNumberMethods*.**nb_remainder**

binaryfunc *PyNumberMethods*.**nb_divmod**

ternaryfunc *PyNumberMethods*.**nb_power**

unaryfunc *PyNumberMethods*.**nb_negative**

unaryfunc *PyNumberMethods*.**nb_positive**

unaryfunc *PyNumberMethods*.**nb_absolute**

inquiry *PyNumberMethods*.**nb_bool**

unaryfunc *PyNumberMethods*.**nb_invert** *binaryfunc* `PyNumberMethods`**.nb_lshift**

binaryfunc `PyNumberMethods`**.nb_rshift**

binaryfunc `PyNumberMethods`**.nb_and**

binaryfunc `PyNumberMethods`**.nb_xor**

binaryfunc `PyNumberMethods`**.nb_or**

unaryfunc `PyNumberMethods`**.nb_int**

void *`PyNumberMethods`**.nb_reserved**

unaryfunc `PyNumberMethods`**.nb_float**

binaryfunc `PyNumberMethods`**.nb_inplace_add**

binaryfunc `PyNumberMethods`**.nb_inplace_subtract**

binaryfunc `PyNumberMethods`**.nb_inplace_multiply**

binaryfunc `PyNumberMethods`**.nb_inplace_remainder**

ternaryfunc `PyNumberMethods`**.nb_inplace_power**

binaryfunc `PyNumberMethods`**.nb_inplace_lshift**

binaryfunc `PyNumberMethods`**.nb_inplace_rshift**

binaryfunc `PyNumberMethods`**.nb_inplace_and**

binaryfunc `PyNumberMethods`**.nb_inplace_xor**

binaryfunc `PyNumberMethods`**.nb_inplace_or**

binaryfunc `PyNumberMethods`**.nb_floor_divide**

binaryfunc `PyNumberMethods`**.nb_true_divide**

binaryfunc `PyNumberMethods`**.nb_inplace_floor_divide**

binaryfunc `PyNumberMethods`**.nb_inplace_true_divide**

unaryfunc `PyNumberMethods`**.nb_index**

binaryfunc `PyNumberMethods`**.nb_matrix_multiply**

binaryfunc `PyNumberMethods`**.nb_inplace_matrix_multiply**

12.5 Mapping Object Structures

type **PyMappingMethods**

This structure holds pointers to the functions which an object uses to implement the mapping protocol. It has three members:

lenfunc `PyMappingMethods`**.mp_length**

This function is used by `PyMapping_Size()` and `PyObject_Size()`, and has the same signature. This slot may be set to `NULL` if the object has no defined length.

binaryfunc PyMappingMethods.**mp_subscript**

This function is used by PyObject_GetItem() and PySequence_GetSlice(), and has the same signature as PyObject_GetItem(). This slot must be filled for the PyMapping_Check() function to return 1, it can be NULL otherwise.

objobjargproc PyMappingMethods.**mp_ass_subscript**

This function is used by PyObject_SetItem(), PyObject_DelItem(), PySequence_SetSlice() and PySequence_DelSlice(). It has the same signature as PyObject_SetItem(), but *v* can also be set to NULL to delete an item. If this slot is NULL, the object does not support item assignment and deletion.

12.6 Sequence Object Structures

type **PySequenceMethods**

This structure holds pointers to the functions which an object uses to implement the sequence protocol.

lenfunc PySequenceMethods.**sq_length**

This function is used by PySequence_Size() and PyObject_Size(), and has the same signature. It is also used for handling negative indices via the sq_item and the sq_ass_item slots.

binaryfunc PySequenceMethods.**sq_concat**

This function is used by PySequence_Concat() and has the same signature. It is also used by the + operator, after trying the numeric addition via the nb_add slot.

ssizeargfunc PySequenceMethods.**sq_repeat**

This function is used by PySequence_Repeat() and has the same signature. It is also used by the * operator, after trying numeric multiplication via the nb_multiply slot.

ssizeargfunc PySequenceMethods.**sq_item**

This function is used by PySequence_GetItem() and has the same signature. It is also used by PyObject_GetItem(), after trying the subscription via the mp_subscript slot. This slot must be filled for the PySequence_Check() function to return 1, it can be NULL otherwise.

Negative indexes are handled as follows: if the sq_length slot is filled, it is called and the sequence length is used to compute a positive index which is passed to sq_item. If sq_length is NULL, the index is passed as is to the function.

ssizeobjargproc PySequenceMethods.**sq_ass_item**

This function is used by PySequence_SetItem() and has the same signature. It is also used by PyObject_SetItem() and PyObject_DelItem(), after trying the item assignment and deletion via the mp_ass_subscript slot. This slot may be left to NULL if the object does not support item assignment and deletion.

objobjproc PySequenceMethods.**sq_contains**

This function may be used by PySequence_Contains() and has the same signature. This slot may be left to NULL, in this case PySequence_Contains() simply traverses the sequence until it finds a match.

binaryfunc PySequenceMethods.**sq_inplace_concat**

This function is used by PySequence_InPlaceConcat() and has the same signature. It should modify its first operand, and return it. This slot may be left to NULL, in this case PySequence_InPlaceConcat() will fall back to PySequence_Concat(). It is also used by the augmented assignment +=, after trying numeric in-place addition via the nb_inplace_add slot.

ssizeargfunc PySequenceMethods.**sq_inplace_repeat**

This function is used by PySequence_InPlaceRepeat() and has the same signature. It should modify its first operand, and return it. This slot may be left to NULL, in this case PySequence_InPlaceRepeat() will fall back to PySequence_Repeat(). It is also used by the augmented assignment *=, after trying numeric in-place multiplication via the nb_inplace_multiply slot.

12.7 Buffer Object Structures

type **PyBufferProcs**

This structure holds pointers to the functions required by the *Buffer protocol*. The protocol defines how an exporter object can expose its internal data to consumer objects.

getbufferproc `PyBufferProcs`.**bf_getbuffer**

The signature of this function is:

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

Handle a request to *exporter* to fill in *view* as specified by *flags*. Except for point (3), an implementation of this function MUST take these steps:

(1) Check if the request can be met. If not, raise `BufferError`, set `view->obj` to `NULL` and return `-1`.

(2) Fill in the requested fields.

(3) Increment an internal counter for the number of exports.

(4) Set `view->obj` to *exporter* and increment `view->obj`.

(5) Return `0`.

If *exporter* is part of a chain or tree of buffer providers, two main schemes can be used:

- Re-export: Each member of the tree acts as the exporting object and sets `view->obj` to a new reference to itself.
- Redirect: The buffer request is redirected to the root object of the tree. Here, `view->obj` will be a new reference to the root object.

