The Python/C API

發 3.10.11

Guido van Rossum
and the Python development team

5 月 24, 2023

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

CHAPTER 1

簡介

Python 的应用编程接口（API）使得 C 和 C++ 程序员可以在多个层级上访问 Python 解释器。该 API 在 C++ 中同样可用，但为简化描述，通常将其称为 Python/C API。使用 Python/C API 有两个基本的理由。第一个理由是为了特定目的而编写 *扩展模块*；它们是扩展 Python 解释器功能的 C 模块。这可能是最常见的使用场景。第二个理由是将 Python 用作更大规模应用的组件；这种技巧通常被称为在一个应用中 *embedding* Python。

编写扩展模块的过程相对来说更易于理解，可以通过"菜谱"的形式分步骤介绍。使用某些工具可在一定程度上自动化这一过程。虽然人们在其他应用中嵌入 Python 的做法早已有之，但嵌入 Python 的过程没有编写扩展模块那样方便直观。

许多 API 函数在你嵌入或是扩展 Python 这两种场景下都能发挥作用；此外，大多数嵌入 Python 的应用程序也需要提供自定义扩展，因此在尝试在实际应用中嵌入 Python 之前先熟悉编写扩展应该会是个好主意。

1.1 代码标准

如果你想要编写可包含于 CPython 的 C 代码，你 **必须**遵循在 **PEP 7** 中定义的指导原则和标准。这些指导原则适用于任何你所要扩展的 Python 版本。在编写你自己的第三方扩展模块时可以不必遵循这些规范，除非你准备在日后向 Python 贡献这些模块。

1.2 包含文件

使用 Python/C API 所需要的全部函数、类型和宏定义可通过下面这行语句包含到你的代码之中：

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

这意味着包含以下标准头文件：`<stdio.h>`，`<string.h>`，`<errno.h>`，`<limits.h>`，`<assert.h>` 和 `<stdlib.h>`（如果可用）。

備[F]: 由于 Python 可能会定义一些能在某些系统上影响标准头文件的预处理器定义，因此在包含任何标准头文件之前，你必须先包含 Python.h。

推荐总是在 Python.h 前定义 PY_SSIZE_T_CLEAN 。查看解析参数并构建值变量 来了解这个宏的更多内容。

Python.h 所定义的全部用户可见名称（由包含的标准头文件所定义的除外）都带有前缀 Py 或者 _Py。以 _Py 打头的名称是供 Python 实现内部使用的，不应被扩展编写者使用。结构成员名称没有保留前缀。

備[F]: 用户代码永远不应该定义以 Py 或 _Py 开头的名称。这会使读者感到困惑，并危及用户代码对未来 Python 版本的可移植性，这些版本可能会定义以这些前缀之一开头的其他名称。

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

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

C++ 用户应该注意，尽管 API 是完全使用 C 来定义的，但头文件正确地将入口点声明为 extern "C"，因此 API 在 C++ 中使用此 API 不必再做任何特殊处理。

1.3 有用的宏

Python 头文件中定义了一些有用的宏。许多是在靠近它们被使用的地方定义的（例如*Py_RETURN_NONE*）。其他更为通用的则定义在这里。这里所显示的并不是一个完整的列表。

Py_UNREACHABLE()

这个可以在你有一个设计上无法到达的代码路径时使用。例如，当一个 switch 语句中所有可能的值都已被 case 子句覆盖了，就可将其用在 default: 子句中。当你非常想在某个位置放一个 assert(0) 或 abort() 调用时也可以用这个。

在 release 模式下，该宏帮助编译器优化代码，并避免发出不可到达代码的警告。例如，在 GCC 的 release 模式下，该宏使用 __builtin_unreachable() 实现。

Py_UNREACHABLE() 的一个用法是调用一个不会返回，但却没有声明 _Py_NO_RETURN 的函数之后。

如果一个代码路径不太可能是正常代码，但在特殊情况下可以到达，就不能使用该宏。例如，在低内存条件下，或者一个系统调用返回超出预期范围值，诸如此类，最好将错误报告给调用者。如果无法将错误报告给调用者，可以使用*Py_FatalError()* 。

3.7 版新加入.

Py_ABS(*x*)

回傳 x 的[F]對值。

3.3 版新加入.

Py_MIN(*x*, *y*)

返回 x 和 y 当中的最小值。

3.3 版新加入.

Py_MAX (*x*, *y*)
返回 x 和 y 当中的最大值。

3.3 版新加入.

Py_STRINGIFY (*x*)
将 x 转换为 C 字符串。例如 Py_STRINGIFY(123) 返回 "123"。

3.4 版新加入.

Py_MEMBER_SIZE (*type*, *member*)
返回结构 (type) member 的大小，以字节表示。

3.6 版新加入.

Py_CHARMASK (*c*)
参数必须为 [-128, 127] 或 [0, 255] 范围内的字符或整数类型。这个宏将 c 强制转换为 unsigned char 返回。

Py_GETENV (*s*)
与 getenv(s) 类似，但是如果命令行上传递了 -E ，则返回 NULL （即如果设置了 Py_IgnoreEnvironmentFlag ）。

Py_UNUSED (*arg*)
用于函数定义中未使用的参数，从而消除编译器警告。例如：int func(int a, int Py_UNUSED(b)) { return a; } 。

3.4 版新加入.

Py_DEPRECATED (*version*)
弃用声明。该宏必须放置在符号名称前。

範例：

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

3.8 版更變: 添加了 MSVC 支持。

PyDoc_STRVAR (*name*, *str*)
创建一个可以在文档字符串中使用的，名字为 name 的变量。如果不和文档字符串一起构建 Python，该值将为空。

如 **PEP 7** 所述，使用*PyDoc_STRVAR* 作为文档字符串，以支持不和文档字符串一起构建 Python 的情况。

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*)
为给定的字符串输入创建一个文档字符串，或者当文档字符串被禁用时，创建一个空字符串。

如 **PEP 7** 所述，使用*PyDoc_STR* 指定文档字符串，以支持不和文档字符串一起构建 Python 的情况。

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 对象、类型和引用计数

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

所有 Python 对象（甚至 Python 整数）都有一个 *type* 和一个 *reference count*。对象的类型确定它是什么类型的对象（例如整数、列表或用户定义函数；还有更多，如 types 中所述）。对于每个众所周知的类型，都有一个宏来检查对象是否属于该类型；例如，当（且仅当）*a* 所指的对象是 Python 列表时 `PyList_Check(a)` 为真。

1.4.1 引用计数

引用计数非常重要，因为现代计算机内存（通常十分）有限；它计算有多少不同的地方引用同一个对象。这样的地方可以是某个对象，或者是某个全局（或静态）C 变量，亦或是某个 C 函数的局部变量。当一个对象的引用计数变为 0，释放该对象。如果这个已释放的对象包含其它对象的引用计数，则递减这些对象的引用计数。如果这些对象的引用计数减少为零，则可以依次释放这些对象，依此类推。（这里有一个很明显的问题——对象之间相互引用；目前，解决方案是“不要那样做”。）

总是显式操作引用计数。通常的方法是使用宏*Py_INCREF()* 来增加一个对象的引用计数，使用宏*Py_DECREF()* 来减少一个对象的引用计数。宏*Py_DECREF()* 必须检查引用计数是否为零，然后调用对象的释放器，因此它比 incref 宏复杂得多。释放器是一个包含在对象类型结构中的函数指针。如果对象是复合对象类型（例如列表），则类型特定的释放器负责递减包含在对象中的其他对象的引用计数，并执行所需的终结。引用计数不会溢出，至少用与虚拟内存中不同内存位置一样多的位用于保存引用计数（即 `sizeof(Py_ssize_t) >= sizeof(void*)`）。因此，引用计数递增是一个简单的操作。

没有必要为每个包含指向对象的指针的局部变量增加对象的引用计数。理论上，当变量指向对象时，对象的引用计数增加 1 ，当变量超出范围时，对象的引用计数减少 1 。但是，这两者相互抵消，所以最后引用计数没有改变。使用引用计数的唯一真正原因是只要我们的变量指向它，就可以防止对象被释放。如果知道至少有一个对该对象的其他引用存活时间至少和我们的变量一样长，则没必要临时增加引用计数。一个典型的情形是，对象作为参数从 Python 中传递给被调用的扩展模块中的 C 函数时，调用机制会保证在调用期间持有对所有参数的引用。

但是，有一个常见的陷阱是从列表中提取一个对象，并将其持有一段时间，而不增加其引用计数。某些操作可能会从列表中删除某个对象，减少其引用计数，并有可能重新分配这个对象。真正的危险是，这个看似无害的操作可能会调用任意 Python 代码——也许有一个代码路径允许控制流从*Py_DECREF()* 回到用户，因此在复合对象上的操作都存在潜在的风险。

一个安全的方式是始终使用泛型操作（名称以 `PyObject_`，`PyNumber_`，`PySequence_` 或 `PyMapping_` 开头的函数）。这些操作总是增加它们返回的对象的引用计数。这让调用者有责任在获得结果后调用*Py_DECREF()* 。习惯这种方式很简单。

引用计数细节

Python/C API 中函数引用计数行为最好是通过 *引用所有权*来解释。所有权是关联到引用，而不是对象（对象没有所有权：它们总是会被共享）。“获得引用所有权”意味着当不再需要该引用时必须在其上调用 Py_DECREF。所有权也可以被转移，这意味着接受该引用所有权的代码当不再需要该引用时必须通过调用*Py_DECREF()* 或*Py_XDECREF()* 来最终撤销引用 --- 或是将这个责任转移出去（通常是转给其调用方）。当一个函数将引用所有权转给其调用方时，则称调用方收到了一个 *新的*引用。当所有权未被转移时，则称调用方 *借入*该引用。对于*borrowed reference* 来说不需任何额外操作。

相反地，当调用方函数传入一个对象的引用时，存在两种可能：该函数 *窃取*了一个对象的引用，或是没有窃取。*窃取引用*意味着当你向一个函数传入引用时，该函数会假定它拥有该引用，而你将不再对它负有责任。

很少有函数会窃取引用；两个重要的例外是*PyList_SetItem()* 和*PyTuple_SetItem()*，它们会窃取对条目的引用（但不是条目所在的元组或列表！）。这些函数被设计为会窃取引用是因为在使用新创建的对象来填充元组或列表时有一个通常的惯例；例如，创建元组 `(1, 2, "three")` 的代码看起来可以是这样的（暂时不要管错误处理；下面会显示更好的代码编写方式）：

```
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"));
```

在这里，*PyLong_FromLong()* 返回了一个新的引用并且它立即被*PyTuple_SetItem()* 所窃取。当你想要继续使用一个对象而对它的引用将被窃取时，请在调用窃取引用的函数之前使用*Py_INCREF()* 来抓取另一个引用。

顺便提一下，*PyTuple_SetItem()* 是设置元组条目的 *唯一*方式；*PySequence_SetItem()* 和*PyObject_SetItem()* 会拒绝这样做因为元组是不可变数据类型。你应当只对你自己创建的元组使用*PyTuple_SetItem()*。

等价于填充一个列表的代码可以使用*PyList_New()* 和*PyList_SetItem()* 来编写。

然而，在实践中，你很少会使用这些创建和填充元组或列表的方式。有一个通用的函数*Py_BuildValue()* 可以根据 C 值来创建大多数常用对象，由一个 *格式字符串*来指明。例如，上面的两个代码块可以用下面的代码来代替（还会负责错误检测）：

```
PyObject *tuple, *list;

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

使用*PyObject_SetItem()* 等来处理那些你只是借入引用的条目是更为常见的，例如传给你正在编写的函数的参数。在这种情况下，他们对于引用计数的行为会更为理智，因为你不需要递增引用计数以便你可以将引用计数转出去（“让它被窃取”）。例如，这个函数将一个列表（实例上是任何可变序列）中的所有项设置为一个给定的条目:

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

    n = PyObject_Length(target);
    if (n < 0)
        return -1;
    for (i = 0; i < n; i++) {
        PyObject *index = PyLong_FromSsize_t(i);
```

(下页继续)

(繼續上一頁)

```
        if (!index)
            return -1;
        if (PyObject_SetItem(target, index, item) < 0) {
            Py_DECREF(index);
            return -1;
        }
        Py_DECREF(index);
    }
    return 0;
}
```

对于函数返回值的情况略有不同。虽然向大多数函数传递一个引用不会改变你对该引用的所有权责任，但许多返回一个引用的函数会给你该引用的所有权。原因很简单：在许多情况下，返回的对象是临时创建的，而你得到的引用是对该对象的唯一引用。因此，返回对象引用的通用函数，如*PyObject_GetItem()*和*PySequence_GetItem()*，将总是返回一个新的引用（调用方将成为该引用的所有者）。

一个需要了解的重点在于你是否拥有一个由函数返回的引用只取决于你所调用的函数 --- 附带物 (作为参数传给函数的对象的类型) 不会带来额外影响！因此，如果你使用*PyList_GetItem()* 从一个列表提取条目，你并不会拥有其引用 --- 但是如果你使用*PySequence_GetItem()* (它恰好接受完全相同的参数) 从同一个列表获取同样的条目，你就会拥有一个对所返回对象的引用。

下面是说明你要如何编写一个函数来计算一个整数列表中条目的示例；一个是使用*PyList_GetItem()*，而另一个是使用*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++) {
```

(下页继续)

(繼續上一頁)

```
        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 类型

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

type Py_ssize_t

Part of the Stable ABI. 一个使得 `sizeof(Py_ssize_t) == sizeof(size_t)` 的有符号整数类型。C99 没有直接定义这样的东西（size_t 是一个无符号整数类型）。请参阅 **PEP 353** 了解详情。`PY_SSIZE_T_MAX` 是*Py_ssize_t* 类型的最大正数值。

1.5 例外

Python 程序员只需要处理特定需要处理的错误异常；未处理的异常会自动传递给调用者，然后传递给调用者的调用者，依此类推，直到他们到达顶级解释器，在那里将它们报告给用户并伴随堆栈回溯。

然而，对于 C 程序员来说，错误检查必须总是显式进行的。Python/C API 中的所有函数都可以引发异常，除非在函数的文档中另外显式声明。一般来说，当一个函数遇到错误时，它会设置一个异常，丢弃它所拥有的任何对象引用，并返回一个错误标示。如果没有说明例外的文档，这个标示将为 `NULL` 或 `-1`，具体取决于函数的返回类型。有少量函数会返回一个布尔真/假结果值，其中假值表示错误。有极少的函数没有显式的错误标示或是具有不明确的返回值，并需要用*PyErr_Occurred()* 来进行显式的检测。这些例外总是会被明确地记入文档中。

异常状态是在各个线程的存储中维护的（这相当于在一个无线程的应用中使用全局存储）。一个线程可以处在两种状态之一：异常已经发生，或者没有发生。函数*PyErr_Occurred()* 可以被用来检查此状态：当异常发生时它将返回一个借入的异常类型对象的引用，在其他情况下则返回 `NULL`。有多个函数可以设置异常状态：*PyErr_SetString()* 是最常见的（尽管不是最通用的）设置异常状态的函数，而*PyErr_Clear()* 可以清除异常状态。

完整的异常状态由三个对象组成 (它为都可以为 `NULL`): 异常类型、相应的异常值，以及回溯信息。这些对象的含义与 Python 中 `sys.exc_info()` 的结果相同；然而，它们并不是一样的：Python 对象代表由 Python `try` ... `except` 语句所处理的最后一个异常，而 C 层级的异常状态只在异常被传入到 C 函数或在它们之间传递时存在直至其到达 Python 字节码解释器的主事件循环，该事件循环会负责将其转移至 `sys.exc_info()` 等处。

请注意自 Python 1.5 开始，从 Python 代码访问异常状态的首选的、线程安全的方式是调用函数 sys.exc_info()，它将返回 Python 代码的分线程异常状态。此外，这两种访问异常状态的方式的语义都发生了变化因而捕获到异常的函数将保存并恢复其线程的异常状态以保留其调用方的异常状态。这将防止异常处理代码中由一个看起来很无辜的函数覆盖了正在处理的异常所造成的常见错误；它还减少了在回溯由栈帧所引用的对象的往往不被需要的生命其延长。

作为一般的原则，一个调用另一个函数来执行某些任务的函数应当检查被调用的函数是否引发了异常，并在引发异常时将异常状态传递给其调用方。它应当丢弃它所拥有的任何对象引用，并返回一个错误标示，但它不应设置另一个异常 --- 那会覆盖刚引发的异常，并丢失有关错误确切原因的重要信息。

一个检测异常并传递它们的简单例子在上面的 sum_sequence() 示例中进行了演示。这个例子恰好在检测到错误时不需要清理所拥有的任何引用。下面的示例函数演示了一些错误清理操作。首先，为了向你提示 Python 的优势，我们展示了等效的 Python 代码：

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

下面是对应的闪耀荣光的 C 代码：

```
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 */
```

(下页继续)

(繼續上一頁)

```
    /* 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 */
}
```

这个例子代表了 C 语言中 goto 语句一种受到认可的用法！它说明了如何使用*PyErr_ExceptionMatches()* 和*PyErr_Clear()* 来处理特定的异常，以及如何使用*Py_XDECREF()* 来处理可能为 NULL 的自有引用（注意名称中的 'X'；*Py_DECREF()* 在遇到 NULL 引用时将会崩溃）。重要的一点在于用来保存自有引用的变量要被初始化为 NULL 才能发挥作用；类似地，建议的返回值也要被初始化为 -1 (失败) 并且只有在最终执行的调用成功后才会被设置为成功。

1.6 嵌入式 Python

只有 Python 解释器的嵌入方（相对于扩展编写者而言）才需要担心的一项重要任务是它的初始化，可能还有它的最终化。解释器的大多数功能只有在解释器被初始化之后才能被使用。

基本的初始化函数是*Py_Initialize()*。此函数将初始化已加载模块表，并创建基本模块 builtins, __main__ 和 sys。它还将初始化模块搜索路径 (sys.path)。

Py_Initialize() 不会设置" 脚本参数列表" (sys.argv)。如果随后将要执行的 Python 代码需要此变量，则必须在调用*Py_Initialize()* 之后通过调用 PySys_SetArgvEx(argc, argv, updatepath) 来显式地设置它。

在大多数系统上（特别是 Unix 和 Windows，虽然在细节上有所不同），*Py_Initialize()* 将根据对标准 Python 解释器可执行文件的位置的最佳猜测来计算模块搜索路径，并设定 Python 库可在相对于 Python 解释器可执行文件的固定位置上找到。特别地，它将相对于在 shell 命令搜索路径 (环境变量 PATH) 上找到的名为 python 的可执行文件所在父目录中查找名为 lib/python*X.Y* 的目录。

举例来说，如果 Python 可执行文件位于 /usr/local/bin/python，它将假定库位于 /usr/local/lib/python*X.Y*。(实际上，这个特定路径还将成为 "回退" 位置，会在当无法在 PATH 中找到名为 python 的可执行文件时被使用。) 用户可以通过设置环境变量 PYTHONHOME，或通过设置 PYTHONPATH 在标准路径之前插入额外的目录来覆盖此行为。

嵌入的应用程序可以通过在调用*Py_Initialize()* 之前调用 Py_SetProgramName(file) 来改变搜索次序。请注意 PYTHONHOME 仍然会覆盖此设置并且 PYTHONPATH 仍然会被插入到标准路径之前。需要完全控制权的应用程序必须提供它自己的*Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()* 和*Py_GetProgramFullPath()* 实现（这些函数均在 Modules/getpath.c 中定义）。

有时，还需要对 Python 进行 "反初始化"。例如，应用程序可能想要重新启动 (再次调用*Py_Initialize()*) 或者应用程序对 Python 的使用已经完成并想要释放 Python 所分配的内存。这可以通过调用*Py_FinalizeEx()* 来实现。如果当前 Python 处于已初始化状态则*Py_IsInitialized()* 函数将返回真值。有关这些函数的更多信息将在之后的章节中给出。请注意*Py_FinalizeEx()* 不会释放所有由 Python 解释器所分配的内存，例如由扩展模块所分配的内存目前是不会被释放的。

1.7 调试构建

Python 可以附带某些宏来编译以启用对解释器和扩展模块的额外检查。这些检查会给运行时增加大量额外开销因此它们默认未被启用。

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

附带定义 `Py_DEBUG` 宏来编译解释器将产生通常所称的 Python 调试编译版。`Py_DEBUG` 在 Unix 编译中启用是通过添加 `--with-pydebug` 到 `./configure` 命令来实现的。它也可通过提供非 Python 专属的 `_DEBUG` 宏来启用。当 `Py_DEBUG` 在 Unix 编译中启用时，编译器优化将被禁用。

除了下文描述的引用计数调试，还会执行额外检查，请参阅 Python Debug Build。

定义 `Py_TRACE_REFS` 将启用引用追踪 (参见 `configure --with-trace-refs` 选项)。当定义了此宏时，将通过在每个 *PyObject* 上添加两个额外字段来维护一个活动对象的循环双链列表。总的分配量也会被追踪。在退出时，所有现存的引用将被打印出来。(在交互模式下这将在解释器运行每条语句之后发生)。

有关更多详细信息，请参阅 Python 源代码中的 `Misc/SpecialBuilds.txt` 。

CHAPTER 2

C API 的稳定性

Python 的 C 语言 API 包含于向下兼容政策 **PEP 387** 中。C API 会跟随小版本的发布而发生变化（比如 3.9 到 3.10 的时候），不过大多数变化都是源代码级兼容的，通常只会增加新的 API。已有 API 的修改或删除，只有在废止期过后或修复严重问题时才会进行。

CPython 的应用二进制接口（ABI）可以跨小版本实现前后兼容（只要以同样方式编译；参见下面的平台的考虑）。因此，用 Python 3.10.0 编译的代码可以在 3.10.8 上运行，反之亦然，但针对 3.9.x 和 3.10.x 则需分别进行编译。

带下划线前缀的是私有 API，如 `_Py_InternalState`，即便是补丁发布版本中也可能不加通知地进行改动。

2.1 应用程序二进制接口的稳定版

Python 3.2 引入了 *受限 API*，Python 的 C API 的一个子集。只使用受限 API 的扩展可以被一次性编译而适用于多个 Python 版本。受限 API 的内容如下所示。

为了实现这一点，Python 提供了一个 *稳定 ABI*: 一个将在各 Python 3.x 版本中保持兼容性的符号集合。稳定 ABI 包含了在受限 API 中暴露的符号，但还包含其他符号–例如，支持旧版受限 API 所需的函数。

（简单起见，本文档只讨论了 *扩展*，但受限 API 和稳定 ABI 对于 API 的所有用法都同样适用–例如，嵌入 Python 等。）

`Py_LIMITED_API`

请在包括 `Python.h` 之前定义这个宏以选择只使用受限 API，并选择受限 API 的版本。

将 `Py_LIMITED_API` 定义为对应你的扩展所支持的最低 Python 版本的 *`PY_VERSION_HEX`* 的值。扩展无需重编译即可适用于从指定版本开始的所有 Python 3 发布版，并可使用到该版本为止所引入的受限 API。

不直接使用 `PY_VERSION_HEX` 宏，而是硬编码一个最小的次要版本（例如 `0x030A0000` 表示 Python 3.10）以便在使用未来的 Python 版本进行编译时保持稳定。

你还可以将 `Py_LIMITED_API` 定义为 `3`。其效果与 `0x03020000` 相同（即 Python 3.2，引入受限 API 的版本）。

在 Windows 上，使用稳定 ABI 的扩展应当被链接到 python3.dll 而不是版本专属的库如 python39.dll。

在某些平台上，Python 将查找并载入名称中带有 abi3 标签的共享库文件 (例如 mymodule.abi3.so)。它不会检查这样的扩展是否兼容稳定 ABI。使用方 (或其打包工具) 需要确保这一些，例如，基于 3.10+ 受限 API 编译的扩展不可被安装于更低版本的 Python 中。

稳定 ABI 中的所有函数都会作为 Python 的共享库中的函数存在，而不仅是作为宏。这使得它们可以在不使用 C 预处理器的语言中使用。

2.1.1 受限 API 的作用域和性能

受限 API 的目标是允许使用在完整 C API 中可用的任何东西，但可能会有性能上的损失。

例如，虽然*PyList_GetItem()*是可用的，但其 "不安全的" 宏版本*PyList_GET_ITEM()*则是不可用的。这个宏的运行速度更快因为它可以利用版本专属的列表对象实现细节。

在未定义 Py_LIMITED_API 的情况下，某些 C API 函数将由宏来执行内联或替换。定义 Py_LIMITED_API 会禁用这样的内联，允许提升 Python 的数据结构稳定性，但有可能降低性能。

通过省略 Py_LIMITED_API 定义，可以使基于版本专属的 ABI 来编译受限 API 扩展成为可能。这能提升其在相应 Python 版本上的性能，但也将限制其兼容性。基于 Py_LIMITED_API 进行编译将产生一个可在版本专属扩展不可用的场合分发的扩展–例如，针对即将发布的 Python 版本的预发布包。

2.1.2 受限 API 警示

请注意基于 Py_LIMITED_API 进行编译 不能完全保证代码兼容受限 API 或稳定 ABI。Py_LIMITED_API 仅涵盖了定义，但是一个 API 还包括其他因素，例如预期的语义等。

Py_LIMITED_API 不能处理的一个问题是附带在较低 Python 版本中无效的参数调用某个函数。例如，考虑一个接受 NULL 作为参数的函数。在 Python 3.9 中，NULL 现在会选择一个默认行为，但在 Python 3.8 中，该参数将被直接使用，导致一个 NULL 引用被崩溃。类似的参数也适用于结构体的字段。

另一个问题是当定义了 Py_LIMITED_API 时某些结构体字段目前不会被隐藏，即使它们是受限 API 的一部分。

出于这些原因，我们建议用要支持的 *所有* Python 小版本号来测试一个扩展，并最好是用其中 *最低的*版本来编译它。

我们还建议查看所使用 API 的全部文档以检查其是否显式指明为受限 API 的一部分。即使定义了 Py_LIMITED_API，少数私有声明还是会出于技术原因（或者甚至是作为程序缺陷在无意中）被暴露出来。

还要注意受限 API 并不必然是稳定的：在 Python 3.8 上用 Py_LIMITED_API 编译扩展意味着该扩展能在 Python 3.12 上运行，但它将不一定能用 Python 3.12 *编译*。特别地，在稳定 ABI 保持稳定的情况下，部分受限 API 可能会被弃用并被移除。

2.2 平台的考虑

ABI 的稳定性不仅取决于 Python，还取决于所使用的编译器、低层库和编译器选项。对于稳定 ABI 的目标来说，这些细节定义了一个 "平台"。它们通常会取决于 OS 类型和处理器架构。

确保在特定平台上的所有 Python 版本都以不破坏稳定 ABI 的方式构建是每个特定 Python 分发方的责任。来自 python.org 以及许多第三方分发商的 Windows 和 macOS 发布版都必于这种情况。

2.3 受限 API 的内容

目前，受限 API 包括下面这些项:

- *PyAIter_Check()*
- *PyArg_Parse()*
- *PyArg_ParseTuple()*
- *PyArg_ParseTupleAndKeywords()*
- *PyArg_UnpackTuple()*
- *PyArg_VaParse()*
- *PyArg_VaParseTupleAndKeywords()*
- *PyArg_ValidateKeywordArguments()*
- PyBaseObject_Type
- *PyBool_FromLong()*
- PyBool_Type
- 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_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_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_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_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_CheckReadBuffer()*
- *PyObject_ClearWeakRefs()*
- *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_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()*
- 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_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_XNewRef()*
- 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

极高层级 API

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

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

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

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

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

Part of the Stable ABI *since version 3.8.* 类似于 `Py_Main()` 但 *argv* 是一个包含字节串的数组。

3.8 版新加入.

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

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

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

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

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

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

int **PyRun_AnyFileExFlags** (FILE **fp*, **const** char **filename*, int *closeit*, *PyCompilerFlags* **flags*)
如果 *fp* 指向一个关联到交互设备（控制台或终端输入或 Unix 伪终端）的文件，则返回`PyRun_InteractiveLoop()` 的值，否则返回`PyRun_SimpleFile()` 的结果。*filename* 会使用文件系统的编码格式 (`sys.getfilesystemencoding()`) 来解码。如果 *filename* 为 `NULL`，此函数会使用 `"???"` 作为文件名。如果 *closeit* 为真值，文件会在 `PyRun_SimpleFileExFlags()` 返回之前被关闭。

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

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

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

int **PyRun_SimpleFile** (FILE **fp*, **const** char **filename*)
这是针对下面`PyRun_SimpleFileExFlags()` 的简化版接口，将 *closeit* 设为 `0` 而将 *flags* 设为 `NULL`。

int **PyRun_SimpleFileEx** (FILE **fp*, **const** char **filename*, int *closeit*)
这是针对下面`PyRun_SimpleFileExFlags()` 的简化版接口，将 *flags* 设为 `NULL`。

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

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

int **PyRun_InteractiveOne** (FILE **fp*, **const** char **filename*)
这是针对下面`PyRun_InteractiveOneFlags()` 的简化版接口，将 *flags* 设为 `NULL`。

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

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

int **PyRun_InteractiveLoop** (FILE **fp*, **const** char **filename*)
这是针对下面`PyRun_InteractiveLoopFlags()` 的简化版接口，将 *flags* 设为 `NULL`。

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

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

char *(***PyOS_ReadlineFunctionPointer**) (FILE*, FILE*, **const** char*)
可以被设为指向一个原型为 `char *func(FILE *stdin, FILE *stdout, char *prompt)` 的

函数，重载被用来读取解释器提示符的一行输入的默认函数。该函数被预期为如果字符串 *prompt* 不为 `NULL` 就输出它，然后从所提供的标准输入文件读取一行输入，并返回结果字符串。例如，`readline` 模块将这个钩子设置为提供行编辑和 tab 键补全等功能。

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

3.4 版更變: 结果必须由 `PyMem_RawMalloc()` 或 `PyMem_RawRealloc()` 分配，而不是由 `PyMem_Malloc()` 或 `PyMem_Realloc()` 分配。

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

Return value: New reference. 这是针对下面 `PyRun_StringFlags()` 的简化版接口，将 *flags* 设为 `NULL`。

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

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

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

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

Return value: New reference. 这是针对下面 `PyRun_FileExFlags()` 的简化版接口，将 *closeit* 设为 `0` 并将 *flags* 设为 `NULL`。

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

Return value: New reference. 这是针对下面 `PyRun_FileExFlags()` 的简化版接口，将 *flags* 设为 `NULL`。

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

Return value: New reference. 这是针对下面 `PyRun_FileExFlags()` 的简化版接口，将 *closeit* 设为 `0`。

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

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

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

Return value: New reference. Part of the Stable ABI. 这是针对下面 `Py_CompileStringFlags()` 的简化版接口，将 *flags* 设为 `NULL`。

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

Return value: New reference. 这是针对下面 `Py_CompileStringExFlags()` 的简化版接口，将 *optimize* 设为 `-1`。

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

Return value: New reference. 解析并编译 *str* 中的 Python 源代码，返回结果代码对象。开始形符由 *start* 给出；这可被用来限制可被编译的代码并且应为 `Py_eval_input`, `Py_file_input` 或 `Py_single_input`。由 *filename* 指定的文件名会被用来构造代码对象并可能出现在回溯信息或 `SyntaxError` 异常消息中。如果代码无法被解析或编译则此函数将返回 `NULL`。

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

3.4 版新加入.

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

Return value: New reference. 与`Py_CompileStringObject()` 类似，但 *filename* 是以*filesystem encoding and error handler* 解码出的字节串。

3.2 版新加入.

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

Return value: New reference. Part of the Stable ABI. 这是针对`PyEval_EvalCodeEx()` 的简化版接口，只附带代码对象，以及全局和局部变量。其他参数均设为 `NULL`。

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

Return value: New reference. Part of the Stable ABI. 对一个预编译的代码对象求值，为其求值给出特定的环境。此环境由全局变量的字典，局部变量映射对象，参数、关键字和默认值的数组，仅限关键字 参数的默认值的字典和单元的封闭元组构成。

type PyFrameObject

Part of the Limited API *(as an opaque struct)*. 用于描述帧对象的 C 对象结构体。此类型的字段可能在任何时候被改变。

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

Return value: New reference. Part of the Stable ABI. 对一个执行帧求值。这是针对`PyEval_EvalFrameEx()` 的简化版接口，用于保持向下兼容性。

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

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

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

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

此函数会修改当前求值帧的旗标，并在成功时返回真值，失败时返回假值。

int **Py_eval_input**

Python 语法中用于孤立表达式的起始符号；配合`Py_CompileString()` 使用。

int **Py_file_input**

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

int **Py_single_input**

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

struct PyCompilerFlags

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

当 `PyCompilerFlags *flags` 为 `NULL` 时，`cf_flags` 将被当作等于 `0` 来处理，而任何 `from __future__ import` 所导致的修改都会被丢弃。

int **cf_flags**

编译器旗标。

int **cf_feature_version**

cf_feature_version 是 Python 的小版本号。它应当被初始化为 `PY_MINOR_VERSION`。

此字段默认会被忽略，当且仅当在 *cf_flags* 中设置了 `PyCF_ONLY_AST` 旗标它才会被使用。

3.8 版更變: 新增 *cf_feature_version* 欄位。

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

CHAPTER 4

参照計數

本节介绍的宏被用于管理 Python 对象的引用计数。

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

增加对象 *o* 的引用计数。

此函数通常被用来将*borrowed reference* 原地转换为*strong reference*。*Py_NewRef()* 函数可被用来创建新的*strong reference*。

此对象必须不为 NULL；如果你不能确定它不为 NULL，请使用*Py_XINCREF()*。

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

增加对象 *o* 的引用计数。对象可以为 NULL，在此情况下该宏不产生任何效果。

另請見*Py_XNewRef()*。

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

Part of the Stable ABI *since version 3.10.* 新建指向一个对象的*strong reference*: 增加对象 *o* 的引用计数并返回对象 *o*。

当不再需要这个*strong reference* 时，应当在对象上调用*Py_DECREF()* 来有减少该对象的引用计数。

对象 *o* 必须不为 NULL；如果 *o* 可以为 NULL 则应改用*Py_XNewRef()*。

舉例來說：

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

可以寫成：

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

另請参閱*Py_INCREF()*。

3.10 版新加入.

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

Part of the Stable ABI *since version 3.10.* 类似于*Py_NewRef()*，但对象 *o* 可以为 NULL。

如果对象 *o* 为 NULL，该函数也 · 将返回 NULL。

3.10 版新加入.

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

减少对象 *o* 的引用计数。

如果引用计数达到零，则会发起调用对象类型的撤销分配函数 (该函数必须不为 NULL)。

此函数通常被用于在退出作用域之前删除一个*strong reference*。

此对象必须不为 NULL；如果你不能确定它不为 NULL，请使用*Py_XDECREF()*。

> **警告：** 释放函数可导致任意 Python 代码被发起调用（例如当一个带有 `__del__()` 方法的类实例被释放时就是如此）。虽然此类代码中的异常不会被传播，但被执行的代码能够自由访问所有 Python 全局变量。这意味着任何可通过全局变量获取的对象在*Py_DECREF()* 被发起调用之前都应当处于完好状态。例如，从一个列表中删除对象的代码应当将被删除对象的引用拷贝到一个临时变量中，更新列表数据结构，然后再为临时变量调用*Py_DECREF()*。

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

减少对象 *o* 的引用计数。对象可以为 NULL，在此情况下该宏不产生任何效果；在其他情况下其效果与*Py_DECREF()* 相同，并会应用同样的警告。

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

减少对象 *o* 的引用计数。对象可以为 NULL，在此情况下该宏不产生任何效果；在其他情况下其效果与*Py_DECREF()* 相同，区别在于其参数也会被设为 NULL。针对*Py_DECREF()* 的警告不适用于所传递的对象，因为该宏会细心地使用一个临时变量并在减少其引用计数之前将参数设为 NULL。

每当要减少在垃圾回收期间可能会被遍历的对象的引用计数时，使用该宏是一个好主意。

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

Part of the Stable ABI. 增加对象 *o* 的引用计数。*Py_XINCREF()* 的函数版本。它可被用于 Python 的运行时动态嵌入。

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

Part of the Stable ABI. 增加对象 *o* 的引用计数。*Py_XDECREF()* 的函数版本。它可被用于 Python 的运行时动态嵌入。

以下函数或宏仅可在解释器核心内部使用：`_Py_Dealloc()`，`_Py_ForgetReference()`，`_Py_NewReference()` 以及全局变量 `_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).

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

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

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

5.1 打印和清理

void **PyErr_Clear**()

Part of the Stable ABI. 清除错误指示器。如果没有设置错误指示器，则不会有作用。

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

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

只有在错误指示器被设置时才需要调用这个函数，否则这会导致错误！

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

void **PyErr_Print**()
Part of the Stable ABI. `PyErr_PrintEx(1)` 的􏰀名。

void **PyErr_WriteUnraisable**(*PyObject* *obj)
Part of the Stable ABI. 使用当前异常和 *obj* 参数调用 `sys.unraisablehook()`。

当设置了异常，但解释器不可能实际地触发异常时，这个实用函数向 `sys.stderr` 打印一个警告信息。例如，当 `__del__()` 方法中发生异常时使用这个函数。

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

调用此函数时必须设置一个异常。

5.2 抛出异常

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

void **PyErr_SetString**(*PyObject* *type, **const** char *message)
Part of the Stable ABI. 这是设置错误标记最常用的方式。第一个参数指定异常类型；它通常为某个标准异常，例如 `PyExc_RuntimeError`。你不需要增加它的引用计数。第二个参数是错误消息；它是用 `'utf-8'` 解码的。

void **PyErr_SetObject**(*PyObject* *type, *PyObject* *value)
Part of the Stable ABI. 此函数类似于 `PyErr_SetString()`，但是允许你为异常的“值”指定任意一个 Python 对象。

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

PyObject ***PyErr_FormatV**(*PyObject* *exception, **const** char *format, va_list vargs)
Return value: Always NULL. Part of the Stable ABI *since version 3.5.* 和 `PyErr_Format()` 相同，但它接受一个 `va_list` 类型的参数而不是可变数量的参数集。

3.5 版新加入.

void **PyErr_SetNone**(*PyObject* *type)
Part of the Stable ABI. 这是 `PyErr_SetObject(type, Py_None)` 的简写。

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

PyObject ***PyErr_NoMemory**()
Return value: Always NULL. Part of the Stable ABI. 这是 `PyErr_SetNone(PyExc_MemoryError)` 的简写；它返回 `NULL`，以便当内存耗尽时，对象分配函数可以写 `return PyErr_NoMemory();`。

PyObject ***PyErr_SetFromErrno**(*PyObject* *type)
Return value: Always NULL. Part of the Stable ABI. 这是个便捷函数，当 C 库函数返回错误并设置 `errno` 时，这个函数会触发异常。它构造一个元组对象，其第一项是整数值 `errno`，第二项是相应的错误消息（从 `strerror()` 获取），然后调用 `PyErr_SetObject(type, object)`。在 Unix 上，当 `errno` 值是 `EINTR`，即中断的系统调用时，这个函数会调用 `PyErr_CheckSignals()`，如果设置了错误指

示器，则将其设置为该值。该函数永远返回 `NULL`，因此当系统调用返回错误时，围绕系统调用的包装函数可以写成 `return PyErr_SetFromErrno(type);`。

PyObject ***PyErr_SetFromErrnoWithFilenameObject**(*PyObject* *type, *PyObject* *filenameObject)
Return value: Always NULL. Part of the Stable ABI. 类似于 `PyErr_SetFromErrno()`，附加的行为是如果 *filenameObject* 不为 `NULL`，它将作为第三个参数传递给 *type* 的构造函数。举个例子，在 `OSError` 异常中，*filenameObject* 将用来定义异常实例的 `filename` 属性。

PyObject ***PyErr_SetFromErrnoWithFilenameObjects**(*PyObject* *type, *PyObject* *filenameObject, *PyObject* *filenameObject2)
Return value: Always NULL. Part of the Stable ABI *since version 3.7.* 类似于 `PyErr_SetFromErrnoWithFilenameObject()`，但接受第二个 filename 对象，用于当一个接受两个 filename 的函数失败时触发错误。

3.4 版新加入.

PyObject ***PyErr_SetFromErrnoWithFilename**(*PyObject* *type, **const** char *filename)
Return value: Always NULL. Part of the Stable ABI. 类似于 `PyErr_SetFromErrnoWithFilenameObject()`，但文件名以 C 字符串形式给出。*filename* 是用 *filesystem encoding and error handler* 解码的。

PyObject ***PyErr_SetFromWindowsErr**(int *ierr*)
Return value: Always NULL. Part of the Stable ABI *on Windows since version 3.7.* This is a convenience function to raise `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`.

适用：Windows。

PyObject ***PyErr_SetExcFromWindowsErr**(*PyObject* *type, int *ierr*)
Return value: Always NULL. Part of the Stable ABI *on Windows since version 3.7.* 类似于 `PyErr_SetFromWindowsErr()`，额外的参数指定要触发的异常类型。

适用：Windows。

PyObject ***PyErr_SetFromWindowsErrWithFilename**(int *ierr*, **const** char *filename)
Return value: Always NULL. Part of the Stable ABI *on Windows since version 3.7.* 类似于 `PyErr_SetFromWindowsErrWithFilenameObject()`，但是 filename 是以 C 字符串形式给出的。*filename* 是从文件系统编码（`os.fsdecode()`）解码出来的。

适用：Windows。

PyObject ***PyErr_SetExcFromWindowsErrWithFilenameObject**(*PyObject* *type, int *ierr*, *PyObject* *filename)
Return value: Always NULL. Part of the Stable ABI *on Windows since version 3.7.* 类似于 `PyErr_SetFromWindowsErrWithFilenameObject()`，额外参数指定要触发的异常类型。

适用：Windows。

PyObject ***PyErr_SetExcFromWindowsErrWithFilenameObjects**(*PyObject* *type, int *ierr*, *PyObject* *filename, *PyObject* *filename2)
Return value: Always NULL. Part of the Stable ABI *on Windows since version 3.7.* 类似于 `PyErr_SetExcFromWindowsErrWithFilenameObject()`，但是接受第二个 filename 对象。

适用：Windows。

3.4 版新加入.

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

Return value: Always NULL. Part of the Stable ABI *on Windows since version 3.7.* 类似于`PyErr_SetFromWindowsErrWithFilename()`，额外参数指定要触发的异常类型。

适用：Windows。

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

Return value: Always NULL. Part of the Stable ABI *since version 3.7.* 这是触发 `ImportError` 的便捷函数。*msg* 将被设为异常的消息字符串。*name* 和 *path* ，（都可以为 `NULL` ），将用来被设置 `ImportError` 对应的属性 `name` 和 `path`。

3.3 版新加入.

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

Return value: Always NULL. Part of the Stable ABI *since version 3.6.* 和`PyErr_SetImportError()` 很类似，但这个函数允许指定一个 `ImportError` 的子类来触发。

3.6 版新加入.

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

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

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.* 类似于`PyErr_SyntaxLocationObject()`，但 *filename* 是用*filesystem encoding and error handler* 解码的字节串。

3.2 版新加入.

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

Part of the Stable ABI. 类似于`PyErr_SyntaxLocationEx()`，但省略了 *col_offset* parameter 形参。

void **PyErr_BadInternalCall**()

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

5.3 发出警告

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

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

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

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

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

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

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

int **PyErr_WarnFormat**(*PyObject* **category*, *Py_ssize_t* *stack_level*, **const** char **format*, ...)
Part of the Stable ABI. 类似于`PyErr_WarnEx()` 的函数，但使用`PyUnicode_FromFormat()` 来格式化警告消息。*format* 是使用 ASCII 编码的字符串。

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.* 类似于`PyErr_WarnFormat()` 的函数，但 *category* 是 `ResourceWarning` 并且它会将 *source* 传给 `warnings.WarningMessage()`。

3.6 版新加入.

5.4 查询错误指示器

PyObject ***PyErr_Occurred**()
Return value: Borrowed reference. *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.

呼叫者必须持有 GIL。

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Part of the Stable ABI *since version 3.7.* 提取异常信息，即从 `sys.exc_info()` 所得到的。这是指一个已被捕获的异常，而不是刚被引发的异常。返回分别指向三个对象的新引用，其中任何一个均可以为 NULL。不会修改异常信息的状态。

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

3.3 版新加入.

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

Part of the Stable ABI *since version 3.7.* 设置异常信息，即从 `sys.exc_info()` 所得到的。这是指一个已被捕获的异常，而不是刚被引发的异常。此函数会偷取对参数的引用。要清空异常状态，请为所有三个参数传入 NULL。对于有关三个参数的一般规则，请参阅 `PyErr_Restore()`。

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

3.3 版新加入.

5.5 信号处理

int **PyErr_CheckSignals**()

Part of the Stable ABI. 这个函数与 Python 的信号处理交互。

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

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

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

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

備註： 针对 `SIGINT` 的默认 Python 信号处理句柄会引发 `KeyboardInterrupt` 异常。

void **PyErr_SetInterrupt**()

Part of the Stable ABI. 模拟一个 `SIGINT` 信号到达的效果。这等价于 `PyErr_SetInterruptEx(SIGINT)`。

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

int **PyErr_SetInterruptEx**(int *signum*)

Part of the Stable ABI *since version 3.10.* 模拟一个信号到达的效果。当下次*PyErr_CheckSignals()* 被调用时，将会调用针对指定的信号编号的 Python 信号处理句柄。

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

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

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

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

3.10 版新加入.

int **PySignal_SetWakeupFd**(int *fd*)

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

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

3.5 版更變: 在 Windows 上，此函数现在也支持套接字处理。

5.6 例外類F

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

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

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

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

Return value: New reference. Part of the Stable ABI. 和 *PyErr_NewException()* 一样，除了可以轻松地给新的异常类一个文档字符串：如果 *doc* 属性非空，它将用作异常类的文档字符串。

3.2 版新加入.

5.7 例外物件

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

Return value: New reference. Part of the Stable ABI. 将与异常相关联的回溯作为一个新引用返回，可以通过 `__traceback__` 在 Python 中访问。如果没有已关联的回溯，则返回 `NULL`。

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

Part of the Stable ABI. 将异常关联的回溯设置为 *tb* 。使用 "Py_None"清除它。

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

Return value: New reference. Part of the Stable ABI. 将与异常相关联的上下文（在处理 *ex* 的过程中引发的另一个异常实例）作为一个新引用返回，可以通过 `__context__` 在 Python 中访问。如果没有已关联的上下文，则返回 `NULL`。

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

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

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

Return value: New reference. Part of the Stable ABI. 将与异常相关联的原因（一个异常实例，或是 `None`，由 `raise ... from ...` 设置）作为一个新引用返回，可在 Python 中通过 `__cause__` 来访问。

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

Part of the Stable ABI. 将与异常相关联的原因设置为 *cause*。使用 `NULL` 来清空它。它没有用来确保 *cause* 是一个异常实例或 `None` 的类型检查。这将偷取一个指向 *cause* 的引用。

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

5.8 Unicode 异常对象

下列函数被用于创建和修改来自 C 的 Unicode 异常。

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

Return value: New reference. Part of the Stable ABI. 创建一个 `UnicodeDecodeError` 对象并附带 *encoding*, *object*, *length*, *start*, *end* 和 *reason* 等属性。*encoding* 和 *reason* 为 UTF-8 编码的字符串。

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

Return value: New reference. 创建一个 `UnicodeEncodeError` 对象并附带 *encoding*, *object*, *length*, *start*, *end* 和 *reason*。*encoding* 和 *reason* 都是以 UTF-8 编码的字符串。

3.3 版後已棄用: 3.11

`Py_UNICODE` 自 Python 3.3 起已被弃用。请迁移至 `PyObject_CallFunction(PyExc_UnicodeEncodeError, "sOnns", ...)`。

PyObject ***PyUnicodeTranslateError_Create**(**const** *Py_UNICODE *object*, *Py_ssize_t length*, *Py_ssize_t start*, *Py_ssize_t end*, **const** char **reason*)

Return value: New reference. 创建一个 `UnicodeTranslateError` 对象并附带 *object*, *length*, *start*, *end* 和 *reason*。*reason* 是一个以 UTF-8 编码的字符串。

3.3 版後已棄用: 3.11

`Py_UNICODE` 自 Python 3.3 起已被弃用。请迁移至 `PyObject_CallFunction(PyExc_UnicodeTranslateError, "Onns", ...)`。

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

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

Return value: New reference. Part of the Stable ABI. 返回给定异常对象的 *encoding* 属性

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

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

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

Return value: New reference. Part of the Stable ABI. 返回给定异常对象的 *object* 属性

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

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

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

Part of the Stable ABI. 获取给定异常对象的 *start* 属性并将其放入 **start*。*start* 必须不为 `NULL`。成功时返回 `0`，失败时返回 `-1`。

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

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

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

Part of the Stable ABI. 将给定异常对象的 *start* 属性设为 *start*。成功时返回 `0`，失败时返回 `-1`。

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

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

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

Part of the Stable ABI. 获取给定异常对象的 *end* 属性并将其放入 **end*。*end* 必须不为 `NULL`。成功时返回 `0`，失败时返回 `-1`。

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

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

Part of the Stable ABI. 将给定异常对象的 *end* 属性设为 *end*。成功时返回 0，失败时返回 -1。

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

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

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

Return value: New reference. Part of the Stable ABI. 返回给定异常对象的 *reason* 属性

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

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

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

Part of the Stable ABI. 将给定异常对象的 *reason* 属性设为 *reason*。成功时返回 0，失败时返回 -1。

5.9 递归控制

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

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

Part of the Stable ABI *since version 3.9.* 标记一个递归的 C 层级调用即将被执行的点位。

如果定义了 `USE_STACKCHECK`，此函数会使用`PyOS_CheckStack()` 来检查操作系统堆栈是否溢出。在这种情况下，它将设置一个 `MemoryError` 并返回非零值。

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

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

3.9 版更變: 此函数现在也在受限 API 中可用。

void **Py_LeaveRecursiveCall** (void)

Part of the Stable ABI *since version 3.9.* 结束一个`Py_EnterRecursiveCall()`。必须针对`Py_EnterRecursiveCall()` 的每个 成功的发起调用操作执行一次调用。

3.9 版更變: 此函数现在也在受限 API 中可用。

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

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

Part of the Stable ABI. 在`tp_repr` 实现的开头被调用以检测循环。

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

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

在其他情况下，此函数将返回零而`tp_repr` 实现将可正常继续。

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

Part of the Stable ABI. 结束一个`Py_ReprEnter()`。必须针对每个返回零的`Py_ReprEnter()` 的发起调用操作调用一次。

5.10 标准异常

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 名称	Python 名称	🄵解
PyExc_BaseException	BaseException	[1]
PyExc_Exception	Exception	[1]
PyExc_ArithmeticError	ArithmeticError	[1]
PyExc_AssertionError	AssertionError	
PyExc_AttributeError	AttributeError	
PyExc_BlockingIOError	BlockingIOError	
PyExc_BrokenPipeError	BrokenPipeError	
PyExc_BufferError	BufferError	
PyExc_ChildProcessError	ChildProcessError	
PyExc_ConnectionAbortedError	ConnectionAbortedError	
PyExc_ConnectionError	ConnectionError	
PyExc_ConnectionRefusedError	ConnectionRefusedError	
PyExc_ConnectionResetError	ConnectionResetError	
PyExc_EOFError	EOFError	
PyExc_FileExistsError	FileExistsError	
PyExc_FileNotFoundError	FileNotFoundError	
PyExc_FloatingPointError	FloatingPointError	
PyExc_GeneratorExit	GeneratorExit	
PyExc_ImportError	ImportError	
PyExc_IndentationError	IndentationError	
PyExc_IndexError	IndexError	
PyExc_InterruptedError	InterruptedError	
PyExc_IsADirectoryError	IsADirectoryError	
PyExc_KeyError	KeyError	
PyExc_KeyboardInterrupt	KeyboardInterrupt	
PyExc_LookupError	LookupError	[1]
PyExc_MemoryError	MemoryError	
PyExc_ModuleNotFoundError	ModuleNotFoundError	
PyExc_NameError	NameError	
PyExc_NotADirectoryError	NotADirectoryError	
PyExc_NotImplementedError	NotImplementedError	
PyExc_OSError	OSError	[1]
PyExc_OverflowError	OverflowError	
PyExc_PermissionError	PermissionError	
PyExc_ProcessLookupError	ProcessLookupError	
PyExc_RecursionError	RecursionError	
PyExc_ReferenceError	ReferenceError	
PyExc_RuntimeError	RuntimeError	
PyExc_StopAsyncIteration	StopAsyncIteration	
PyExc_StopIteration	StopIteration	
PyExc_SyntaxError	SyntaxError	
PyExc_SystemError	SystemError	
PyExc_SystemExit	SystemExit	
PyExc_TabError	TabError	

下页继续

表 1 – 繼續上一頁

C 名称	Python 名称	F解
PyExc_TimeoutError	TimeoutError	
PyExc_TypeError	TypeError	
PyExc_UnboundLocalError	UnboundLocalError	
PyExc_UnicodeDecodeError	UnicodeDecodeError	
PyExc_UnicodeEncodeError	UnicodeEncodeError	
PyExc_UnicodeError	UnicodeError	
PyExc_UnicodeTranslateError	UnicodeTranslateError	
PyExc_ValueError	ValueError	
PyExc_ZeroDivisionError	ZeroDivisionError	

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。

3.5 版新加入: PyExc_StopAsyncIteration 和 PyExc_RecursionError。

3.6 版新加入: PyExc_ModuleNotFoundError。

这些是兼容性别名 PyExc_OSError:

C 名称	F解
PyExc_EnvironmentError	
PyExc_IOError	
PyExc_WindowsError	[2]

3.3 版更變: 这些别名曾经是单独的异常类型。

F解:

5.11 标准警告类别

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 名称	Python 名称	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	

[1] 这是其他标准异常的基类。
[2] 仅在 Windows 中定义；检测是否定义了预处理程序宏 MS_WINDOWS ，以便保护用到它的代码。

3.2 版新加入: PyExc_ResourceWarning.

F解:

[3] 这是其他标准警告类别的基类。

CHAPTER 6

工具

本章中的函式可用來執行各種工具任務，包括幫助 C 程式碼提升跨平臺可攜性 (portable)、在 C 中使用 Python module（模組）、以及剖析函式引數並基於 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*)

Return value: New reference. Part of the Stable ABI *since version 3.6.* 返回 *path* 在文件系统中的表示形式。如果该对象是一个 `str` 或 `bytes` 对象，则它的引用计数将会增加。如果该对象实现了 `os.PathLike` 接口，则只要它是一个 `str` 或 `bytes` 对象就将返回 `__fspath__()`。在其他情况下将引发 `TypeError` 并返回 `NULL`。

3.6 版新加入.

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

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

void **PyOS_BeforeFork**()

Part of the Stable ABI *on platforms with fork() since version 3.7.* 在进程分叉之前准备某些内部状态的函数。此函数应当在调用 `fork()` 或者任何类似的克隆当前进程的函数之前被调用。只适用于定义了 `fork()` 的系统。

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

3.7 版新加入.

void **PyOS_AfterFork_Parent**()

Part of the Stable ABI *on platforms with fork() since version 3.7.* 在进程分叉之后更新某些内部状态的函数。

此函数应当在调用 fork() 或任何类似的克隆当前进程的函数之后被调用，无论进程克隆是否成功。只适用于定义了 fork() 的系统。

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

3.7 版新加入.

void **PyOS_AfterFork_Child**()

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

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

3.7 版新加入.

也参考:

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

void **PyOS_AfterFork**()

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

3.7 版後已F用: 此函数已被 *PyOS_AfterFork_Child()* 取代。

int **PyOS_CheckStack**()

Part of the Stable ABI *on platforms with USE_STACKCHECK since version 3.7.* 当解释器的栈空间耗尽时返回真值。这是一个可靠的检查，但仅在定义了 USE_STACKCHECK 时可用（目前在 Windows 上使用 Microsoft Visual C++ 编译器）。USE_STACKCHECK 将被自动定义；你绝不应该在你自己的代码中改变此定义。

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 is a typedef alias for void (*)int.

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! PyOS_sighandler_t is a typedef alias for void (*)int.

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

Part of the Stable ABI *since version 3.7.*

> **警告：** 此函数不应当被直接调用：请使用 *PyConfig* API 以及可确保 *对 Python 进行预初始化* 的 PyConfig_SetBytesString() 函数。
>
> 此函数不可在 This function must not be called before *对 Python 进行预初始化* 之前被调用以便正确地配置 LC_CTYPE 语言区域：请参阅 *Py_PreInitialize()* 函数。

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

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

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

filesystem encoding and error handler 是由 `PyConfig_Read()` 来选择的：参见`PyConfig` 的`filesystem_encoding` 和`filesystem_errors` 等成员。

解码错误绝对不应当发生，除非 C 库有程序缺陷。

请使用`Py_EncodeLocale()` 函数来将字符串编码回字节串。

也參考:

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

3.5 版新加入.

3.7 版更變: 现在此函数在 Python UTF-8 模式下将使用 UTF-8 编码格式。

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

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

Part of the Stable ABI *since version 3.7.* 将一个由宽字符组成的字符串编码为*filesystem encoding and error handler*。如果错误处理句柄为 surrogateescape 错误处理句柄，则在 U+DC80..U+DCFF 范围内的代理字符会被转换为字节值 0x80..0xFF。

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

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

filesystem encoding and error handler 是由 `PyConfig_Read()` 来选择的：参见`PyConfig` 的`filesystem_encoding` 和`filesystem_errors` 等成员。

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

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

也參考:

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

3.5 版新加入.

3.7 版更變: 现在此函数在 Python UTF-8 模式下将使用 UTF-8 编码格式。

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

6.2 系統函式

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

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

Return value: Borrowed reference. Part of the Stable ABI. 返回来自 sys 模块的对象 *name* 或者如果它不存在则返回 NULL，并且不会设置异常。

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

Part of the Stable ABI. 将 sys 模块中的 *name* 设为 *v* 除非 *v* 为 NULL，在此情况下 *name* 将从 sys 模块中被删除。成功时返回 0，发生错误时返回 -1。

void **PySys_ResetWarnOptions**()

Part of the Stable ABI. 将 sys.warnoptions 重置为空列表。此函数可在 *Py_Initialize()* 之前被调用。

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

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

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

Part of the Stable ABI. 将 *unicode* 添加到 sys.warnoptions。

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

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

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

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

Part of the Stable ABI. 将以 *format* 描述的输出字符串写入到 sys.stdout。不会引发任何异常，即使发生了截断（见下文）。

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

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

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

Part of the Stable ABI. 类似 *PySys_WriteStdout()*，但改为写入到 sys.stderr 或 *stderr*。

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

Part of the Stable ABI. 类似 PySys_WriteStdout() 的函数将会使用 *PyUnicode_FromFormatV()* 来格式化消息并且不会将消息截短至任意长度。

3.2 版新加入.

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

Part of the Stable ABI. 类似 *PySys_FormatStdout()*，但改为写入到 sys.stderr 或 *stderr*。

3.2 版新加入.

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

Part of the Stable ABI *since version 3.7.* 将 *s* 解析为一个由 -X 选项组成的集合并将它们添加到 *PySys_GetXOptions()* 所返回的当前选项映射。此函数可以在 *Py_Initialize()* 之前被调用。

3.2 版新加入.

PyObject ***PySys_GetXOptions**()

Return value: Borrowed reference. Part of the Stable ABI *since version 3.7.* 返回当前 `-X` 选项的字典，类似于 `sys._xoptions`。发生错误时，将返回 `NULL` 并设置一个异常。

3.2 版新加入.

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

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

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

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

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

3.8 版新加入.

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

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

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

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

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

The hook function is of type `int (*)`**`const`**` char *event, `*`PyObject`*` *args, void *userData`, where *args* is guaranteed to be a *PyTupleObject*. The hook function is always called with the GIL held by the Python interpreter that raised the event.

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

引发一个 审计事件 `sys.addaudithook`，没有附带参数。

3.8 版新加入.

6.3 行程（Process）控制

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

Part of the Stable ABI. 打印一个致命错误消息并杀掉进程。不会执行任何清理。此函数应当仅在检测到可能令继续使用 Python 解释器变得危险的条件时被发起调用；例如，当对象管理已被破坏的时候。在 Unix 上，标准 C 库函数 `abort()` 会被调用并将由它来尝试产生一个 `core` 文件。

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

3.9 版更變: 自动记录函数名称。

void **Py_Exit**(int *status*)

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

3.6 版更變: 来自最终化的错误不会再被忽略。

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

6.4 滙入模組

PyObject ***PyImport_ImportModule**(**const** char *name)
Return value: New reference. Part of the Stable ABI. 这是下面`PyImport_ImportModuleEx()` 的简化版接口，将 *globals* 和 *locals* 参数设为 `NULL` 并将 *level* 设为 0。当 *name* 参数包含一个点号（即指定了一个包的子模块）时，*fromlist* 参数会被设为列表 `['*']` 这样返回值将为所指定的模块而不像在其他情况下那样为包含模块的最高层级包。(不幸的是，这在 *name* 实际上是指定一个子包而非子模块时将有一个额外的副作用：在包的 `__all__` 变量中指定的子模块会被加载。) 返回一个对所导入模块的新引用，或是在导入失败时返回 `NULL` 并设置一个异常。模块导入失败同模块不会留在 `sys.modules` 中。

该函数总是使用绝对路径导入。

PyObject ***PyImport_ImportModuleNoBlock**(**const** char *name)
Return value: New reference. Part of the Stable ABI. 该函数是`PyImport_ImportModule()` 的一个被遗弃的别名。

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

PyObject ***PyImport_ImportModuleEx**(**const** char *name, *PyObject* *globals, *PyObject* *locals, *PyObject* *fromlist)
Return value: New reference. 导入一个模块。请参阅内置 Python 函数 `__import__()` 获取完善的相关描述。

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

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

PyObject ***PyImport_ImportModuleLevelObject**(*PyObject* *name, *PyObject* *globals, *PyObject* *locals, *PyObject* *fromlist, int level)
Return value: New reference. Part of the Stable ABI *since version 3.7.* 导入一个模块。关于此函数的最佳说明请参考内置 Python 函数 `__import__()`，因为标准 `__import__()` 函数会直接调用此函数。

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

3.3 版新加入.

PyObject ***PyImport_ImportModuleLevel**(**const** char *name, *PyObject* *globals, *PyObject* *locals, *PyObject* *fromlist, int level)
Return value: New reference. Part of the Stable ABI. 类似于`PyImport_ImportModuleLevelObject()`，但其名称为 UTF-8 编码的字符串而不是 Unicode 对象。

3.3 版更變: 不再接受 *level* 为负数值。

PyObject ***PyImport_Import**(*PyObject* *name)
Return value: New reference. Part of the Stable ABI. 这是一个调用了当前"导入钩子函数"的更高层级

接口（显式指定 *level* 为 0，表示绝对导入）。它将发起调用当前全局作用域下 `__builtins__` 中的 `__import__()` 函数。这意味着将使用当前环境下安装的任何导入钩子来完成导入。

该函数总是使用绝对路径导入。

PyObject ***PyImport_ReloadModule**(*PyObject* *m*)
Return value: New reference. Part of the Stable ABI. 重载一个模块。返回一个指向被重载模块的新引用，或者在失败时返回 NULL 并设置一个异常（在此情况下模块仍然会存在）。

PyObject ***PyImport_AddModuleObject**(*PyObject* *name*)
Return value: Borrowed reference. Part of the Stable ABI *since version 3.7.* 返回对应于某个模块名称的模块对象。*name* 参数的形式可以为 `package.module`。如果存在 modules 字典则首先检查该字典，如果找不到，则创建一个新模块并将其插入到 modules 字典。在失败时返回 NULL 并设置一个异常。

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

3.3 版新加入.

PyObject ***PyImport_AddModule**(**const** char *name*)
Return value: Borrowed reference. Part of the Stable ABI. 类似于 *PyImport_AddModuleObject()*，但其名称为 UTF-8 编码的字符串而不是 Unicode 对象。object.

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

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

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

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

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

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

PyObject ***PyImport_ExecCodeModuleEx**(**const** char *name*, *PyObject* *co*, **const** char *pathname*)
Return value: New reference. Part of the Stable ABI. 类似于 *PyImport_ExecCodeModule()*，但如果 *pathname* 不为 NULL 则会被设为模块对象的 `__file__` 属性的值。

也請見 *PyImport_ExecCodeModuleWithPathnames()*。

PyObject ***PyImport_ExecCodeModuleObject**(*PyObject* *name*, *PyObject* *co*, *PyObject* *pathname*, *PyObject* *cpathname*)
Return value: New reference. Part of the Stable ABI *since version 3.7.* 类似于 *PyImport_ExecCodeModuleEx()*，但如果 *cpathname* 不为 NULL 则会被设为模块对象的 `__cached__` 值。在三个函数中，这是推荐使用的一个。

3.3 版新加入.

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

Return value: New reference. Part of the Stable ABI. 类似于*PyImport_ExecCodeModuleObject()*，但 *name*, *pathname* 和 *cpathname* 为 UTF-8 编码的字符串。如果 *pathname* 也被设为 NULL 则还会尝试根据 *cpathname* 推断出前者的值。

3.2 版新加入.

3.3 版更變: 如果只提供了字节码路径则会使用 imp.source_from_cache() 来计算源路径。

long **PyImport_GetMagicNumber**()

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

3.3 版更變: 當失敗時回傳 -1。

const char ***PyImport_GetMagicTag**()

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

3.2 版新加入.

PyObject ***PyImport_GetModuleDict**()

Return value: Borrowed reference. Part of the Stable ABI. 返回用于模块管理的字典 (即 sys.modules)。请注意这是针对每个解释器的变量。

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

Return value: New reference. Part of the Stable ABI *since version 3.8.* 返回给定名称的已导入模块。如果模块尚未导入则返回 NULL 但不会设置错误。如果查找失败则返回 NULL 并设置错误。

3.7 版新加入.

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

Return value: New reference. Part of the Stable ABI. 返回针对一个 sys.path/pkg.__path__ 中条目 *path* 的查找器对象，可能会通过 sys.path_importer_cache 字典来获取。如果它尚未被缓存，则会遍历 sys.path_hooks 直至找到一个能处理该 path 条目的钩子。如果没有可用的钩子则返回 None；这将告知调用方*path based finder* 无法为该 path 条目找到查找器。结果将缓存到 sys.path_importer_cache。返回一个指向查找器对象的新引用。

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

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

3.3 版新加入.

3.4 版更變: __file__ 属性将不再在模块上设置。

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

Part of the Stable ABI. 类似于*PyImport_ImportFrozenModuleObject()*，但其名称为 UTF-8 编码的字符串而不是 Unicode 对象。

struct _frozen

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

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

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

struct _inittab

描述内置模块列表中的一个条目的结构体。每个结构体都给出了内置在解释器中的某个模块的名称和初始化函数。名称是一个 ASCII 编码的字符串。嵌入了 Python 的程序可以使用该结构体的数组来与`PyImport_ExtendInittab()` 相结合以提供额外的内置模块。该结构体在 `Include/import.h` 中被定义为:

```
struct _inittab {
    const char *name;           /* ASCII encoded string */
    PyObject* (*initfunc)(void);
};
```

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

将内置模块表添加一组模块。*newtab* 数组必须以一个包含以 `NULL` 作为 `name` 字段的岗哨条目结束；未能提供岗哨值会导致内存错误。成功时返回 `0` 或者如果无法分配足够内存来扩展内部表则返回 `-1`。当发生失败时，将不会添加模块到内部表。此函数必须在`Py_Initialize()` 之前调用。

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

6.5 数据 marshal 操作支持

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

数字值在存储时会将最低位字节放在开头。

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

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

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.

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

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

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

Return value: New reference. 返回一个包含 *value* 的 marshal 表示形式的字节串对象。*version* 指明文件格式的版本。

以下函数允许读取并恢复存储为 marshal 格式的值。

long **PyMarshal_ReadLongFromFile**(FILE **file*)

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

发生错误时，将设置适当的异常 (EOFError) 并返回 -1。

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

发生错误时，将设置适当的异常 (EOFError) 并返回 -1。

PyObject ***PyMarshal_ReadObjectFromFile**(FILE **file*)
Return value: New reference. Return a Python object from the data stream in a FILE* opened for reading.

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

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

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

PyObject ***PyMarshal_ReadObjectFromString**(**const** char **data*, *Py_ssize_t* *len*)
Return value: New reference. 从包含指向 *data* 的 *len* 个字节的字节缓冲区对应的数据流返回一个 Python 对象。

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

6.6 解析参数并构建值变量

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

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

6.6.1 解析参数

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

字符串和缓存区

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

除非另有说明，缓冲区是不会以空终止的。

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).

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

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

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

3.5 版更變: 以前，当 Python 字符串中遇到了嵌入的 null 代码点会引发 TypeError 。

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

s# (str, read-only *bytes-like object*) [const char *, *Py_ssize_t*] 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 *] 与 s 类似，但 Python 对象也可能为 None，在这种情况下，C 指针设置为 NULL。

z*（str、*bytes-like object* 或 None）[Py_buffer] 与 s* 类似，但 Python 对象也可能为 None，在这种情况下，*Py_buffer* 结构的 buf 成员设置为 NULL。

z# (str, read-only *bytes-like object* 或者 None) [const char *, *Py_ssize_t*] 与 s# 类似，但 Python 对象也可能为 None，在这种情况下，C 指针设置为 NULL。

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

3.5 版更變: 以前，当字节缓冲区中遇到了嵌入的 null 字节会引发 TypeError 。

y* (*bytes-like object*) [Py_buffer] s* 的变式，不接受 Unicode 对象，只接受类字节类型变量。**这是接受二进制数据的推荐方法。**

y# (read-only *bytes-like object*) [const char *, *Py_ssize_t*] s# 的变式，不接受 Unicode 对象，只接受类字节类型变量。

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

3.5 版更變: 以前，当 Python 字符串中遇到了嵌入的 null 代码点会引发 TypeError 。

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: 这是旧版样式*Py_UNICODE* API; 请迁移至*PyUnicode_AsWideCharString()*.

u# (str) [const Py_UNICODE *, *Py_ssize_t*] u 的变式，存储两个 C 变量，第一个指针指向一个 Unicode 数据缓存区，第二个是它的长度。它允许 null 代码点。

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: 这是旧版样式*Py_UNICODE* API; 请迁移至*PyUnicode_AsWideCharString()*.

Z (str 或 None) [const Py_UNICODE *] 与 u 类似，但 Python 对象也可能为 None，在这种情况下*Py_UNICODE* 指针设置为 NULL。

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: 这是旧版样式*Py_UNICODE* API; 请迁移至*PyUnicode_AsWideCharString()*.

Z# (str 或 None) [const Py_UNICODE *, *Py_ssize_t*] 与 u# 类似，但 Python 对象也可能为 None，在这种情况下*Py_UNICODE* 指针设置为 NULL。

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: 这是旧版样式*Py_UNICODE* API; 请迁移至*PyUnicode_AsWideCharString()*.

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

es (str) [const char *encoding, char **buffer] s 的变式，它将编码后的 Unicode 字符存入字符缓冲区。它只处理没有嵌 NUL 字节的已编码数据。

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

et (str, bytes or bytearray) [const char *encoding, char **buffer] 和 es 相同，除了不用重编码传入的字符串对象。相反，它假设传入的参数是编码后的字符串类型。

es# (str) [const char *encoding, char **buffer, *Py_ssize_t* *buffer_length] s# 的变式，它将已编码的 Unicode 字符存入字符缓冲区。不像 es 表达式，它允许传入的数据包含 NUL 字符。

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

有两种操作方式：

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

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

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

`et#` (`str`, `bytes` 或 `bytearray`) [const char *encoding, char **buffer, `Py_ssize_t` *buffer_length] 和 `es#` 相同，除了不用重编码传入的字符串对象。相反，它假设传入的参数是编码后的字符串类型。

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□成 C 的 `short int`。

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

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

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

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

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

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

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

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

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

3.3 版更變: 允許 `bytearray` 物件。

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

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

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

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

其他物件

O（物件）[PyObject *] 将 Python 对象（不进行任何转换）存储在 C 对象指针中。因此，C 程序接收已传递的实际对象。对象的引用计数不会增加。存储的指针不是 `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.

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

3.1 版更變: 加入 `Py_CLEANUP_SUPPORTED`。

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

3.3 版新加入.

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

传递"long" 整型 (整型的值超过了平台的 `LONG_MAX` 限制) 是可能的，然而没有进行适当的范围检测——当接收字段太小而接收不到值时，最重要的位被静默地截断 (实际上，C 语言会在语义继承的基础上强制类型转换——期望的值可能会发生变化)。

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

| 表明在 Python 参数列表中剩下的参数都是可选的。C 变量对应的可选参数需要初始化为默认值——当一个可选参数没有指定时，`PyArg_ParseTuple()` 不能访问相应的 C 变量 (变量集) 的内容。

$ `PyArg_ParseTupleAndKeywords()` only：表明在 Python 参数列表中剩下的参数都是强制关键字参数。当前，所有强制关键字参数都必须也是可选参数，所以格式化字符串中 `|` 必须一直在 `$` 前面。

3.3 版新加入.

: 格式单元的列表结束标志；冒号后的字符串被用来作为错误消息中的函数名 (`PyArg_ParseTuple()` 函数引发的"关联值"异常)。

; 格式单元的列表结束标志；分号后的字符串被用来作为错误消息取代默认的错误消息。**:** 和 **;** 相互排斥。

注意任何由调用者提供的 Python 对象引用是 借来的引用；不要递减它们的引用计数！

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

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

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

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

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

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

3.6 版更變: 添加了*positional-only parameters* 的支持。

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

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

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

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

3.2 版新加入.

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

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

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

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.