The individual fields of *view* are described in section *Buffer structure*, the rules how an exporter must react to specific requests are in section *Buffer request types*.

All memory pointed to in the `Py_buffer` structure belongs to the exporter and must remain valid until there are no consumers left. `format`, `shape`, `strides`, `suboffsets` and `internal` are read-only for the consumer.

`PyBuffer_FillInfo()` provides an easy way of exposing a simple bytes buffer while dealing correctly with all request types.

`PyObject_GetBuffer()` is the interface for the consumer that wraps this function.

releasebufferproc `PyBufferProcs`.**bf_releasebuffer**

The signature of this function is:

```
void (PyObject *exporter, Py_buffer *view);
```

Handle a request to release the resources of the buffer. If no resources need to be released, `PyBufferProcs.bf_releasebuffer` may be `NULL`. Otherwise, a standard implementation of this function will take these optional steps:

(1) Decrement an internal counter for the number of exports.

(2) If the counter is `0`, free all memory associated with *view*.

The exporter MUST use the `internal` field to keep track of buffer-specific resources. This field is guaranteed to remain constant, while a consumer MAY pass a copy of the original buffer as the *view* argument.

This function MUST NOT decrement `view->obj`, since that is done automatically in `PyBuffer_Release()` (this scheme is useful for breaking reference cycles).

`PyBuffer_Release()` is the interface for the consumer that wraps this function.

12.8 Async Object Structures

在 3.5 版新加入.

type **PyAsyncMethods**

This structure holds pointers to the functions required to implement *awaitable* and *asynchronous iterator* objects.

Here is the structure definition:

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
    sendfunc am_send;
} PyAsyncMethods;
```

unaryfunc *PyAsyncMethods*.**am_await**

The signature of this function is:

```
PyObject *am_await(PyObject *self);
```

The returned object must be an *iterator*, i.e. *PyIter_Check()* must return 1 for it.

This slot may be set to NULL if an object is not an *awaitable*.

unaryfunc *PyAsyncMethods*.**am_aiter**

The signature of this function is:

```
PyObject *am_aiter(PyObject *self);
```

Must return an *asynchronous iterator* object. See `__anext__()` for details.

This slot may be set to NULL if an object does not implement asynchronous iteration protocol.

unaryfunc *PyAsyncMethods*.**am_anext**

The signature of this function is:

```
PyObject *am_anext(PyObject *self);
```

Must return an *awaitable* object. See `__anext__()` for details. This slot may be set to NULL.

sendfunc *PyAsyncMethods*.**am_send**

The signature of this function is:

```
PySendResult am_send(PyObject *self, PyObject *arg, PyObject **result);
```

See *PyIter_Send()* for details. This slot may be set to NULL.

在 3.10 版新加入.

12.9 Slot Type typedefs

typedef *PyObject* *(***allocfunc**)(*PyTypeObject* *cls, *Py_ssize_t* nitems)

Part of the Stable ABI. The purpose of this function is to separate memory allocation from memory initialization. It should return a pointer to a block of memory of adequate length for the instance, suitably aligned, and initialized to zeros, but with *ob_refcnt* set to 1 and *ob_type* set to the type argument. If the type's *tp_itemsize* is non-zero, the object's *ob_size* field should be initialized to *nitems* and the length of the allocated memory block should be `tp_basicsize + nitems*tp_itemsize`, rounded up to a multiple of `sizeof(void*)`; otherwise, *nitems* is not used and the length of the block should be *tp_basicsize*.

This function should not do any other instance initialization, not even to allocate additional memory; that should be done by *tp_new*.

typedef void (***destructor**)(*PyObject**)

Part of the Stable ABI.

typedef void (***freefunc**)(void*)

請見*tp_free*。

typedef *PyObject* *(***newfunc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the Stable ABI. 請見*tp_new*。

typedef int (***initproc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the Stable ABI. 請見*tp_init*。

typedef *PyObject* *(***reprfunc**)(*PyObject**)

Part of the Stable ABI. 請見*tp_repr*。

typedef *PyObject* *(***getattrfunc**)(*PyObject* *self, char *attr)

Part of the Stable ABI. Return the value of the named attribute for the object.

typedef int (***setattrfunc**)(*PyObject* *self, char *attr, *PyObject* *value)

Part of the Stable ABI. Set the value of the named attribute for the object. The value argument is set to NULL to delete the attribute.

typedef *PyObject* *(***getattrofunc**)(*PyObject* *self, *PyObject* *attr)

Part of the Stable ABI. Return the value of the named attribute for the object.

請見*tp_getattro*。

typedef int (***setattrofunc**)(*PyObject* *self, *PyObject* *attr, *PyObject* *value)

Part of the Stable ABI. Set the value of the named attribute for the object. The value argument is set to NULL to delete the attribute.

請見*tp_setattro*。

typedef *PyObject* *(***descrgetfunc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the Stable ABI. 請見*tp_descr_get*。

typedef int (***descrsetfunc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the Stable ABI. 請見*tp_descr_set*。

typedef *Py_hash_t* (***hashfunc**)(*PyObject**)

Part of the Stable ABI. 請見*tp_hash*。

typedef *PyObject* *(***richcmpfunc**)(*PyObject**, *PyObject**, int)

Part of the Stable ABI. 請見*tp_richcompare*。

typedef *PyObject* *(***getiterfunc**)(*PyObject**)

Part of the Stable ABI. 請見*tp_iter*。

typedef *PyObject* *(***iternextfunc**)(*PyObject**)

Part of the Stable ABI. 請見*tp_iternext*。

typedef *Py_ssize_t* (***lenfunc**)(*PyObject**)

Part of the Stable ABI.

typedef int (***getbufferproc**)(*PyObject**, *Py_buffer**, int)

typedef void (***releasebufferproc**)(*PyObject**, *Py_buffer**)

typedef *PyObject* *(***unaryfunc**)(*PyObject**)

Part of the Stable ABI.

typedef *PyObject* *(***binaryfunc**)(*PyObject**, *PyObject**)

Part of the Stable ABI.

typedef *PySendResult* (***sendfunc**)(*PyObject**, *PyObject**, *PyObject***)

請見*am_send*。

typedef *PyObject* *(***ternaryfunc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the Stable ABI.

typedef *PyObject* *(***ssizeargfunc**)(*PyObject**, *Py_ssize_t*)

Part of the Stable ABI.

typedef int (***ssizeobjargproc**)(*PyObject**, *Py_ssize_t*, *PyObject**)

Part of the Stable ABI.

typedef int (***objobjproc**)(*PyObject**, *PyObject**)

Part of the Stable ABI.

typedef int (***objobjargproc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the Stable ABI.

12.10 范例

The following are simple examples of Python type definitions. They include common usage you may encounter. Some demonstrate tricky corner cases. For more examples, practical info, and a tutorial, see defining-new-types and new-types-topics.

A basic *static type*:

```
typedef struct {
    PyObject_HEAD
    const char *data;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_new = myobj_new,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
};
```

You may also find older code (especially in the CPython code base) with a more verbose initializer:

```
static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "mymod.MyObject",               /* tp_name */
    sizeof(MyObject),               /* tp_basicsize */
    0,                              /* tp_itemsize */
    (destructor)myobj_dealloc,      /* tp_dealloc */
    0,                              /* tp_vectorcall_offset */
    0,                              /* tp_getattr */
    0,                              /* tp_setattr */
    0,                              /* tp_as_async */
    (reprfunc)myobj_repr,           /* tp_repr */
    0,                              /* tp_as_number */
    0,                              /* tp_as_sequence */
    0,                              /* tp_as_mapping */
```

(繼續下一頁)

(繼續上一頁)

```
    0,                                  /* tp_hash */
    0,                                  /* tp_call */
    0,                                  /* tp_str */
    0,                                  /* tp_getattro */
    0,                                  /* tp_setattro */
    0,                                  /* tp_as_buffer */
    0,                                  /* tp_flags */
    PyDoc_STR("My objects"),            /* tp_doc */
    0,                                  /* tp_traverse */
    0,                                  /* tp_clear */
    0,                                  /* tp_richcompare */
    0,                                  /* tp_weaklistoffset */
    0,                                  /* tp_iter */
    0,                                  /* tp_iternext */
    0,                                  /* tp_methods */
    0,                                  /* tp_members */
    0,                                  /* tp_getset */
    0,                                  /* tp_base */
    0,                                  /* tp_dict */
    0,                                  /* tp_descr_get */
    0,                                  /* tp_descr_set */
    0,                                  /* tp_dictoffset */
    0,                                  /* tp_init */
    0,                                  /* tp_alloc */
    myobj_new,                          /* tp_new */
};
```

A type that supports weakrefs, instance dicts, and hashing:

```
typedef struct {
    PyObject_HEAD
    const char *data;
    PyObject *inst_dict;
    PyObject *weakreflist;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_weaklistoffset = offsetof(MyObject, weakreflist),
    .tp_dictoffset = offsetof(MyObject, inst_dict),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
    .tp_new = myobj_new,
    .tp_traverse = (traverseproc)myobj_traverse,
    .tp_clear = (inquiry)myobj_clear,
    .tp_alloc = PyType_GenericNew,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
    .tp_hash = (hashfunc)myobj_hash,
    .tp_richcompare = PyBaseObject_Type.tp_richcompare,
};
```

A str subclass that cannot be subclassed and cannot be called to create instances (e.g. uses a separate factory func) using *Py_TPFLAGS_DISALLOW_INSTANTIATION* flag:

```
typedef struct {
    PyUnicodeObject raw;
    char *extra;
} MyStr;
```

(繼續下一頁)

(繼續上一頁)

```
static PyTypeObject MyStr_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyStr",
    .tp_basicsize = sizeof(MyStr),
    .tp_base = NULL,  // set to &PyUnicode_Type in module init
    .tp_doc = PyDoc_STR("my custom str"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_DISALLOW_INSTANTIATION,
    .tp_repr = (reprfunc)myobj_repr,
};
```

The simplest *static type* with fixed-length instances:

```
typedef struct {
    PyObject_HEAD
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
};
```

The simplest *static type* with variable-length instances:

```
typedef struct {
    PyObject_VAR_HEAD
    const char *data[1];
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject) - sizeof(char *),
    .tp_itemsize = sizeof(char *),
};
```

12.11 Supporting Cyclic Garbage Collection

Python's support for detecting and collecting garbage which involves circular references requires support from object types which are "containers" for other objects which may also be containers. Types which do not store references to other objects, or which only store references to atomic types (such as numbers or strings), do not need to provide any explicit support for garbage collection.

To create a container type, the `tp_flags` field of the type object must include the `Py_TPFLAGS_HAVE_GC` and provide an implementation of the `tp_traverse` handler. If instances of the type are mutable, a `tp_clear` implementation must also be provided.

`Py_TPFLAGS_HAVE_GC`

Objects with a type with this flag set must conform with the rules documented here. For convenience these objects will be referred to as container objects.

Constructors for container types must conform to two rules:

1. The memory for the object must be allocated using `PyObject_GC_New` or `PyObject_GC_NewVar`.
2. Once all the fields which may contain references to other containers are initialized, it must call `PyObject_GC_Track()`.

Similarly, the deallocator for the object must conform to a similar pair of rules:

1. Before fields which refer to other containers are invalidated, `PyObject_GC_UnTrack()` must be called.

2. The object's memory must be deallocated using *PyObject_GC_Del()*.

> **警告：** If a type adds the Py_TPFLAGS_HAVE_GC, then it *must* implement at least a *tp_traverse* handler or explicitly use one from its subclass or subclasses.
>
> When calling *PyType_Ready()* or some of the APIs that indirectly call it like *PyType_FromSpecWithBases()* or *PyType_FromSpec()* the interpreter will automatically populate the *tp_flags*, *tp_traverse* and *tp_clear* fields if the type inherits from a class that implements the garbage collector protocol and the child class does *not* include the *Py_TPFLAGS_HAVE_GC* flag.

PyObject_GC_New(TYPE, typeobj)

Analogous to *PyObject_New* but for container objects with the *Py_TPFLAGS_HAVE_GC* flag set.

PyObject_GC_NewVar(TYPE, typeobj, size)

Analogous to *PyObject_NewVar* but for container objects with the *Py_TPFLAGS_HAVE_GC* flag set.

PyObject_GC_Resize(TYPE, op, newsize)

Resize an object allocated by *PyObject_NewVar*. Returns the resized object of type `TYPE*` (refers to any C type) or `NULL` on failure.

op must be of type *PyVarObject** and must not be tracked by the collector yet. *newsize* must be of type *Py_ssize_t*.

void **PyObject_GC_Track**(*PyObject* *op)

Part of the Stable ABI. Adds the object *op* to the set of container objects tracked by the collector. The collector can run at unexpected times so objects must be valid while being tracked. This should be called once all the fields followed by the *tp_traverse* handler become valid, usually near the end of the constructor.

int **PyObject_IS_GC**(*PyObject* *obj)

Returns non-zero if the object implements the garbage collector protocol, otherwise returns 0.

The object cannot be tracked by the garbage collector if this function returns 0.

int **PyObject_GC_IsTracked**(*PyObject* *op)

Part of the Stable ABI *since version 3.9.* Returns 1 if the object type of *op* implements the GC protocol and *op* is being currently tracked by the garbage collector and 0 otherwise.

This is analogous to the Python function `gc.is_tracked()`.

在 3.9 版新加入.

int **PyObject_GC_IsFinalized**(*PyObject* *op)

Part of the Stable ABI *since version 3.9.* Returns 1 if the object type of *op* implements the GC protocol and *op* has been already finalized by the garbage collector and 0 otherwise.

This is analogous to the Python function `gc.is_finalized()`.

在 3.9 版新加入.

void **PyObject_GC_Del**(void *op)

Part of the Stable ABI. Releases memory allocated to an object using *PyObject_GC_New* or *PyObject_GC_NewVar*.

void **PyObject_GC_UnTrack**(void *op)

Part of the Stable ABI. Remove the object *op* from the set of container objects tracked by the collector. Note that *PyObject_GC_Track()* can be called again on this object to add it back to the set of tracked objects. The deallocator (*tp_dealloc* handler) should call this for the object before any of the fields used by the *tp_traverse* handler become invalid.

在 3.8 版的變更: The `_PyObject_GC_TRACK()` and `_PyObject_GC_UNTRACK()` macros have been removed from the public C API.

The `tp_traverse` handler accepts a function parameter of this type:

typedef int (***visitproc**)(*PyObject* *object, void *arg)

Part of the Stable ABI. Type of the visitor function passed to the `tp_traverse` handler. The function should be called with an object to traverse as *object* and the third parameter to the `tp_traverse` handler as *arg*. The Python core uses several visitor functions to implement cyclic garbage detection; it's not expected that users will need to write their own visitor functions.

The `tp_traverse` handler must have the following type:

typedef int (***traverseproc**)(*PyObject* *self, *visitproc* visit, void *arg)

Part of the Stable ABI. Traversal function for a container object. Implementations must call the *visit* function for each object directly contained by *self*, with the parameters to *visit* being the contained object and the *arg* value passed to the handler. The *visit* function must not be called with a `NULL` object argument. If *visit* returns a non-zero value that value should be returned immediately.

To simplify writing `tp_traverse` handlers, a `Py_VISIT()` macro is provided. In order to use this macro, the `tp_traverse` implementation must name its arguments exactly *visit* and *arg*:

void **Py_VISIT**(*PyObject* *o)

If *o* is not `NULL`, call the *visit* callback, with arguments *o* and *arg*. If *visit* returns a non-zero value, then return it. Using this macro, `tp_traverse` handlers look like:

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

The `tp_clear` handler must be of the `inquiry` type, or `NULL` if the object is immutable.

typedef int (***inquiry**)(*PyObject* *self)

Part of the Stable ABI. Drop references that may have created reference cycles. Immutable objects do not have to define this method since they can never directly create reference cycles. Note that the object must still be valid after calling this method (don't just call `Py_DECREF()` on a reference). The collector will call this method if it detects that this object is involved in a reference cycle.

12.11.1 Controlling the Garbage Collector State

The C-API provides the following functions for controlling garbage collection runs.

Py_ssize_t **PyGC_Collect**(void)

Part of the Stable ABI. Perform a full garbage collection, if the garbage collector is enabled. (Note that `gc.collect()` runs it unconditionally.)

Returns the number of collected + unreachable objects which cannot be collected. If the garbage collector is disabled or already collecting, returns `0` immediately. Errors during garbage collection are passed to `sys.unraisablehook`. This function does not raise exceptions.

int **PyGC_Enable**(void)

Part of the Stable ABI *since version 3.10.* Enable the garbage collector: similar to `gc.enable()`. Returns the previous state, 0 for disabled and 1 for enabled.