这是一个使用此函数的示例，取自 `_weakref` 帮助模块用来弱化引用的源代码：

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

(下页继续)

(繼續上一頁)

```
    PyObject *result = NULL;

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

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

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

6.6.2 创建变量

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

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

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

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

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

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

s（str 或 None）[const char *] 使用 'utf-8' 编码将空终止的 C 字符串转换为 Python str 对象。如果 C 字符串指针为 NULL，则使用 None。

s#（str 或 None）[const char *, *Py_ssize_t*] 使用 'utf-8' 编码将 C 字符串及其长度转换为 Python str 对象。如果 C 字符串指针为 NULL，则长度将被忽略，并返回 None。

y (bytes) [const char *] 这将 C 字符串转换为 Python bytes 对象。如果 C 字符串指针为 NULL，则返回 None。

y# (bytes) [const char *, *Py_ssize_t*] 这会将 C 字符串及其长度转换为一个 Python 对象。如果该 C 字符串指针为 NULL，则返回 None。

z（str 或 None）[const char *] 和 s 相同。

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

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

U（str 或 None）[const char *] 和 s 相同。

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

O（物件）[PyObject *] 将 Python 对象传递不变（其引用计数除外，该计数由 1 递增）。如果传入的对象是 NULL 指针，则假定这是由于生成参数的调用发现错误并设置异常而引起的。因此，*Py_BuildValue()* 将返回 NULL，但不会引发异常。如果尚未引发异常，则设置 SystemError。

S（物件）[PyObject *] 和 O 相同。

N（物件）[PyObject *] 和 O 相同，然而它并不增加对象的引用计数。当通过调用参数列表中的对象构造器创建对象时很实用。

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*] 将一个 C 变量序列转换成 Python 元组并保持相同的元素数量。

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

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

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

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

Return value: New reference. Part of the Stable ABI. 和*Py_BuildValue()* 相同，然而它接受一个 va_list 类型的参数而不是可变数量的参数集。

6.7 字串轉F與格式化

數字轉F函數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. 根据格式字符串 *format* 和额外参数，输出不超过 *size* 个字节到 *str*。参见 Unix 手册页面 *snprintf(3)*。

int **PyOS_vsnprintf**(char *str, size_t size, **const** char *format, va_list va)
Part of the Stable ABI. 根据格式字符串 *format* 和变量参数列表 *va*，输出不超过 *size* 个字节到 *str*。参见 Unix 手册页面 *vsnprintf(3)*。

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

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*) 給這些函數應該被編譯如下：

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

以下函数提供与语言环境无关的字符串到数字转换。

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.

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

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

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

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

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*.

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

flags 可以为零或者其他值 Py_DTSF_SIGN, Py_DTSF_ADD_DOT_0 或 Py_DTSF_ALT 或其组合：

- Py_DTSF_SIGN 表示总是在返回的字符串前附加一个符号字符，即使 *val* 为非负数。
- Py_DTSF_ADD_DOT_0 表示确保返回的字符串看起来不像是一个整数。
- Py_DTSF_ALT 表示应用" 替代的" 格式化规则。相关细节请参阅PyOS_snprintf() '#' 定义文档。

如果 *ptype* 不为 NULL，则它指向的值将被设为 Py_DTST_FINITE, Py_DTST_INFINITE 或 Py_DTST_NAN 中的一个，分别表示 *val* 是一个有限数字、无限数字或非数字。

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

3.1 版新加入.

int **PyOS_stricmp**(const char **s1*, const char **s2*)
字符串不区分大小写。该函数几乎与 strcmp() 的工作方式相同，只是它忽略了大小写。

int **PyOS_strnicmp**(const char **s1*, const char **s2*, *Py_ssize_t size*)
字符串不区分大小写。该函数几乎与 strncmp() 的工作方式相同，只是它忽略了大小写。

6.8 反射

PyObject ***PyEval_GetBuiltins**(void)
Return value: Borrowed reference. Part of the Stable ABI. 返回当前执行帧中内置函数的字典，如果当前没有帧正在执行，则返回线程状态的解释器。

PyObject ***PyEval_GetLocals**(void)
Return value: Borrowed reference. Part of the Stable ABI. 返回当前执行帧中局部变量的字典，如果没有当前执行的帧则返回 NULL。

PyObject ***PyEval_GetGlobals**(void)
Return value: Borrowed reference. Part of the Stable ABI. 返回当前执行帧中全局变量的字典，如果没有当前执行的帧则返回 NULL。

PyFrameObject ***PyEval_GetFrame**(void)
Return value: Borrowed reference. Part of the Stable ABI. 返回当前线程状态的帧，如果没有当前执行的帧则返回 NULL。

另請見PyThreadState_GetFrame()。

PyFrameObject ***PyFrame_GetBack**(*PyFrameObject* **frame*)
获取 *frame* 为下一个外部帧。

参考返回一个*strong reference*，或者如果 *frame* 没有外部帧则返回 NULL。

frame 不可F NULL。

3.9 版新加入.

PyCodeObject ***PyFrame_GetCode**(*PyFrameObject* **frame*)
Part of the Stable ABI *since version 3.10.* 获取 *frame* 的代码。

返回一个*strong reference*。

frame 必须不为 NULL。结果（帧的代码）不能为 NULL。

3.9 版新加入.

int **PyFrame_GetLineNumber**(*PyFrameObject* **frame*)
Part of the Stable ABI *since version 3.10.* 返回 *frame* 当前正在执行的行号。

frame 不可F NULL。

const char ***PyEval_GetFuncName** (*PyObject* **func*)
Part of the Stable ABI. 如果 *func* 是函数、类或实例对象，则返回它的名称，否则返回 *func* 的类型的名称。

const char ***PyEval_GetFuncDesc** (*PyObject* **func*)
Part of the Stable ABI. 根据 *func* 的类型返回描述字符串。返回值包括函数和方法的"()", " constructor", " instance" 和" object"。与 `PyEval_GetFuncName()` 的结果连接，结果将是 *func* 的描述。

6.9 编解码器注册与支持功能

int **PyCodec_Register** (*PyObject* **search_function*)
Part of the Stable ABI. 注册一个新的编解码器搜索函数。

作为副作用，其尝试加载 `encodings` 包，如果尚未完成，请确保它始终位于搜索函数列表的第一位。

int **PyCodec_Unregister** (*PyObject* **search_function*)
Part of the Stable ABI *since version 3.10.* 注销一个编解码器搜索函数并清空注册表缓存。如果指定搜索函数未被注册，则不做任何操作。成功时返回 0。出错时引发一个异常并返回 -1。

3.10 版新加入.

int **PyCodec_KnownEncoding** (**const** char **encoding*)
Part of the Stable ABI. 根据注册的给定 *encoding* 的编解码器是否已存在而返回 `1` 或 `0`。此函数总能成功。

PyObject ***PyCodec_Encode** (*PyObject* **object*, **const** char **encoding*, **const** char **errors*)
Return value: New reference. Part of the Stable ABI. 泛型编解码器基本编码 API。

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

PyObject ***PyCodec_Decode** (*PyObject* **object*, **const** char **encoding*, **const** char **errors*)
Return value: New reference. Part of the Stable ABI. 泛型编解码器基本解码 API。

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

6.9.1 Codec 查找 API

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

PyObject ***PyCodec_Encoder** (**const** char **encoding*)
Return value: New reference. Part of the Stable ABI. 为给定的 *encoding* 获取一个编码器函数。

PyObject ***PyCodec_Decoder** (**const** char **encoding*)
Return value: New reference. Part of the Stable ABI. 为给定的 *encoding* 获取一个解码器函数。

PyObject ***PyCodec_IncrementalEncoder** (**const** char **encoding*, **const** char **errors*)
Return value: New reference. Part of the Stable ABI. 为给定的 *encoding* 获取一个 `IncrementalEncoder` 对象。

PyObject ***PyCodec_IncrementalDecoder** (**const** char **encoding*, **const** char **errors*)
Return value: New reference. Part of the Stable ABI. 为给定的 *encoding* 获取一个 `IncrementalDecoder` 对象。

PyObject ***PyCodec_StreamReader** (**const** char **encoding*, *PyObject* **stream*, **const** char **errors*)
Return value: New reference. Part of the Stable ABI. 为给定的 *encoding* 获取一个 `StreamReader` 工厂函数。

PyObject ***PyCodec_StreamWriter** (**const** char **encoding*, *PyObject* **stream*, **const** char **errors*)
Return value: New reference. Part of the Stable ABI. 为给定的 *encoding* 获取一个 `StreamWriter` 工厂函数。

6.9.2 用于 Unicode 编码错误处理程序的注册表 API

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

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

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

成功则返回 "0"，失败则返回 "-1"

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

Return value: New reference. Part of the Stable ABI. 查找在 *name* 之下注册的错误处理回调函数。作为特例还可以传入 `NULL`，在此情况下将返回针对"strict" 的错误处理回调函数。

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

Return value: Always NULL. Part of the Stable ABI. 引发 *exc* 作为异常。

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

Return value: New reference. Part of the Stable ABI. 忽略 unicode 错误，跳过错误的输入。

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

Return value: New reference. Part of the Stable ABI. 使用 `?` 或 `U+FFFD` 替换 unicode 编码错误。

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

Return value: New reference. Part of the Stable ABI. 使用 XML 字符引用替换 unicode 编码错误。

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

Return value: New reference. Part of the Stable ABI. 使用反斜杠转义符 (`\x`, `\u` 和 `\U`) 替换 unicode 编码错误。

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

Return value: New reference. Part of the Stable ABI *since version 3.7.* 使用 `\N{...}` 转义符替换 unicode 编码错误。

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□、或具有廣泛類□的物件型□（例如所有數值型□或所有序列型□）。當使用於不適用的物件型□時，他們會引發一個 Python □常 (exception)。

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

7.1 对象协议

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

Py_RETURN_NOTIMPLEMENTED
: C 函数内部应正确处理`Py_NotImplemented` 的返回过程（即增加 NotImplemented 的引用计数并返回之）。

int **PyObject_Print**(*PyObject* **o*, FILE **fp*, int *flags*)
: 将对象 *o* 写入到文件 *fp*。出错时返回 `-1` 。旗标参数被用于启用特定的输出选项。目前唯一支持的选项是 `Py_PRINT_RAW`；如果给出该选项，则将写入对象的 `str()` 而不是 `repr()`。

int **PyObject_HasAttr**(*PyObject* **o*, *PyObject* **attr_name*)
: *Part of the* Stable ABI. 如果 *o* 带有属性 *attr_name*，则返回 `1`，否则返回 `0`。这相当于 Python 表达式 `hasattr(o, attr_name)`。此函数总是成功。

  注意，在调用 `__getattr__()` 和 `__getattribute__()` 方法时发生的异常将被抑制。若要获得错误报告，请换用`PyObject_GetAttr()` 。

int **PyObject_HasAttrString**(*PyObject* **o*, **const** char **attr_name*)
: *Part of the* Stable ABI. 如果 *o* 带有属性 *attr_name*，则返回 `1`，否则返回 `0`。这相当于 Python 表达式 `hasattr(o, attr_name)`。此函数总是成功。

  注意，在调用 `__getattr__()` 和 `__getattribute__()` 方法并创建一个临时字符串对象时，异常将被抑制。若要获得错误报告，请换用`PyObject_GetAttrString()` 。

PyObject ***PyObject_GetAttr**(*PyObject* **o*, *PyObject* **attr_name*)
Return value: New reference. Part of the Stable ABI. 从对象 *o* 中读取名为 *attr_name* 的属性。成功返回属性值，失败则返回 NULL。这相当于 Python 表达式 o.attr_name。

PyObject ***PyObject_GetAttrString**(*PyObject* **o*, **const** char **attr_name*)
Return value: New reference. Part of the Stable ABI. 从对象 *o* 中读取一个名为 *attr_name* 的属性。成功时返回属性值，失败则返回 NULL。这相当于 Python 表达式 o.attr_name。

PyObject ***PyObject_GenericGetAttr**(*PyObject* **o*, *PyObject* **name*)
Return value: New reference. Part of the Stable ABI. 通用的属性获取函数，用于放入类型对象的 tp_getattro 槽中。它在类的字典中（位于对象的 MRO 中）查找某个描述符，并在对象的 __dict__ 中查找某个属性。正如 descriptors 所述，数据描述符优先于实例属性，而非数据描述符则不优先。失败则会触发 AttributeError 。

int **PyObject_SetAttr**(*PyObject* **o*, *PyObject* **attr_name*, *PyObject* **v*)
Part of the Stable ABI. 将对象 *o* 中名为 *attr_name* 的属性值设为 *v* 。失败时引发异常并返回 -1；成功时返回 "0" 。这相当于 Python 语句 o.attr_name = v。

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

int **PyObject_SetAttrString**(*PyObject* **o*, **const** char **attr_name*, *PyObject* **v*)
Part of the Stable ABI. 将对象 *o* 中名为 *attr_name* 的属性值设为 *v* 。失败时引发异常并返回 -1；成功时返回 "0" 。这相当于 Python 语句 o.attr_name = v。

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

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

int **PyObject_DelAttr**(*PyObject* **o*, *PyObject* **attr_name*)
删除对象 *o* 中名为 *attr_name* 的属性。失败时返回 -1。这相当于 Python 语句 del o.attr_name。

int **PyObject_DelAttrString**(*PyObject* **o*, **const** char **attr_name*)
删除对象 *o* 中名为 *attr_name* 的属性。失败时返回 -1。这相当于 Python 语句 del o.attr_name。

PyObject ***PyObject_GenericGetDict**(*PyObject* **o*, void **context*)
Return value: New reference. Part of the Stable ABI *since version 3.10.* __dict__ 描述符的获取函数的一种通用实现。必要时会创建该字典。

3.3 版新加入.

int **PyObject_GenericSetDict**(*PyObject* **o*, *PyObject* **value*, void **context*)
Part of the Stable ABI *since version 3.7.* __dict__ 描述符设置函数的一种通用实现。这里不允许删除该字典。

3.3 版新加入.

PyObject ***PyObject_RichCompare**(*PyObject* **o1*, *PyObject* **o2*, int *opid*)
Return value: New reference. Part of the Stable ABI. 用 *opid* 指定的操作比较 *o1* 和 *o2* 的值，必须是 Py_LT 、Py_LE 、Py_EQ 、Py_NE 、Py_GT 或 Py_GE 之一，分别对应于 "<、``<= 、== 、!= 、> 或 >=。这相当于 Python 表达式 o1 op o2，其中 op 是对应于 *opid* 的操作符。成功时返回比较值，失败时返回 NULL。

int **PyObject_RichCompareBool**(*PyObject* **o1*, *PyObject* **o2*, int *opid*)
Part of the Stable ABI. 用 *opid* 指定的操作比较 *o1* 和 *o2* 的值，必须是 Py_LT 、Py_LE 、Py_EQ 、Py_NE 、Py_GT 或 Py_GE 之一，分别对应于 < 、<=、== 、!= 、> 或 >=。错误时返回 -1，若结果为 false 则返回 0，否则返回 1。这相当于 Python 表达式 o1 op o2，其中 op 是对应于 *opid* 的操作符。

備註: 如果 *o1* 和 *o2* 是同一个对象，*PyObject_RichCompareBool()* 为 Py_EQ 则返回 1 ，为 Py_NE 则返回 0。

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

Part of the Stable ABI. Format *obj* using *format_spec*. This is equivalent to the Python expression `format(obj, format_spec)`.

format_spec may be NULL. In this case the call is equivalent to `format(obj)`. Returns the formatted string on success, NULL on failure.

PyObject ***PyObject_Repr**(*PyObject* *o)

Return value: New reference. Part of the Stable ABI. 计算对象 *o* 的字符串形式。成功时返回字符串，失败时返回 NULL。这相当于 Python 表达式 `repr(o)`。由内置函数 `repr()` 调用。

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

PyObject ***PyObject_ASCII**(*PyObject* *o)

Return value: New reference. Part of the Stable ABI. 与*PyObject_Repr()* 一样，计算对象 *o* 的字符串形式，但在*PyObject_Repr()* 返回的字符串中用 \x、\u 或 \U 转义非 ASCII 字符。这将生成一个类似于 Python 2 中由*PyObject_Repr()* 返回的字符串。由内置函数 `ascii()` 调用。

PyObject ***PyObject_Str**(*PyObject* *o)

Return value: New reference. Part of the Stable ABI. 计算对象 *o* 的字符串形式。成功时返回字符串，失败时返回 NULL。这相当于 Python 表达式 `str(o)`。由内置函数 `str()` 调用，因此也由 `print()` 函数调用。

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

PyObject ***PyObject_Bytes**(*PyObject* *o)

Return value: New reference. Part of the Stable ABI. 计算对象 *o* 的字节形式。失败时返回 NULL，成功时返回一个字节串对象。这相当于 *o* 不是整数时的 Python 表达式 `bytes(o)` 。与 `bytes(o)` 不同的是，当 *o* 是整数而不是初始为 0 的字节串对象时，会触发 TypeError。

int **PyObject_IsSubclass**(*PyObject* *derived, *PyObject* *cls)

Part of the Stable ABI. 如果 *derived* 类与 *cls* 类相同或为其派生类，则返回 1，否则返回 0。如果出错则返回 -1。

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

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

通常只有类对象才会被视为类，即 `type` 或派生类的实例。然而，对象可以通过拥有 `__bases__` 属性（必须是基类的元组）来覆盖这一点。

int **PyObject_IsInstance**(*PyObject* *inst, *PyObject* *cls)

Part of the Stable ABI. 如果 *inst* 是 *cls* 类或其子类的实例，则返回 1，如果不是则返回 "0"。如果出错则返回 -1 并设置一个异常。

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

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

实例 *inst* 可以通过 `__class__` 属性来覆盖其所属类。

对象 *cls* 是否被认作类，以及基类是什么，均可通过 `__bases__` 属性（必须是基类的元组）进行覆盖。

Py_hash_t **PyObject_Hash**(*PyObject* *o)

Part of the Stable ABI. 计算并返回对象的哈希值 *o*。失败时返回 -1。这相当于 Python 表达式 `hash(o)`。

3.2 版更變: 现在的返回类型是 Py_hash_t。这是一个大小与`Py_ssize_t` 相同的有符号整数。

Py_hash_t **PyObject_HashNotImplemented**(*PyObject* *o)
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. 如果对象 *o* 被认为是 true，则返回 `1`，否则返回 `0`。这相当于 Python 表达式 `not not o`。失败则返回 `-1`。

int **PyObject_Not**(*PyObject* *o)
Part of the Stable ABI. 如果对象 *o* 被认为是 true，则返回 `1`，否则返回 `0`。这相当于 Python 表达式 `not not o`。失败则返回 `-1`。

PyObject ***PyObject_Type**(*PyObject* *o)
Return value: New reference. Part of the Stable ABI. When *o* is non-`NULL`, returns a type object corresponding to the object type of object *o*. On failure, raises `SystemError` and returns `NULL`. This is equivalent to the Python expression `type(o)`. This function increments the reference count of 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 the incremented reference count is needed.

int **PyObject_TypeCheck**(*PyObject* *o, *PyTypeObject* *type)
如果对象 *o* 是 *type* 类型或其子类型，则返回非零，否则返回 `0`。两个参数都必须非 `NULL`。

Py_ssize_t **PyObject_Size**(*PyObject* *o)

Py_ssize_t **PyObject_Length**(*PyObject* *o)
Part of the Stable ABI. 返回对象 *o* 的长度。如果对象 *o* 支持序列和映射协议，则返回序列长度。出错时返回 `-1`。这等同于 Python 表达式 `len(o)`。

Py_ssize_t **PyObject_LengthHint**(*PyObject* *o, *Py_ssize_t* defaultvalue)
返回对象 *o* 的估计长度。首先尝试返回实际长度，然后用 `__length_hint__()` 进行估计，最后返回默认值。出错时返回 "-1"。这等同于 Python 表达式 `operator.length_hint(o, defaultvalue)`。

3.4 版新加入.

PyObject ***PyObject_GetItem**(*PyObject* *o, *PyObject* *key)
Return value: New reference. Part of the Stable ABI. 返回对象 *key* 对应的 *o* 元素，或在失败时返回 `NULL`。这等同于 Python 表达式 `o[key]`。

int **PyObject_SetItem**(*PyObject* *o, *PyObject* *key, *PyObject* *v)
Part of the Stable ABI. 将对象 *key* 映射到值 *v*。失败时引发异常并返回 `-1`；成功时返回 `0`。这相当于 Python 语句 `o[key] = v`。该函数 不会偷取 *v* 的引用计数。

int **PyObject_DelItem**(*PyObject* *o, *PyObject* *key)
Part of the Stable ABI. 从对象 *o* 中移除对象 *key* 的映射。失败时返回 `-1`。这相当于 Python 语句 `del o[key]`。

PyObject ***PyObject_Dir**(*PyObject* *o)
Return value: New reference. Part of the Stable ABI. 相当于 Python 表达式 `dir(o)`，返回一个（可能为空）适合对象参数的字符串列表，如果出错则返回 `NULL`。如果参数为 `NULL`，类似 Python 的 `dir()`，则返回当前 locals 的名字；这时如果没有活动的执行框架，则返回 `NULL`，但`PyErr_Occurred()` 将返回 false。

PyObject ***PyObject_GetIter**(*PyObject* *o)
Return value: New reference. Part of the Stable ABI. 等同于 Python 表达式 `iter(o)`。为对象参数返回一个新的迭代器，如果该对象已经是一个迭代器，则返回对象本身。如果对象不能被迭代，会引发 `TypeError`，并返回 `NULL`。

PyObject ***PyObject_GetAIter**(*PyObject* *o)
Return value: New reference. Part of the Stable ABI *since version 3.10.* 等同于 Python 表达式 `aiter(o)`。接

受一个 AsyncIterable 对象，并为其返回一个 AsyncIterator。通常返回的是一个新迭代器，但如果参数是一个 AsyncIterator，将返回其自身。如果该对象不能被迭代，会引发 TypeError，并返回 NULL。

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[F]之實例都是可呼叫的。該擴充槽 (slot) 的簽章[F]：

```
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[F] NULL（如果[F]有引數，會使用一個空 tuple），但如果[F]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

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[F]部呼叫中會更傾向使用 vectorcall。然而，這[F]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[F]也必須實作 tp_call。此外，無論使用哪種協定，可呼叫物件的行[F]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[F] PyVectorcall_Call()。這值得一再提醒：

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

如果一個類[F]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[F][F] args tuple（引數元組）和 kwargs dict（關鍵字引數字典），那[F]實作 vectorcall 就[F]有意義。

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

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

- *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[F]有引數，這個值可以是 *NULL*。
- ***nargsf* 是位置引數的數量加上可能會有的** PY_VECTORCALL_ARGUMENTS_OFFSET 旗標。如果要從 *nargsf* 獲得實際的位置引數數量，請使用 PyVectorcall_NARGS()。

- ***kwnames* 是一個包含所有關鍵字引數名稱的 tuple;** ⬚句話⬚，就是 kwargs 字典的鍵。這些名字必須是字串（`str` 或其子類⬚的實例），⬚且它們必須是不重⬚的。如果⬚有關鍵字引數，那⬚ *kwnames* 可以用 *NULL* 代替。

`PY_VECTORCALL_ARGUMENTS_OFFSET`

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

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

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

要呼叫一個實作了 vectorcall 的物件，請就像其他可呼叫物件一樣使用*呼叫 API* 中的函式。*`PyObject_Vectorcall()`* 通常是最有效率的。

備⬚: 在 CPython 3.8 中，vectorcall API 和相關函式暫定以帶開頭底⬚的名稱提供：`_PyObject_Vectorcall`、`_Py_TPFLAGS_HAVE_VECTORCALL`、`_PyObject_VectorcallMethod`、`_PyVectorcall_Function`、`_PyObject_CallOneArg`、`_PyObject_CallMethodNoArgs`、`_PyObject_CallMethodOneArg`。此外，`PyObject_VectorcallDict` 也以 `_PyObject_FastCallDict` 名稱提供。這些舊名稱仍有被定義，做⬚不帶底⬚的新名稱的⬚名。

遞⬚控制

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

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

Vectorcall 支援 API

Py_ssize_t **`PyVectorcall_NARGS`**(size_t *nargsf*)

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。

3.8 版新加入.

vectorcallfunc **`PyVectorcall_Function`**(*PyObject* *op*)

如果 *op* 不支援 vectorcall 協定（因⬚型⬚不支援或特定實例不支援），就回傳 *NULL*。否則，回傳儲存在 *op* 中的 vectorcall 函式指標。這個函式不會引發例外。

這大多在檢查 *op* 是否支援 vectorcall 時能派上用場，可以透過檢查 `PyVectorcall_Function(op) != NULL` 來達成。

3.8 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⬚以 tuple 和 dict 格式給定。

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

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*)

Return value: New reference. 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 必須不為 *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* 可以為 *NULL*。

成功時回傳結果，或在失敗時引發一個例外並回傳 *NULL*。

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

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

Part of the Stable ABI *since version 3.10.* 呼叫一個可呼叫的 Python 物件 *callable* 並不附帶任何引數。這是不帶引數呼叫 Python 可呼叫物件的最有效方式。

成功時回傳結果，或在失敗時引發一個例外並回傳 *NULL*。

3.9 版新加入.

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

呼叫一個可呼叫的 Python 物件 *callable* 並附帶正好一個位置引數 *arg* 而沒有關鍵字引數。

成功時回傳結果，或在失敗時引發一個例外並回傳 *NULL*。

3.9 版新加入.

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

Return value: New reference. 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* 可以為 *NULL*。

成功時回傳結果，或在失敗時引發一個例外並回傳 *NULL*。

這等價於 Python 運算式 `callable(*args)`。

PyObject ***PyObject_CallFunction**(*PyObject* **callable*, **const** char **format*, ...)
Return value: New reference. 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。格式可以為 *NULL*，表示沒有提供任何引數。

成功時回傳結果，或在失敗時引發一個例外並回傳 *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。

3.4 版更變: 這個 *format* 的型別已從 `char *` 更改。

PyObject ***PyObject_CallMethod**(*PyObject* **obj*, **const** char **name*, **const** char **format*, ...)
Return value: New reference. Part of the Stable ABI. 呼叫 *obj* 物件中名為 *name* 的 method 並附帶數量可變的 C 引數。這些 C 引數由`Py_BuildValue()` 格式字串來描述，並應當生成一個 tuple。

格式可以為 *NULL*，表示未有提供任何引數。

成功時回傳結果，或在失敗時引發一個例外並回傳 *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。

3.4 版更變: *name* 和 *format* 的型別已從 `char *` 更改。

PyObject ***PyObject_CallFunctionObjArgs**(*PyObject* **callable*, ...)
Return value: New reference. 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並回傳 *NULL*。

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

PyObject ***PyObject_CallMethodObjArgs**(*PyObject* **obj*, *PyObject* **name*, ...)
Return value: New reference. 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並回傳 *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並回傳 *NULL*。

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並回傳 *NULL*。

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並回傳 *NULL*。

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。

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

3.9 版新加入.

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

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

如果物件具有`Py_TPFLAGS_METHOD_DESCRIPTOR` 特性，這將以完整的 *args* 向量作為引數來呼叫 unbound method（未綁定方法）物件。

成功時回傳結果，或在失敗時引發一個例外並回傳 *NULL*。

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 数字协议

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

Part of the Stable ABI. 如果对象 *o* 提供数字的协议，返回真 `1`，否则返回假。这个函数不会调用失败。

3.8 版更變: 如果 *o* 是一个索引整数则返回 `1`。

PyObject ***PyNumber_Add**(*PyObject* **o1*, *PyObject* **o2*)

Return value: New reference. Part of the Stable ABI. 返回 *o1* 、*o2* 相加的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 + o2`。

PyObject ***PyNumber_Subtract**(*PyObject* **o1*, *PyObject* **o2*)

Return value: New reference. Part of the Stable ABI. 返回 *o1* 减去 *o2* 的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 - o2`。

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

Return value: New reference. Part of the Stable ABI. 返回 *o1* 、*o2* 相乘的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 * o2`。

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

Return value: New reference. Part of the Stable ABI *since version 3.7.* 返回 *o1* 、*o2* 做矩阵乘法的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 @ o2`。

3.5 版新加入.

PyObject ***PyNumber_FloorDivide**(*PyObject* **o1*, *PyObject* **o2*)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 除以 *o2* 向下取整的值，失败时返回 `NULL`。这等价于 Python 表达式 `o1 // o2`。

PyObject ***PyNumber_TrueDivide**(*PyObject* **o1*, *PyObject* **o2*)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 除以 *o2* 的数学值的合理近似值，或失败时返回 `NULL`。返回的是" 近似值" 因为二进制浮点数本身就是近似值；不可能以二进制精确表示所有实数。此函数可以在传入两个整数时返回一个浮点值。此函数等价于 Python 表达式 `o1 / o2`。

PyObject ***PyNumber_Remainder**(*PyObject* **o1*, *PyObject* **o2*)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 除以 *o2* 得到的余数，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 % o2`。

PyObject ***PyNumber_Divmod**(*PyObject* **o1*, *PyObject* **o2*)
Return value: New reference. Part of the Stable ABI. 参考内置函数 `divmod()`。如果失败，返回 `NULL`。等价于 Python 表达式 `divmod(o1, o2)`。

PyObject ***PyNumber_Power**(*PyObject* **o1*, *PyObject* **o2*, *PyObject* **o3*)
Return value: New reference. Part of the Stable ABI. 请参阅内置函数 `pow()`。如果失败，返回 `NULL`。等价于 Python 中的表达式 `pow(o1, o2, o3)`，其中 *o3* 是可选的。如果要忽略 *o3*，则需传入 `Py_None` 作为代替（如果传入 `NULL` 会导致非法内存访问）。

PyObject ***PyNumber_Negative**(*PyObject* **o*)
Return value: New reference. Part of the Stable ABI. 返回 *o* 的负值，如果失败，返回 `NULL` 。等价于 Python 表达式 `-o`。

PyObject ***PyNumber_Positive**(*PyObject* **o*)
Return value: New reference. Part of the Stable ABI. 返回 *o*，如果失败，返回 `NULL` 。等价于 Python 表达式 `+o`。

PyObject ***PyNumber_Absolute**(*PyObject* **o*)
Return value: New reference. Part of the Stable ABI. 返回 *o* 的绝对值，如果失败，返回 `NULL`。等价于 Python 表达式 `abs(o)`。

PyObject ***PyNumber_Invert**(*PyObject* **o*)
Return value: New reference. Part of the Stable ABI. 返回 *o* 的按位取反后的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `~o`。

PyObject ***PyNumber_Lshift**(*PyObject* **o1*, *PyObject* **o2*)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 左移 *o2* 个比特后的结果，如果失败，返回 `NULL`。等价于 Python 表达式 `o1 << o2`。

PyObject ***PyNumber_Rshift**(*PyObject* **o1*, *PyObject* **o2*)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 右移 *o2* 个比特后的结果，如果失败，返回 `NULL` 。等价于 Python 表达式 `o1 >> o2`。

PyObject ***PyNumber_And**(*PyObject* **o1*, *PyObject* **o2*)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 和 *o2* "按位与" 的结果，如果失败，返回 `NULL` 。等价于 Python 表达式 `o1 & o2`。

PyObject ***PyNumber_Xor**(*PyObject* **o1*, *PyObject* **o2*)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 和 *o2* "按位异或" 的结果，如果失败，返回 `NULL` 。等价于 Python 表达式 `o1 ^ o2`。

PyObject ***PyNumber_Or**(*PyObject* **o1*, *PyObject* **o2*)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 和 *o2* "按位或" 的结果，如果失败，返回 `NULL` 。等价于 Python 表达式 `o1 | o2`。

PyObject ***PyNumber_InPlaceAdd**(*PyObject* **o1*, *PyObject* **o2*)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 、*o2* 相加的结果，如果失败，返回 `NULL`。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 += o2`。

PyObject ***PyNumber_InPlaceSubtract**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 、*o2* 相减的结果，如果失败，返回 NULL 。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 -= o2`。

PyObject ***PyNumber_InPlaceMultiply**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 、*o2** 相乘的结果，如果失败，返回 *"NULL"*。当 **o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 *= o2`。

PyObject ***PyNumber_InPlaceMatrixMultiply**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Part of the Stable ABI *since version 3.7.* 返回 *o1* 、*o2* 做矩阵乘法后的结果，如果失败，返回 NULL 。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 @= o2`。

3.5 版新加入.

PyObject ***PyNumber_InPlaceFloorDivide**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 除以 *o2* 后向下取整的结果，如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 //= o2`。

PyObject ***PyNumber_InPlaceTrueDivide**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 除以 *o2* 的数学值的合理近似值，或失败时返回 NULL。返回的是" 近似值" 因为二进制浮点数本身就是近似值；不可能以二进制精确表示所有实数。此函数可以在传入两个整数时返回一个浮点数。此运算在 *o1* 支持的时候会 原地执行。此函数等价于 Python 语句 `o1 /= o2`。

PyObject ***PyNumber_InPlaceRemainder**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 除以 *o2* 得到的余数，如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 %= o2`。

PyObject ***PyNumber_InPlacePower**(*PyObject* *o1, *PyObject* *o2, *PyObject* *o3)
Return value: New reference. Part of the Stable ABI. 请参阅内置函数 `pow()`。如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。当 *o3* 是 `Py_None` 时，等价于 Python 语句 `o1 **= o2`；否则等价于在原来位置储存结果的 `pow(o1, o2, o3)` 。如果要忽略 *o3*，则需传入 `Py_None` （传入 NULL 会导致非法内存访问)。

PyObject ***PyNumber_InPlaceLshift**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 左移 *o2* 个比特后的结果，如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 <<= o2`。

PyObject ***PyNumber_InPlaceRshift**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Part of the Stable ABI. 返回 *o1* 右移 *o2* 个比特后的结果，如果失败，返回 NULL。当 *o1* 支持时，这个运算直接使用它储存结果。等价于 Python 语句 `o1 >>= o2`。

PyObject ***PyNumber_InPlaceAnd**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Part of the Stable ABI. 成功时返回 *o1* 和 *o2* " 按位与" 的结果，失败时返回 NULL。在 *o1* 支持的前提下该操作将 原地执行。等价与 Python 语句 `o1 &= o2`。

PyObject ***PyNumber_InPlaceXor**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Part of the Stable ABI. 成功时返回 *o1* 和 *o2* " 按位异或的结果，失败时返回 NULL。在 *o1* 支持的前提下该操作将 原地执行。等价与 Python 语句 `o1 ^= o2`。

PyObject ***PyNumber_InPlaceOr**(*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Part of the Stable ABI. 成功时返回 *o1* 和 *o2* " 按位或" 的结果，失败时返回 NULL。在 *o1* 支持的前提下该操作将 原地执行。等价于 Python 语句 `o1 |= o2`。

PyObject ***PyNumber_Long**(*PyObject* *o)
Return value: New reference. Part of the Stable ABI. 成功时返回 *o* 转换为整数对象后的结果，失败时返回 NULL。等价于 Python 表达式 `int(o)`。

PyObject ***PyNumber_Float**(*PyObject* *o)
Return value: New reference. Part of the Stable ABI. 成功时返回 *o* 转换为浮点对象后的结果，失败时返回 NULL。等价于 Python 表达式 `float(o)`。

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

Return value: New reference. Part of the Stable ABI. 成功时返回 *o* 转换为 Python int 类型后的结果，失败时返回 NULL 并引发 TypeError 异常。

3.10 版更變: 结果总是为 int 类型。在之前版本中，结果可能为 int 的子类的实例。

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

Return value: New reference. Part of the Stable ABI. 返回整数 *n* 转换成以 *base* 为基数的字符串后的结果。这个 *base* 参数必须是 2，8，10 或者 16 。对于基数 2，8，或 16 ，返回的字符串将分别加上基数标识 '0b', '0o', or '0x'。如果 *n* 不是 Python 中的整数 *int* 类型，就先通过 PyNumber_Index() 将它转换成整数类型。

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

Part of the Stable ABI. 如果 *o* 可以被解读为一个整数则返回 *o* 转换成的 Py_ssize_t 值。如果调用失败，则会引发一个异常并返回 -1。

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

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

Part of the Stable ABI *since version 3.8.* 返回 1 如果 *o* 是一个索引整数（将 nb_index 槽位填充到 tp_as_number 结构体)，或者在其他情况下返回 0。此函数总是会成功执行。

7.4 序列协议

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

Part of the Stable ABI. 如果对象提供了序列协议则返回 1，否则返回 0。请注意它将为具有 __getitem__() 方法的 Python 类返回 1，除非它们是 dict 的子类，因为在通常情况下无法确定这种类支持哪种键类型。此函数总是会成功执行。

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

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

Part of the Stable ABI. 成功时返回序列中 *o* 的对象数, 失败时返回 "-1". 相当于 Python 的 "len(o)"表达式.

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

Return value: New reference. Part of the Stable ABI. 成功时返回 *o1* 和 *o2* 的拼接，失败时返回 NULL。这等价于 Python 表达式 o1 + o2。

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

Return value: New reference. Part of the Stable ABI. 返回序列对象 *o* 重复 *count* 次的结果，失败时返回 NULL。这等价于 Python 表达式 o * count。

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

Return value: New reference. Part of the Stable ABI. 成功时返回 *o1* 和 *o2* 的拼接，失败时返回 NULL。在 *o1* 支持的情况下操作将 原地完成。这等价于 Python 表达式 o1 += o2。

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

Return value: New reference. Part of the Stable ABI. Return the result of repeating sequence object 返回序列对象 *o* 重复 *count* 次的结果，失败时返回 NULL。在 *o* 支持的情况下该操作会 原地完成。这等价于 Python 表达式 o *= count。

PyObject ***PySequence_GetItem***(*PyObject* **o*, *Py_ssize_t* *i*)

Return value: New reference. Part of the Stable ABI. 返回 *o* 中的第 *i* 号元素，失败时返回 NULL。这等价于 Python 表达式 o[i]。

PyObject ***PySequence_GetSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

Return value: New reference. Part of the Stable ABI. 返回序列对象 *o* 的 *i1* 到 *i2* 的切片，失败时返回 NULL。这等价于 Python 表达式 `o[i1:i2]`。

int **PySequence_SetItem**(*PyObject* *o, *Py_ssize_t* i, *PyObject* *v)

Part of the Stable ABI. 将对象 *v* 赋值给 *o* 的第 *i* 号元素。失败时会引发异常并返回 -1；成功时返回 0。这相当于 Python 语句 `o[i] = v`。此函数 不会改变对 *v* 的引用。

如果 *v* 为 NULL，元素将被删除，但是此特性已被弃用而应改用`PySequence_DelItem()`。

int **PySequence_DelItem**(*PyObject* *o, *Py_ssize_t* i)

Part of the Stable ABI. 删除对象 *o* 的第 *i* 号元素。失败时返回 -1。这相当于 Python 语句 `del o[i]`。

int **PySequence_SetSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2, *PyObject* *v)

Part of the Stable ABI. 将序列对象 *v* 赋值给序列对象 *o* 的从 *i1* 到 *i2* 切片。这相当于 Python 语句 `o[i1:i2] = v`。

int **PySequence_DelSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

Part of the Stable ABI. 删除序列对象 *o* 的从 *i1* 到 *i2* 的切片。失败时返回 -1。这相当于 Python 语句 `del o[i1:i2]`。

Py_ssize_t **PySequence_Count**(*PyObject* *o, *PyObject* *value)

Part of the Stable ABI. 返回 *value* 在 *o* 中出现的次数，即返回使得 `o[key] == value` 的键的数量。失败时返回 -1。这相当于 Python 表达式 `o.count(value)`。

int **PySequence_Contains**(*PyObject* *o, *PyObject* *value)

Part of the Stable ABI. 确定 *o* 是否包含 *value*。如果 *o* 中的某一项等于 *value*，则返回 1，否则返回 0。出错时，返回 -1。这相当于 Python 表达式 `value in o`。

Py_ssize_t **PySequence_Index**(*PyObject* *o, *PyObject* *value)

Part of the Stable ABI. 返回第一个索引 *i*, 其中 `o[i] == value`. 出错时, 返回 -1. 相当于 Python 的 "o.index(value)"表达式.

PyObject ***PySequence_List**(*PyObject* *o)

Return value: New reference. Part of the Stable ABI. 返回一个列表对象，其内容与序列或可迭代对象 *o* 相同，失败时返回 NULL。返回的列表保证是一个新对象。这等价于 Python 表达式 `list(o)`。

PyObject ***PySequence_Tuple**(*PyObject* *o)

Return value: New reference. Part of the Stable ABI. 返回一个元组对象，其内容与序列或可迭代对象 *o* 相同，失败时返回 NULL。如果 *o* 为元组，则将返回一个新的引用，在其他情况下将使用适当的内容构造一个元组。这等价于 Python 表达式 `tuple(o)`。

PyObject ***PySequence_Fast**(*PyObject* *o, **const** char *m)

Return value: New reference. Part of the Stable ABI. 将序列或可迭代对象 *o* 作为其他 `PySequence_Fast*` 函数族可用的对象返回。如果该对象不是序列或可迭代对象，则会引发 `TypeError` 并将 *m* 作为消息文本。失败时返回 NULL。

`PySequence_Fast*` 函数之所以这样命名，是因为它们会假定 *o* 是一个`PyTupleObject` 或`PyListObject` 并直接访问 *o* 的数据字段。

作为 CPython 的实现细节，如果 *o* 已经是一个序列或列表，它将被直接返回。

Py_ssize_t **PySequence_Fast_GET_SIZE**(*PyObject* *o)

在 *o* 由`PySequence_Fast()` 返回且 *o* 不为 NULL 的情况下返回 *o* 长度。也可以通过在 *o* 上调用`PySequence_Size()` 来获取大小，但是`PySequence_Fast_GET_SIZE()` 的速度更快因为它可以假定 *o* 为列表或元组。

PyObject ***PySequence_Fast_GET_ITEM**(*PyObject* *o, *Py_ssize_t* i)

Return value: Borrowed reference. 在 *o* 由`PySequence_Fast()` 返回且 *o* 不 NULL，并且 *i* d 在索引范围内的情况下返回 *o* 的第 *i* 号元素。

PyObject ****PySequence_Fast_ITEMS**(*PyObject* **o*)
返回 PyObject 指针的底层数组。假设 *o* 由`PySequence_Fast()` 返回且 *o* 不为 `NULL`。

请注意, 如果列表调整大小, 重新分配可能会重新定位 items 数组. 因此, 仅在序列无法更改的上下文中使用基础数组指针.

PyObject ***PySequence_ITEM**(*PyObject* **o*, *Py_ssize_t* *i*)
Return value: New reference. 返回 *o* 的第 *i* 个元素或在失败时返回 `NULL`。此形式比`PySequence_GetItem()` 理馔，但不会检查 *o* 上的`PySequence_Check()` 是否为真值，也不会对负序号进行调整。

7.5 映射协议

参见`PyObject_GetItem()`、`PyObject_SetItem()` 与`PyObject_DelItem()`。

int **PyMapping_Check**(*PyObject* **o*)
Part of the Stable ABI. 如果对象提供了映射协议或是支持切片则返回 `1`，否则返回 `0`。请注意它将为具有 `__getitem__()` 方法的 Python 类返回 `1`，因为在通常情况下无法确定该类所支持的键类型。此函数总是会成功执行。

Py_ssize_t **PyMapping_Size**(*PyObject* **o*)

Py_ssize_t **PyMapping_Length**(*PyObject* **o*)
Part of the Stable ABI. 成功时返回对象 *o* 中键的数量，失败时返回 `-1`。这相当于 Python 表达式 `len(o)`。

PyObject ***PyMapping_GetItemString**(*PyObject* **o*, **const** char **key*)
Return value: New reference. Part of the Stable ABI. 返回 *o* 中对应于字符串 *key* 的元素，或者失败时返回 `NULL`。这相当于 Python 表达式 `o[key]`。另请参见 also `PyObject_GetItem()`。

int **PyMapping_SetItemString**(*PyObject* **o*, **const** char **key*, *PyObject* **v*)
Part of the Stable ABI. 在对象 *o* 中将字符串 *key* 映射到值 *v*。失败时返回 `-1`。这相当于 Python 语句 `o[key] = v`。另请参见`PyObject_SetItem()`。此函数 不会增加对 *v* 的引用。

int **PyMapping_DelItem**(*PyObject* **o*, *PyObject* **key*)
从对象 *o* 中移除对象 *key* 的映射。失败时返回 `-1`。这相当于 Python 语句 `del o[key]`。这是`PyObject_DelItem()` 的一个别名。

int **PyMapping_DelItemString**(*PyObject* **o*, **const** char **key*)
从对象 *o* 中移除字符串 *key* 的映射。失败时返回 `-1`。这相当于 Python 语句 `del o[key]`。

int **PyMapping_HasKey**(*PyObject* **o*, *PyObject* **key*)
Part of the Stable ABI. 如果映射对象具有键 *key* 则返回 `1`，否则返回 `0`。这相当于 Python 表达式 `key in o`。此函数总是会成功执行。

请注意在调用 `__getitem__()` 方法期间发生的异常将会被屏蔽。要获取错误报告请改用`PyObject_GetItem()`。

int **PyMapping_HasKeyString**(*PyObject* **o*, **const** char **key*)
Part of the Stable ABI. 如果映射对象具有键 *key* 则返回 `1`，否则返回 `0`。这相当于 Python 表达式 `key in o`。此函数总是会成功执行。

请注意在调用 `__getitem__()` 方法期间发生的异常将会被屏蔽。要获取错误报告请改用`PyMapping_GetItemString()`。

PyObject ***PyMapping_Keys**(*PyObject* **o*)
Return value: New reference. Part of the Stable ABI. 成功时，返回对象 *o* 中的键的列表。失败时，返回 `NULL`。

3.7 版更變: 在之前版本中，此函数返回一个列表或元组。

PyObject ***PyMapping_Values**(*PyObject* **o*)
Return value: New reference. Part of the Stable ABI. 成功时，返回对象 *o* 中的值的列表。失败时，返回 NULL。

3.7 版更變: 在之前版本中，此函数返回一个列表或元组。

PyObject ***PyMapping_Items**(*PyObject* **o*)
Return value: New reference. Part of the Stable ABI. 成功时，返回对象 *o* 中条目的列表，其中每个条目是一个包含键值对的元组。失败时，返回 NULL。

3.7 版更變: 在之前版本中，此函数返回一个列表或元组。

7.6 迭代器协议

迭代器有两个函数。

int **PyIter_Check**(*PyObject* **o*)
Part of the Stable ABI *since version 3.8.* 如果对象 Return non-zero if the object *o* 可以被安全把传给*PyIter_Next()* 则返回非零值，否则返回 0。此函数总是会成功执行。

int **PyAIter_Check**(*PyObject* **o*)
Part of the Stable ABI *since version 3.10.* 如果对象 *o* 提供了 AsyncIterator 协议则返回非零值，否则返回 0。此函数总是会成功执行。

3.10 版新加入.

PyObject ***PyIter_Next**(*PyObject* **o*)
Return value: New reference. Part of the Stable ABI. 从迭代器 *o* 返回下一个值。对象必须可被*PyIter_Check()* 确认为迭代器（需要调用方来负责检查）。如果没有剩余的值，则返回 NULL 并且不设置异常。如果在获取条目时发生了错误，则返回 NULL 并且传递异常。

要为迭代器编写一个一个循环，C 代码应该看起来像这样

```
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;

if (iterator == NULL) {
    /* propagate error */
}

while ((item = PyIter_Next(iterator))) {
    /* do something with item */
    ...
    /* release reference when done */
    Py_DECREF(item);
}

Py_DECREF(iterator);

if (PyErr_Occurred()) {
    /* propagate error */
}
else {
    /* continue doing useful work */
}
```

type PySendResult
用于代表*PyIter_Send()* 的不同结果的枚举值。

3.10 版新加入.

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

Part of the Stable ABI *since version 3.10.* 将 *arg* 值发送到迭代器 *iter*。返回:

- PYGEN_RETURN，如果迭代器返回的话。返回值会通过 *presult* 来返回。
- PYGEN_NEXT，如果迭代器生成值的话。生成的值会通过 *presult* 来返回。
- PYGEN_ERROR，如果迭代器引发异常的话。*presult* 会被设为 NULL。

3.10 版新加入.

7.7 緩沖協定 (Buffer Protocol)

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

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

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

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

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

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

对于缓冲区接口的使用者而言，有两种方式来获取一个目的对象的缓冲：

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

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

7.7.1 缓冲区结构

缓冲区结构 (或者简单地称为 "buffers") 对于将二进制数据从另一个对象公开给 Python 程序员非常有用。它们还可以用作零拷贝切片机制。使用它们引用内存块的能力，可以很容易地将任何数据公开给 Python 程序员。内存可以是 C 扩展中的一个大的常量数组，也可以是在传递到操作系统库之前用于操作的原始内存块，或者可以用来传递本机内存格式的结构化数据。

与 Python 解释器公开的大多部数据类型不同，缓冲区不是*PyObject* 指针而是简单的 C 结构。这使得它们可以非常简单地创建和复制。当需要为缓冲区加上泛型包装器时，可以创建一个内存视图 对象。

有关如何编写并导出对象的简短说明，请参阅缓冲区对象结构。要获取缓冲区对象，请参阅*PyObject_GetBuffer()*。

type Py_buffer

void *__buf__
指向由缓冲区字段描述的逻辑结构开始的指针。这可以是导出程序底层物理内存块中的任何位置。例如，使用负的 *strides* 值可能指向内存块的末尾。

对于 *contiguous*，'邻接'数组，值指向内存块的开头。

PyObject *__obj__
对导出对象的新引用。该引用归使用者所有，并由 *PyBuffer_Release()* 自动递减并设置为 NULL。该字段等于任何标准 C-API 函数的返回值。

作为一种特殊情况，对于由 *PyMemoryView_FromBuffer()* 或 *PyBuffer_FillInfo()* 包装的 *temporary* 缓冲区，此字段为 NULL。通常，导出对象不得使用此方案。

Py_ssize_t __len__
product(shape) * itemsize。对于连续数组，这是基础内存块的长度。对于非连续数组，如果逻辑结构复制到连续表示形式，则该长度将具有该长度。

仅当缓冲区是通过保证连续性的请求获取时，才访问 ((char *)buf)[0] up to ((char *)buf)[len-1] 时才有效。在大多数情况下，此类请求将为 *PyBUF_SIMPLE* 或 *PyBUF_WRITABLE*。

int __readonly__
缓冲区是否为只读的指示器。此字段由 *PyBUF_WRITABLE* 标志控制。

Py_ssize_t __itemsize__
单个元素的项大小（以字节为单位）。与 struct.calcsize() 调用非 NULL *format* 的值相同。

重要例外：如果使用者请求的缓冲区没有 *PyBUF_FORMAT* 标志，*format* 将设置为 NULL，但 *itemsize* 仍具有原始格式的值。

如果 *shape* 存在，则相等的 product(shape) * itemsize == len 仍然存在，使用者可以使用 *itemsize* 来导航缓冲区。

如果 *shape* 是 NULL，因为结果为 *PyBUF_SIMPLE* 或 *PyBUF_WRITABLE* 请求，则使用者必须忽略 *itemsize*，并假设 itemsize == 1。

__const__ char *__format__
在 struct 模块样式语法中 *NUL* 字符串，描述单个项的内容。如果这是 NULL，则假定为 "”B”"（无符号字节）。

此字段由 *PyBUF_FORMAT* 标志控制。

int __ndim__
内存表示为 n 维数组的维数。如果是 "0"，*buf* 指向表示标量的单个项目。在这种情况下，*shape*、*strides* 和 *suboffsets* 必须是 "NULL"。

宏 PyBUF_MAX_NDIM 将最大维度数限制为 64。导出程序必须遵守这个限制，多维缓冲区的使用者应该能够处理最多 PyBUF_MAX_NDIM 维度。

Py_ssize_t *__shape__
一个长度为 *Py_ssize_t* 的数组 *ndim* 表示作为 n 维数组的内存形状。请注意，shape[0] * ... * shape[ndim-1] * itemsize 必须等于 *len*。

Shape 形状数组中的值被限定在 shape[n] >= 0。shape[n] == 0 这一情形需要特别注意。更多信息请参阅 *complex arrays*。

shape 数组对于使用者来说是只读的。

Py_ssize_t *__strides__
一个长度为 *Py_ssize_t* 的数组 *ndim* 给出要跳过的字节数以获取每个尺寸中的新元素。

Stride 步幅数组中的值可以为任何整数。对于常规数组，步幅通常为正数，但是使用者必须能够处理 strides[n] <= 0 的情况。更多信息请参阅 *complex arrays*。

strides 数组对用户来说是只读的。

Py_ssize_t ***suboffsets**
一个长度为*ndim* 类型为*Py_ssize_t* 的数组。如果 suboffsets[n] >= 0，则第 n 维存储的是指针，suboffset 值决定了解除引用时要给指针增加多少字节的偏移。suboffset 为负值，则表示不应解除引用（在连续内存块中移动）。

如果所有子偏移均为负（即无需取消引用），则此字段必须为 NULL （默认值）。

Python Imaging Library (PIL) 中使用了这种类型的数组表达方式。请参阅*complex arrays* 来了解如何从这样一个数组中访问元素。

suboffsets 数组对于使用者来说是只读的。

void ***internal**
供输出对象内部使用。比如可能被输出程序重组为一个整数，用于存储一个标志，标明在缓冲区释放时是否必须释放 shape、strides 和 suboffsets 数组。消费者程序 不得修改该值。

7.7.2 缓冲区请求的类型

通常，通过*PyObject_GetBuffer()* 向输出对象发送缓冲区请求，即可获得缓冲区。由于内存的逻辑结构复杂，可能会有很大差异，缓冲区使用者可用 *flags* 参数指定其能够处理的缓冲区具体类型。

所有*Py_buffer* 字段均由请求类型明确定义。

与请求无关的字段

以下字段不会被 *flags* 影响，并且必须总是用正确的值填充：*obj*, *buf*，*len*，*itemsize*，*ndim*。

只读，格式

PyBUF_WRITABLE
控制*readonly* 字段。如果设置了，输出程序 必须提供一个可写的缓冲区，否则报告失败。若未设置，输出程序 可以提供只读或可写的缓冲区，但对所有消费者程序 必须保持一致。

PyBUF_FORMAT
控制*format* 字段。如果设置，则必须正确填写此字段。其他情况下，此字段必须为 "NULL"。

PyBUF_WRITABLE 可以和下一节的所有标志联用。由于*PyBUF_SIMPLE* 定义为 0，所以*PyBUF_WRITABLE* 可以作为一个独立的标志，用于请求一个简单的可写缓冲区。

PyBUF_FORMAT 可以被设为除了*PyBUF_SIMPLE* 之外的任何标志。后者已经按暗示了 "B"(无符号字节串) 格式。

形状，步幅，子偏移量

控制内存逻辑结构的标志按照复杂度的递减顺序列出。注意，每个标志包含它下面的所有标志。

请求	形状	步幅	子偏移量
`PyBUF_INDIRECT`	是	是	如果需要的话
`PyBUF_STRIDES`	是	是	NULL
`PyBUF_ND`	是	NULL	NULL
`PyBUF_SIMPLE`	NULL	NULL	NULL

连续性的请求

可以显式地请求 C 或 Fortran 连续，不管有没有步幅信息。若没有步幅信息，则缓冲区必须是 C-连续的。

请求	形状	步幅	子偏移量	邻接
`PyBUF_C_CONTIGUOUS`	是	是	NULL	C
`PyBUF_F_CONTIGUOUS`	是	是	NULL	F
`PyBUF_ANY_CONTIGUOUS`	是	是	NULL	C 或 F
PyBUF_ND	是	NULL	NULL	C

复合请求

所有可能的请求都由上一节中某些标志的组合完全定义。为方便起见，缓冲区协议提供常用的组合作为单个标志。

在下表中，*U* 代表连续性未定义。消费者程序必须调用`PyBuffer_IsContiguous()` 以确定连续性。

请求	形状	步幅	子偏移量	邻接	只读	format
PyBUF_FULL	是	是	如果需要的话	U	0	是
PyBUF_FULL_RO	是	是	如果需要的话	U	1 或 0	是
PyBUF_RECORDS	是	是	NULL	U	0	是
PyBUF_RECORDS_RO	是	是	NULL	U	1 或 0	是
PyBUF_STRIDED	是	是	NULL	U	0	NULL
PyBUF_STRIDED_RO	是	是	NULL	U	1 或 0	NULL
PyBUF_CONTIG	是	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	是	NULL	NULL	C	1 或 0	NULL

7.7.3 复杂数组

NumPy-风格：形状和步幅

NumPy 风格数组的逻辑结构由 *itemsize* 、*ndim* 、*shape* 和 *strides* 定义。

如果 `ndim == 0` ，*buf* 指向的内存位置被解释为大小为 *itemsize* 的标量。这时，*shape* 和 *strides* 都为 `NULL`。

如果 *strides* 为 `NULL`，则数组将被解释为一个标准的 n 维 C 语言数组。否则，消费者程序必须按如下方式访问 n 维数组：

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

如上所述，*buf* 可以指向实际内存块中的任意位置。输出者程序可以用该函数检查缓冲区的有效性。

```
def verify_structure(memlen, itemsize, ndim, shape, strides, offset):
    """Verify that the parameters represent a valid array within
       the bounds of the allocated memory:
           char *mem: start of the physical memory block
           memlen: length of the physical memory block
           offset: (char *)buf - mem
    """
    if offset % itemsize:
        return False
    if offset < 0 or offset+itemsize > memlen:
        return False
    if any(v % itemsize for v in strides):
        return False
```

(下页继续)

(繼續上一頁)

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

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

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

PIL-风格：形状，步幅和子偏移量

除了常规项之外，PIL 风格的数组还可以包含指针，必须跟随这些指针才能到达维度的下一个元素。例如，常规的三维 C 语言数组 `char v[2][2][3]` 可以看作是一个指向 2 个二维数组的 2 个指针：`char (*v[2])[2][3]`。在子偏移表示中，这两个指针可以嵌入在*buf* 的开头，指向两个可以位于内存任何位置的 `char x[2][3]` 数组。