在 3.10 版新加入.

int **PyGC_Disable**(void)

Part of the Stable ABI *since version 3.10.* Disable the garbage collector: similar to `gc.disable()`. Returns the previous state, 0 for disabled and 1 for enabled.

在 3.10 版新加入.

int **PyGC_IsEnabled**(void)

Part of the Stable ABI *since version 3.10.* Query the state of the garbage collector: similar to `gc.isenabled()`. Returns the current state, 0 for disabled and 1 for enabled.

在 3.10 版新加入.

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 >= ...`。

This version is also available via the symbol *Py_Version*.

const unsigned long **Py_Version**

Part of the Stable ABI *since version 3.11.* 編碼為單個常數整數的 Python 執行環境版本號，格式與*PY_VERSION_HEX* 巨集相同。這包含在執行環境使用的 Python 版本。

在 3.11 版新加入.

所有提到的巨集都定義在 Include/patchlevel.h。

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 在可以標准函式庫中以 `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，以識別字（identifier，例如 `name=`）開頭的引數，或是以 `**` 後面 dictionary（字典）內的值被傳遞的引數。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）作為 `*` 之後的*iterable*（可迭代物件）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，請參閱 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。

另請參閱術語表的*parameter*（參數）條目、常見問題中的引數和參數之間的差異，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（非同步產生器）
一個會回傳*asynchronous generator iterator*（非同步產生器迭代器）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` 迴圈的值。

這個術語通常用來表示一個非同步產生器函式，但在某些情境中，也可能是表示非同步產生器迭代器 *(asynchronous generator iterator)*。萬一想表達的意思不夠清楚，那就使用完整的術語，以避免歧義。

一個非同步產生器函式可能包含 `await` 運算式，以及 `async for` 和 `async with` 陳述式。

asynchronous generator iterator（非同步產生器迭代器）
一個由*asynchronous generator*（非同步產生器）函式所建立的物件。

這是一個*asynchronous iterator*（非同步迭代器），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產生器的函式主體，直到遇到下一個 `yield` 運算式。

每個 `yield` 會暫停處理程序，並記住位置執行狀態（包括區域變數及擱置中的 try 陳述式）。當非同步產生器迭代器以另一個被 `__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。請參閱 **PEP 492** 和 **PEP 525**。

asynchronous iterable（非同步可迭代物件）
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*（非同步迭代器）。由 **PEP 492** 引入。

asynchronous iterator（非同步迭代器）
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` 會解析非同步迭代器的 `__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 所定義之識別符 (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。另請參閱 **PEP 492**。

BDFL

Benevolent Dictator For Life（終身仁慈獨裁者），又名 Guido van Rossum，Python 的創造者。

binary file（二進制檔案）

A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

另請參考*text file*（文字檔案），它是一個能夠讀取和寫入 `str` 物件的檔案物件。

borrowed reference（借用參照）

In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

對*borrowed reference* 呼叫`Py_INCREF()` 以將它原地 (in-place) 轉換為*strong reference* 是被建議的做法，除非該物件不能在最後一次使用借用參照之前被銷毀。`Py_NewRef()` 函式可用於建立一個新的*strong reference*。

bytes-like object（類位元組串物件）

一個支援緩衝協定 *(Buffer Protocol)*且能夠匯出 C-*contiguous* 緩衝區的物件。這包括所有的 `bytes`、`bytearray` 和 `array.array` 物件，以及許多常見的 `memoryview` 物件。類位元組串物件可用於處理二進制資料的各種運算；這些運算包括壓縮、儲存至二進制檔案和透過 socket（插座）發送。

有些運算需要二進制資料是可變的。說明文件通常會將這些物件稱為「可讀寫的類位元組串物件」。可變緩衝區的物件包括 `bytearray`，以及 `bytearray` 的 `memoryview`。其他的運算需要讓二進制資料被儲存在不可變物件（「唯讀的類位元組串物件」）中；這些物件包括 `bytes`，以及 `bytes` 物件的 `memoryview`。

bytecode（位元組碼）

Python 的原始碼會被編譯成位元組碼，它是 Python 程式在 CPython 直譯器中的內部表示法。該位元組碼也會被暫存在 `.pyc` 檔案中，以便第二次執行同一個檔案時能夠更快速（可以不用從原始碼重新編譯為位元組碼）。這種「中間語言 (intermediate language)」據說是運行在一個*virtual machine*（虛擬機器）上，該虛擬機器會執行與每個位元組碼對應的機器碼 (machine code)。要注意的是，位元組碼理論上是無法在不同的 Python 虛擬機器之間運作的，也不能在不同版本的 Python 之間保持穩定。

位元組碼的指令列表可以在 dis 模組的說明文件中找到。

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）

作為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（類別）

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（類別變數）

一個在 class 中被定義，且應該只能在 class 層次（意即不是在 class 的實例中）被修改的變數。

complex number（複數）

一個我們熟悉的實數系統的擴充，在此所有數字都會被表示為一個實部和一個虛部之和。虛數就是虛數單位（`-1` 的平方根）的實數倍，此單位通常在數學中被寫為 `i`，在工程學中被寫為 `j`。Python 內建了對複數的支援，它是用後者的記法來表示虛數；虛部會帶著一個後綴的 `j` 被編寫，例如 `3+1j`。若要將 `math` 模組內的工具等效地用於複數，請使用 `cmath` 模組。複數的使用是一個相當進階的數學功能。如果你沒有察覺到對它們的需求，那麼幾乎能確定你可以安全地忽略它們。

context manager（情境管理器）
An object which controls the environment seen in a `with` statement by defining `__enter__()` and `__exit__()` methods. See **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 程式語言的標準實作 (canonical implementation)，被發布在 python.org 上。「CPython」這個術語在必要時被使用，以區分此實作與其它語言的實作，例如 Jython 或 IronPython。

decorator（裝飾器）
一個函式，它會回傳另一個函式，通常它會使用 `@wrapper` 語法，被應用□一種函式的變□ (function transformation)。裝飾器的常見範例是 `classmethod()` 和 `staticmethod()`。

裝飾器語法只是語法糖。以下兩個函式定義在語義上是等效的：

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

Class 也存在相同的概念，但在那□比較不常用。關於裝飾器的更多□容，請參□函式定義和 class 定義的□明文件。

descriptor（描述器）
Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

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）
An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

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)}` 會產生一個字典，它包含了鍵 `n` 映射到值 `n ** 2`。請參閱 comprehensions。

dictionary view（字典檢視）
從 `dict.keys()`、`dict.values()` 及 `dict.items()` 回傳的物件被稱為字典檢視。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。若要強制將字典檢視轉為完整的 list（串列），須使用 `list(dictview)`。請參閱 dict-views。

docstring（說明字串）
A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing（鴨子型別）
一種程式設計風格，它不是藉由檢查一個物件的型別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。（「如果它看起來像一隻鴨子而且叫起來像一隻鴨子，那麼它一定是一隻鴨子。」）因為強調介面而非特定型別，精心設計的程式碼能讓多形替代 (polymorphic substitution) 來增進它的靈活性。鴨子型別要避免使用 `type()` 或 `isinstance()` 進行測試。（但是請注意，鴨子型別可以用抽象基底類別 *(abstract base class)* 來補充。）然而，它通常會採用 `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，並在該假設被推翻時再捕獲例外。這種乾淨且快速的風格，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）
一段可以被評估並求值的語法。換句話說，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，並非所有的 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'` 為前綴的字串文本通常被稱為「f 字串」，它是格式化的字串文本的縮寫。另請參閱 **PEP 498**。

file object（檔案物件）
An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

實際上，有三種檔案物件：原始的二進制檔案、緩衝的二進制檔案和文字檔案。它們的介面在 `io` 模組中被定義。建立檔案物件的標準方法是使用 `open()` 函式。

file-like object（類檔案物件）
file object（檔案物件）的同義字。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）
Python 所使用的一種編碼和錯誤處理函式，用來解碼來自作業系統的位元組，以及將 Unicode 編碼到作業系統。

檔案系統編碼必須保證能成功解碼所有小於 128 的位元組。如果檔案系統編碼無法提供此保證，則 API 函式會引發 `UnicodeError`。

`sys.getfilesystemencoding()` 和 `sys.getfilesystemencodeerrors()` 函式可用於取得檔案系統編碼和錯誤處理函式。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）會在 Python 啟動時由*PyConfig_Read()* 函式來配置：請參閱*filesystem_encoding*，以及*PyConfig* 的成

員*filesystem_errors*。

另請參閱*locale encoding*（區域編碼）。

finder（尋檢器）
一個物件，它會嘗試為正在被 import 的模組尋找*loader*（載入器）。

從 Python 3.3 開始，有兩種類型的尋檢器：*元路徑尋檢器 (meta path finder)* 會使用 `sys.meta_path`，而*路徑項目尋檢器 (path entry finder)* 會使用 `sys.path_hooks`。

請參閱 **PEP 302**、**PEP 420** 和 **PEP 451** 以了解更多細節。

floor division（向下取整除法）
向下無條件舍去到最接近整數的數學除法。向下取整除法的運算子是 `//`。例如，運算式 `11 // 4` 的計算結果為 `2`，與 float（浮點數）真除法所回傳的 `2.75` 不同。請注意，`(-11) // 4` 的結果是 `-3`，因為是 `-2.75` 被向下無條件舍去。請參閱 **PEP 238**。

function（函式）
一連串的陳述式，它能夠向呼叫者回傳一些值。它也可以被傳遞零個或多個引數，這些引數可被使用於函式本體的執行。另請參閱*parameter*（參數）、*method*（方法），以及 function 章節。

function annotation（函式註釋）
函式參數或回傳值的一個*annotation*（註釋）。

函式註釋通常被使用於型別提示：例如，這個函式預期會得到兩個 `int` 引數，並會有一個 `int` 回傳值：

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

函式註釋的語法在 function 章節有詳細解釋。

請參閱*variable annotation* 和 **PEP 484**，皆有此功能的描述。關於註釋的最佳實踐方法，另請參閱 annotations-howto。

__future__
future 陳述式：`from __future__ import <feature>`，會指示編譯器使用那些在 Python 未來的發布版本中將成為標準的語法或語義，來編譯當前的模組。而 `__future__` 模組則記錄了 *feature*（功能）可能的值。透過 import 此模組並對其變數求值，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）成為預設的功能：

```
>>> 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)，以及一個能夠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（產生器）
一個會回傳*generator iterator*（產生器疊代器）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，能產生一系列的值，這些值可用於 for 迴圈，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產生器函式，但在某些情境中，也可能是表示產生器疊代器。萬一想表達的意思不夠清楚，那就使用完整的術語，以避免歧義。

generator iterator（產生器疊代器）
一個由*generator*（產生器）函式所建立的物件。

每個 `yield` 會暫停處理程序，並記住位置執行狀態（包括區域變數及擱置中的 try 陳述式）。當產生器疊代器回復時，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□實作相同的運算。呼叫期間應該使用哪種實作，是由調度演算法 (dispatch algorithm) 來□定。

另請參□*single dispatch*（單一調度）術語表條目、`functools.singledispatch()` 裝飾器和 **PEP 443**。

generic type（泛型型□）

一個能□被參數化 (parameterized) 的*type*（型□）；通常是一個 容器型□，像是 `list` 和 `dict`。它被用於型□提示和□釋。

詳情請參□泛型□名型□、**PEP 483**、**PEP 484**、**PEP 585** 和 `typing` 模組。

GIL

請參□*global interpreter lock*（全域直譯器鎖）。

global interpreter lock（全域直譯器鎖）

CPython 直譯器所使用的機制，用以確保每次都只有一個執行緒能執行 Python 的*bytecode*（位元組碼）。透過使物件模型（包括關鍵的□建型□，如 `dict`）自動地避免□行存取 (concurrent access) 的危險，此機制可以簡化 CPython 的實作。鎖定整個直譯器，會使直譯器更容易成□多執行緒 (multi-threaded)，但代價是會犧牲掉多處理器的機器能□提供的一大部分平行性 (parallelism)。

然而，有些擴充模組，無論是標准的或是第三方的，它們被設計成在執行壓縮或雜□等計算密集 (computationally intensive) 的任務時，可以解除 GIL。另外，在執行 I/O 時，GIL 總是會被解除。

過去對於建立「無限制執行緒」直譯器（以更高的精細度鎖定共享資料的直譯器）的努力□未成功，因□在一般的單一處理器情□下，效能會有所損失。一般認□，若要克服這個效能問題，會使實作變得□雜許多，進而付出更高的維護成本。

hash-based pyc（雜□架構的 pyc）

一個位元組碼 (bytecode) 暫存檔，它使用雜□值而不是對應原始檔案的最後修改時間，來確定其有效性。請參□ pyc-invalidation。

hashable（可雜□的）

An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

可雜□性 (hashability) 使一個物件可用作 dictionary（字典）的鍵和 set（集合）的成員，因□這些資料結構都在其□部使用了雜□值。

大多數的 Python 不可變□建物件都是可雜□的；可變的容器（例如 list 或 dictionary）□不是；而不可變的容器（例如 tuple（元組）和 frozenset），只有當它們的元素是可雜□的，它們本身才是可雜□的。若物件是使用者自定 class 的實例，則這些物件會被預設□可雜□的。它們在互相比較時都是不相等的（除非它們與自己比較），而它們的雜□值則是衍生自它們的 `id()`。

IDLE

Python 的 Integrated Development and Learning Environment（整合開發與學習環境）。idle 是一個基本的編輯器和直譯器環境，它和 Python 的標准發行版本一起被提供。

immutable（不可變物件）

一個具有固定值的物件。不可變物件包括數字、字串和 tuple（元組）。這類物件是不能被改變的。如果一個不同的值必須被儲存，則必須建立一個新的物件。它們在需要□定雜□值的地方，扮演重要的角色，例如 dictionary（字典）中的一個鍵。

import path（引入路徑）

一個位置（或路徑項目）的列表，而那些位置就是在 import 模組時，會被*path based finder*（基於路

徑的尋檢器）搜尋模組的位置。在 import 期間，此位置列表通常是來自 `sys.path`，但對於子套件 (subpackage) 而言，它也可能是來自父套件的 `__path__` 屬性。

importing（引入）
一個過程。一個模組中的 Python 程式碼可以透過此過程，被另一個模組中的 Python 程式碼使用。

importer（引入器）
一個能夠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，立即執行它們並且看到它們的結果。只要啟動 `python`，不需要任何引數（可能藉由從你的電腦的主選單選擇它）。這是測試新想法或檢查模塊和包的非常強大的方法（請記住 help(x)）。

interpreted（直譯的）
Python 是一種直譯語言，而不是編譯語言，不過這個區分可能有些模糊，因為有位元組碼 (bytecode) 編譯器的存在。這表示原始檔案可以直接被運行，而不需明確地建立另一個執行檔，然後再執行它。直譯語言通常比編譯語言有更短的開發 / 除錯週期，不過它們的程式通常也運行得較慢。另請參閱*interactive*（互動的）。

interpreter shutdown（直譯器關閉）
當 Python 直譯器被要求關閉時，它會進入一個特殊階段，在此它逐漸釋放所有被配置的資源，例如模組和各種關鍵內部結構。它也會多次呼叫垃圾回收器 *(garbage collector)*。這能夠觸發使用者自定的解構函式 (destructor) 或弱引用的回呼 (weakref callback)，並執行其中的程式碼。在關閉階段被執行的程式碼會遇到各種例外，因為它所依賴的資源可能不再有作用了（常見的例子是函式庫模組或是警告機制）。

直譯器關閉的主要原因，是 `__main__` 模組或正被運行的腳本已經執行完成。

iterable（可疊代物件）
An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *sequence* semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator（疊代器）
An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

在 typeiter 文中可以找到更多資訊。

CPython 實作細節： CPython does not consistently apply the requirement that an iterator define `__iter__()`.

key function（鍵函式）
鍵函式或理序函式 (collation function) 是一個可呼叫 (callable) 函式，它會回傳一個用於排序 (sorting) 或定序 (ordering) 的值。例如，`locale.strxfrm()` 被用來產生一個了解區域特定排序慣例的排序鍵。

Python 中的許多工具，都接受以鍵函式來控制元素被定序或分組的方式。它們包括 `min()`、`max()`、`sorted()`、`list.sort()`、`heapq.merge()`、`heapq.nsmallest()`、`heapq.nlargest()`

和 `itertools.groupby()`。

有幾種方法可以建立一個鍵函式。例如，`str.lower()` method 可以作為不分大小寫排序的鍵函式。或者，一個鍵函式也可以從 `lambda` 運算式被建造，例如 `lambda r: (r[0], r[2])`。另外，`operator.attrgetter()`、`operator.itemgetter()` 和 `operator.methodcaller()` 為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，請參閱如何排序。

keyword argument（關鍵字引數）
請參閱*argument*（引數）。

lambda
由單一*expression*（運算式）所組成的一個匿名行內函式 (inline function)，於該函式被呼叫時求值。建立 lambda 函式的語法是 `lambda [parameters]: expression`

LBYL
Look before you leap.（三思而後行。）這種編碼風格會在進行呼叫或查找之前，明確地測試先決條件。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」之間引入了競爭條件 (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 編碼方式來解決。

list（串列）
A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is $O(1)$.