这是一个函数，当 n 维索引所指向的 N-D 数组中有 "NULL"步长和子偏移量时，它返回一个指针

```
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
                       Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >=0 ) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

7.7.4 缓冲区相关函数

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

如果 *obj* 支持缓冲区接口，则返回 `1`，否则返回 `0`。返回 `1` 时不保证*PyObject_GetBuffer()* 一定成功。本函数一定调用成功。

int **PyObject_GetBuffer** (*PyObject* *exporter, *Py_buffer* *view, int *flags*)

向输出器程序发送请求，按照 *flags* 指定的内容填充 *view*。如果输出器程序不能提供准确类型的缓冲区，必须触发 `PyExc_BufferError`，设置 `view->obj` 为 `NULL` 并返回 `-1`。

成功时，填充 *view*，将 `view->obj` 设为对 *exporter* 的新引用，并返回 0。当链式缓冲区提供程序将请求重定向到一个对象时，`view->obj` 可以引用该对象而不是 *exporter* (参见缓冲区对象结构)。

PyObject_GetBuffer() 必须与*PyBuffer_Release()* 同时调用成功，类似于 `malloc()` 和 `free()`。因此，消费者程序用完缓冲区后，*PyBuffer_Release()* 必须保证被调用一次。

void **PyBuffer_Release** (*Py_buffer* *view)

释放缓冲区 *view* 并递减 `view->obj` 的引用计数。该函数必须在缓冲区不再使用时才能调用，否则可能会发生引用泄漏。

若该函数针对的缓冲区不是通过*PyObject_GetBuffer()* 获得的，将会出错。

Py_ssize_t **PyBuffer_SizeFromFormat** (**const** char **format*)
返回 `itemsize` 中隐含的 `format` 。如果出错，会触发异常并返回 -1。

3.9 版新加入.

int **PyBuffer_IsContiguous** (*Py_buffer* **view*, char *order*)
如果 *view* 定义的内存是 C 风格（*order* 为 `'C'`）或 Fortran 风格（*order* 为 `'F'`）*contiguous* 或其中之一（*order* 是 `'A'`），则返回 `1`。否则返回 `0`。该函数总会成功。

void ***PyBuffer_GetPointer** (*Py_buffer* **view*, *Py_ssize_t* **indices*)
获取给定 *view* 内的 *indices* 所指向的内存区域。*indices* 必须指向一个 `view->ndim` 索引的数组。

int **PyBuffer_FromContiguous** (*Py_buffer* **view*, void **buf*, *Py_ssize_t* *len*, char *fort*)
从 *buf* 复制连续的 *len* 字节到 *view* 。*fort* 可以是 `'C'` 或 `'F'`` (对应于 C 风格或 Fortran 风格的顺序)。成功时返回 ``0`，错误时返回 `-1`。

int **PyBuffer_ToContiguous** (void **buf*, *Py_buffer* **src*, *Py_ssize_t* *len*, char *order*)
从 *src* 复制 *len* 字节到 *buf* ，成为连续字节串的形式。*order* 可以是 `'C'` 或 `'F'` 或 `'A'`` (对应于 C 风格、Fortran 风格的顺序或其中任意一种)。成功时返回 ``0`，出错时返回 `-1`。

如果 *len* != *src->len* 则此函数将报错。

void **PyBuffer_FillContiguousStrides** (int *ndims*, *Py_ssize_t* **shape*, *Py_ssize_t* **strides*, int *itemsize*, char *order*)
用给定形状的*contiguous* 字节串数组 (如果 *order* 为 `'C'` 则为 C 风格，如果 *order* 为 `'F'` 则为 Fortran 风格) 来填充 *strides* 数组，每个元素具有给定的字节数。

int **PyBuffer_FillInfo** (*Py_buffer* **view*, *PyObject* **exporter*, void **buf*, *Py_ssize_t* *len*, int *readonly*, int *flags*)
处理导出程序的缓冲区请求，该导出程序要公开大小为 *len* 的 *buf* ，并根据 *readonly* 设置可写性。*bug* 被解释为一个无符号字节序列。

参数 *flags* 表示请求的类型。该函数总是按照 flag 指定的内容填入 *view*，除非 *buf* 设为只读，并且 *flag* 中设置了 `PyBUF_WRITABLE` 标志。

成功时，将 `view->obj` 设为 *exporter* 的新引用，并返回 0。否则，引发 `PyExc_BufferError` ，将 `view->obj` 设为 `NULL`，并返回 `-1`。

如果此函数用作*getbufferproc* 的一部分，则 *exporter* 必须设置为导出对象，并且必须在未修改的情况下传递 *flags*。否则，*exporter* 必须是 `NULL`。

7.8 舊式緩冲協定 (Buffer Protocol)

3.0 版後已棄用.

这些函数是 Python 2 中“旧缓冲协议”API 的组成部分。在 Python 3 中，此协议已不复存在，但这些函数仍然被公开以便移植 2.x 的代码。它们被用作新缓冲协议 的兼容性包装器，但它们并不会在缓冲被导出时向你提供对所获资源的生命周期控制。

因此，推荐你调用`PyObject_GetBuffer()` (或者配合`PyArg_ParseTuple()` 函数族使用 `y*` 或 `w*` 格式码) 来获取一个对象的缓冲视图，并在缓冲视图可被释放时调用`PyBuffer_Release()`。

int **PyObject_AsCharBuffer** (*PyObject* **obj*, **const** char ***buffer*, *Py_ssize_t* **buffer_len*)
Part of the Stable ABI. 返回一个指向可用作基于字符的输入的只读内存地址的指针。*obj* 参数必须支持单段字符缓冲接口。成功时返回 `0`，将 *buffer* 设为内存地址并将 *buffer_len* 设为缓冲区长度。出错时返回 `-1` 并设置一个 `TypeError`。

int **PyObject_AsReadBuffer** (*PyObject* **obj*, **const** void ***buffer*, *Py_ssize_t* **buffer_len*)
Part of the Stable ABI. 返回一个指向包含任意数据的只读内存地址的指针。*obj* 参数必须支持单段可读缓

冲接口。成功时返回 `0`，将 *buffer* 设为内存地址并将 *buffer_len* 设为缓冲区长度。出错时返回 `-1` 并设置一个 `TypeError`。

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

Part of the Stable ABI. 如果 *o* 支持单段可读缓冲接口则返回 `1`。否则返回 `0`。此函数总是会成功执行。

请注意此函数会尝试获取并释放一个缓冲区，并且在调用对应函数期间发生的异常会被屏蔽。要获取错误报告则应改用*PyObject_GetBuffer()*。

int **PyObject_AsWriteBuffer**(*PyObject* **obj*, void ***buffer*, *Py_ssize_t* **buffer_len*)

Part of the Stable ABI. 返回一个指向可写内存地址的指针。*obj* 必须支持单段字符缓冲接口。成功时返回 `0`，将 *buffer* 设为内存地址并将 *buffer_len* 设为缓冲区长度。出错时返回 `-1` 并设置一个 `TypeError`。

CHAPTER 8

具体的对象层

本章中的函数特定于某些 Python 对象类型。将错误类型的对象传递给它们并不是一个好主意；如果您从 Python 程序接收到一个对象，但不确定它是否具有正确的类型，则必须首先执行类型检查；例如，要检查对象是否为字典，请使用`PyDict_Check()`。本章的结构类似于 Python 对象类型的“家族树”。

> **警告：** 虽然本章所描述的函数会仔细检查传入对象的类型，但是其中许多函数不会检查传入的对象是否为 `NULL`。允许传入 `NULL` 可能导致内存访问冲突和解释器的立即终止。

8.1 基礎物件

本节描述 Python 类型对象和单一实例对象 象 None。

8.1.1 类型对象

type PyTypeObject
Part of the Limited API *(as an opaque struct).* 对象的 C 结构用于描述 built-in 类型。

PyTypeObject **PyType_Type**
Part of the Stable ABI. 这是属于 type 对象的 type object，它在 Python 层面和 `type` 是相同的对象。

int **PyType_Check**(*PyObject* *o)
如果对象 *o* 是一个类型对象，包括派生自标准类型对象的类型实例则返回非零值。在所有其它情况下都返回 0。此函数将总是成功执行。

int **PyType_CheckExact**(*PyObject* *o)
如果对象 *o* 是一个类型对象，但不是标准类型对象的子类型则返回非零值。在所有其它情况下都返回 0。此函数将总是成功执行。

unsigned int **PyType_ClearCache**()
Part of the Stable ABI. 清空内部查找缓存。返回当前版本标签。

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.

3.2 版新加入.

3.4 版更變: 返回类型现在是 `unsigned long` 而不是 `long`。

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

Part of the Stable ABI. 使该类型及其所有子类型的内部查找缓存失效。此函数必须在对该类型的属性或基类进行任何手动修改之后调用。

int **PyType_HasFeature** (*PyTypeObject* *o, int *feature*)

如果类型对象 *o* 设置了特性 *feature* 则返回非零值。类型特性是用单个比特位旗标来表示的。

int **PyType_IS_GC** (*PyTypeObject* *o)

如果类型对象包括对循环检测器的支持则返回真值；这会测试类型旗标`Py_TPFLAGS_HAVE_GC`。

int **PyType_IsSubtype** (*PyTypeObject* *a, *PyTypeObject* *b)

Part of the Stable ABI. 如果 *a* 是 *b* 的子类型则返回真值。

此函数只检查实际的子类型，这意味着 `__subclasscheck__()` 不会在 *b* 上被调用。请调用`PyObject_IsSubclass()` 来执行与 `issubclass()` 所做的相同检查。

PyObject ***PyType_GenericAlloc** (*PyTypeObject* *type, *Py_ssize_t* *nitems*)

Return value: New reference. Part of the Stable ABI. 类型对象的`tp_alloc` 槽位的通用处理句柄。请使用 Python 的默认内存分配机制来分配一个新的实例并将其所有内容初始化为 `NULL`。

PyObject ***PyType_GenericNew** (*PyTypeObject* *type, *PyObject* *args, *PyObject* *kwds)

Return value: New reference. Part of the Stable ABI. 类型对象的`tp_new` 槽位的通用处理句柄。请使用类型的`tp_alloc` 槽位来创建一个新的实例。

int **PyType_Ready** (*PyTypeObject* *type)

Part of the Stable ABI. 最终化一个类型对象。这应当在所有类型对象上调用以完成它们的初始化。此函数会负责从一个类型的基类添加被继承的槽位。成功时返回 `0`，或是在出错时返回 `-1` 并设置一个异常。

> **備F:** 如果某些基类实现了 GC 协议并且所提供的类型的旗标中未包括`Py_TPFLAGS_HAVE_GC`，则将自动从其父类实现 GC 协议。相反地，如果被创建的类型的旗标中未包括`Py_TPFLAGS_HAVE_GC` 则它 **必须**自己通过实现`tp_traverse` 句柄来实现 GC 协议。

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

Part of the Stable ABI *since version 3.4.* 返回存储在给定槽位中的函数指针。如果结果为 `NULL`，则表示或者该槽位为 `NULL`，或者该函数调用传入了无效的形参。调用方通常要将结果指针转换到适当的函数类型。

请参阅 `PyType_Slot.slot` 查看可用的 *slot* 参数值。

3.4 版新加入.

3.10 版更變: `PyType_GetSlot()` 现在可以接受所有类型。在此之前，它被限制为堆类型。

PyObject ***PyType_GetModule** (*PyTypeObject* *type)

Part of the Stable ABI *since version 3.10.* 返回当使用`PyType_FromModuleAndSpec()` 创建类型时关联到给定类型的模块对象。

如果没有关联到给定类型的模块，则设置 `TypeError` 并返回 `NULL`。

此函数通常被用于获取方法定义所在的模块。请注意在这样的方法中，`PyType_GetModule(Py_TYPE(self))` 可能不会返回预期的结果。`Py_TYPE(self)` 可以是

目标类的一个 子类，而子类并不一定是在与其上级类相同的模块中定义的。请参阅*PyCMethod* 了解如何获取方法定义所在的类。

3.9 版新加入.

void ***PyType_GetModuleState**(*PyTypeObject* *type*)

Part of the Stable ABI *since version 3.10.* 返回关联到给定类型的模块对象的状态。这是一个在*PyType_GetModule()* 的结果上调用*PyModule_GetState()* 的快捷方式。

如果没有关联到给定类型的模块，则设置 TypeError 并返回 NULL。

如果 *type* 有关联的模块但其状态为 NULL，则返回 NULL 且不设置异常。

3.9 版新加入.

创建堆分配类型

下列函数和结构体可被用来创建堆类型。

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

Return value: New reference. Part of the Stable ABI *since version 3.10.* 根据 *spec* (*Py_TPFLAGS_HEAPTYPE*) 创建并返回一个堆类型。

bases 参数可被用来指定基类；它可以是单个类或由多个类组成的元组。如果 *bases* 为 NULL，则会改用 *Py_tp_bases* 槽位。如果该槽位也为 NULL，则会改用 *Py_tp_base* 槽位。如果该槽位同样为 NULL，则新类型将派生自 object。

module 参数可被用来记录新类定义所在的模块。它必须是一个模块对象或为 NULL。如果不为 NULL，则该模块会被关联到新类型并且可在之后通过*PyType_GetModule()* 来获取。这个关联模块不可被子类继承；它必须为每个类单独指定。

此函数会在新类型上调用*PyType_Ready()*。

3.9 版新加入.

3.10 版更變: 此函数现在接受一个单独类作为 *bases* 参数并接受 NULL 作为 tp_doc 槽位。

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

Return value: New reference. Part of the Stable ABI *since version 3.3.* 等價於 PyType_FromModuleAndSpec(NULL, spec, bases)。

3.3 版新加入.

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

Return value: New reference. Part of the Stable ABI. 等價於 PyType_FromSpecWithBases(spec, NULL)。

type PyType_Spec

Part of the Stable ABI *(including all members).* 定义一个类型的行为的结构体。

const char *PyType_Spec.**name**

类型的名称，用来设置*PyTypeObject.tp_name*。

int PyType_Spec.**basicsize**

int PyType_Spec.**itemsize**

以字节数表示的实例大小，用来设置*PyTypeObject.tp_basicsize* 和*PyTypeObject.tp_itemsize*。

int PyType_Spec.**flags**

类型旗标，用来设置*PyTypeObject.tp_flags*。

如果未设置 Py_TPFLAGS_HEAPTYPE 旗标，则*PyType_FromSpecWithBases()* 会自动设置它。

PyType_Slot *PyType_Spec.**slots**

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

type PyType_Slot

Part of the Stable ABI *(including all members).* 定义一个类型的可选功能的结构体，包含一个槽位 ID 和一个值指针。

int PyType_Slot.**slot**

槽位 ID。

槽位 ID 的类名像是结构体*PyTypeObject*, *PyNumberMethods*, *PySequenceMethods*, *PyMappingMethods* 和*PyAsyncMethods* 的字段名附加一个 Py_ 前缀。举例来说，使用:

- Py_tp_dealloc 设置*PyTypeObject.tp_dealloc*
- Py_nb_add 设置*PyNumberMethods.nb_add*
- Py_sq_length 设置*PySequenceMethods.sq_length*

下列字段完全无法使得*PyType_Spec* 和 PyType_Slot 来设置:

- *tp_dict*
- *tp_mro*
- *tp_cache*
- *tp_subclasses*
- *tp_weaklist*
- *tp_vectorcall*
- *tp_weaklistoffset* (参见*PyMemberDef*)
- *tp_dictoffset* (参见*PyMemberDef*)
- *tp_vectorcall_offset* (参见*PyMemberDef*)

下列字段在受限 API 下无法使用*PyType_Spec* 和 PyType_Slot 来设置:

- *bf_getbuffer*
- *bf_releasebuffer*

设置 Py_tp_bases 或 Py_tp_base 在某些平台上可能会有问题。为了避免问题，请改用 PyType_FromSpecWithBases() 的 *bases* 参数。

3.9 版更變: *PyBufferProcs* 中的槽位可能会在不受限 API 中被设置。

void *PyType_Slot.**pfunc**

该槽位的预期值。在大多数情况下，这将是一个指向函数的指针。

Py_tp_doc 以外的槽位均不可为 NULL。

8.1.2 None 物件

请注意，None 的*PyTypeObject* 不会直接在 Python / C API 中公开。由于 None 是单例，测试对象标识（在 C 中使用 == ）就足够了。由于同样的原因，没有 PyNone_Check() 函数。

PyObject ***Py_None**
    Python None 对象，表示缺乏值。这个对象没有方法。它需要像引用计数一样处理任何其他对象。

Py_RETURN_NONE
    正确处理来自 C 函数内的*Py_None* 返回（也就是说，增加 None 的引用计数并返回它。）

8.2 數值物件

8.2.1 整數物件

所有整数都实现为长度任意的长整数对象。

在出错时，大多数 PyLong_As* API 都会返回 (return type)-1，这与数字无法区分开。请采用*PyErr_Occurred()* 来加以区分。

type PyLongObject
    Part of the Limited API (as an opaque struct). 表示 Python 整数对象的*PyObject* 子类型。

PyTypeObject **PyLong_Type**
    Part of the Stable ABI. 这个*PyTypeObject* 的实例表示 Python 的整数类型。与 Python 语言中的 int 相同。

int **PyLong_Check**(*PyObject* *p)
    如果参数是*PyLongObject* 或*PyLongObject* 的子类型，则返回 True。该函数一定能够执行成功。

int **PyLong_CheckExact**(*PyObject* *p)
    如果其参数属于*PyLongObject*，但不是*PyLongObject* 的子类型则返回真值。此函数总是会成功执行。

PyObject ***PyLong_FromLong**(long *v*)
    Return value: New reference. Part of the Stable ABI. 由 *v* 返回一个新的*PyLongObject* 对象，失败时返回 NULL 。

    当前的实现维护着一个整数对象数组，包含 -5 和 256 之间的所有整数对象。若创建一个位于该区间的 int 时，实际得到的将是对已有对象的引用。

PyObject ***PyLong_FromUnsignedLong**(unsigned long *v*)
    Return value: New reference. Part of the Stable ABI. Return a new *PyLongObject* object from a C unsigned long, or NULL on failure.

PyObject ***PyLong_FromSsize_t**(*Py_ssize_t* *v*)
    Return value: New reference. Part of the Stable ABI. 由 C *Py_ssize_t* 返回一个新的*PyLongObject* 对象，失败时返回 NULL 。

PyObject ***PyLong_FromSize_t**(size_t *v*)
    Return value: New reference. Part of the Stable ABI. 由 C size_t 返回一个新的*PyLongObject* 对象，失败则返回 NULL 。

PyObject ***PyLong_FromLongLong**(long long *v*)
    Return value: New reference. Part of the Stable ABI. Return a new *PyLongObject* object from a C long long, or NULL on failure.

PyObject ***PyLong_FromUnsignedLongLong*** (unsigned long long *v*)
Return value: New reference. Part of the Stable ABI. Return a new *PyLongObject* object from a C `unsigned long long`, or `NULL` on failure.

PyObject ***PyLong_FromDouble*** (double *v*)
Return value: New reference. Part of the Stable ABI. 由 *v* 的整数部分返回一个新的 *PyLongObject* 对象，失败则返回 `NULL` 。

PyObject ***PyLong_FromString*** (**const** char **str*, char ***pend*, int *base*)
Return value: New reference. Part of the Stable ABI. 根据 *str* 字符串值返回一个新的 *PyLongObject* ，*base* 指定了整数的基。如果 *pend* 不为 `NULL` ，则 */*pend* 将指向 *str* 中表示数字部分后面的第一个字符。如果 *base* 为 `0` ，*str* 将采用 integers 的定义进行解释；这时非零十进制数的前导零会触发 `ValueError` 。如果 *base* 不为 `0` ，则须位于 `2` 和 `36` 之间（含 2 和 36）。基之后及数字之间的前导空格、单下划线将被忽略。如果不存在数字，将触发 `ValueError`。

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*)
Return value: New reference. 将字符串 *u* 中的 Unicode 数字序列转换为 Python 整数值。

3.3 版新加入.

PyObject ***PyLong_FromVoidPtr*** (void **p*)
Return value: New reference. Part of the Stable ABI. 从指针 *p* 创建一个 Python 整数。可以使用 *PyLong_AsVoidPtr()* 返回的指针值。

long ***PyLong_AsLong*** (*PyObject* **obj*)
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`.

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

3.8 版更變: 如果可用将使用 `__index__()`。

3.10 版更變: 本函数不再使用 `__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*.

如果 *obj* 的值大于 `LONG_MAX` 或小于 `LONG_MIN`，则会把 **overflow* 分别置为 “1“ 或 `-1`，并返回 `1`；否则，将 **overflow* 置为 *0*。如果发生其他异常，则会按常规把 **overflow* 置为 `0`，并返回 `-1`。

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

3.8 版更變: 如果可用将使用 `__index__()`。

3.10 版更變: 本函数不再使用 `__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`.

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

3.8 版更變: 如果可用将使用 `__index__()`。

3.10 版更變: 本函数不再使用 `__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*.

如果 *obj* 的值大于 `LLONG_MAX` 或小于 `LLONG_MIN`，则按常规将 **overflow* 分别置为 `1` 或 `-1`，并返回 -1，否则将 **overflow* 置为 0。如果触发其他异常则 **overflow* 置为 0 并返回 -1。

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

3.2 版新加入.

3.8 版更變: 如果可用将使用 `__index__()`。

3.10 版更變: 本函数不再使用 `__int__()` 。

Py_ssize_t **PyLong_AsSsize_t**(*PyObject* **pylong*)

Part of the Stable ABI. 返回 *pylong* 的 C 语言*Py_ssize_t* 形式。*pylong* 必须是*PyLongObject* 的实例。

如果 *pylong* 的值超出了*Py_ssize_t* 的取值范围则会引发 `OverflowError`。

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

unsigned long **PyLong_AsUnsignedLong**(*PyObject* **pylong*)

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`.

出错时返回 `(unsigned long)-1` ，请利用*PyErr_Occurred()* 辨别具体问题。

size_t **PyLong_AsSize_t**(*PyObject* **pylong*)

Part of the Stable ABI. 返回 *pylong* 的 C 语言 `size_t` 形式。*pylong* 必须是*PyLongObject* 的实例。

如果 *pylong* 的值超出了 `size_t` 的取值范围则会引发 `OverflowError`。

出错时返回 `(size_t)-1` ，请利用*PyErr_Occurred()* 辨别具体问题。

unsigned long long **PyLong_AsUnsignedLongLong**(*PyObject* **pylong*)

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`.

出错时返回 `(unsigned long long)-1`，请利用*PyErr_Occurred()* 辨别具体问题。

3.1 版更變: 现在 *pylong* 为负值会触发 `OverflowError`，而不是 `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`.

出错时返回 `(unsigned long)-1`，请利用*PyErr_Occurred()* 辨别具体问题。

3.8 版更變: 如果可用将使用 `__index__()`。

3.10 版更變: 本函数不再使用 `__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`.

出错时返回 (unsigned long long)-1，请利用*PyErr_Occurred()* 辨别具体问题。

3.8 版更變: 如果可用将使用 __index__()。

3.10 版更變: 本函数不再使用 __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.

出错时返回 -1.0 ，请利用*PyErr_Occurred()* 辨别具体问题。

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()*.

出错时返回 NULL，请利用*PyErr_Occurred()* 辨别具体问题。

8.2.2 Boolean（布林）物件

Python 中的 boolean 是以整數子類別化來實現的。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。

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*)

Return value: New reference. 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

这个 C 类型*PyObject* 的子类型代表一个 Python 浮点数对象。

PyTypeObject **PyFloat_Type**

Part of the Stable ABI. 这是个属于 C 类型*PyTypeObject* 的代表 Python 浮点类型的实例。在 Python 层面的类型 float 是同一个对象。

int **PyFloat_Check** (*PyObject* *p)

如果它的参数是一个*PyFloatObject* 或者*PyFloatObject* 的子类型则返回真值。此函数总是会成功执行。

int **PyFloat_CheckExact**(*PyObject* *p)
如果它的参数是一个`PyFloatObject`但不是`PyFloatObject`的子类型则返回真值。此函数总是会成功执行。

PyObject ***PyFloat_FromString**(*PyObject* *str)
Return value: New reference. Part of the Stable ABI. 根据字符串 *str* 的值创建一个`PyFloatObject`，失败时返回 NULL。

PyObject ***PyFloat_FromDouble**(double *v*)
Return value: New reference. Part of the Stable ABI. 根据 *v* 创建一个`PyFloatObject`对象，失败时返回 NULL。

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.

3.8 版更變: 如果可用将使用 `__index__()`。

double **PyFloat_AS_DOUBLE**(*PyObject* *pyfloat)
Return a C `double` representation of the contents of *pyfloat*, but without error checking.

PyObject ***PyFloat_GetInfo**(void)
Return value: New reference. Part of the Stable ABI. 返回一个 structseq 实例，其中包含有关 float 的精度、最小值和最大值的信息。它是头文件 `float.h` 的一个简单包装。

double **PyFloat_GetMax**()
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`.

8.2.4 复数对象

从 C API 看，Python 的复数对象由两个不同的部分实现：一个是在 Python 程序使用的 Python 对象，另外的是一个代表真正复数值的 C 结构体。API 提供了函数共同操作两者。

表示复数的 C 结构体

需要注意的是接受这些结构体的作为参数并当做结果返回的函数，都是传递“值”而不是引用指针。此规则适用于整个 API。

type Py_complex
这是一个对应 Python 复数对象的值部分的 C 结构体。绝大部分处理复数对象的函数都用这类型的结构体作为输入或者输出值，它可近似地定义为：

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

Py_complex **_Py_c_sum**(*Py_complex* *left*, *Py_complex* *right*)
返回两个复数的和，用 C 类型`Py_complex`表示。

Py_complex **_Py_c_diff**(*Py_complex* *left*, *Py_complex* *right*)
返回两个复数的差，用 C 类型`Py_complex`表示。

Py_complex **_Py_c_neg** (*Py_complex num*)
返回复数 *num* 的负值，用 C `Py_complex` 表示。

Py_complex **_Py_c_prod** (*Py_complex left*, *Py_complex right*)
返回两个复数的乘积，用 C 类型`Py_complex` 表示。

Py_complex **_Py_c_quot** (*Py_complex dividend*, *Py_complex divisor*)
返回两个复数的商，用 C 类型`Py_complex` 表示。

如果 *divisor* 为空，这个方法返回零并设置 `errno` 为 `EDOM`。

Py_complex **_Py_c_pow** (*Py_complex num*, *Py_complex exp*)
返回 *num* 的 *exp* 次幂，用 C 类型`Py_complex` 表示。

如果 *num* 为空且 *exp* 不是正实数，这个方法返回零并设置 `errno` 为 `EDOM`。

表示复数的 Python 对象

type PyComplexObject
这个 C 类型`PyObject` 的子类型代表一个 Python 复数对象。

PyTypeObject **PyComplex_Type**
Part of the Stable ABI. 这是个属于`PyTypeObject` 的代表 Python 复数类型的实例。在 Python 层面的类型 `complex` 是同一个对象。

int **PyComplex_Check** (*PyObject *p*)
如果它的参数是一个`PyComplexObject` 或者`PyComplexObject` 的子类型则返回真值。此函数总是会成功执行。

int **PyComplex_CheckExact** (*PyObject *p*)
如果它的参数是一个`PyComplexObject` 但不是`PyComplexObject` 的子类型则返回真值。此函数总是会成功执行。

PyObject ***PyComplex_FromCComplex** (*Py_complex v*)
Return value: New reference. 根据 C 类型`Py_complex` 的值生成一个新的 Python 复数对象。

PyObject ***PyComplex_FromDoubles** (double *real*, double *imag*)
Return value: New reference. Part of the Stable ABI. 根据 *real* 和 *imag* 返回一个新的 C 类型`PyComplexObject` 对象。

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*)
返回复数 *op* 的 C 类型`Py_complex` 值。

如果 *op* 不是一个 Python 复数对象，但是具有 `__complex__()` 方法，此方法将首先被调用，将 *op* 转换为一个 Python 复数对象。如果 `__complex__()` 未定义则将回退至 `__float__()`，如果 `__float__()` 未定义则将回退至 `__index__()`。如果失败，此方法将返回 `-1.0` 作为实数值。

3.8 版更變: 如果可用将使用 `__index__()`。

8.3 序列物件

序列对象的一般操作在前一章中讨论过; 本节介绍 Python 语言固有的特定类型的序列对象。

8.3.1 bytes 对象

这些函数在期望附带一个字节串形参但却附带了一个非字节串形参被调用时会引发 `TypeError`。

type PyBytesObject
这种 `PyObject` 的子类型表示一个 Python 字节对象。

PyTypeObject **PyBytes_Type**
Part of the Stable ABI. `PyTypeObject` 的实例代表一个 Python 字节类型，在 Python 层面它与 `bytes` 是相同的对象。

int **PyBytes_Check** (*PyObject* *o)
如果对象 *o* 是一个 bytes 对象或者 bytes 类型的子类型的实例则返回真值。此函数总是会成功执行。

int **PyBytes_CheckExact** (*PyObject* *o)
如果对象 *o* 是一个 bytes 对象但不是 bytes 类型的子类型的实例则返回真值。此函数总是会成功执行。

PyObject ***PyBytes_FromString** (**const** char *v)
Return value: New reference. Part of the Stable ABI. 成功时返回一个以字符串 *v* 的副本为值的新字节串对象，失败时返回 `NULL`。形参 *v* 不可为 `NULL`；它不会被检查。

PyObject ***PyBytes_FromStringAndSize** (**const** char *v, *Py_ssize_t* len)
Return value: New reference. Part of the Stable ABI. 成功时返回一个以字符串 *v* 的副本为值且长度为 *len* 的新字节串对象，失败时返回 `NULL`。如果 *v* 为 `NULL`，则不初始化字节串对象的内容。

PyObject ***PyBytes_FromFormat** (**const** char *format, ...)
Return value: New reference. Part of the Stable ABI. 接受一个 C `printf()` 风格的 *format* 字符串和可变数量的参数，计算结果 Python 字节串对象的大小并返回参数值经格式化后的字节串对象。可变数量的参数必须均为 C 类型并且必须恰好与 *format* 字符串中的格式字符相对应。允许使用下列格式字符串:

格式字符	类型	注释
`%%`	*n/a*	文字% 字符。
`%c`	int	一个字节，被表示为一个 C 语言的整型
`%d`	int	等價於 `printf("%d")`.[1]
`%u`	unsigned int	等價於 `printf("%u")`.[1]
`%ld`	long	等價於 `printf("%ld")`.[1]
`%lu`	unsigned long	等價於 `printf("%lu")`.[1]
`%zd`	`Py_ssize_t`	等價於 `printf("%zd")`.[1]
`%zu`	size_t	等價於 `printf("%zu")`.[1]
`%i`	int	等價於 `printf("%i")`.[1]
`%x`	int	等價於 `printf("%x")`.[1]
`%s`	const char*	以 null 为终止符的 C 字符数组。
`%p`	const void*	一个 C 指针的十六进制表示形式。基本等价于 `printf("%p")` 但它会确保以字面值 `0x` 开头，不论系统平台上 `printf` 的输出是什么。

无法识别的格式字符会导致将格式字符串的其余所有内容原样复制到结果对象，并丢弃所有多余的参数。

[1] 对于整数说明符（d，u，ld，lu，zd，zu，i，x）：当给出精度时，0 转换标志是有效的。

PyObject ***PyBytes_FromFormatV**(**const** char **format*, va_list *vargs*)
Return value: New reference. Part of the Stable ABI. 与`PyBytes_FromFormat()` 完全相同，除了它需要两个参数。

PyObject ***PyBytes_FromObject**(*PyObject* **o*)
Return value: New reference. Part of the Stable ABI. 返回字节表示实现缓冲区协议的对象 *o*。

Py_ssize_t **PyBytes_Size**(*PyObject* **o*)
Part of the Stable ABI. 返回字节对象 *o* 中字节的长度。

Py_ssize_t **PyBytes_GET_SIZE**(*PyObject* **o*)
宏版本的`PyBytes_Size()` 但是不带错误检测。

char ***PyBytes_AsString**(*PyObject* **o*)
Part of the Stable ABI. 返回对应 *o* 的内容的指针。该指针指向 *o* 的内部缓冲区，其中包含 `len(o) + 1` 个字节。缓冲区的最后一个字节总是为空，不论是否存在其他空字节。该数据不可通过任何形式来修改，除非是刚使用 `PyBytes_FromStringAndSize(NULL, size)` 创建该对象。它不可被撤销分配。如果 *o* 根本不是一个字节串对象，则`PyBytes_AsString()` 将返回 `NULL` 并引发 `TypeError`。

char ***PyBytes_AS_STRING**(*PyObject* **string*)
宏版本的`PyBytes_AsString()` 但是不带错误检测。

int **PyBytes_AsStringAndSize**(*PyObject* **obj*, char ***buffer*, *Py_ssize_t* **length*)
Part of the Stable ABI. 通过输出变量 *buffer* 和 *length* 返回以 null 为终止符的对象 *obj* 的内容。

如果 *length* 为 `NULL`，字节串对象就不包含嵌入的空字节；如果包含，则该函数将返回 `-1` 并引发 `ValueError`。

该缓冲区指向 *obj* 的内部缓冲，它的末尾包含一个额外的空字节（不算在 *length* 当中）。该数据不可通过任何方式来修改，除非是刚使用 `PyBytes_FromStringAndSize(NULL, size)` 创建该对象。它不可被撤销分配。如果 *obj* 根本不是一个字节串对象，则`PyBytes_AsStringAndSize()` 将返回 `-1` 并引发 `TypeError`。

3.5 版更變: 以前，当字节串对象中出现嵌入的空字节时将引发 `TypeError`。

void **PyBytes_Concat**(*PyObject* ***bytes*, *PyObject* **newpart*)
Part of the Stable ABI. 在 **bytes* 中创建新的字节串对象，其中包含添加到 *bytes* 的 *newpart* 的内容；调用者将获得新的引用。对 *bytes* 原值的引用将被收回。如果无法创建新对象，对 *bytes* 的旧引用仍将被丢弃且 **bytes* 的值将被设为 `NULL`；并将设置适当的异常。

void **PyBytes_ConcatAndDel**(*PyObject* ***bytes*, *PyObject* **newpart*)
Part of the Stable ABI. 在 **bytes* 中创建新的字节串对象，其中包含添加到 *bytes* 的 *newpart* 的内容。此版本会减少 *newpart* 的引用计数。

int **_PyBytes_Resize**(*PyObject* ***bytes*, *Py_ssize_t* *newsize*)
改变字节串大小的一种方式，即使其为"不可变对象"。此方式仅用于创建全新的字节串对象；如果字节串在代码的其他部分已知则不可使用此方式。如果输入字节串对象的引用计数不为一则调用此函数将报错。传入一个现有字节串对象的地址作为 lvalue（它可能会被写入），并传入希望的新大小。当成功时，**bytes* 将存放改变大小后的字节串对象并返回 `0`；**bytes* 中的地址可能与其输入值不同。如果重新分配失败，则 **bytes* 上的原字节串对象将被撤销分配，**bytes* 会被设为 `NULL`，同时设置 `MemoryError` 并返回 `-1`。

8.3.2 字节数组对象

type PyByteArrayObject
这个 *PyObject* 的子类型表示一个 Python 字节数组对象。

PyTypeObject **PyByteArray_Type**
Part of the Stable ABI. Python bytearray 类型表示为 *PyTypeObject* 的实例；这与 Python 层面的 bytearray 是相同的对象。

类型检查宏

int **PyByteArray_Check** (*PyObject* *o)
如果对象 *o* 是一个 bytearray 对象或者 bytearray 类型的子类型的实例则返回真值。此函数总是会成功执行。

int **PyByteArray_CheckExact** (*PyObject* *o)
如果对象 *o* 是一个 bytearray 对象但不是 bytearray 类型的子类型的实例则返回真值。此函数总是会成功执行。

直接 API 函数

PyObject ***PyByteArray_FromObject** (*PyObject* *o)
Return value: New reference. Part of the Stable ABI. 根据任何实现了缓冲区协议 的对象 *o*，返回一个新的字节数组对象。

PyObject ***PyByteArray_FromStringAndSize** (**const** char *string, *Py_ssize_t* len)
Return value: New reference. Part of the Stable ABI. 根据 *string* 及其长度 *len* 创建一个新的 bytearray 对象。当失败时返回 NULL。

PyObject ***PyByteArray_Concat** (*PyObject* *a, *PyObject* *b)
Return value: New reference. Part of the Stable ABI. 连接字节数组 *a* 和 *b* 并返回一个带有结果的新的字节数组。

Py_ssize_t **PyByteArray_Size** (*PyObject* *bytearray)
Part of the Stable ABI. 在检查 NULL 指针后返回 *bytearray* 的大小。

char ***PyByteArray_AsString** (*PyObject* *bytearray)
Part of the Stable ABI. 在检查 NULL 指针后返回将 *bytearray* 返回为一个字符数组。返回的数组总是会附加一个额外的空字节。

int **PyByteArray_Resize** (*PyObject* *bytearray, *Py_ssize_t* len)
Part of the Stable ABI. 将 *bytearray* 的内部缓冲区的大小调整为 *len*。

宏

这些宏减低安全性以换取性能，它们不检查指针。

char ***PyByteArray_AS_STRING** (*PyObject* *bytearray)
C 函数 *PyByteArray_AsString()* 的宏版本。

Py_ssize_t **PyByteArray_GET_SIZE** (*PyObject* *bytearray)
C 函数 *PyByteArray_Size()* 的宏版本。

8.3.3 Unicode 物件與編碼

Unicode 对象

自从 python3.3 中实现了 **PEP 393** 以来，Unicode 对象在内部使用各种表示形式，以便在保持内存效率的同时处理完整范围的 Unicode 字符。对于所有代码点都低于 128、256 或 65536 的字符串，有一些特殊情况；否则，代码点必须低于 1114112（这是完整的 Unicode 范围）。

*Py_UNICODE** 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 类型

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*.

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.

3.3 版更變: 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.

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 really C macros and can be used to do fast checks and to access internal read-only data of Unicode objects:

int **PyUnicode_Check** (*PyObject* *o)
Return true if the object *o* is a Unicode object or an instance of a Unicode subtype. This function always succeeds.

int **PyUnicode_CheckExact** (*PyObject* *o)
Return true if the object *o* is a Unicode object, but not an instance of a subtype. This function always succeeds.

int **PyUnicode_READY** (*PyObject* *o)
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.

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* *o)
Return the length of the Unicode string, in code points. *o* has to be a Unicode object in the "canonical" representation (not checked).

3.3 版新加入.

Py_UCS1 ***PyUnicode_1BYTE_DATA** (*PyObject* *o)

Py_UCS2 ***PyUnicode_2BYTE_DATA** (*PyObject* *o)

Py_UCS4 ***PyUnicode_4BYTE_DATA** (*PyObject* *o)
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.

3.3 版新加入.

PyUnicode_WCHAR_KIND

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND

PyUnicode_4BYTE_KIND
返回`PyUnicode_KIND()` 宏的值。

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。

unsigned int **PyUnicode_KIND** (*PyObject* *o)
Return one of the PyUnicode kind constants (see above) that indicate how many bytes per character this Unicode object uses to store its data. *o* has to be a Unicode object in the "canonical" representation (not checked).

3.3 版新加入.

void ***PyUnicode_DATA** (*PyObject* *o)
Return a void pointer to the raw Unicode buffer. *o* has to be a Unicode object in the "canonical" representation (not checked).

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 macro does not do any sanity checks and is intended for usage in loops. The caller should cache the *kind* value and *data* pointer as obtained from other macro 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.

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.

3.3 版新加入.

Py_UCS4 **PyUnicode_READ_CHAR** (*PyObject* **o*, *Py_ssize_t* *index*)
Read a character from a Unicode object *o*, which must be in the "canonical" representation. This is less efficient than `PyUnicode_READ()` if you do multiple consecutive reads.

3.3 版新加入.

PyUnicode_MAX_CHAR_VALUE (*o*)
Return the maximum code point that is suitable for creating another string based on *o*, which must be in the "canonical" representation. This is always an approximation but more efficient than iterating over the string.

3.3 版新加入.

Py_ssize_t **PyUnicode_GET_SIZE** (*PyObject* **o*)
Return the size of the deprecated `Py_UNICODE` representation, in code units (this includes surrogate pairs as 2 units). *o* has to be a Unicode object (not checked).

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: 旧式 Unicode API 的一部分，请迁移到使用`PyUnicode_GET_LENGTH()`。

Py_ssize_t **PyUnicode_GET_DATA_SIZE** (*PyObject* **o*)
Return the size of the deprecated `Py_UNICODE` representation in bytes. *o* has to be a Unicode object (not checked).

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: 旧式 Unicode API 的一部分，请迁移到使用`PyUnicode_GET_LENGTH()`。

Py_UNICODE ***PyUnicode_AS_UNICODE** (*PyObject* **o*)

const char ***PyUnicode_AS_DATA** (*PyObject* **o*)
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 *o* argument has to be a Unicode object (not checked).

3.3 版更變: This macro is now inefficient -- because in many cases the `Py_UNICODE` representation does not exist and needs to be created -- and can fail (return `NULL` with an exception set). Try to port the code to use the new `PyUnicode_nBYTE_DATA()` macros or use `PyUnicode_WRITE()` or `PyUnicode_READ()`.

自從版本 3.3 後不推薦使用，將會自版本 3.12 中移除。: 旧式 Unicode API 的一部分，请迁移到使用 `PyUnicode_nBYTE_DATA()` 宏族。

int **PyUnicode_IsIdentifier** (*PyObject* **o*)
Part of the Stable ABI. Return `1` if the string is a valid identifier according to the language definition, section identifiers. Return `0` otherwise.

3.9 版更變: The function does not call `Py_FatalError()` anymore if the string is not ready.

Unicode 字符属性

Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration.

int **Py_UNICODE_ISSPACE**(*Py_UCS4 ch*)
根据 *ch* 是否为空白字符返回 `1` 或 `0`。

int **Py_UNICODE_ISLOWER**(*Py_UCS4 ch*)
根据 *ch* 是否为小写字符返回 `1` 或 `0`。

int **Py_UNICODE_ISUPPER**(*Py_UCS4 ch*)
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.

3.3 版後已F用: This function uses simple case mappings.

Py_UCS4 **Py_UNICODE_TOUPPER**(*Py_UCS4 ch*)
Return the character *ch* converted to upper case.

3.3 版後已F用: This function uses simple case mappings.

Py_UCS4 **Py_UNICODE_TOTITLE**(*Py_UCS4 ch*)
Return the character *ch* converted to title case.

3.3 版後已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*)
Return value: New reference. 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.

3.3 版新加入.

PyObject ***PyUnicode_FromKindAndData**(int *kind*, **const** void **buffer*, *Py_ssize_t size*)
Return value: New reference. 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.

3.3 版新加入.

PyObject ***PyUnicode_FromStringAndSize**(**const** char **u*, *Py_ssize_t size*)
Return value: New reference. Part of the Stable ABI. Create a Unicode object from the char buffer *u*. 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 *u* 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 **u*)
Return value: New reference. Part of the Stable ABI. Create a Unicode object from a UTF-8 encoded null-terminated char buffer *u*.

PyObject ***PyUnicode_FromFormat**(**const** char **format*, ...)
Return value: New reference. Part of the Stable ABI. Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python Unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* ASCII-encoded string. The following format characters are allowed:

格式字符	类型	注释
%%	*n/a*	文字% 字符。
%c	int	单个字符，表示为 C 语言的整型。
%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*	以 null 为终止符的 C 字符数组。
%p	const void*	一个 C 指针的十六进制表示形式。基本等价于 printf("%p") 但它会确保以字面值 0x 开头，不论系统平台上 printf 的输出是什么。
%A	PyObject*	ascii() 调用的结果。
%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F: 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).

3.2 版更變: Support for "%lld" and "%llu" added.

3.3 版更變: Support for "%li", "%lli" and "%zi" added.

3.4 版更變: Support width and precision formatter for "%s", "%A", "%U", "%V", "%S", "%R" added.

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

Return value: New reference. Part of the Stable ABI. Identical to PyUnicode_FromFormat() except that it takes exactly two arguments.

PyObject ***PyUnicode_FromEncodedObject**(*PyObject* **obj*, **const** char **encoding*, **const** char **errors*)

Return value: New reference. 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).

[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.

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.

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.

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.

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).

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 the macro version `PyUnicode_READ_CHAR()`.

3.3 版新加入.

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

Return value: New reference. Part of the Stable ABI *since version 3.7.* Return a substring of *str*, from character index *start* (included) to character index *end* (excluded). Negative indices are not supported.

3.3 版新加入.

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

Part of the Stable ABI *since version 3.7.* Copy the string *u* 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 *u*). *buffer* is returned on success.

3.3 版新加入.

Py_UCS4 ***PyUnicode_AsUCS4Copy**(*PyObject* **u*)

Part of the Stable ABI *since version 3.7.* Copy the string *u* 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.

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)

Return value: New reference. 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.

PyObject ***PyUnicode_TransformDecimalToASCII**(*Py_UNICODE* *s, *Py_ssize_t* size)

Return value: New reference. Create a Unicode object by replacing all decimal digits in `Py_UNICODE` buffer of the given *size* by ASCII digits 0--9 according to their decimal value. Return `NULL` if an exception occurs.

自從版本 3.3 後不推薦使用，將會自版本 3.11 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `Py_UNICODE_TODECIMAL()`.

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.

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。: 旧式 Unicode API 的一部分，请迁移到使用`PyUnicode_GET_LENGTH()`。

PyObject ***PyUnicode_FromObject**(*PyObject* *obj)

Return value: New reference. Part of the Stable ABI. Copy an instance of a Unicode subtype to a new true Unicode object if necessary. If *obj* is already a true Unicode object (not a subtype), return the reference with incremented refcount.

Objects other than Unicode or its subtypes will cause a `TypeError`.

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* len, const char *errors)

Return value: New reference. 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。

3.3 版新加入.

3.7 版更變: 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)

Return value: New reference. Part of the Stable ABI *since version 3.7.* Similar to `PyUnicode_DecodeLocaleAndSize()`, but compute the string length using `strlen()`.

3.3 版新加入.

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

Return value: New reference. 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。

3.3 版新加入.

3.7 版更變: 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`.

文件系統編碼格式

To encode and decode file names and other environment strings, Py_FileSystemDefaultEncoding should be used as the encoding, and Py_FileSystemDefaultEncodeErrors should be used as the error handler (**PEP 383** and **PEP 529**). To encode file names to bytes during argument parsing, the "O&" converter should be used, passing *PyUnicode_FSConverter()* as the conversion function:

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

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.

3.1 版新加入.

3.6 版更變: 接受一个*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.

3.2 版新加入.

3.6 版更變: 接受一个*path-like object*。

PyObject ***PyUnicode_DecodeFSDefaultAndSize**(**const** char *s, *Py_ssize_t* size)

Return value: New reference. 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。

3.6 版更變: Use Py_FileSystemDefaultEncodeErrors error handler.

PyObject ***PyUnicode_DecodeFSDefault**(**const** char *s)

Return value: New reference. 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.

3.6 版更變: Use Py_FileSystemDefaultEncodeErrors error handler.

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

Return value: New reference. 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。

3.2 版新加入.

3.6 版更變: Use `Py_FileSystemDefaultEncodeErrors` error handler.

wchar_t Support

`wchar_t` support for platforms which support it:

PyObject ***PyUnicode_FromWideChar**(**const** wchar_t **w*, *Py_ssize_t size*)

Return value: New reference. Part of the Stable ABI. Create a Unicode object from the `wchar_t` buffer *w* 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 **w*, *Py_ssize_t size*)

Part of the Stable ABI. Copy the Unicode object contents into the `wchar_t` buffer *w*. 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. 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_Alloc()` (use *PyMem_Free()* to free it) on success. On error, returns `NULL` and **size* is undefined. Raises a `MemoryError` if memory allocation is failed.

3.2 版新加入.

3.7 版更變: 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 **s*, *Py_ssize_t* *size*, **const** char **encoding*, **const** char **errors*)

Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the encoded string *s*. *encoding* and *errors* have the same meaning as the parameters of the same name in the `str()` built-in function. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsEncodedString**(*PyObject* **unicode*, **const** char **encoding*, **const** char **errors*)

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

PyObject ***PyUnicode_Encode**(**const** *Py_UNICODE* **s*, *Py_ssize_t* *size*, **const** char **encoding*, **const** char **errors*)

Return value: New reference. Encode the `Py_UNICODE` buffer *s* of the given *size* and return a 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.

自從版本 3.3 後不推薦使用，將會自版本 3.11 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsEncodedString()`.

UTF-8 编解码器

以下是 UTF-8 编解码器 API:

PyObject ***PyUnicode_DecodeUTF8**(**const** char **s*, *Py_ssize_t* *size*, **const** char **errors*)

Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the UTF-8 encoded string *s*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeUTF8Stateful**(**const** char **s*, *Py_ssize_t* *size*, **const** char **errors*, *Py_ssize_t* **consumed*)

Return value: New reference. 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*)

Return value: New reference. 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.

3.3 版新加入.

3.7 版更變: The return type is now `const char *` rather of `char *`.

3.10 版更變: This function is a part of the *limited API*.

const char ***PyUnicode_AsUTF8**(*PyObject* **unicode*)
As `PyUnicode_AsUTF8AndSize()`, but does not store the size.

3.3 版新加入.

3.7 版更變: The return type is now `const char *` rather of `char *`.

PyObject ***PyUnicode_EncodeUTF8**(**const** *Py_UNICODE* **s*, *Py_ssize_t size*, **const** char **errors*)
Return value: New reference. Encode the `Py_UNICODE` buffer *s* of the given *size* using UTF-8 and return a Python bytes object. Return `NULL` if an exception was raised by the codec.

自從版本 3.3 後不推薦使用，將會自版本 3.11 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsUTF8String()`, `PyUnicode_AsUTF8AndSize()` or `PyUnicode_AsEncodedString()`.

UTF-32 Codecs

These are the UTF-32 codec APIs:

PyObject ***PyUnicode_DecodeUTF32**(**const** char **s*, *Py_ssize_t size*, **const** char **errors*, int **byteorder*)
Return value: New reference. 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 **s*, *Py_ssize_t size*, **const** char **errors*, int **byteorder*, *Py_ssize_t* **consumed*)
Return value: New reference. 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*)
Return value: New reference. 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.

PyObject ***PyUnicode_EncodeUTF32**(**const** *Py_UNICODE* **s*, *Py_ssize_t size*, **const** char **errors*, int *byteorder*)
Return value: New reference. Return a Python bytes object holding the UTF-32 encoded value of the Unicode data in *s*. Output is written according to the following byte order:

```
byteorder == -1: little endian
byteorder == 0:  native byte order (writes a BOM mark)
byteorder == 1:  big endian
```

If byteorder is `0`, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.

If `Py_UNICODE_WIDE` is not defined, surrogate pairs will be output as a single code point.

Return `NULL` if an exception was raised by the codec.

自從版本 3.3 後不推薦使用，將會自版本 3.11 中移除。: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsUTF32String()* or *PyUnicode_AsEncodedString()*.

UTF-16 Codecs

These are the UTF-16 codec APIs:

PyObject ***PyUnicode_DecodeUTF16**(**const** char **s*, *Py_ssize_t size*, **const** char **errors*, int **byteorder*)

Return value: New reference. 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 `\uffe` 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 **s*, *Py_ssize_t size*, **const** char **errors*, int **byteorder*, *Py_ssize_t *consumed*)

Return value: New reference. 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*)

Return value: New reference. 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.

PyObject ***PyUnicode_EncodeUTF16**(**const** *Py_UNICODE *s*, *Py_ssize_t size*, **const** char **errors*, int *byteorder*)

Return value: New reference. Return a Python bytes object holding the UTF-16 encoded value of the Unicode data in *s*. Output is written according to the following byte order:

```
byteorder == -1: little endian
byteorder == 0:  native byte order (writes a BOM mark)
byteorder == 1:  big endian
```

If byteorder is `0`, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.

If `Py_UNICODE_WIDE` is defined, a single `Py_UNICODE` value may get represented as a surrogate pair. If it is not defined, each `Py_UNICODE` values is interpreted as a UCS-2 character.

Return `NULL` if an exception was raised by the codec.

自從版本 3.3 後不推薦使用，將會自版本 3.11 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsUTF16String()` or `PyUnicode_AsEncodedString()`.

UTF-7 Codecs

These are the UTF-7 codec APIs:

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

Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the UTF-7 encoded string *s*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeUTF7Stateful**(**const** char **s*, *Py_ssize_t* *size*, **const** char **errors*, *Py_ssize_t* **consumed*)

Return value: New reference. 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*.

PyObject ***PyUnicode_EncodeUTF7**(**const** *Py_UNICODE* **s*, *Py_ssize_t* *size*, int *base64SetO*, int *base64WhiteSpace*, **const** char **errors*)

Return value: New reference. Encode the `Py_UNICODE` buffer of the given size using UTF-7 and return a Python bytes object. Return `NULL` if an exception was raised by the codec.

If *base64SetO* is nonzero, "Set O" (punctuation that has no otherwise special meaning) will be encoded in base-64. If *base64WhiteSpace* is nonzero, whitespace will be encoded in base-64. Both are set to zero for the Python "utf-7" codec.

自從版本 3.3 後不推薦使用，將會自版本 3.11 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsEncodedString()`.

Unicode-Escape 编解码器

These are the "Unicode Escape" codec APIs:

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

Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the Unicode-Escape encoded string *s*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsUnicodeEscapeString**(*PyObject* **unicode*)

Return value: New reference. 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.

PyObject ***PyUnicode_EncodeUnicodeEscape**(**const** *Py_UNICODE* **s*, *Py_ssize_t* *size*)

Return value: New reference. Encode the `Py_UNICODE` buffer of the given *size* using Unicode-Escape and return a bytes object. Return `NULL` if an exception was raised by the codec.

自從版本 3.3 後不推薦使用，將會自版本 3.11 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsUnicodeEscapeString()`.

Raw-Unicode-Escape Codecs

These are the "Raw Unicode Escape" codec APIs:

PyObject ***PyUnicode_DecodeRawUnicodeEscape**(const char *s, *Py_ssize_t* size, const char *errors)
Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the Raw-Unicode-Escape encoded string *s*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsRawUnicodeEscapeString**(*PyObject* *unicode)
Return value: New reference. 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.

PyObject ***PyUnicode_EncodeRawUnicodeEscape**(const *Py_UNICODE* *s, *Py_ssize_t* size)
Return value: New reference. Encode the `Py_UNICODE` buffer of the given *size* using Raw-Unicode-Escape and return a bytes object. Return `NULL` if an exception was raised by the codec.

自從版本 3.3 後不推薦使用，將會自版本 3.11 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsRawUnicodeEscapeString()` or `PyUnicode_AsEncodedString()`.

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 *s, *Py_ssize_t* size, const char *errors)
Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the Latin-1 encoded string *s*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsLatin1String**(*PyObject* *unicode)
Return value: New reference. 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.

PyObject ***PyUnicode_EncodeLatin1**(const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors)
Return value: New reference. Encode the `Py_UNICODE` buffer of the given *size* using Latin-1 and return a Python bytes object. Return `NULL` if an exception was raised by the codec.

自從版本 3.3 後不推薦使用，將會自版本 3.11 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsLatin1String()` or `PyUnicode_AsEncodedString()`.

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 *s, *Py_ssize_t* size, const char *errors)
Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the ASCII encoded string *s*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsASCIIString**(*PyObject* *unicode)
Return value: New reference. 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.

PyObject ***PyUnicode_EncodeASCII**(const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors)
Return value: New reference. Encode the `Py_UNICODE` buffer of the given *size* using ASCII and return a Python bytes object. Return `NULL` if an exception was raised by the codec.

自從版本 3.3 後不推薦使用，將會自版本 3.11 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsASCIIString()` or `PyUnicode_AsEncodedString()`.

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 **data*, *Py_ssize_t size*, *PyObject *mapping*, **const** char **errors*)

Return value: New reference. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the encoded string *s* 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*)

Return value: New reference. 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.

PyObject ***PyUnicode_EncodeCharmap**(**const** *Py_UNICODE *s*, *Py_ssize_t size*, *PyObject *mapping*, **const** char **errors*)

Return value: New reference. Encode the `Py_UNICODE` buffer of the given *size* using the given *mapping* object and return the result as a bytes object. Return `NULL` if an exception was raised by the codec.

自從版本 3.3 後不推薦使用，將會自版本 3.11 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsCharmapString()` or `PyUnicode_AsEncodedString()`.

The following codec API is special in that maps Unicode to Unicode.

PyObject ***PyUnicode_Translate**(*PyObject *str*, *PyObject *table*, **const** char **errors*)

Return value: New reference. 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.

PyObject ***PyUnicode_TranslateCharmap**(**const** *Py_UNICODE *s*, *Py_ssize_t size*, *PyObject *mapping*, **const** char **errors*)

Return value: New reference. Translate a `Py_UNICODE` buffer of the given *size* by applying a character *mapping* table to it and return the resulting Unicode object. Return `NULL` when an exception was raised by the codec.

自從版本 3.3 後不推薦使用，將會自版本 3.11 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_Translate()`. or *generic codec based API*

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 *s, *Py_ssize_t* size, **const** char *errors)
    Return value: New reference. Part of the Stable ABI *on Windows since version 3.7.* Create a Unicode object by decoding *size* bytes of the MBCS encoded string *s*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeMBCSStateful**(**const** char *s, *Py_ssize_t* size, **const** char *errors, *Py_ssize_t* *consumed)
    Return value: New reference. Part of the Stable ABI *on Windows since version 3.7.* 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)
    Return value: New reference. 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)
    Return value: New reference. Part of the Stable ABI *on Windows since version 3.7.* Encode the Unicode object using the specified code page and return a Python bytes object. Return `NULL` if an exception was raised by the codec. Use `CP_ACP` code page to get the MBCS encoder.

    3.3 版新加入.

PyObject ***PyUnicode_EncodeMBCS**(**const** *Py_UNICODE* *s, *Py_ssize_t* size, **const** char *errors)
    Return value: New reference. Encode the `Py_UNICODE` buffer of the given *size* using MBCS and return a Python bytes object. Return `NULL` if an exception was raised by the codec.

    自從版本 3.3 後不推薦使用，將會自版本 4.0 中移除。: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsMBCSString()`, `PyUnicode_EncodeCodePage()` or `PyUnicode_AsEncodedString()`.

Methods & Slots

方法与槽位函数

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)
    Return value: New reference. Part of the Stable ABI. Concat two strings giving a new Unicode string.

PyObject ***PyUnicode_Split**(*PyObject* *s, *PyObject* *sep, *Py_ssize_t* maxsplit)
    Return value: New reference. 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* *s, int keepend)
    Return value: New reference. Part of the Stable ABI. Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If *keepend* is `0`, the line break characters are not included in the resulting strings.

PyObject ***PyUnicode_Join**(*PyObject* *separator, *PyObject* *seq)
Return value: New reference. 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* *str, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)
Part of the Stable ABI. Return `1` if *substr* matches `str[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* *str, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)
Part of the Stable ABI. Return the first position of *substr* in `str[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* *str, *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 `str[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.

3.3 版新加入.

3.7 版更變: *start* and *end* are now adjusted to behave like `str[start:end]`.

Py_ssize_t **PyUnicode_Count**(*PyObject* *str, *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 `str[start:end]`. Return `-1` if an error occurred.

PyObject ***PyUnicode_Replace**(*PyObject* *str, *PyObject* *substr, *PyObject* *replstr, *Py_ssize_t* maxcount)
Return value: New reference. Part of the Stable ABI. Replace at most *maxcount* occurrences of *substr* in *str* 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* *uni, **const** char *string)
Part of the Stable ABI. Compare a Unicode object, *uni*, 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)
Return value: New reference. Part of the Stable ABI. 对两个 Unicode 字符串执行富比较并返回以下值之一:

- `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)
Return value: New reference. Part of the Stable ABI. Return a new string object from *format* and *args*; this is analogous to `format % args`.

int **PyUnicode_Contains**(*PyObject* *container, *PyObject* *element)
Part of the Stable ABI. Check whether *element* is contained in *container* and return true or false accordingly.

element has to coerce to a one element Unicode string. `-1` is returned if there was an error.

void **PyUnicode_InternInPlace**(*PyObject* ***string*)

Part of the Stable ABI. Intern the argument **string* 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 **string*, it sets **string* to it (decrementing the reference count of the old string object and incrementing the reference count of the interned string object), otherwise it leaves **string* alone and interns it (incrementing its reference count). (Clarification: even though there is a lot of talk about reference counts, think of this function as reference-count-neutral; you own the object after the call if and only if you owned it before the call.)

PyObject ***PyUnicode_InternFromString**(**const** char **v*)

Return value: New reference. 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

这个`PyObject`的子类型代表一个 Python 的元组对象。

PyTypeObject **PyTuple_Type**

Part of the Stable ABI. `PyTypeObject` 的实例代表一个 Python 元组类型，这与 Python 层面的 `tuple` 是相同的对象。

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

如果 *p* 是一个 tuple 对象或者 tuple 类型的子类型的实例则返回真值。此函数总是会成功执行。

int **PyTuple_CheckExact**(*PyObject* **p*)

如果 *p* 是一个 tuple 对象但不是 tuple 类型的子类型的实例则返回真值。此函数总是会成功执行。

PyObject ***PyTuple_New**(*Py_ssize_t* *len*)

Return value: New reference. Part of the Stable ABI. 成功时返回一个新的元组对象，长度为 *len*，失败时返回 "NULL"。

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

Return value: New reference. Part of the Stable ABI. 成功时返回一个新的元组对象，大小为 *n* ，失败时返回 `NULL`。元组值初始化为指向 Python 对象的后续 *n* 个 C 参数。`PyTuple_Pack(2, a, b)` 和 `Py_BuildValue("(OO)", a, b)` 相等。

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

Part of the Stable ABI. 获取指向元组对象的指针，并返回该元组的大小。

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

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

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

Return value: Borrowed reference. Part of the Stable ABI. 返回 *p* 所指向的元组中位于 *pos* 处的对象。如果 *pos* 为负值或超出范围，则返回 `NULL` 并设置一个 `IndexError` 异常。

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

Return value: Borrowed reference. 类似于`PyTuple_GetItem()`，但不检查其参数。

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

Return value: New reference. Part of the Stable ABI. 返回 *p* 所指向的元组的切片，在 *low* 和 *high* 之间，或者在失败时返回 `NULL`。这等同于 Python 表达式 `p[low:high]`。不支持从列表末尾索引。

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

Part of the Stable ABI. 在 *p* 指向的元组的 *pos* 位置插入对对象 *o* 的引用。成功时返回 `0`；如果 *pos* 越界，则返回 `-1`，并抛出一个 `IndexError` 异常。

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

void **PyTuple_SET_ITEM**(*PyObject* **p*, *Py_ssize_t* *pos*, *PyObject* **o*)
类似于`PyTuple_SetItem()`，但不进行错误检查，并且应该 只是被用来填充全新的元组。

備F: 这个宏会“偷走”一个对 *o* 的引用，但与`PyTuple_SetItem()`不同，它 不会丢弃对任何被替换项的引用；元组中位于 *pos* 位置的任何引用都将被泄漏。

int **_PyTuple_Resize**(*PyObject* ***p*, *Py_ssize_t* *newsize*)
可以用于调整元组的大小。*newsize* 将是元组的新长度。因为元组 被认为是不可变的，所以只有在对象仅有一个引用时，才应该使用它。如果元组已经被代码的其他部分所引用，请不要使用此项。元组在最后总是会增长或缩小。把它看作是销毁旧元组并创建一个新元组，只会更有效。成功时返回 `0` 。客户端代码不应假定 `*p` 的结果值将与调用此函数之前的值相同。如果替换了 `*p` 引用的对象，则原始的 `*p` 将被销毁。失败时，返回“-1“，将 `*p` 设置为 `NULL`，并引发 `MemoryError` 或者 `SystemError`。

8.3.5 结构序列对象

结构序列对象是等价于 `namedtuple()` 的 C 对象，即一个序列，其中的条目也可以通过属性访问。要创建结构序列，你首先必须创建特定的结构序列类型。

PyTypeObject ***PyStructSequence_NewType**(*PyStructSequence_Desc* **desc*)
Return value: New reference. Part of the Stable ABI. 根据 *desc* 中的数据创建一个新的结构序列类型，如下所述。可以使用`PyStructSequence_New()`创建结果类型的实例。

void **PyStructSequence_InitType**(*PyTypeObject* **type*, *PyStructSequence_Desc* **desc*)
从 *desc* 就地初始化结构序列类型 *type*。

int **PyStructSequence_InitType2**(*PyTypeObject* **type*, *PyStructSequence_Desc* **desc*)
与 `PyStructSequence_InitType` 相同，但成功时返回 `0` ，失败时返回 `-1` 。

3.4 版新加入.

type **PyStructSequence_Desc**
Part of the Stable ABI *(including all members).* 包含要创建的结构序列类型的元信息。

域	C Type	含意
`name`	`const char *`	结构序列类型的名称
`doc`	`const char *`	指向要忽略类型的文档字符串或 `NULL` 的指针
`fields`	`PyStructSequence_Field *`	指向以 `NULL` 结尾的数组的指针，其字段名称为新类型
`n_in_sequence`	`int`	Python 侧可见的字段数（如果用作元组）

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.

域	C Type	含意
`name`	`const char *`	字段的名称或 `NULL` ，若要结束命名字段的列表，请设置为`PyStructSequence_UnnamedField` 以保留未命名字段
`doc`	`const char *`	要忽略的字段文档字符串或 `NULL`

const char ***const PyStructSequence_UnnamedField**

字段名的特殊值将保持未命名状态。

3.9 版更變: 这个类型已从 `char *` 更改。

PyObject ***PyStructSequence_New**(*PyTypeObject* **type*)

Return value: New reference. Part of the Stable ABI. 创建 *type* 的实例，该实例必须使用*PyStructSequence_NewType()* 创建。

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

Return value: Borrowed reference. Part of the Stable ABI. 返回 *p* 所指向的结构序列中，位于 *pos* 处的对象。不需要进行边界检查。

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

Return value: Borrowed reference. *PyStructSequence_GetItem()* 的宏版本。

void **PyStructSequence_SetItem**(*PyObject* **p*, *Py_ssize_t pos*, *PyObject* **o*)

Part of the Stable ABI. 将结构序列 *p* 的索引 *pos* 处的字段设置为值 *o*。与*PyTuple_SET_ITEM()* 一样，它应该只用于填充全新的实例。

備註： 这个函数 "窃取" 了指向 *o* 的一个引用。

void **PyStructSequence_SET_ITEM**(*PyObject* **p*, *Py_ssize_t* **pos*, *PyObject* **o*)

PyStructSequence_SetItem() 的宏版本。

備註： 这个函数 "窃取" 了指向 *o* 的一个引用。

8.3.6 List（串列）物件

type PyListObject

这个 C 类型*PyObject* 的子类型代表一个 Python 列表对象。

PyTypeObject **PyList_Type**

Part of the Stable ABI. 这是个属于*PyTypeObject* 的代表 Python 列表类型的实例。在 Python 层面和类型 `list` 是同一个对象。

int **PyList_Check**(*PyObject* **p*)

如果 *p* 是一个 list 对象或者 list 类型的子类型的实例则返回真值。此函数总是会成功执行。

int **PyList_CheckExact**(*PyObject* **p*)

如果 *p* 是一个 list 对象但不是 list 类型的子类型的实例则返回真值。此函数总是会成功执行。

PyObject ***PyList_New**(*Py_ssize_t len*)

Return value: New reference. Part of the Stable ABI. 成功时返回一个长度为 *len* 的新列表，失败时返回 `NULL`。

備註： 当 *len* 大于零时，被返回的列表对象项目被设成 `NULL`。因此你不能用类似 C 函数*PySequence_SetItem()* 的抽象 API 或者用 C 函数*PyList_SetItem()* 将所有项目设置成真实对象前对 Python 代码公开这个对象。

Py_ssize_t **PyList_Size**(*PyObject* **list*)

Part of the Stable ABI. 返回 *list* 中列表对象的长度；这等于在列表对象调用 `len(list)` 。

Py_ssize_t **PyList_GET_SIZE**(*PyObject* **list*)
宏版本的 C 函数`PyList_Size()`，没有错误检测。

PyObject ***PyList_GetItem**(*PyObject* **list*, *Py_ssize_t* *index*)
Return value: Borrowed reference. Part of the Stable ABI. 返回 *list* 所指向列表中 *index* 位置上的对象。位置值必须为非负数；不支持从列表末尾进行索引。如果 *index* 超出边界 (<0 or >=len(list))，则返回 `NULL` 并设置 `IndexError` 异常。

PyObject ***PyList_GET_ITEM**(*PyObject* **list*, *Py_ssize_t* *i*)
Return value: Borrowed reference. 宏版本的 C 函数`PyList_GetItem()`，没有错误检测。

int **PyList_SetItem**(*PyObject* **list*, *Py_ssize_t* *index*, *PyObject* **item*)
Part of the Stable ABI. 将列表中索引为 *index* 的项设为 *item*。成功时返回 `0`。如果 *index* 超出范围则返回 `-1` 并设定 `IndexError` 异常。

備註： 此函数会“偷走”一个对 *item* 的引用并丢弃一个对列表中受影响位置上的已有条目的引用。

void **PyList_SET_ITEM**(*PyObject* **list*, *Py_ssize_t* *i*, *PyObject* **o*)
不带错误检测的宏版本`PyList_SetItem()`。这通常只被用于新列表中之前没有内容的位置进行填充。

備註： 该宏会“偷走”一个对 *item* 的引用，但与`PyList_SetItem()`不同的是它 不会丢弃对任何被替换条目的引用；在 *list* 的 *i* 位置上的任何引用都将被泄露。

int **PyList_Insert**(*PyObject* **list*, *Py_ssize_t* *index*, *PyObject* **item*)
Part of the Stable ABI. 将条目 *item* 插入到列表 *list* 索引号 *index* 之前的位置。如果成功将返回 `0`；如果不成功则返回 `-1` 并设置一个异常。相当于 `list.insert(index, item)`。

int **PyList_Append**(*PyObject* **list*, *PyObject* **item*)
Part of the Stable ABI. 将对象 *item* 添加到列表 *list* 的末尾。如果成功将返回 `0`；如果不成功则返回 `-1` 并设置一个异常。相当于 `list.append(item)`。

PyObject ***PyList_GetSlice**(*PyObject* **list*, *Py_ssize_t* *low*, *Py_ssize_t* *high*)
Return value: New reference. Part of the Stable ABI. 返回一个对象列表，包含 *list* 当中位于 *low* 和 *high* 之间的对象。如果不成功则返回 `NULL` 并设置异常。相当于 `list[low:high]`。不支持从列表末尾进行索引。

int **PyList_SetSlice**(*PyObject* **list*, *Py_ssize_t* *low*, *Py_ssize_t* *high*, *PyObject* **itemlist*)
Part of the Stable ABI. 将 *list* 当中 *low* 与 *high* 之间的切片设为 *itemlist* 的内容。相当于 `list[low:high] = itemlist`。*itemlist* 可以为 `NULL`，表示赋值为一个空列表（删除切片）。成功时返回 `0`，失败时返回 `-1`。这里不支持从列表末尾进行索引。

int **PyList_Sort**(*PyObject* **list*)
Part of the Stable ABI. 对 *list* 中的条目进行原地排序。成功时返回 `0`，失败时返回 `-1`。这等价于 `list.sort()`。

int **PyList_Reverse**(*PyObject* **list*)
Part of the Stable ABI. 对 *list* 中的条目进行原地反转。成功时返回 `0`，失败时返回 `-1`。这等价于 `list.reverse()`。

PyObject ***PyList_AsTuple**(*PyObject* **list*)
Return value: New reference. Part of the Stable ABI. 返回一个新的元组对象，其中包含 *list* 的内容；等价于 `tuple(list)`。

8.4 容器物件

8.4.1 字典物件

type PyDictObject
: *PyObject* 子型態代表一個 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**()
: *Return value: New reference. Part of the* Stable ABI. 返回一个新的空字典，失败时返回 `NULL`。

PyObject ***PyDictProxy_New**(*PyObject* *mapping)
: *Return value: New reference. Part of the* Stable ABI. 返回 `types.MappingProxyType` 对象，用于强制执行只读行为的映射。这通常用于创建视图以防止修改非动态类类型的字典。

void **PyDict_Clear**(*PyObject* *p)
: *Part of the* Stable ABI. 清空现有字典的所有键值对。

int **PyDict_Contains**(*PyObject* *p, *PyObject* *key)
: *Part of the* Stable ABI. 确定 *key* 是否包含在字典 *p* 中。如果 *key* 匹配上 *p* 的某一项，则返回 `1`，否则返回 `0`。返回 `-1` 表示出错。这等同于 Python 表达式 `key in p`。

PyObject ***PyDict_Copy**(*PyObject* *p)
: *Return value: New reference. Part of the* Stable ABI. 返回与 *p* 包含相同键值对的新字典。

int **PyDict_SetItem**(*PyObject* *p, *PyObject* *key, *PyObject* *val)
: *Part of the* Stable ABI. 使用 *key* 作为键将 *val* 插入字典 *p*。*key* 必须为*hashable*；如果不是，则将引发 `TypeError`。成功时返回 `0`，失败时返回 `-1`。此函数 不会附带对 *val* 的引用。

int **PyDict_SetItemString**(*PyObject* *p, **const** char *key, *PyObject* *val)
: *Part of the* Stable ABI. Insert *val* into the dictionary *p* using *key* as a key. *key* should be a **const** `char*`. The key object is created using `PyUnicode_FromString(key)`. Return `0` on success or `-1` on failure. This function *does not* steal a reference to *val*.

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. 移除字典 *p* 中由字符串 *key* 指定的键的条目。如果字典中没有 *key*，则会引发 `KeyError`。成功时返回 `0`，失败时返回 `-1`。

PyObject ***PyDict_GetItem**(*PyObject* *p, *PyObject* *key)
: *Return value: Borrowed reference. Part of the* Stable ABI. 从字典 *p* 中返回以 *key* 为键的对象。如果键名 *key* 不存在但 没有设置一个异常则返回 `NULL`。

  需要注意的是，调用 `__hash__()` 和 `__eq__()` 方法产生的异常不会被抛出。改用*PyDict_GetItemWithError()* 获得错误报告。

  3.10 版更變: 在不保持*GIL* 的情况下调用此 API 曾因历史原因而被允许。现在已不再被允许。

PyObject ***PyDict_GetItemWithError**(*PyObject* **p*, *PyObject* **key*)
Return value: Borrowed reference. Part of the Stable ABI. `PyDict_GetItem()` 的变种，它不会屏蔽异常。当异常发生时将返回 `NULL` **并且**设置一个异常。如果键不存在则返回 `NULL` **并且不会**设置一个异常。

PyObject ***PyDict_GetItemString**(*PyObject* **p*, **const** char **key*)
Return value: Borrowed reference. Part of the Stable ABI. This is the same as `PyDict_GetItem()`, but *key* is specified as a **const** `char*`, rather than a `PyObject*`.

需要注意的是，调用 `__hash__()` 、`__eq__()` 方法和创建一个临时的字符串对象时产生的异常不会被抛出。改用`PyDict_GetItemWithError()` 获得错误报告。

PyObject ***PyDict_SetDefault**(*PyObject* **p*, *PyObject* **key*, *PyObject* **defaultobj*)
Return value: Borrowed reference. 这跟 Python 层面的 `dict.setdefault()` 一样。如果键 *key* 存在，它返回在字典 *p* 里面对应的值。如果键不存在，它会和值 *defaultobj* 一起插入并返回 *defaultobj* 。这个函数只计算 *key* 的哈希函数一次，而不是在查找和插入时分别计算它。

3.4 版新加入.

PyObject ***PyDict_Items**(*PyObject* **p*)
Return value: New reference. Part of the Stable ABI. 返回一个包含字典中所有键值项的`PyListObject`。

PyObject ***PyDict_Keys**(*PyObject* **p*)
Return value: New reference. Part of the Stable ABI. 返回一个包含字典中所有键 (keys) 的`PyListObject`。

PyObject ***PyDict_Values**(*PyObject* **p*)
Return value: New reference. Part of the Stable ABI. 返回一个包含字典中所有值 (values) 的`PyListObject`。

Py_ssize_t **PyDict_Size**(*PyObject* **p*)
Part of the Stable ABI. 返回字典中项目数，等价于对字典 *p* 使用 `len(p)`。

int **PyDict_Next**(*PyObject* **p*, *Py_ssize_t* **ppos*, *PyObject* ***pkey*, *PyObject* ***pvalue*)
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.

舉例來說:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}
```

字典 *p* 不应该在遍历期间发生改变。在遍历字典时，改变键中的值是安全的，但仅限于键的集合不发生改变。例如:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    long i = PyLong_AsLong(value);
    if (i == -1 && PyErr_Occurred()) {
        return -1;
    }
    PyObject *o = PyLong_FromLong(i + 1);
```

(下页继续)

(繼續上一頁)

```
    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. 对映射对象 *b* 进行迭代，将键值对添加到字典 *a*。*b* 可以是一个字典，或任何支持*PyMapping_Keys()* 和*PyObject_GetItem()* 的对象。如果 *override* 为真值，则如果在 *b* 中找到相同的键则 *a* 中已存在的相应键值对将被替换，否则如果在 *a* 中没有相同的键则只是添加键值对。当成功时返回 0 或者当引发异常时返回 -1。

int **PyDict_Update** (*PyObject* *a, *PyObject* *b)

Part of the Stable ABI. 这与 C 中的 PyDict_Merge(a, b, 1) 一样，也类似于 Python 中的 a.update(b)，差别在于*PyDict_Update()* 在第二个参数没有"keys" 属性时不会回退到迭代键值对的序列。当成功时返回 0 或者当引发异常时返回 -1。

int **PyDict_MergeFromSeq2** (*PyObject* *a, *PyObject* *seq2, int *override*)

Part of the Stable ABI. 将 *seq2* 中的键值对更新或合并到字典 *a*。*seq2* 必须为产生长度为 2 的用作键值对的元素的可迭代对象。当存在重复的键时，如果 *override* 真值则最后出现的键胜出。当成功时返回 0 或者当引发异常时返回 -1。等价的 Python 代码（返回值除外）：

```
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value
```

8.4.2 集合对象

这一节详细介绍了针对 set 和 frozenset 对象的公共 API。任何未在下面列出的功能最好是使用抽象对象协议 (包括*PyObject_CallMethod()*, *PyObject_RichCompareBool()*, *PyObject_Hash()*, *PyObject_Repr()*, *PyObject_IsTrue()*, *PyObject_Print()* 以及*PyObject_GetIter()*) 或者抽象数字协议 (包括*PyNumber_And()*, *PyNumber_Subtract()*, *PyNumber_Or()*, *PyNumber_Xor()*, *PyNumber_InPlaceAnd()*, *PyNumber_InPlaceSubtract()*, *PyNumber_InPlaceOr()* 以及*PyNumber_InPlaceXor()*)。

type PySetObject

这个*PyObject* 的子类型被用来保存 set 和 frozenset 对象的内部数据。它类似于*PyDictObject* 的地方在于对小尺寸集合来说它是固定大小的（很像元组的存储方式），而对于中等和大尺寸集合来说它将指向单独的可变大小的内存块（很像列表的存储方式）。此结构体的字段不应被视为公有并且可能发生改变。所有访问都应当通过已写入文档的 API 来进行而不可通过直接操纵结构体中的值。

PyTypeObject **PySet_Type**

Part of the Stable ABI. 这是一个*PyTypeObject* 实例，表示 Python set 类型。

PyTypeObject **PyFrozenSet_Type**

Part of the Stable ABI. 这是一个*PyTypeObject* 实例，表示 Python frozenset 类型。

下列类型检查宏适用于指向任意 Python 对象的指针。类似地，这些构造函数也适用于任意可迭代的 Python 对象。

int **PySet_Check** (*PyObject* *p)

如果 *p* 是一个 set 对象或者是其子类型的实例则返回真值。此函数总是会成功执行。

int **PyFrozenSet_Check** (*PyObject* *p)
如果 *p* 是一个 `frozenset` 对象或者是其子类型的实例则返回真值。此函数总是会成功执行。

int **PyAnySet_Check** (*PyObject* *p)
如果 *p* 是一个 `set` 对象、`frozenset` 对象或者是其子类型的实例则返回真值。此函数总是会成功执行。

int **PySet_CheckExact** (*PyObject* *p)
如果 *p* 是一个 `set` 对象但不是其子类型的实例则返回真值。此函数总是会成功执行。

3.10 版新加入.

int **PyAnySet_CheckExact** (*PyObject* *p)
如果 *p* 是一个 `set` 或 `frozenset` 对象但不是其子类型的实例则返回真值。此函数总是会成功执行。

int **PyFrozenSet_CheckExact** (*PyObject* *p)
如果 *p* 是一个 `frozenset` 对象但不是其子类型的实例则返回真值。此函数总是会成功执行。

PyObject ***PySet_New** (*PyObject* *iterable)
Return value: New reference. Part of the Stable ABI. 返回一个新的 `set`，其中包含 *iterable* 所返回的对象。*iterable* 可以为 `NULL` 表示创建一个新的空集合。成功时返回新的集合，失败时返回 `NULL`。如果 *iterable* 实际上不是可迭代对象则引发 `TypeError`。该构造器也适用于拷贝集合 (`c=set(s)`)。

PyObject ***PyFrozenSet_New** (*PyObject* *iterable)
Return value: New reference. Part of the Stable ABI. 返回一个新的 `frozenset`，其中包含 *iterable* 所返回的对象。*iterable* 可以为 `NULL` 表示创建一个新的空冻结集合。成功时返回新的冻结集合，失败时返回 `NULL`。如果 *iterable* 实际上不是可迭代对象则引发 `TypeError`。

下列函数和宏适用于 `set` 或 `frozenset` 的实例或是其子类型的实例。

Py_ssize_t **PySet_Size** (*PyObject* *anyset)
Part of the Stable ABI. 返回 `set` 或 `frozenset` 对象的长度。等价于 `len(anyset)`。如果 *anyset* 不是 `set`, `frozenset` 或其子类型的实例则会引发 `PyExc_SystemError`。

Py_ssize_t **PySet_GET_SIZE** (*PyObject* *anyset)
宏版本的 `PySet_Size()`，不带错误检测。

int **PySet_Contains** (*PyObject* *anyset, *PyObject* *key)
Part of the Stable ABI. 如果找到返回 `1`，如果未找到返回 `0`，如果遇到错误则返回 `-1`。不同于 Python `__contains__()` 方法，此函数不会自动将不可哈希的集合转换为临时的冻结集合。如果 *key* 为不可哈希对象则会引发 `TypeError`。如果 *anyset* 不是 `set`, `frozenset` 或其子类型的实例则会引发 `PyExc_SystemError`。

int **PySet_Add** (*PyObject* *set, *PyObject* *key)
Part of the Stable ABI. 添加 *key* 到一个 `set` 实例。也可用于 `frozenset` 实例 (与 `PyTuple_SetItem()` 的类似之处是它也可被用来为全新的冻结集合在公开给其他代码之前填充全新的值)。成功时返回 `0` 而失败时返回 `-1`。如果 *key* 为不可哈希对象则会引发 `TypeError`。如果没有增长空间则会引发 `MemoryError`。如果 *set* 不是 `set` 或其子类型的实例则会引发 `SystemError`。

下列函数适用于 `set` 或其子类型的实例，但不可用于 `frozenset` 或其子类型的实例。

int **PySet_Discard** (*PyObject* *set, *PyObject* *key)
Part of the Stable ABI. 如果找到并移除返回 `1`，如果未找到 (无操作) 返回 `0`，如果遇到错误则返回 `-1`。对于不存在的键不会引发 `KeyError`。如果 *key* 为不可哈希对象则会引发 `TypeError`。不同于 Python `discard()` 方法，此函数不会自动将不可哈希的集合转换为临时的冻结集合。如果 *set* 不是 `set` 或其子类型的实例则会引发 `PyExc_SystemError`。

PyObject ***PySet_Pop** (*PyObject* *set)
Return value: New reference. Part of the Stable ABI. 返回 *set* 中任意对象的新引用，并从 *set* 中移除该对象。失败时返回 `NULL`。如果集合为空则会引发 `KeyError`。如果 *set* 不是 `set` 或其子类型的实例则会引发 `SystemError`。

int **PySet_Clear**(*PyObject* *set)
Part of the Stable ABI. 清空现有字典的所有键值对。

8.5 函式物件

8.5.1 函式（Function）物件

這有一些少數 Python 函數的於具體說明。

type PyFunctionObject
用于函数的 C 结构体。

PyTypeObject **PyFunction_Type**
这是一个`PyTypeObject` 实例并表示 Python 函数类型。它作为 `types.FunctionType` 向 Python 程序员公开。

int **PyFunction_Check**(*PyObject* *o)
如果 *o* 是一个函数对象 (类型为`PyFunction_Type`) 则返回真值。形参必须不为 `NULL`。此函数总是会成功执行。

PyObject ***PyFunction_New**(*PyObject* *code, *PyObject* *globals)
Return value: New reference. 返回与代码对象 *code* 关联的新函数对象。*globals* 必须是一个字典，该函数可以访问全局变量。

从代码对象中提取函数的文档字符串和名称。*__module__* 会从 *globals* 中提取。参数 defaults, annotations 和 closure 设为 `NULL`。*__qualname__* 设为与函数名称相同的值。

PyObject ***PyFunction_NewWithQualName**(*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)
Return value: New reference. 类似`PyFunction_New()`，但还允许设置函数对象的 `__qualname__` 属性。*qualname* 应当是 unicode 对象或 `NULL`；如果是 `NULL` 则 `__qualname__` 属性设为与其 `__name__` 属性相同的值。

3.3 版新加入.

PyObject ***PyFunction_GetCode**(*PyObject* *op)
Return value: Borrowed reference. 回傳與程式碼物件相關的函數物件 *op*。

PyObject ***PyFunction_GetGlobals**(*PyObject* *op)
Return value: Borrowed reference. 回傳與全域函數字典相關的函數物件 *op*。

PyObject ***PyFunction_GetModule**(*PyObject* *op)
Return value: Borrowed reference. 向函数对象 *op* 的 *__module__* 属性返回一个*borrowed reference*。该值可以为 *NULL*。

这通常为一个包含模块名称的字符串，但可以通过 Python 代码设为任何其他对象。

PyObject ***PyFunction_GetDefaults**(*PyObject* *op)
Return value: Borrowed reference. 返回函数对象 *op* 的参数默认值。这可以是一个参数元组或 `NULL`。

int **PyFunction_SetDefaults**(*PyObject* *op, *PyObject* *defaults)
为函数对象 *op* 设置参数默认值。*defaults* 必须为 `Py_None` 或一个元组。

失败时引发 `SystemError` 异常并返回 `-1` 。

PyObject ***PyFunction_GetClosure**(*PyObject* *op)
Return value: Borrowed reference. 返回关联到函数对象 *op* 的闭包。这可以是 `NULL` 或 cell 对象的元组。

int **PyFunction_SetClosure**(*PyObject* *op, *PyObject* *closure)
设置关联到函数对象 *op* 的闭包。*closure* 必须为 `Py_None` 或 cell 对象的元组。

失败时引发 `SystemError` 异常并返回 `-1` 。

PyObject ***PyFunction_GetAnnotations**(*PyObject* **op*)
: *Return value: Borrowed reference.* 返回函数对象 *op* 的标注。这可以是一个可变字典或 `NULL`。

int **PyFunction_SetAnnotations**(*PyObject* **op*, *PyObject* **annotations*)
: 设置函数对象 *op* 的标注。*annotations* 必须为一个字典或 `Py_None`。

失败时引发 `SystemError` 异常并返回 `-1` 。

8.5.2 實例方法物件 (Instance Method Objects)

實例方法是*PyCFunction* 的包裝器 (wrapper)，也是將*PyCFunction* □結 (bind) 到類□物件的一種新方式。它替代了原先對 `PyMethod_New(func, NULL, class)` 的呼叫。

PyTypeObject **PyInstanceMethod_Type**
: *PyTypeObject* 的實例代表 Python 實例方法型□。它不會公開 (expose) 給 Python 程式。

int **PyInstanceMethod_Check**(*PyObject* **o*)
: 如果 *o* 是一個實例方法物件（型□□*PyInstanceMethod_Type*）則回傳 true。參數必須不□ `NULL`。此函式總是會成功執行。

PyObject ***PyInstanceMethod_New**(*PyObject* **func*)
: *Return value: New reference.* 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*)
: *Return value: Borrowed reference.* 回傳關聯到實例方法 *im* 的函式物件。

PyObject ***PyInstanceMethod_GET_FUNCTION**(*PyObject* **im*)
: *Return value: Borrowed reference.* 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*)
: *Return value: New reference.* 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*)
: *Return value: Borrowed reference.* 回傳關聯到方法 *meth* 的函式物件。

PyObject ***PyMethod_GET_FUNCTION**(*PyObject* **meth*)
: *Return value: Borrowed reference.* 巨集版本的*PyMethod_Function()*，忽略了錯誤檢查。

PyObject ***PyMethod_Self**(*PyObject* **meth*)
: *Return value: Borrowed reference.* 回傳關聯到方法 *meth* 的實例。

PyObject ***PyMethod_GET_SELF**(*PyObject *meth*)

Return value: Borrowed reference. 巨集版本的`PyMethod_Self()`，忽略了錯誤檢查。

8.5.4 Cell 物件

"Cell" 对象用于实现由多个作用域引用的变量。对于每个这样的变量，一个 "Cell" 对象为了存储该值而被创建；引用该值的每个堆栈框架的局部变量包含同样使用该变量的对外部作用域的 "Cell" 引用。访问该值时，将使用 "Cell" 中包含的值而不是单元格对象本身。这种对 "Cell" 对象的非关联化的引用需要支持生成的字节码；访问时不会自动非关联化这些内容。"Cell" 对象在其他地方可能不太有用。

type PyCellObject

C 結構的 cell 物件

PyTypeObject **PyCell_Type**

對應 cell 物件的物件型F。

int **PyCell_Check**(ob)

如果 *ob* 是一个 cell 对象则返回真值；*ob* 必须不为 `NULL`。此函数总是会成功执行。

PyObject ***PyCell_New**(*PyObject *ob*)

Return value: New reference. 创建并返回一个包含值 *ob* 的新 cell 对象。形参可以为 `NULL`。

PyObject ***PyCell_Get**(*PyObject *cell*)

Return value: New reference. 回傳 cell F容中的 *cell*。

PyObject ***PyCell_GET**(*PyObject *cell*)

Return value: Borrowed reference. 返回 cell 对象 *cell* 的内容，但是不检测 *cell* 是否非 `NULL` 并且为一个 cell 对象。

int **PyCell_Set**(*PyObject *cell*, *PyObject *value*)

将 cell 对象 *cell* 的内容设为 *value*。这将释放任何对 cell 对象当前内容的引用。*value* 可以为 `NULL`。*cell* 必须为非 `NULL`；如果它不是一个 cell 对象则将返回 `-1`。如果设置成功则将返回 `0`。

void **PyCell_SET**(*PyObject *cell*, *PyObject *value*)

将 cell 对象 *cell* 的值设为 *value*。不会调整引用计数，并且不会进行检测以保证安全；*cell* 必须为非 `NULL` 并且为一个 cell 对象。

8.5.5 代码对象

代码对象是 CPython 实现的低级细节。每个代表一块尚未绑定到函数中的可执行代码。

type PyCodeObject

用于描述代码对象的对象的 C 结构。此类型字段可随时更改。

PyTypeObject **PyCode_Type**

这是一个`PyTypeObject` 实例，其表示 Python 的 `code` 类型。

int **PyCode_Check**(*PyObject *co*)

如果 *co* 是一个 `code` 对象则返回真值。此函数总是会成功执行。

int **PyCode_GetNumFree**(*PyCodeObject *co*)

返回 *co* 中的自由变量数。

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*, int *firstlineno*, *PyObject *lnotab*)

Return value: New reference. 返回一个新的代码对象。如果你需要一个虚拟代码对象来创建一个代码帧，

请使用`PyCode_NewEmpty()`。调用`PyCode_New()` 直接可以绑定到准确的 Python 版本，因为字节码的定义经常变化。

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*, int *firstlineno*, *PyObject* *lnotab*)

Return value: New reference. 类似于`PyCode_New()`，但带有一个额外的"posonlyargcount" 用于仅限位置参数。

3.8 版新加入.

PyCodeObject ***PyCode_NewEmpty**(**const** char **filename*, **const** char **funcname*, int *firstlineno*)

Return value: New reference. 返回具有指定文件名、函数名和第一行号的新空代码对象。对于 `exec()` 或 `eval()` 生成的代码对象是非法的。

int **PyCode_Addr2Line**(*PyCodeObject* **co*, int *byte_offset*)

返回在 `byte_offset` 位置或之前以及之后发生的指令的行号。如果你只需要一个帧的行号，请改用`PyFrame_GetLineNumber()`。

要高效地迭代一个代码对象中的行号，请使用 PEP 626 描述的 API。

8.6 其他物件

8.6.1 檔案（File）物件

These APIs are a minimal emulation of the Python 2 C API for built-in file objects, which used to rely on the buffered I/O (`FILE*`) support from the C standard library. In Python 3, files and streams use the new `io` module, which defines several layers over the low-level unbuffered I/O of the operating system. The functions described below are convenience C wrappers over these new APIs, and meant mostly for internal error reporting in the interpreter; third-party code is advised to access the `io` APIs instead.

PyObject ***PyFile_FromFd**(int *fd*, **const** char **name*, **const** char **mode*, int *buffering*, **const** char **encoding*, **const** char **errors*, **const** char **newline*, int *closefd*)

Return value: New reference. Part of the Stable ABI. 根据已打开文件 *fd* 的文件描述符创建一个 Python 文件对象。参数 *name*, *encoding*, *errors* 和 *newline* 可以为 `NULL` 表示使用默认值；*buffering* 可以为 *-1* 表示使用默认值。*name* 会被忽略仅保留用于向下兼容。失败时返回 `NULL`。有关参数的更全面描述，请参阅 `io.open()` 函数的文档。

> **警告：** 由于 Python 流具有自己的缓冲层，因此将它们与 OS 级文件描述符混合会产生各种问题（例如数据的意外排序）。

3.2 版更變: 忽略 *name* 屬性。

int **PyObject_AsFileDescriptor**(*PyObject* **p*)

Part of the Stable ABI. Return the file descriptor associated with *p* as an `int`. If the object is an integer, its value is returned. If not, the object's `fileno()` method is called if it exists; the method must return an integer, which is returned as the file descriptor value. Sets an exception and returns `-1` on failure.

PyObject ***PyFile_GetLine**(*PyObject* **p*, int *n*)

Return value: New reference. Part of the Stable ABI. 等价于 `p.readline([n])`，这个函数从对象 *p* 中读取一行。*p* 可以是文件对象或具有 `readline()` 方法的任何对象。如果 *n* 是 `0`，则无论该行的长度如何，都会读取一行。如果 *n* 大于 "0"，则从文件中读取不超过 *n* 个字节；可以返回行的一部分。在这两

种情况下，如果立即到达文件末尾，则返回空字符串。但是，如果 *n* 小于 0 ，则无论长度如何都会读取一行，但是如果立即到达文件末尾，则引发 `EOFError`。

int **PyFile_SetOpenCodeHook**(Py_OpenCodeHookFunction *handler*)

重载 `io.open_code()` 的正常行为，将其形参通过所提供的处理程序来传递。

The handler is a function of type `PyObject *(*)PyObject *path, void *userData`, where *path* is guaranteed to be `PyUnicodeObject`.