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，並將處理結果以一個 list 回傳的簡要方法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 會產生一個字串 list，其中包含 0 到 255 範圍內，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）
一個能夠載入模組的物件。它必須定義一個名為 `load_module()` 的 method（方法）。載入器通常是被*finder*（尋檢器）回傳。更多細節請參閱 **PEP 302**，關於*abstract base class*（抽象基底類別），請參閱 `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"` 作為區域編碼。

`locale.getencoding()` can be used to get the locale encoding.

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（抽象基底類別）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，請參閱 `importlib.abc.MetaPathFinder`。

metaclass（元類別）
一種 class 的 class。Class 定義過程會建立一個 class 名稱、一個 class dictionary（字典），以及一個

base class（基底類□）的列表。Metaclass 負責接受這三個引數，□建立該 class。大多數的物件導向程式語言會提供一個預設的實作。Python 的特□之處在於它能□建立自訂的 metaclass。大部分的使用者從未需要此工具，但是當需要時，metaclass 可以提供□大且優雅的解□方案。它們已被用於記□屬性存取、增加執行緒安全性、追□物件建立、實作單例模式 (singleton)，以及許多其他的任務。

更多資訊可以在 metaclasses 章節中找到。

method（方法）
一個在 class 本體□被定義的函式。如果 method 作□其 class 實例的一個屬性被呼叫，則它將會得到該實例物件成□它的第一個*argument*（引數）（此引數通常被稱□ `self`）。請參□*function*（函式）和*nested scope*（巢狀作用域）。

method resolution order（方法解析順序）
方法解析順序是在查找某個成員的過程中，base class（基底類□）被搜尋的順序。關於第 2.3 版至今，Python 直譯器所使用的演算法細節，請參□ Python 2.3 版方法解析順序。

module（模組）
一個擔任 Python 程式碼的組織單位 (organizational unit) 的物件。模組有一個命名空間，它包含任意的 Python 物件。模組是藉由*importing* 的過程，被載入至 Python。

另請參□*package*（套件）。

module spec（模組規格）
一個命名空間，它包含用於載入模組的 import 相關資訊。它是 `importlib.machinery.ModuleSpec` 的一個實例。

MRO
請參□*method resolution order*（方法解析順序）。

mutable（可變物件）
可變物件可以改變它們的值，但維持它們的 `id()`。另請參□*immutable*（不可變物件）。

named tuple（附名元組）
術語「named tuple（附名元組）」是指從 tuple 繼承的任何型□或 class，且它的可索引 (indexable) 元素也可以用附名屬性來存取。這些型□或 class 也可以具有其他的特性。

有些□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
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand, or it can be created by inheriting `typing.NamedTuple`, or with the factory function `collections.namedtuple()`. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

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）
能夠參照外層定義 (enclosing definition) 中的變數的能力。舉例來說，一個函式如果是在另一個函式中被定義，則它便能夠參照外層函式中的變數。請注意，在預設情況下，巢狀作用域僅適用於參照，而無法用於賦值。區域變數能在最內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（新式類別）
Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

object（物件）
具有狀態（屬性或值）及被定義的行為（method）的任何資料。它也是任何*new-style class*（新式類別）的最終 base class（基底類別）。

package（套件）
一個 Python 的*module*（模組），它可以包含子模組 (submodule) 或是遞迴的子套件 (subpackage)。技術上而言，套件就是具有 `__path__` 屬性的一個 Python 模組。

另請參閱*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)，它指明該函式能夠接受的一個*argument*（引數），或在某些情況下指示多個引數。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為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，也可以為一些選擇性的引數指定預設值。

另請參閱術語表的*argument*（引數）條目、常見問題中的引數和參數之間的差異、`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，請參閱 `importlib.abc.PathEntryFinder`。

path entry hook（路徑項目鉤）
A callable on the sys.path_hooks list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

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換為 str 或 bytes 檔案系統路徑；而 os.fsdecode() 及 os.fsencode() 則分別可用於確保 str 及 bytes 的結果。由 **PEP 519** 引入。

PEP
Python Enhancement Proposal（Python 增強提案）。PEP 是一份設計說明文件，它能為 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，是要成為重大新功能的提案、社群中關於某個問題的意見收集，以及已納入 Python 的設計決策的記錄，這些過程的主要機制。PEP 的作者要負責在社群內建立共識並記錄反對意見。

請參閱 **PEP 1**。

portion（部分）
在單一目錄中的一組檔案（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）
請參閱*argument*（引數）。

provisional API（暫行 API）
暫行 API 是指，從標准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，只要它們被標示為暫行的，理論上並不會有重大的變更，但如果核心開發人員認為有必要，也可能會出現向後不相容的變更（甚至包括移除該介面）。這種變更並不會無端地產生——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為「最後的解決方案」——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決方案。

這個過程使得標准函式庫能隨著時間不斷進化，而避免耗費過長的時間去鎖定有問題的設計錯誤。請參閱 **PEP 411** 了解更多細節。

provisional package（暫行套件）
請參閱*provisional API*（暫行 API）。

Python 3000
Python 3.x 系列版本的暱稱（很久以前創造的，當時第 3 版的發布是在遙遠的未來。）也可以縮寫為「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，對一個可迭代物件的所有元素進行迴圈。許多其他語言並沒有這種類型的架構，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)。參照計數通常在 Python 程式碼中看不到，但它卻是*CPython* 實作的一個關鍵元素。程式設計師可以呼叫 getrefcount() 函式來回傳一個特定物件的參照計數。

regular package（正規套件）
一個傳統的*package*（套件），例如一個包含 __init__.py 檔案的目錄。

另請參閱*namespace package*（命名空間套件）。

__slots__
在 class 內部的一個宣告，它藉由預先宣告實例屬性的空間，以及消除實例 dictionary（字典），來節省記憶體。雖然該技術很普遍，但它有點難以正確地使用，最好保留給那種在一個記憶體關鍵 (memory-critical) 的應用程式中存在大量實例的罕見情況。

sequence（序列）
An *iterable* which supports efficient element access using integer indices via the __getitem__() special method and defines a __len__() method that returns the length of the sequence. Some built-in sequence types are list, str, tuple, and bytes. Note that dict also supports __getitem__() and __len__(), but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