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

鉴于这个钩子专门在导入期间使用的，请避免在新模块执行期间进行导入操作，除非已知它们为冻结状态或者是在 `sys.modules` 中可用。

一旦钩子被设定，它就不能被移除或替换，之后对`PyFile_SetOpenCodeHook()` 的调用也将失败，如果解释器已经被初始化，函数将返回 -1 并设置一个异常。

此函数可以安全地在`Py_Initialize()` 之前调用。

引发一个 审计事件 `setopencodehook`，不附带任何参数。

3.8 版新加入.

int **PyFile_WriteObject**(PyObject **obj*, PyObject **p*, int *flags*)

Part of the Stable ABI. 将对象 *obj* 写入文件对象 *p* 。*flags* 唯一支持的标志是 `Py_PRINT_RAW`；如果给定，则写入对象的 `str()` 而不是 `repr()`。成功时返回 `0`，失败时返回 `-1`。将设置适当的例外。

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，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. 这个 C 类型实例`PyTypeObject` 用来表示 Python 中的模块类型。在 Python 程序中该实例被暴露为 "types.ModuleType"。

int **PyModule_Check**(PyObject **p*)

当 *p* 为模块类型的对象，或是模块子类型的对象时返回真值。该函数永远有返回值。

int **PyModule_CheckExact**(PyObject **p*)

当 *p* 为模块类型的对象且不是`PyModule_Type` 的子类型的对象时返回真值。该函数永远有返回值。

PyObject ***PyModule_NewObject**(PyObject **name*)

Return value: New reference. Part of the Stable ABI *since version 3.7.* 返回新的模块对象，其属性 `__name__` 为 *name* 。模块的如下属性 `__name__`, `__doc__`, `__package__`, and `__loader__` 都会被自动填充。（所有属性除了 `__name__` 都被设为 "None"）。调用时应当提供 `__file__` 属性。

3.3 版新加入.

3.4 版更變: `__package__` 和 `__loader__` 被設為 `None`。

PyObject ***PyModule_New**(const char **name*)

Return value: New reference. Part of the Stable ABI. 这类似于`PyModule_NewObject()`，但其名称为 UTF-8 编码的字符串而不是 Unicode 对象。

PyObject ***PyModule_GetDict**(PyObject **module*)

Return value: Borrowed reference. Part of the Stable ABI. 返回实现 *module* 的命名空间的字典对象；此对象与模块对象的 `__dict__` 属性相同。如果 *module* 不是一个模块对象（或模块对象的子类型），则会引发 `SystemError` 并返回 `NULL`。

It is recommended extensions use other `PyModule_*` and `PyObject_*` functions rather than directly manipulate a module's `__dict__`.

PyObject ***PyModule_GetNameObject**(*PyObject* **module*)
Return value: New reference. Part of the Stable ABI *since version 3.7.* 返回 *module* 的 `__name__` 值。如果模块未提供该值，或者如果它不是一个字符串，则会引发 `SystemError` 并返回 `NULL`。

3.3 版新加入.

const char ***PyModule_GetName**(*PyObject* **module*)
Part of the Stable ABI. 类似于 *PyModule_GetNameObject()* 但返回 `'utf-8'` 编码的名称。

void ***PyModule_GetState**(*PyObject* **module*)
Part of the Stable ABI. 返回模块的"状态"，也就是说，返回指向在模块创建时分配的内存块的指针，或者 `NULL`。参见 *PyModuleDef.m_size*。

PyModuleDef ***PyModule_GetDef**(*PyObject* **module*)
Part of the Stable ABI. 返回指向模块创建所使用的 *PyModuleDef* 结构体的指针，或者如果模块不是使用结构体定义创建的则返回 `NULL`。

PyObject ***PyModule_GetFilenameObject**(*PyObject* **module*)
Return value: New reference. Part of the Stable ABI. 返回使用 *module* 的 `__file__` 属性所加载的 模块的文件名。如果属性未定义，或者如果它不是一个 Unicode 字符串，则会引发 `SystemError` 并返回 `NULL`；在其他情况下将返回一个指向 Unicode 对象的引用。

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'.

3.2 版後已棄用: *PyModule_GetFilename()* raises `UnicodeEncodeError` on unencodable filenames, use *PyModule_GetFilenameObject()* instead.

初始化 C 模块

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**
新模块的名称。

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`.

3.5 版更變: 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`.

3.9 版更變: 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.

3.9 版更變: 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`.

3.9 版更變: 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*)

Return value: New reference. 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*)

Return value: New reference. 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*)

Return value: Borrowed reference. 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.

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.

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*)
Return value: New reference. 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`.

3.5 版新加入.

PyObject ***PyModule_FromDefAndSpec2**(*PyModuleDef* **def*, *PyObject* **spec*, int *module_api_version*)
Return value: New reference. Part of the Stable ABI *since version 3.7.* Create a new module object, given the definition in *def* and the ModuleSpec *spec*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

備F: Most uses of this function should be using *PyModule_FromDefAndSpec()* instead; only use this if you are sure you need it.

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*.

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.

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.

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)
```

(下页继续)

(繼續上一頁)

```
{
    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`.

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 decrements the reference count of *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) {
        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.

int **PyModule_AddIntMacro** (*PyObject* **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.

int **PyModule_AddStringMacro** (*PyObject* **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.

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*)

Return value: Borrowed reference. 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).

调用时必须携带 GIL。

Return 0 on success or -1 on failure.

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.

调用时必须携带 GIL。

3.3 版新加入.

8.6.3 F代器（Iterator）物件

Python 提供了两个通用迭代器对象。第一个是序列迭代器，它使用支持 `__getitem__()` 方法的任意序列。第二个使用可调用对象和一个 sentinel 值，为序列中的每个项调用可调用对象，并在返回 sentinel 值时结束迭代。

PyTypeObject **PySeqIter_Type**
: *Part of the* Stable ABI. *PySeqIter_New()* 返回迭代器对象的类型对象和内置序列类型内置函数 `iter()` 的单参数形式。

int **PySeqIter_Check**(op)
: 如果 *op* 的类型为 *PySeqIter_Type* 则返回真值。此函数总是会成功执行。

PyObject ***PySeqIter_New**(*PyObject* **seq*)
: *Return value: New reference. Part of the* Stable ABI. 返回一个与常规序列对象一起使用的迭代器 *seq*。当序列订阅操作引发 `IndexError` 时，迭代结束。

PyTypeObject **PyCallIter_Type**
: *Part of the* Stable ABI. 由函数 *PyCallIter_New()* 和 `iter()` 内置函数的双参数形式返回的迭代器对象类型对象。

int **PyCallIter_Check**(op)
: 如果 *op* 的类型为 *PyCallIter_Type* 则返回真值。此函数总是会成功执行。

PyObject ***PyCallIter_New**(*PyObject* **callable*, *PyObject* **sentinel*)
: *Return value: New reference. Part of the* Stable ABI. 返回一个新的迭代器。第一个参数 *callable* 可以是任何可以在没有参数的情况下调用的 Python 可调用对象；每次调用都应该返回迭代中的下一个项目。当 *callable* 返回等于 *sentinel* 的值时，迭代将终止。

8.6.4 Descriptor（描述器）物件

"Descriptor" 是描述物件某些屬性的物件，它們存在於型F物件的 dictionary（字典）中。

PyTypeObject **PyProperty_Type**
: *Part of the* Stable ABI. F建 descriptor 型F的型F物件。

PyObject ***PyDescr_NewGetSet**(*PyTypeObject* **type*, **struct** *PyGetSetDef* **getset*)
: *Return value: New reference. Part of the* Stable ABI.

PyObject ***PyDescr_NewMember**(*PyTypeObject* **type*, **struct** *PyMemberDef* **meth*)
: *Return value: New reference. Part of the* Stable ABI.

PyObject ***PyDescr_NewMethod**(*PyTypeObject* **type*, **struct** *PyMethodDef* **meth*)
: *Return value: New reference. Part of the* Stable ABI.

PyObject ***PyDescr_NewWrapper**(*PyTypeObject* **type*, **struct** wrapperbase **wrapper*, void **wrapped*)
: *Return value: New reference.*

PyObject ***PyDescr_NewClassMethod**(*PyTypeObject* **type*, *PyMethodDef* **method*)
: *Return value: New reference. Part of the* Stable ABI.

int **PyDescr_IsData**(*PyObject* **descr*)
: 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F一個 descriptor 物件；F有錯誤檢查。

PyObject ***PyWrapper_New**(*PyObject**, *PyObject**)
: *Return value: New reference. Part of the* Stable ABI.

8.6.5 切片物件

PyTypeObject **PySlice_Type**

Part of the Stable ABI. 切片对象的类型对象。它与 Python 层面的 `slice` 是相同的对象。

int **PySlice_Check** (*PyObject* *ob)

如果 *ob* 是一个 slice 对象则返回真值；*ob* 必须不为 `NULL`。此函数总是会成功执行。

PyObject ***PySlice_New** (*PyObject* *start, *PyObject* *stop, *PyObject* *step)

Return value: New reference. Part of the Stable ABI. 返回一个具有给定值的新切片对象。*start*, *stop* 和 *step* 形参会被用作 slice 对象相应名称的属性的值。这些值中的任何一个都可以为 `NULL`，在这种情况下将使用 `None` 作为对应属性的值。如果新对象无法被分配则返回 `NULL`。

int **PySlice_GetIndices** (*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)

Part of the Stable ABI. 从切片对象 *slice* 提取 start, stop 和 step 索引号，将序列长度视为 *length*。大于 *length* 的序列号将被当作错误。

成功时返回 `0`，出错时返回 `-1` 并且不设置异常（除非某个序列号不为 `None` 且无法被转换为整数，在这种情况下会返回 `-1` 并且设置一个异常)。

你可能不会打算使用此函数。

3.2 版更變: 之前 *slice* 形参的形参类型是 `PySliceObject*`。

int **PySlice_GetIndicesEx** (*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step, *Py_ssize_t* *slicelength)

Part of the Stable ABI. `PySlice_GetIndices()` 的可用替代。从切片对象 *slice* 提取 start, stop 和 step 索引号，将序列长度视为 *length*，并将切片的长度保存在 *slicelength* 中，超出范围的索引号会以与普通切片一致的方式进行剪切。

成功时返回 `0`，出错时返回 `-1` 并且不设置异常。

備註： 此函数对于可变大小序列来说是不安全的。对它的调用应被替换为`PySlice_Unpack()` 和`PySlice_AdjustIndices()` 的组合，其中

```
if (PySlice_GetIndicesEx(slice, length, &start, &stop, &step, &slicelength) < 0) {
    // return error
}
```

会被替换为

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {
    // return error
}
slicelength = PySlice_AdjustIndices(length, &start, &stop, step);
```

3.2 版更變: 之前 *slice* 形参的形参类型是 `PySliceObject*`。

3.6.1 版更變: 如果 `Py_LIMITED_API` 未设置或设置为 `0x03050400` 与 `0x03060000` 之间的值（不包括边界）或 `0x03060100` 或更大则 `PySlice_GetIndicesEx()` 会被实现为一个使用 `PySlice_Unpack()` 和 `PySlice_AdjustIndices()` 的宏。参数 *start*, *stop* 和 *step* 会被多被求值。

3.6.1 版後已棄用: 如果 `Py_LIMITED_API` 设置为小于 `0x03050400` 或 `0x03060000` 与 `0x03060100` 之间的值（不包括边界）则 `PySlice_GetIndicesEx()` 为已弃用的函数。

int **PySlice_Unpack** (*PyObject* *slice, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)

Part of the Stable ABI *since version 3.7.* 从切片对象中将 start, stop 和 step 数据成员提取为 C 整数。会静默地将大于 `PY_SSIZE_T_MAX` 的值减小为 `PY_SSIZE_T_MAX`，静默地将小于 `PY_SSIZE_T_MIN`

的 start 和 stop 值增大为 `PY_SSIZE_T_MIN`，并静默地将小于 `-PY_SSIZE_T_MAX` 的 step 值增大为 `-PY_SSIZE_T_MAX`。

出错时返回 `-1`，成功时返回 `0`。

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.* 将 start/end 切片索引号根据指定的序列长度进行调整。超出范围的索引号会以与普通切片一致的方式进行剪切。

返回切片的长度。此操作总是会成功。不会调用 Python 代码。

3.6.1 版新加入.

8.6.6 Ellipsis 对象

PyObject ***Py_Ellipsis**

Python 的 `Ellipsis` 对象。该对象没有任何方法。它必须以与任何其他对象一样的方式遵循引用计数。它与 `Py_None` 一样属于单例对象。

8.6.7 MemoryView 物件

一个 `memoryview` 对象 C 级别的缓冲区接口 暴露为一个可以像任何其他对象一样传递的 Python 对象。

PyObject ***PyMemoryView_FromObject** (*PyObject *obj*)

Return value: New reference. Part of the Stable ABI. 从提供缓冲区接口的对象创建 memoryview 对象。如果 *obj* 支持可写缓冲区导出，则 memoryview 对象将可以被读/写，否则它可能是只读的，也可以是导出器自行决定的读/写。

PyObject ***PyMemoryView_FromMemory** (char **mem*, *Py_ssize_t size*, int *flags*)

Return value: New reference. Part of the Stable ABI *since version 3.7.* 使用 *mem* 作为底层缓冲区创建一个 memoryview 对象。*flags* 可以是 `PyBUF_READ` 或者 `PyBUF_WRITE` 之一.

3.3 版新加入.

PyObject ***PyMemoryView_FromBuffer** (*Py_buffer *view*)

Return value: New reference. 创建一个包含给定缓冲区结构 *view* 的 memoryview 对象。对于简单的字节缓冲区，`PyMemoryView_FromMemory()` 是首选函数。

PyObject ***PyMemoryView_GetContiguous** (*PyObject *obj*, int *buffertype*, char *order*)

Return value: New reference. Part of the Stable ABI. 从定义缓冲区接口的对象创建一个 memoryview 对象*contiguous* 内存块（在'C' 或'Fortran *order* 中）。如果内存是连续的，则 memoryview 对象指向原始内存。否则，复制并且 memoryview 指向新的 bytes 对象。

int **PyMemoryView_Check** (*PyObject *obj*)

如果 *obj* 是一个 memoryview 对象则返回真值。目前不允许创建 `memoryview` 的子类。此函数总是会成功执行。

Py_buffer ***PyMemoryView_GET_BUFFER** (*PyObject *mview*)

返回指向 memoryview 的导出缓冲区私有副本的指针。*mview* **必须**是一个 memoryview 实例；这个宏不检查它的类型，你必须自己检查，否则你将面临崩溃风险。

PyObject ***PyMemoryView_GET_BASE** (*PyObject *mview*)

返回 memoryview 所基于的导出对象的指针，或者如果 memoryview 已由函数`PyMemoryView_FromMemory()` 或`PyMemoryView_FromBuffer()` 创建则返回 `NULL`。*mview* **必须**是一个 memoryview 实例。

8.6.8 弱参照物件

Python 支持“弱引用”作为一类对象。具体来说，有两种直接实现弱引用的对象。第一种就是简单的引用对象，第二种尽可能地作用为一个原对象的代理。

int **PyWeakref_Check**(ob)
如果 *ob* 是一个引用或代理对象则返回真值。此函数总是会成功执行。

int **PyWeakref_CheckRef**(ob)
如果 *ob* 是一个引用对象则返回真值。此函数总是会成功执行。

int **PyWeakref_CheckProxy**(ob)
如果 *ob* 是一个代理对象则返回真值。此函数总是会成功执行。

PyObject ***PyWeakref_NewRef**(*PyObject* **ob*, *PyObject* **callback*)
Return value: New reference. 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*)
Return value: New reference. 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*)
Return value: Borrowed reference. Part of the Stable ABI. 返回弱引用对象 ref 的被引用对象。如果被引用对象不再存在，则返回 `Py_None`。

備註： 该函数返回被引用对象的一个*borrowed reference*。这意味着应该总是在该对象上调用*Py_INCREF()*，除非是当它在借入引用的最后一次被使用之前无法被销毁的时候。

PyObject ***PyWeakref_GET_OBJECT**(*PyObject* **ref*)
Return value: Borrowed reference. 类似*PyWeakref_GetObject()*，但实现为一个不做类型检查的宏。

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.

8.6.9 Capsule 对象

有关使用这些对象的更多信息请参阅 using-capsules。

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. Capsule 的析构器回调的类型。定义如下：

```
typedef void (*PyCapsule_Destructor)(PyObject *);
```

参阅 `PyCapsule_New()` 来获取 PyCapsule_Destructor 返回值的语义。

int **PyCapsule_CheckExact** (*PyObject *p*)

如果参数是一个 `PyCapsule` 则返回真值。此函数总是会成功执行。

PyObject ***PyCapsule_New** (void **pointer*, **const** char **name*, *PyCapsule_Destructor destructor*)

Return value: New reference. Part of the Stable ABI. 创建一个封装了 *pointer* 的 `PyCapsule`。*pointer* 参考可以不为 `NULL`。

在失败时设置一个异常并返回 `NULL`。

字符串 *name* 可以是 `NULL` 或是一个指向有效的 C 字符串的指针。如果不为 `NULL`，则此字符串必须比 capsule 长（虽然也允许在 *destructor* 中释放它。）

如果 *destructor* 参数不为 `NULL`，则当它被销毁时将附带 capsule 作为参数来调用。

如果此 capsule 将被保存为一个模块的属性，则 *name* 应当被指定为 `modulename.attributename`。这将允许其他模块使用 `PyCapsule_Import()` 来导入此 capsule。

void ***PyCapsule_GetPointer** (*PyObject *capsule*, **const** char **name*)

Part of the Stable ABI. 提取保存在 capsule 中的 *pointer*。在失败时设置一个异常并返回 `NULL`。

name 形参必须与保存在 capsule 中的名称进行精确比较。如果保存在 capsule 中的名称为 `NULL`，则传入的 *name* 也必须为 `NULL`。Python 会使用 C 函数 `strcmp()` 来比较 capsule 名称。

PyCapsule_Destructor **PyCapsule_GetDestructor** (*PyObject *capsule*)

Part of the Stable ABI. 返回保存在 capsule 中的当前析构器。在失败时设置一个异常并返回 `NULL`。

capsule 具有 `NULL` 析构器是合法的。这会使得 `NULL` 返回码有些歧义；请使用 `PyCapsule_IsValid()` 或 `PyErr_Occurred()` 来消除歧义。

void ***PyCapsule_GetContext** (*PyObject *capsule*)

Part of the Stable ABI. 返回保存在 capsule 中的当前上下文。在失败时设置一个异常并返回 `NULL`。

capsule 具有 `NULL` 上下文是全法的。这会使得 `NULL` 返回码有些歧义；请使用 `PyCapsule_IsValid()` 或 `PyErr_Occurred()` 来消除歧义。

const char ***PyCapsule_GetName** (*PyObject *capsule*)

Part of the Stable ABI. 返回保存在 capsule 中的当前名称。在失败时设置一个异常并返回 `NULL`。

capsule 具有 `NULL` 名称是合法的。这会使得 `NULL` 返回码有些歧义；请使用 `PyCapsule_IsValid()` 或 `PyErr_Occurred()` 来消除歧义。

void ***PyCapsule_Import** (**const** char **name*, int *no_block*)

Part of the Stable ABI. 从一个模块的 capsule 属性导入指向 C 对象的指针。*name* 形参应当指定属性的完整名称，与 `module.attribute` 中的一致。保存在 capsule 中的 *name* 必须完全匹配此字符串。如果 *no_block* 为真值，则以无阻塞模式导入模块 (使用 `PyImport_ImportModuleNoBlock()`)。如果 *no_block* 为假值，则以传统模式导入模块 (使用 `PyImport_ImportModule()`)。

成功时返回 capsule 的内部 指针。在失败时设置一个异常并返回 `NULL`。

int **PyCapsule_IsValid** (*PyObject *capsule*, **const** char **name*)

Part of the Stable ABI. 确定 *capsule* 是否是一个有效的。有效的 capsule 必须不为 `NULL`，传递 `PyCapsule_CheckExact()`，在其中存储一个不为 `NULL` 的指针，并且其内部名称与 *name* 形参相匹配。（请参阅 `PyCapsule_GetPointer()` 了解如何对 capsule 名称进行比较的有关信息。）

换句话说，如果 `PyCapsule_IsValid()` 返回真值，则任何对访问器（以 `PyCapsule_Get()` 开头的任何函数）的调用都保证会成功。

如果对象有效并且匹配传入的名称则返回非零值。否则返回 0。此函数一定不会失败。

int **PyCapsule_SetContext** (*PyObject* **capsule*, void **context*)
Part of the Stable ABI. 将 *capsule* 内部的上下文指针设为 *context*。

成功时返回 0。失败时返回非零值并设置一个异常。

int **PyCapsule_SetDestructor** (*PyObject* **capsule*, *PyCapsule_Destructor* *destructor*)
Part of the Stable ABI. 将 *capsule* 内部的析构器设为 *destructor*。

成功时返回 0。失败时返回非零值并设置一个异常。

int **PyCapsule_SetName** (*PyObject* **capsule*, **const** char **name*)
Part of the Stable ABI. 将 *capsule* 内部的名称设为 *name*。如果不为 NULL，则名称的存在期必须比 capsule 更长。如果之前保存在 capsule 中的 *name* 不为 NULL，则不会尝试释放它。

成功时返回 0。失败时返回非零值并设置一个异常。

int **PyCapsule_SetPointer** (*PyObject* **capsule*, void **pointer*)
Part of the Stable ABI. 将 *capsule* 内部的空指针设为 *pointer*。指针不可为 NULL。

成功时返回 0。失败时返回非零值并设置一个异常。

8.6.10 F生器物件

生成器对象是 Python 用来实现生成器迭代器的对象。它们通常通过迭代产生值的函数来创建，而不是显式调用*PyGen_New()* 或*PyGen_NewWithQualName()*。

type PyGenObject
用于生成器对象的 C 结构体。

PyTypeObject **PyGen_Type**
与生成器对象对应的类型对 象。

int **PyGen_Check** (*PyObject* **ob*)
如果 *ob* 是一个 generator 对象则返回真值；*ob* 必须不为 NULL。此函数总是会成功执行。

int **PyGen_CheckExact** (*PyObject* **ob*)
如果 *ob* 的类型是*PyGen_Type* 则返回真值；*ob* 必须不为 NULL。此函数总是会成功执行。

PyObject ***PyGen_New** (*PyFrameObject* **frame*)
Return value: New reference. 基于 *frame* 对象创建并返回一个新的生成器对象。此函数会取走一个对 *frame* 的引用。参数必须不为 NULL。

PyObject ***PyGen_NewWithQualName** (*PyFrameObject* **frame*, *PyObject* **name*, *PyObject* **qualname*)
Return value: New reference. 基于 *frame* 对象创建并返回一个新的生成器对象，其中 `__name__` 和 `__qualname__` 设为 *name* 和 *qualname*。此函数会取走一个对 *frame* 的引用。*frame* 参数必须不为 NULL。

8.6.11 Coroutine（協程）物件

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F物件。

int **PyCoro_CheckExact** (*PyObject* *ob)
    如果 *ob* 的型F是`PyCoro_Type` 則回傳真值；*ob* 必須不F `NULL`。此函式總是會執行成功。

PyObject ***PyCoro_New** (*PyFrameObject* **frame*, *PyObject* **name*, *PyObject* **qualname*)
    Return value: New reference. 基於 *frame* 物件來建立F回傳一個新的 coroutine 物件，其中 `__name__` 和 `__qualname__` 被設F *name* 和 *qualname*。此函式會取得一個對 *frame* 的參照 (reference)。*frame* 引數必須不F `NULL`。

8.6.12 上下文变量对象

備F: 3.7.1 版更變: 在 Python 3.7.1 中，所有上下文变量 C API 的签名被 **更改**为使用`PyObject` 指针而不是`PyContext`, `PyContextVar` 以及`PyContextToken`，例如:

```
// in 3.7.0:
PyContext *PyContext_New(void);

// in 3.7.1+:
PyObject *PyContext_New(void);
```

更多細節請見 bpo-34762。

3.7 版新加入.

本节深入介绍了 `contextvars` 模块的公用 C API。

type PyContext
    用于表示 `contextvars.Context` 对象的 C 结构体。

type PyContextVar
    用于表示 `contextvars.ContextVar` 对象的 C 结构体。

type PyContextToken
    用于表示 `contextvars.Token` 对象的 C 结构体。

PyTypeObject **PyContext_Type**
    表示 *context* 类型的类型对象。

PyTypeObject **PyContextVar_Type**
    表示 *context variable* 类型的类型对象。

PyTypeObject **PyContextToken_Type**
    表示 *context variable token* 类型的类型对象。

类型检查宏:

int **PyContext_CheckExact** (*PyObject* *o)
    如果 *o* 的类型为`PyContext_Type` 则返回真值。*o* 必须不为 `NULL`。此函数总是会成功执行。

int **PyContextVar_CheckExact** (*PyObject* *o)
    如果 *o* 的类型为`PyContextVar_Type` 则返回真值。*o* 必须不为 `NULL`。此函数总是会成功执行。

int **PyContextToken_CheckExact** (*PyObject* *o)
    如果 *o* 的类型为`PyContextToken_Type` 则返回真值。*o* 必须不为 `NULL`。此函数总是会成功执行。

上下文对象管理函数:

PyObject ***PyContext_New** (void)
    Return value: New reference. 创建一个新的空上下文对象。如果发生错误则返回 `NULL`。

PyObject ***PyContext_Copy**(*PyObject* **ctx*)
Return value: New reference. 创建所传入的 *ctx* 上下文对象的浅拷贝。如果发生错误则返回 NULL。

PyObject ***PyContext_CopyCurrent**(void)
Return value: New reference. 创建当前线程上下文的浅拷贝。如果发生错误则返回 NULL。

int **PyContext_Enter**(*PyObject* **ctx*)
将 *ctx* 设为当前线程的当前上下文。成功时返回 0，出错时返回 -1。

int **PyContext_Exit**(*PyObject* **ctx*)
取消激活 *ctx* 上下文并将之前的上下文恢复为当前线程的当前上下文。成功时返回 0，出错时返回 -1。

上下文变量函数:

PyObject ***PyContextVar_New**(**const** char **name*, *PyObject* **def*)
Return value: New reference. 创建一个新的 ContextVar 对象。形参 *name* 用于自我检查和调试目的。形参 *def* 为上下文变量指定默认值，或为 NULL 表示无默认值。如果发生错误，这个函数会返回 NULL。

int **PyContextVar_Get**(*PyObject* **var*, *PyObject* **default_value*, *PyObject* ***value*)
获取上下文变量的值。如果在查找过程中发生错误，返回''-1''，如果没有发生错误，无论是否找到值，都返回''0''，

如果找到上下文变量，*value* 将是指向它的指针。如果上下文变量 没有找到，*value* 将指向：

- *default_value*，如果非 "NULL";
- *var* 的默认值，如果不是 NULL;
- NULL

除了返回 NULL，这个函数会返回一个新的引用。

PyObject ***PyContextVar_Set**(*PyObject* **var*, *PyObject* **value*)
Return value: New reference. 在当前上下文中将 *var* 设为 *value*。返回针对此修改的新凭据对象，或者如果发生错误则返回 NULL。

int **PyContextVar_Reset**(*PyObject* **var*, *PyObject* **token*)
将上下文变量 *var* 的状态重置为它在返回 *token* 的*PyContextVar_Set()* 被调用之前的状态。此函数成功时返回 0，出错时返回 -1。

8.6.13 DateTime 物件

datetime 模块提供了各种日期和时间对象。在使用任何这些函数之前，必须在你的源码中包含头文件 datetime.h (请注意此文件并未包含在 Python.h 中)，并且宏 PyDateTime_IMPORT 必须被发起调用，通常是作为模块初始化函数的一部分。这个宏会将指向特定 C 结构的指针放入一个静态变量 PyDateTimeAPI 中，它会由下面的宏来使用。

宏访问 UTC 单例:

PyObject ***PyDateTime_TimeZone_UTC**
返回表示 UTC 的时区单例，与 datetime.timezone.utc 为同一对象。

3.7 版新加入.

类型检查宏：

int **PyDate_Check**(*PyObject* **ob*)
如果 *ob* 为 PyDateTime_DateType 类型或 PyDateTime_DateType 的某个子类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyDate_CheckExact**(*PyObject* **ob*)
如果 *ob* 为 PyDateTime_DateType 类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyDateTime_Check** (*PyObject* *ob)
: 如果 *ob* 为 PyDateTime_DateTimeType 类型或 PyDateTime_DateTimeType 的某个子类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyDateTime_CheckExact** (*PyObject* *ob)
: 如果 *ob* 为 PyDateTime_DateTimeType 类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyTime_Check** (*PyObject* *ob)
: 如果 *ob* 的类型是 PyDateTime_TimeType 或是 PyDateTime_TimeType 的子类型则返回真值。*ob* 必须不为 NULL。此函数总是会成功执行。

int **PyTime_CheckExact** (*PyObject* *ob)
: 如果 *ob* 为 PyDateTime_TimeType 类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyDelta_Check** (*PyObject* *ob)
: 如果 *ob* 为 PyDateTime_DeltaType 类型或 PyDateTime_DeltaType 的某个子类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyDelta_CheckExact** (*PyObject* *ob)
: 如果 *ob* 为 PyDateTime_DeltaType 类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

int **PyTZInfo_Check** (*PyObject* *ob)
: 如果 *ob* 的类型是 PyDateTime_TZInfoType 或是 PyDateTime_TZInfoType 的子类型则返回真值。*ob* 必须不为 NULL。此函数总是会成功执行。

int **PyTZInfo_CheckExact** (*PyObject* *ob)
: 如果 *ob* 为 PyDateTime_TZInfoType 类型则返回真值。*ob* 不能为 NULL。此函数总是会成功执行。

用于创建对象的宏：

PyObject ***PyDate_FromDate** (int *year*, int *month*, int *day*)
: *Return value: New reference.* 返回指定年、月、日的 datetime.date 对象。

PyObject ***PyDateTime_FromDateAndTime** (int *year*, int *month*, int *day*, int *hour*, int *minute*, int *second*, int *usecond*)
: *Return value: New reference.* 返回具有指定 year, month, day, hour, minute, second 和 microsecond 属性的 datetime.datetime 对象。

PyObject ***PyDateTime_FromDateAndTimeAndFold** (int *year*, int *month*, int *day*, int *hour*, int *minute*, int *second*, int *usecond*, int *fold*)
: *Return value: New reference.* 返回具有指定 year, month, day, hour, minute, second, microsecond 和 fold 属性的 datetime.datetime 对象。

  3.6 版新加入.

PyObject ***PyTime_FromTime** (int *hour*, int *minute*, int *second*, int *usecond*)
: *Return value: New reference.* 返回具有指定 hour, minute, second and microsecond 属性的 datetime.time 对象。

PyObject ***PyTime_FromTimeAndFold** (int *hour*, int *minute*, int *second*, int *usecond*, int *fold*)
: *Return value: New reference.* 返回具有指定 hour, minute, second, microsecond 和 fold 属性的 datetime.time 对象。

  3.6 版新加入.

PyObject ***PyDelta_FromDSU** (int *days*, int *seconds*, int *useconds*)
: *Return value: New reference.* 返回代表给定天、秒和微秒数的 datetime.timedelta 对象。将执行正规化操作以使最终的微秒和秒数处在 datetime.timedelta 对象的文档指明的区间之内。

PyObject ***PyTimeZone_FromOffset** (PyDateTime_DeltaType *offset*)
: *Return value: New reference.* 返回一个 datetime.timezone 对象，该对象具有以 *offset* 参数表示的未命名固定时差。

3.7 版新加入.

PyObject ***PyTimeZone_FromOffsetAndName**(PyDateTime_DeltaType **offset*, PyUnicode **name*)
Return value: New reference. 返回一个 `datetime.timezone` 对象，该对象具有以 *offset* 参数表示的固定时差和时区名称 *name*。

3.7 版新加入.

一些用来从 date 对象中提取字段的宏。参数必须是 `PyDateTime_Date` 包括其子类 (例如 `PyDateTime_DateTime`) 的实例。参数必须不为 `NULL`，并且类型不被会检查:

int **PyDateTime_GET_YEAR**(PyDateTime_Date **o*)
回傳年份，F正整數。

int **PyDateTime_GET_MONTH**(PyDateTime_Date **o*)
回傳月份，F正整數，從 1 到 12。

int **PyDateTime_GET_DAY**(PyDateTime_Date **o*)
回傳日期，F正整數，從 1 到 31。

一些用来从 datetime 对象中提取字段的宏。参数必须是 `PyDateTime_DateTime` 包括其子类的实例。参数必须不为 `NULL`，并且类型不会被检查:

int **PyDateTime_DATE_GET_HOUR**(PyDateTime_DateTime **o*)
回傳小時，F正整數，從 0 到 23。

int **PyDateTime_DATE_GET_MINUTE**(PyDateTime_DateTime **o*)
回傳分鐘，F正整數，從 0 到 59。

int **PyDateTime_DATE_GET_SECOND**(PyDateTime_DateTime **o*)
回傳秒，F正整數，從 0 到 59。

int **PyDateTime_DATE_GET_MICROSECOND**(PyDateTime_DateTime **o*)
回傳微秒，F正整數，從 0 到 999999。

int **PyDateTime_DATE_GET_FOLD**(PyDateTime_DateTime **o*)
回傳 fold，F 0 或 1 的正整數。

3.6 版新加入.

PyObject ***PyDateTime_DATE_GET_TZINFO**(PyDateTime_DateTime **o*)
返回 tzinfo (可以为 `None`)。

3.10 版新加入.

一些用来从 time 对象中提取字段的宏。参数必须是 `PyDateTime_Time` 包括其子类的实例。参数必须不为 `NULL`，并且类型不会被检查:

int **PyDateTime_TIME_GET_HOUR**(PyDateTime_Time **o*)
回傳小時，F正整數，從 0 到 23。

int **PyDateTime_TIME_GET_MINUTE**(PyDateTime_Time **o*)
回傳分鐘，F正整數，從 0 到 59。

int **PyDateTime_TIME_GET_SECOND**(PyDateTime_Time **o*)
回傳秒，F正整數，從 0 到 59。

int **PyDateTime_TIME_GET_MICROSECOND**(PyDateTime_Time **o*)
回傳微秒，F正整數，從 0 到 999999。

int **PyDateTime_TIME_GET_FOLD**(PyDateTime_Time **o*)
回傳 fold，F 0 或 1 的正整數。

3.6 版新加入.

PyObject ***PyDateTime_TIME_GET_TZINFO**(PyDateTime_Time *o)
　　返回 tzinfo (可以为 None)。

　　3.10 版新加入.

一些用来从 timedelta 对象中提取字段的宏。参数必须是 PyDateTime_Delta 包括其子类的实例。参数必须不为 NULL，并且类型不会被检查:

int **PyDateTime_DELTA_GET_DAYS**(PyDateTime_Delta *o)
　　返回天数，从-999999999 到 999999999 的整数。

　　3.3 版新加入.

int **PyDateTime_DELTA_GET_SECONDS**(PyDateTime_Delta *o)
　　返回秒数，从 0 到 86399 的整数。

　　3.3 版新加入.

int **PyDateTime_DELTA_GET_MICROSECONDS**(PyDateTime_Delta *o)
　　返回微秒数，从 0 到 999999 的整数。

　　3.3 版新加入.

一些便于模块实现 DB API 的宏:

PyObject ***PyDateTime_FromTimestamp**(*PyObject* *args)
　　Return value: New reference. 创建并返回一个给定元组参数的新 datetime.datetime 对象，适合传给 datetime.datetime.fromtimestamp()。

PyObject ***PyDate_FromTimestamp**(*PyObject* *args)
　　Return value: New reference. 创建并返回一个给定元组参数的新 datetime.date 对象，适合传给 datetime.date.fromtimestamp()。

8.6.14 类型注解对象

提供几种用于类型提示的内置类型。目前存在两种类型 -- GenericAlias 和 Union。只有 GenericAlias 会向 C 开放。

PyObject ***Py_GenericAlias**(*PyObject* *origin, *PyObject* *args)
　　Part of the Stable ABI since version 3.9. Create a GenericAlias object. Equivalent to calling the Python class types.GenericAlias. The *origin* and *args* arguments set the GenericAlias's __origin__ and __args__ attributes respectively. *origin* should be a *PyTypeObject**, and *args* can be a *PyTupleObject** or any PyObject*. If *args* passed is not a tuple, a 1-tuple is automatically constructed and __args__ is set to (args,). Minimal checking is done for the arguments, so the function will succeed even if *origin* is not a type. The GenericAlias's __parameters__ attribute is constructed lazily from __args__. On failure, an exception is raised and NULL is returned.

　　下面是一个如何创建一个扩展类型泛型的例子:

```
...
static PyMethodDef my_obj_methods[] = {
    // Other methods.
    ...
    {"__class_getitem__", (PyCFunction)Py_GenericAlias, METH_O|METH_CLASS, "See␣
↪PEP 585"}
    ...
}
```

　　也参考:

　　数据模型的方法 __class_getitem__() 。

3.9 版新加入.

PyTypeObject **Py_GenericAliasType**

Part of the Stable ABI *since version 3.9.* 由`Py_GenericAlias()` 所返回的对象的 C 类型。等价于 Python 中的 `types.GenericAlias` 。

3.9 版新加入.

CHAPTER 9

初始化，终结和线程

请参阅*Python 初始化配置* 。

9.1 在 Python 初始化之前

在一个植入了 Python 的应用程序中，*Py_Initialize()* 函数必须在任何其他 Python/C API 函数之前被调用；例外的只有个别函数和*全局配置变量*。

在初始化 Python 之前，可以安全地调用以下函数：

- 配置函数：
  - *PyImport_AppendInittab()*
  - *PyImport_ExtendInittab()*
  - PyInitFrozenExtensions()
  - *PyMem_SetAllocator()*
  - *PyMem_SetupDebugHooks()*
  - *PyObject_SetArenaAllocator()*
  - *Py_SetPath()*
  - *Py_SetProgramName()*
  - *Py_SetPythonHome()*
  - *Py_SetStandardStreamEncoding()*
  - *PySys_AddWarnOption()*
  - *PySys_AddXOption()*
  - *PySys_ResetWarnOptions()*
- 信息函数：

- *Py_IsInitialized()*
- *PyMem_GetAllocator()*
- *PyObject_GetArenaAllocator()*
- *Py_GetBuildInfo()*
- *Py_GetCompiler()*
- *Py_GetCopyright()*
- *Py_GetPlatform()*
- *Py_GetVersion()*

• 工具

- *Py_DecodeLocale()*

• 内存分配器:

- *PyMem_RawMalloc()*
- *PyMem_RawRealloc()*
- *PyMem_RawCalloc()*
- *PyMem_RawFree()*

備 F: 以下函数 **不应该** 在*Py_Initialize()*: *Py_EncodeLocale()*, *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()*, *Py_GetProgramFullPath()*, *Py_GetPythonHome()*, *Py_GetProgramName()* 和*PyEval_InitThreads()* 前调用。

9.2 全局配置变量

Python 有负责控制全局配置中不同特性和选项的变量。这些标志默认被 命令行选项。

当一个选项设置一个旗标时，该旗标的值将是设置选项的次数。例如，-b 会将*Py_BytesWarningFlag* 设为 1 而 -bb 会将*Py_BytesWarningFlag* 设为 2.

int **Py_BytesWarningFlag**

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 选项设置。

int **Py_DebugFlag**

开启解析器调试输出（限专家使用，依赖于编译选项）。

由 -d 选项和 PYTHONDEBUG 环境变量设置。

int **Py_DontWriteBytecodeFlag**

如果设置为非零, Python 不会在导入源代码时尝试写入 .pyc 文件

由 -B 选项和 PYTHONDONTWRITEBYTECODE 环境变量设置。

int **Py_FrozenFlag**

Suppress error messages when calculating the module search path in *Py_GetPath()*.

Private flag used by _freeze_importlib and frozenmain programs.

int **Py_HashRandomizationFlag**
Set to 1 if the PYTHONHASHSEED environment variable is set to a non-empty string.

If the flag is non-zero, read the PYTHONHASHSEED environment variable to initialize the secret hash seed.

int **Py_IgnoreEnvironmentFlag**
忽略所有 PYTHON* 环境变量，例如，已设置的 PYTHONPATH 和 PYTHONHOME。

由 -E 和 -I 选项设置。

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.

Set by the -i option and the PYTHONINSPECT environment variable.

int **Py_InteractiveFlag**
由 -i 选项设置。

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 选项设置。

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*.

Set to 1 if the PYTHONLEGACYWINDOWSFSENCODING environment variable is set to a non-empty string.

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 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**
禁用 site 的导入及其所附带的基于站点对 sys.path 的操作。如果 site 会在稍后被显式地导入也会禁用这些操作 (如果你希望触发它们则应调用 site.main())。

由 -S 选项设置。

int **Py_NoUserSiteDirectory**
不要将 用户 site-packages 目录添加到 sys.path。

Set by the -s and -I options, and the PYTHONNOUSERSITE environment variable.

int **Py_OptimizeFlag**
Set by the -O option and the PYTHONOPTIMIZE environment variable.

int **Py_QuietFlag**
即使在交互模式下也不显示版权和版本信息。

由 -q 选项设置。

3.2 版新加入.

int **Py_UnbufferedStdioFlag**

强制 stdout 和 stderr 流不带缓冲。

Set by the -u option and the PYTHONUNBUFFERED environment variable.

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.

Set by the -v option and the PYTHONVERBOSE environment variable.

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.

備䪾: 在 Windows 上，将控制台模式从 O_TEXT 改为 O_BINARY，这还将影响使用 C 运行时的非 Python 的控制台使用。

void **Py_InitializeEx**(int *initsigs*)

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.

Raises an auditing event `cpython._PySys_ClearAuditHooks` with no arguments.

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 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).

3.4 版新加入.

void **Py_SetProgramName**(**const** wchar_t **name*)

Part of the Stable ABI. 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.

wchar ***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`.

3.10 版更變: 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`.

3.10 版更變: 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.

3.10 版更變: 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.

3.10 版更變: 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.

3.10 版更變: It now returns NULL if called before *Py_Initialize()*.

void **Py_SetPath**(**const** wchar_t*)

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

3.8 版更變: The program full path is now used for `sys.executable`, instead of the program name.

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`.

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. 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.

備[F]: 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");
```

3.1.3 版新加入.

void **PySys_SetArgv** (int *argc*, wchar_t ***argv*)

Part of the Stable ABI. 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.