The collections.abc.Sequence abstract base class defines a much richer interface that goes beyond just __getitem__() and __len__(), adding count(), index(), __contains__(), and __reversed__(). Types that implement this expanded interface can be registered explicitly using register(). For more documentation on sequence methods generally, see Common Sequence Operations.

set comprehension（集合綜合運算）
一種緊密的方法，用來處理一個可迭代物件中的全部或部分元素，並將處理結果以一個 set 回傳。results = {c for c in 'abracadabra' if c not in 'abc'} 會產生一個字串 set：{'r', 'd'}。請參閱 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別。

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）符號的內部，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
An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the `typing` module.

strong reference（□參照）
In Python's C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

`Py_NewRef()` 函式可用於建立一個對物件的□參照。通常，在退出□參照的作用域之前，必須在該□參照上呼叫`Py_DECREF()` 函式，以避免□漏一個參照。

另請參□*borrowed reference*（借用參照）。

text encoding（文字編碼）
Python 中的字串是一個 Unicode 碼點 (code point) 的序列（範圍在 `U+0000` -- `U+10FFFF` 之間）。若要儲存或傳送一個字串，它必須被序列化□一個位元組序列。

將一個字串序列化□位元組序列，稱□「編碼」，而從位元組序列重新建立該字串則稱□「解碼 (decoding)」。

有多種不同的文字序列化編解碼器 (codecs)，它們被統稱□「文字編碼」。

text file（文字檔案）
一個能□讀取和寫入 `str` 物件的一個*file object*（檔案物件）。通常，文字檔案實際上是存取位元組導向的資料流 (byte-oriented datastream) □會自動處理*text encoding*（文字編碼）。文字檔案的例子有：以文字模式（`'r'` 或 `'w'`）開□的檔案、`sys.stdin`、`sys.stdout` 以及 `io.StringIO` 的實例。

另請參□*binary file*（二進制檔案），它是一個能□讀取和寫入類位元組串物件 *(bytes-like object)* 的檔案物件。

triple-quoted string（三引號□字串）
由三個雙引號 (“) 或單引號 (') 的作□邊界的一個字串。雖然它們□□有提供□於單引號字串的任何額外功能，但基於許多原因，它們仍是很有用的。它們讓你可以在字串中包含未跳□ (unescaped) 的單引號和雙引號，而且它們不需使用連續字元 (continuation character) 就可以跨越多行，這使得它們在編寫□明字串時特□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􀀀。

Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

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，使得這些版本的發行成為可能。

C.2 關於存取或以其他方式使用 Python 的合約條款

Python 軟體和說明文件的授權是基於*PSF 授權合約*。

從 Python 3.8.6 開始，說明文件中的範例、程式庫和其他程式碼，是被雙重授權 (dual licensed) 於 PSF 授權合約以及*Zero-Clause BSD 授權*。

有些被納入 Python 中的軟體是基於不同的授權。這些授權將會與其授權之程式碼一起被列出。關於這些授權的不完整清單，請參閱被收錄軟體的授權與致謝。

C.2.1 用於 PYTHON 3.11.8 的 PSF 授權合約

```
1. This LICENSE AGREEMENT is between the Python Software Foundation␣
↪("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise␣
↪using Python
   3.11.8 software in source or binary form and its associated␣
↪documentation.

2. Subject to the terms and conditions of this License Agreement, PSF␣
↪hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative␣
↪works,
   distribute, and otherwise use Python 3.11.8 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's␣
↪notice of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.11.8 alone or in any derivative␣
↪version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.11.8 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made␣
↪to Python
   3.11.8.

4. PSF is making Python 3.11.8 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY␣
↪OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY␣
↪REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR␣
↪THAT THE
   USE OF PYTHON 3.11.8 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.11.8
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A␣
↪RESULT OF
```

```
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.11.8, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material␣
↪breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee. ␣
↪This License
   Agreement does not grant permission to use PSF trademarks or trade name␣
↪in a
   trademark sense to endorse or promote products or services of Licensee,␣
↪or any
   third party.

8. By copying, installing or otherwise using Python 3.11.8, 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 3.11.8 說明文件內程式碼的 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

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, https://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 非同步 socket 服務

`asynchat` 和 `asyncore` 模組包含以下聲明：

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie 管理

`http.cookies` 模組包含以下聲明：

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
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F

`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

The `test.test_epoll` module contains the following notice:

```
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換。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

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```
                              Apache License
                        Version 2.0, January 2004
                     https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "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
```

(繼續下一頁)

(繼續上一頁)

```
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   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
```

(繼續下一頁)

(繼續上一頁)

```
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

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

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

除非在建置 `_ctypes` 擴充時設定了 `--with-system-libffi`，否則該擴充會用一個內含 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，則該擴充會用一個包含 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` 使用的雜湊表 (hash table) 實作，是以 cfuhash 專案為基礎：

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
```

(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` 模組時設定為 `--with-system-libmpdec`，否則該模組會用一個內含 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 授權被發佈：

```
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

The audioop module uses the code base in g771.c file of the SoX project. https://sourceforge.net/projects/sox/files/sox/12.17.7/sox-12.17.7.tar.gz

> This source code is a product of Sun Microsystems, Inc. and is provided for unrestricted use. Users may copy or modify this source code without charge.
>
> 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.
>
> Sun source code is provided with no support and without any obligation on the part of Sun Microsystems, Inc. to assist in its use, correction, modification or enhancement.
>
> 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.
>
> In no event will Sun Microsystems, Inc. be liable for any lost revenue or profits or other special, indirect and consequential damages, even if Sun has been advised of the possibility of such damages.
>
> Sun Microsystems, Inc. 2550 Garcia Avenue Mountain View, California 94043

C.3.20 asyncio

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

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 和這份說明文件的版權：

Copyright © 2001-2023 Python Software Foundation 保留一切權利。

Copyright © 2000 BeOpen.com 保留一切權利。

Copyright © 1995-2000 Corporation for National Research Initiatives 保留一切權利。

Copyright © 1991-1995 Stichting Mathematisch Centrum 保留一切權利。

完整的授權條款資訊請參見沿革與授權。

索引

非依字母順序

A

B

C

D

E

F

Q

R

S

T