3.4 版更變: The *updatepath* value depends on -I.

void **Py_SetPythonHome** (**const** wchar_t **home*)

Part of the Stable ABI. 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.

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.

3.10 版更變: It now returns NULL if called before *Py_Initialize()*.

9.5 线程状态和全局解释器锁

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.

上面的代码块可扩展为下面的代码:

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

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 非 Python 创建的线程

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 高阶 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 `interp` (`PyInterpreterState*`), which points to this thread's interpreter state.

void **PyEval_InitThreads**()

Part of the Stable ABI. Deprecated function which does nothing.

在 Python 3.6 及更老的版本中，此函数会在 GIL 不存在时创建它。

3.9 版更變: The function now does nothing.

3.7 版更變: This function is now called by *Py_Initialize()*, so you don't have to call it yourself anymore.

3.2 版更變: This function cannot be called before *Py_Initialize()* anymore.

自從版本 3.9 後不推薦使用，將會自版本 3.11 中移除。.

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.

3.7 版更變: The *GIL* is now initialized by *Py_Initialize()*.

自從版本 3.9 後不推薦使用，將會自版本 3.11 中移除。.

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.

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.

---

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.

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. 此宏扩展为 `PyEval_RestoreThread(_save); }`。注意它包含一个右花括号；它必须与之前的*Py_BEGIN_ALLOW_THREADS* 宏匹配。请参阅上文以进一步讨论此宏。

Py_BLOCK_THREADS

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 Low-level API

All of the following functions must be called after *Py_Initialize()*.

3.7 版更變: *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.

Raises an auditing event `cpython.PyInterpreterState_New` with no arguments.

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.

Raises an auditing event `cpython.PyInterpreterState_Clear` with no arguments.

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.

3.9 版更變: 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`。

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`。

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`。

3.9 版新加入.

PyInterpreterState ***PyInterpreterState_Get**(void)

Part of the Stable ABI *since version 3.9.* 获取当前解释器。

Issue a fatal error if there no current Python thread state or no current interpreter. It cannot return NULL.

呼叫者必须持有 GIL。

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.

呼叫者必须持有 GIL。

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.

3.8 版新加入.

typedef *PyObject* *(***_PyFrameEvalFunction**)(*PyThreadState* *tstate, *PyFrameObject* *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.

3.9 版更變: 此函数现在可接受一个 *tstate* 形参。

_PyFrameEvalFunction **_PyInterpreterState_GetEvalFrameFunc**(*PyInterpreterState* **interp*)

Get the frame evaluation function.

See the **PEP 523** "Adding a frame evaluation API to CPython".

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".

3.9 版新加入.

PyObject ***PyThreadState_GetDict**()

Return value: Borrowed reference. Part of the Stable ABI. Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no 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.

3.7 版更變: 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.

3.8 版更變: 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.

3.2 版後已F用: This function does not update the current thread state. Please use *PyEval_RestoreThread()* or *PyEval_AcquireThread()* instead.

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.

3.8 版更變: 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.

3.2 版後已F用: This function does not update the current thread state. Please use *PyEval_SaveThread()* or *PyEval_ReleaseThread()* instead.

9.6 子解释器支持

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 错误和警告

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 异步通知

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*.

3.9 版更變: 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.

3.1 版新加入.

9.8 分析和跟踪

The Python interpreter provides some low-level support for attaching profiling and execution tracing facilities. These are used for profiling, debugging, and coverage analysis tools.

This C interface allows the profiling or tracing code to avoid the overhead of calling through Python-level callable objects, making a direct C function call instead. The essential attributes of the facility have not changed; the interface allows trace functions to be installed per-thread, and the basic events reported to the trace function are the same as had been reported to the Python-level trace functions in previous versions.

typedef int (***Py_tracefunc**)(*PyObject* *obj, *PyFrameObject* *frame, int what, *PyObject* *arg)

The type of the trace function registered using `PyEval_SetProfile()` and `PyEval_SetTrace()`. The first parameter is the object passed to the registration function as *obj*, *frame* is the frame object to which the event pertains, *what* is one of the constants `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION`, `PyTrace_C_RETURN`, or `PyTrace_OPCODE`, and *arg* depends on the value of *what*:

what 的值	*arg* 的含义
PyTrace_CALL	总是 *Py_None*.
PyTrace_EXCEPTION	sys.exc_info() 返回的异常信息。
PyTrace_LINE	总是 *Py_None*.
PyTrace_RETURN	返回给调用方的值，或者如果是由异常导致的则返回 NULL。
PyTrace_C_CALL	正在调用函数对象。
PyTrace_C_EXCEPTION	正在调用函数对象。
PyTrace_C_RETURN	正在调用函数对象。
PyTrace_OPCODE	总是 *Py_None*.

int **PyTrace_CALL**
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`.

The caller must hold the *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.

The caller must hold the *GIL*.

9.9 高级调试器支持

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 线程本地存储支持

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.

備註： 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.

3.7 版新加入.

也参考:

"A New C-API for Thread-Local Storage in CPython" (**PEP 539**)

type Py_tss_t

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*.

方法

The parameter *key* of these functions must not be `NULL`. Moreover, the behaviors of `PyThread_tss_set()` and `PyThread_tss_get()` are undefined if the given `Py_tss_t` has not been initialized by `PyThread_tss_create()`.

int **PyThread_tss_is_created**(*Py_tss_t* **key*)

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

3.7 版後已[F]用: This API is superseded by *Thread Specific Storage (TSS) API*.

備[F]: 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.

由于上面提到的兼容性问题，不应在新代码中使用此版本的 API。

int **PyThread_create_key**()
    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 初始化配置

3.8 版新加入.

Python 可以使用Py_InitializeFromConfig() 和PyConfig 结构体来初始化。它可以使用Py_PreInitialize() 和PyPreConfig 结构体来预初始化。

有两种配置方式:

- 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.

Py_RunMain() 函数可被用来编写定制的 Python 程序。

参见*Initialization, Finalization, and Threads*.

也参考:

PEP 587 "Python 初始化配置".

10.1 范例

定制的 Python 的示例总是会以隔离模式运行:

```
int main(int argc, char **argv)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config.isolated = 1;

```

(下页继续)

(繼續上一頁)

```
    /* Decode command line arguments.
       Implicitly preinitialize Python (in isolated mode). */
    status = PyConfig_SetBytesArgv(&config, argc, argv);
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);

    return Py_RunMain();

exception:
    PyConfig_Clear(&config);
    if (PyStatus_IsExit(status)) {
        return status.exitcode;
    }
    /* Display the error message and exit the process with
       non-zero exit code */
    Py_ExitStatusException(status);
}
```

10.2 PyWideStringList

type PyWideStringList

由 `wchar_t*` 字符串组成的列表。

如果 *length* 为非零值，则 *items* 必须不为 `NULL` 并且所有字符串均必须不为 `NULL`。

方法

PyStatus **PyWideStringList_Append**(*PyWideStringList* **list*, **const** wchar_t **item*)

将 *item* 添加到 *list*。

Python 必须被预初始化以便调用此函数。

PyStatus **PyWideStringList_Insert**(*PyWideStringList* **list*, *Py_ssize_t* *index*, **const** wchar_t **item*)

将 *item* 插入到 *list* 的 *index* 位置上。

如果 *index* 大于等于 *list* 的长度，则将 *item* 添加到 *list*。

index must be greater than or equal to 0.

Python 必须被预初始化以便调用此函数。

Structure fields:

Py_ssize_t **length**

List 长度。

wchar_t ****items**

列表项目。

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)

完成。

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*)

以指定的退出代码退出 Python。

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*)

结果错误吗?

int **PyStatus_IsExit**(*PyStatus status*)

结果是否退出?

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();
```

(下页继续)

(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**

If non-zero, enables the 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 by the `-X utf8` command line option and the `PYTHONUTF8` environment variable.

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 */
```

(下页继续)

(繼續上一頁)

```
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.

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.

3.10 版更變: 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`.

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.

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。

3.10 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"`。

参见 **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.

Default: `-1` in Python mode, `0` in isolated mode.

int **dump_refs**

转储 Python 引用？

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"`.

参见*`filesystem_errors`* 的成员。

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"` (仅支持 UTF-8 编码格式)

参见*`filesystem_encoding`* 的成员。

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:

- `sys.path` contains neither the script's directory (computed from `argv[0]` or the current directory) nor the user's site-packages directory.
- Python REPL doesn't import `readline` nor enable default readline configuration on interactive prompts.
- Set `use_environment` and `user_site_directory` to 0.

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"`).

Part of the *Python Path Configuration* input.

3.9 版新加入.

wchar_t ***pythonpath_env**

Module search paths (`sys.path`) as a string separated by `DELIM` (`os.path.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, the function calculating the *Python Path Configuration* overrides the *`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.

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.

3.10 版更變: 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**

On Unix, if non-zero, calculating the *Python Path Configuration* can log warnings into stderr. If equals to 0, suppress these warnings.

It has no effect on Windows.

Default: 1 in Python mode, 0 in isolated mode.

Part of the *Python Path Configuration* input.

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.

For example, it is set to script.py by the python3 script.py arg command.

Used by *Py_RunMain()*.

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.

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.

3.9 版更變: 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");
```

(下页继续)

(繼續上一頁)

```
    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:

```
PyStatus init_python(const char *program_name)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name before reading the configuration
       (decode byte string from the locale encoding).

       Implicitly preinitialize Python. */
    status = PyConfig_SetBytesString(&config, &config.program_name,
                                     program_name);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Read all configuration at once */
    status = PyConfig_Read(&config);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Append our custom search path to sys.path */
    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);
```

(下页继续)

(繼續上一頁)

```
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. Set the *Python Path Configuration* ("output fields") to ignore these configuration files and avoid the function 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:

- 路径配置输入:
  - *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, path configuration input fields are ignored as well.

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`
- `python._pth` (仅 Windows)
- `pybuilddir.txt` (仅 Unix)

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;
  - 等等.

私有临时 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);
    }

    /* Use sys.stderr because sys.stdout is only created
       by _Py_InitializeMain() */
    int res = PyRun_SimpleString(
        "import sys; "
        "print('Run Python code before _Py_InitializeMain', "
               "file=sys.stderr)");
    if (res < 0) {
        exit(1);
    }

    /* ... put more configuration code here ... */

    status = _Py_InitializeMain();
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
}
```

CHAPTER 11

記憶體管理

11.1 總覽

在 Python 中，内存管理涉及到一个包含所有 Python 对象和数据结构的私有堆（heap)。这个私有堆的管理由内部的 *Python 内存管理器*（*Python memory manager*）保证。Python 内存管理器有不同的组件来处理各种动态存储管理方面的问题，如共享、分割、预分配或缓存。

在最底层，一个原始内存分配器通过与操作系统的内存管理器交互，确保私有堆中有足够的空间来存储所有与 Python 相关的数据。在原始内存分配器的基础上，几个对象特定的分配器在同一堆上运行，并根据每种对象类型的特点实现不同的内存管理策略。例如，整数对象在堆内的管理方式不同于字符串、元组或字典，因为整数需要不同的存储需求和速度与空间的权衡。因此，Python 内存管理器将一些工作分配给对象特定分配器，但确保后者在私有堆的范围内运行。

Python 堆内存的管理是由解释器来执行，用户对它没有控制权，即使他们经常操作指向堆内内存块的对象指针，理解这一点十分重要。Python 对象和其他内部缓冲区的堆空间分配是由 Python 内存管理器按需通过本文档中列出的 Python/C API 函数进行的。

为了避免内存破坏，扩展的作者永远不应该试图用 C 库函数导出的函数来对 Python 对象进行操作，这些函数包括：`malloc()`, `calloc()`, `realloc()` 和 `free()`。这将导致 C 分配器和 Python 内存管理器之间的混用，引发严重后果，这是由于它们实现了不同的算法，并在不同的堆上操作。但是，我们可以安全地使用 C 库分配器为单独的目的分配和释放内存块，如下例所示：

```
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyBytes_FromString(buf);
free(buf); /* malloc'ed */
return res;
```

在这个例子中，I/O 缓冲区的内存请求是由 C 库分配器处理的。Python 内存管理器只参与了分配作为结果返回的字节对象。

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.

也参考:

环境变量 `PYTHONMALLOC` 可被用来配置 Python 所使用的内存分配器。

环境变量 `PYTHONMALLOCSTATS` 可以用来在每次创建和关闭新的 pymalloc 对象区域时打印*pymalloc 内存分配器* 的统计数据。

11.2 分配器域

所有分配函数都属于三个不同的"分配器域"之一（见`PyMemAllocatorDomain`）。这些域代表了不同的分配策略，并为不同目的进行了优化。每个域如何分配内存和每个域调用哪些内部函数的具体细节被认为是实现细节，但是出于调试目的，可以在此处 找到一张简化的表格。没有硬性要求将属于给定域的分配函数返回的内存，仅用于该域提示的目的（虽然这是推荐的做法）。例如，你可以将`PyMem_RawMalloc()` 返回的内存用于分配 Python 对象，或者将`PyObject_Malloc()` 返回的内存用作缓冲区。

三个分配域分别是：

- 原始域：用于为通用内存缓冲区分配内存，分配 * 必须 * 转到系统分配器并且分配器可以在没有*GIL* 的情况下运行。内存直接请求自系统。
- "Mem" 域：用于为 Python 缓冲区和通用内存缓冲区分配内存，分配时必须持有*GIL*。内存取自于 Python 私有堆。
- 对象域：用于分配属于 Python 对象的内存。内存取自于 Python 私有堆。

当释放属于给定域的分配函数先前分配的内存时，必须使用对应的释放函数。例如，`PyMem_Free()` 来释放`PyMem_Malloc()` 分配的内存。

11.3 原始内存接口

以下函数集封装了系统分配器。这些函数是线程安全的，不需要持有全局解释器锁。

default raw memory allocator 使用这些函数：`malloc()`、`calloc()`、`realloc()` 和 `free()`；申请零字节时则调用 `malloc(1)` （或 `calloc(1, 1)`）

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.

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyMem_RawMalloc(1)` 一样。但是内存不会以任何方式被初始化。

void ***PyMem_RawCalloc**(size_t *nelem*, size_t *elsize*)

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.

请求零字节可能返回一个独特的非 NULL 指针，就像调用了 PyMem_RawCalloc(1, 1) 一样。

3.5 版新加入.

void ***PyMem_RawRealloc**(void **p*, size_t *n*)

将 *p* 指向的内存块大小调整为 *n* 字节。以新旧内存块大小中的最小值为准，其中内容保持不变，

如果 *p* 是 NULL ，则相当于调用 PyMem_RawMalloc(n) ；如果 *n* 等于 0，则内存块大小会被调整，但不会被释放，返回非 NULL 指针。

除非 *p* 是 NULL ，否则它必须是之前调用*PyMem_RawMalloc()* 、*PyMem_RawRealloc()* 或*PyMem_RawCalloc()* 所返回的。

如果请求失败，*PyMem_RawRealloc()* 返回 NULL ，*p* 仍然是指向先前内存区域的有效指针。

void **PyMem_RawFree**(void **p*)

释放 *p* 指向的内存块。*p* 必须是之前调用*PyMem_RawMalloc()* 、*PyMem_RawRealloc()* 或*PyMem_RawCalloc()* 所返回的指针。否则，或在 PyMem_RawFree(p) 之前已经调用过的情况下，未定义的行为会发生。

如果 *p* 是 NULL, 那么什么操作也不会进行。

11.4 内存接口

以下函数集，仿照 ANSI C 标准，并指定了请求零字节时的行为，可用于从 Python 堆分配和释放内存。

默认内存分配器 使用了*pymalloc 内存分配器*.

> **警告：** 在使用这些函数时，必须持有全局解释器锁（*GIL*）。

3.6 版更變: 现在默认的分配器是 pymalloc 而非系统的 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.

请求零字节可能返回一个独特的非 NULL 指针，就像调用了 PyMem_Malloc(1) 一样。但是内存不会以任何方式被初始化。

void ***PyMem_Calloc**(size_t *nelem*, size_t *elsize*)

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.

请求零字节可能返回一个独特的非 NULL 指针，就像调用了 PyMem_Calloc(1, 1) 一样。

3.5 版新加入.

void ***PyMem_Realloc**(void **p*, size_t *n*)

Part of the Stable ABI. 将 *p* 指向的内存块大小调整为 *n* 字节。以新旧内存块大小中的最小值为准，其中内容保持不变，

如果 *p* 是 NULL ，则相当于调用 PyMem_Malloc(n) ；如果 *n* 等于 0，则内存块大小会被调整，但不会被释放，返回非 NULL 指针。

除非 *p* 是 NULL ，否则它必须是之前调用*PyMem_Malloc()* 、*PyMem_Realloc()* 或*PyMem_Calloc()* 所返回的。

如果请求失败，*PyMem_Realloc()* 返回 NULL ，*p* 仍然是指向先前内存区域的有效指针。

void **PyMem_Free** (void *p)

Part of the Stable ABI. 释放 *p* 指向的内存块。*p* 必须是之前调用 *PyMem_Malloc()*、*PyMem_Realloc()* 或 *PyMem_Calloc()* 所返回的指针。否则，或在 `PyMem_Free(p)` 之前已经调用过的情况下，未定义的行为会发生。

如果 *p* 是 `NULL`, 那么什么操作也不会进行。

以下面向类型的宏为方便而提供。注意 *TYPE* 可以指任何 C 类型。

TYPE ***PyMem_New** (TYPE, size_t 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.

TYPE ***PyMem_Resize** (void *p, TYPE, size_t 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.

这是一个 C 预处理宏，*p* 总是被重新赋值。请保存 *p* 的原始值，以避免在处理错误时丢失内存。

void **PyMem_Del** (void *p)

和 *PyMem_Free()* 相同。

此外，我们还提供了以下宏集用于直接调用 Python 内存分配器，而不涉及上面列出的 C API 函数。但是请注意，使用它们并不能保证跨 Python 版本的二进制兼容性，因此在扩展模块被弃用。

- `PyMem_MALLOC(size)`
- `PyMem_NEW(type, size)`
- `PyMem_REALLOC(ptr, size)`
- `PyMem_RESIZE(ptr, type, size)`
- `PyMem_FREE(ptr)`
- `PyMem_DEL(ptr)`

11.5 对象分配器

以下函数集，仿照 ANSI C 标准，并指定了请求零字节时的行为，可用于从 Python 堆分配和释放内存。

備F: 当通过自定义内存分配器 部分描述的方法拦截该域中的分配函数时，无法保证这些分配器返回的内存可以被成功地转换成 Python 对象。

默认对象分配器 使用*pymalloc* 内存分配器.

警告: 在使用这些函数时，必须持有全局解释器锁（*GIL*）。

void ***PyObject_Malloc** (size_t n)

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.

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyObject_Malloc(1)` 一样。但是内存不会以任何方式被初始化。

void *`PyObject_Calloc`(size_t *nelem*, size_t *elsize*)

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.

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyObject_Calloc(1, 1)` 一样。

3.5 版新加入.

void *`PyObject_Realloc`(void *p, size_t n)

Part of the Stable ABI. 将 *p* 指向的内存块大小调整为 *n* 字节。以新旧内存块大小中的最小值为准，其中内容保持不变，

如果 *p* 是 "NULL"，则相当于调用 `PyObject_Malloc(n)` ；如果 *n* 等于 0，则内存块大小会被调整，但不会被释放，返回非 `NULL` 指针。

除非 *p* 是 `NULL` ，否则它必须是之前调用 *`PyObject_Malloc()`* 、*`PyObject_Realloc()`* 或 *`PyObject_Calloc()`* 所返回的。

如果请求失败，*`PyObject_Realloc()`* 返回 `NULL` ，*p* 仍然是指向先前内存区域的有效指针。

void `PyObject_Free`(void *p)

Part of the Stable ABI. 释放 *p* 指向的内存块。*p* 必须是之前调用 *`PyObject_Malloc()`* 、*`PyObject_Realloc()`* 或 *`PyObject_Calloc()`* 所返回的指针。否则，或在 `PyObject_Free(p)` 之前已经调用过的情况下，未定义行为会发生。

如果 *p* 是 `NULL`, 那么什么操作也不会进行。

11.6 默认内存分配器

默认内存分配器：

配置	名称	PyMem_RawMalloc	PyMem_Malloc	PyObject_Malloc
发布版本	`"pymalloc"`	`malloc`	`pymalloc`	`pymalloc`
调试构建	`"pymalloc_debug"`	`malloc` + debug	`pymalloc` + debug	`pymalloc` + debug
没有 pymalloc 的发布版本	`"malloc"`	`malloc`	`malloc`	`malloc`
没有 pymalloc 的调试构建	`"malloc_debug"`	`malloc` + debug	`malloc` + debug	`malloc` + debug

说明：

- 名称：`PYTHONMALLOC` 环境变量的值。
- `malloc`：来自 C 标准库的系统分配器，C 函数：`malloc()`、`calloc()`、`realloc()` 和 `free()`。
- `pymalloc`：*pymalloc 内存分配器*.
- "+ debug": 附带*Python 内存分配器的调试钩子*.
- "调试构建"：调试模式下的 Python 构建。

11.7 自定义内存分配器

3.4 版新加入.

type PyMemAllocatorEx
用于描述内存块分配器的结构体。该结构体下列字段:

域	含意
`void *ctx`	作为第一个参数传入的用户上下文
`void* malloc(void *ctx, size_t size)`	分配一个内存块
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	分配一个初始化为 0 的内存块
`void* realloc(void *ctx, void *ptr, size_t new_size)`	分配一个内存块或调整其大小
`void free(void *ctx, void *ptr)`	释放一个内存块

3.5 版更變: The `PyMemAllocator` structure was renamed to *PyMemAllocatorEx* and a new `calloc` field was added.

type PyMemAllocatorDomain
用来识别分配器域的枚举类。域有:

PYMEM_DOMAIN_RAW
函式:

- *PyMem_RawMalloc()*
- *PyMem_RawRealloc()*
- *PyMem_RawCalloc()*
- *PyMem_RawFree()*

PYMEM_DOMAIN_MEM
函式:

- *PyMem_Malloc()*,
- *PyMem_Realloc()*
- *PyMem_Calloc()*
- *PyMem_Free()*

PYMEM_DOMAIN_OBJ
函式:

- *PyObject_Malloc()*
- *PyObject_Realloc()*
- *PyObject_Calloc()*
- *PyObject_Free()*

void **PyMem_GetAllocator**(*PyMemAllocatorDomain domain*, *PyMemAllocatorEx *allocator*)
获取指定域的内存块分配器。

void **PyMem_SetAllocator**(*PyMemAllocatorDomain domain*, *PyMemAllocatorEx *allocator*)
设置指定域的内存块分配器。

当请求零字节时，新的分配器必须返回一个独特的非 NULL 指针。

对于 PYMEM_DOMAIN_RAW 域，分配器必须是线程安全的：当分配器被调用时，不持有*全局解释器锁*。

如果新的分配器不是钩子（不调用之前的分配器），必须调用*PyMem_SetupDebugHooks()* 函数在新分配器上重新安装调试钩子。

> **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)
设置*Python 内存分配器的调试钩子* 以检测内存错误。

11.8 Python 内存分配器的调试钩子

当 Python 在调试模式下构建，*PyMem_SetupDebugHooks()* 函数在*Python 预初始化* 时被调用，以在 Python 内存分配器上设置调试钩子以检测内存错误。

PYTHONMALLOC 环境变量可被用于在以发行模式下编译的 Python 上安装调试钩子（例如：PYTHONMALLOC=debug）。

PyMem_SetupDebugHooks() 函数可被用于在调用了*PyMem_SetAllocator()* 之后设置调试钩子。

这些调试钩子用特殊的、可辨认的位模式填充动态分配的内存块。新分配的内存用字节 0xCD``（``PYMEM_CLEANBYTE）填充，释放的内存用字节 0xDD``（``PYMEM_DEADBYTE）填充。内存块被填充了字节 0xFD``（``PYMEM_FORBIDDENBYTE）的“禁止字节”包围。这些字节串不太可能是合法的地址、浮点数或 ASCII 字符串

运行时检查：

- 检测对 API 的违反。例如：检测对*PyMem_Malloc()* 分配的内存块调用*PyObject_Free()*。
- 检测缓冲区起始位置前的写入（缓冲区下溢）。
- 检测缓冲区终止位置后的写入（缓冲区溢出）。
- 检测当调用 PYMEM_DOMAIN_OBJ（如：*PyObject_Malloc()*）和 PYMEM_DOMAIN_MEM（如：*PyMem_Malloc()*）域的分配器函数时*GIL* 已被持有。

在出错时，调试钩子使用 tracemalloc 模块来回溯内存块被分配的位置。只有当 tracemalloc 正在追踪 Python 内存分配，并且内存块被追踪时，才会显示回溯。

让 S = sizeof(size_t)。将 2*S 个字节添加到每个被请求的 N 字节数据块的两端。内存的布局像是这样，其中 p 代表由类似 malloc 或类似 realloc 的函数所返回的地址 (p[i:j] 表示从 *(p+i) 左侧开始到 *(p+j) 左侧止的字节数据切片；请注意对负索引号的处理与 Python 切片是不同的）：

p[-2*S:-S] 最初所要求的字节数。这是一个 size_t，为大端序（易于在内存转储中读取）。

p[-S] API 标识符（ASCII 字符）：

- 'r' 表示 PYMEM_DOMAIN_RAW。

- 'm' 表示 PYMEM_DOMAIN_MEM。
- 'o' 表示 PYMEM_DOMAIN_OBJ。

p[-S+1:0] PYMEM_FORBIDDENBYTE 的副本。用于捕获下层的写入和读取。

p[0:N] 所请求的内存，用 PYMEM_CLEANBYTE 的副本填充，用于捕获对未初始化内存的引用。当调用 realloc 之类的函数来请求更大的内存块时，额外新增的字节也会用 PYMEM_CLEANBYTE 来填充。当调用 free 之类的函数时，这些字节会用 PYMEM_DEADBYTE 来重写，以捕获对已释放内存的引用。当调用 realloc 之类的函数来请求更小的内存块时，多余的旧字节也会用 PYMEM_DEADBYTE 来填充。

p[N:N+S] PYMEM_FORBIDDENBYTE 的副本。用于捕获超限的写入和读取。

p[N+S:N+2*S] 仅当定义了 PYMEM_DEBUG_SERIALNO 宏时会被使用（默认情况下将不定义）。

一个序列号，每次调用 malloc 之类或 realloc 之类的函数时自增 1。大端序的 size_t。如果之后检测到了"被破坏的内存"，此序列号提供了一个很好的手段用来在下次运行时设置中断点，以捕获该内存块被破坏的瞬间。obmalloc.c 中的静态函数 bumpserialno() 是此序列号会发生自增的唯一地方，它的存在使你可以方便地设置这样的中断点。

一个 realloc 之类或 free 之类的函数会先检查两端的 PYMEM_FORBIDDENBYTE 字节是否完好。如果它们被改变了，则会将诊断输出写入到 stderr，并且程序将通过 Py_FatalError() 中止。另一种主要的失败模式是当程序读到某种特殊的比特模式并试图将其用作地址时触发内存错误。如果你随即进入调试器并查看该对象，你很可能会看到它已完全被填充为 PYMEM_DEADBYTE (意味着已释放的内存被使用) 或 PYMEM_CLEANBYTE (意味着未初始货摊内存被使用)。

3.6 版更變: *PyMem_SetupDebugHooks()* 函数现在也能在使用发布模式编译的 Python 上工作。当发生错误时，调试钩子现在会使用 tracemalloc 来获取已分配内存块的回溯信息。调试钩子现在还会在 PYMEM_DOMAIN_OBJ 和 PYMEM_DOMAIN_MEM 作用域的函数被调用时检查是否持有 GIL。

3.8 版更變: 字节模式 0xCB (PYMEM_CLEANBYTE)、0xDB (PYMEM_DEADBYTE) 和 0xFB (PYMEM_FORBIDDENBYTE) 已被 0xCD 、0xDD 和 0xFD 替代以使用与 Windows CRT 调试 malloc() 和 free() 相同的值。

11.9 pymalloc 分配器

Python 有为具有短生命周期的小对象（小于或等于 512 字节）优化的 *pymalloc* 分配器。它使用固定大小为 256 KiB 的称为"arenas" 的内存映射。对于大于 512 字节的分配，它回到使用*PyMem_RawMalloc()* 和*PyMem_RawRealloc()* 。

pymalloc 是 PYMEM_DOMAIN_MEM (例如: *PyMem_Malloc()*) 和 PYMEM_DOMAIN_OBJ (例如: *PyObject_Malloc()*) 域的默认分配器 。

arena 分配器使用以下函数：

- Windows 上的 VirtualAlloc() 和 VirtualFree() ,
- mmap() 和 munmap() ，如果可用，
- 否则，malloc() 和 free() 。

如果 Python 配置了 --without-pymalloc 选项，那么此分配器将被禁用。也可以在运行时使用 PYTHONMALLOC` (例如: ``PYTHONMALLOC=malloc`) 环境变量来禁用它。

11.9.1 自定义 pymalloc Arena 分配器

3.4 版新加入.

type PyObjectArenaAllocator
用来描述一个 arena 分配器的结构体。这个结构体有三个字段:

域	含意
`void *ctx`	作为第一个参数传入的用户上下文
`void* alloc(void *ctx, size_t size)`	分配一块 size 字节的区域
`void free(void *ctx, void *ptr, size_t size)`	释放一块区域

void **PyObject_GetArenaAllocator**(*PyObjectArenaAllocator* **allocator*)
获取 arena 分配器

void **PyObject_SetArenaAllocator**(*PyObjectArenaAllocator* **allocator*)
设置 arena 分配器

11.10 tracemalloc C API

3.7 版新加入.

int **PyTraceMalloc_Track**(unsigned int *domain*, uintptr_t *ptr*, size_t *size*)
在 `tracemalloc` 模块中跟踪一个已分配的内存块。

成功时返回 `0`, 出错时返回 `-1` (无法分配内存来保存跟踪信息)。如果禁用了 tracemalloc 则返回 `-2`。

如果内存块已被跟踪, 则更新现有跟踪信息。

int **PyTraceMalloc_Untrack**(unsigned int *domain*, uintptr_t *ptr*)
在 `tracemalloc` 模块中取消跟踪一个已分配的内存块。如果内存块未被跟踪则不执行任何操作。

如果 tracemalloc 被禁用则返回 `-2`, 否则返回 `0`。

11.11 范例

以下是来自總覽 小节的示例, 经过重写以使 I/O 缓冲区是通过使用第一个函数集从 Python 堆中分配的:

```
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

使用面向类型函数集的相同代码:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
```

(下页继续)

(繼續上一頁)

```
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

请注意在以上两个示例中，缓冲区总是通过归属于相同集的函数来操纵的。事实上，对于一个给定的内存块必须使用相同的内存 API 族，以便使得混合不同分配器的风险减至最低。以下代码序列包含两处错误，其中一个被标记为 *fatal* 因为它混合了两种在不同堆上操作的不同分配器。

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);  /* Wrong -- should be PyMem_Free() */
free(buf2);       /* Right -- allocated via malloc() */
free(buf1);       /* Fatal -- should be PyMem_Del()  */
```

除了旨在处理来自 Python 堆的原始内存块的函数之外，Python 中的对象是通过`PyObject_New()`，`PyObject_NewVar()` 和`PyObject_Del()` 来分配和释放的。

这些将在有关如何在 C 中定义和实现新对象类型的下一章中讲解。

CHAPTER 12

对象实现支持

本章描述了定义新对象类型时所使用的函数、类型和宏。

12.1 在堆上分配对象

PyObject ***_PyObject_New**(*PyTypeObject* **type*)

Return value: New reference.

PyVarObject ***_PyObject_NewVar**(*PyTypeObject* **type*, *Py_ssize_t* *size*)

Return value: New reference.

PyObject ***PyObject_Init**(*PyObject* **op*, *PyTypeObject* **type*)

Return value: Borrowed reference. Part of the Stable ABI. Initialize a newly allocated object *op* with its type and initial reference. Returns the initialized object. If *type* indicates that the object participates in the cyclic garbage detector, it is added to the detector's set of observed objects. Other fields of the object are not affected.

PyVarObject ***PyObject_InitVar**(*PyVarObject* **op*, *PyTypeObject* **type*, *Py_ssize_t* *size*)

Return value: Borrowed reference. Part of the Stable ABI. 它的功能和`PyObject_Init()`一样，并且会初始化变量大小对象的长度信息。

TYPE ***PyObject_New**(TYPE, *PyTypeObject* **type*)

Return value: New reference. 使用 C 结构类型 *TYPE* 和 Python 类型对象 *type* 分配一个新的 Python 对象。未在该 Python 对象标头中定义的字段不会被初始化；对象的引用计数将为一。内存分配大小由 type 对象的`tp_basicsize`字段来确定。

TYPE ***PyObject_NewVar**(TYPE, *PyTypeObject* **type*, *Py_ssize_t* *size*)

Return value: New reference. 使用 C 的数据结构类型 *TYPE* 和 Python 的类型对象 *type* 分配一个新的 Python 对象。Python 对象头文件中没有定义的字段不会被初始化。被分配的内存空间预留了 *TYPE* 结构加 *type* 对象中`tp_itemsize`字段提供的 *size* 字段的值。这对于实现类似元组这种能够在构造期决定自己大小的对象是很实用的。将字段的数组嵌入到相同的内存分配中可以减少内存分配的次数，这提高了内存分配的效率。

void **PyObject_Del**(void **op*)

释放由`PyObject_New()`或者`PyObject_NewVar()`分配内存的对象。这通常由对象的 type 字段定

义的tp_dealloc 处理函数来调用。调用这个函数以后 op 对象中的字段都不可以被访问，因为原分配的内存空间已不再是一个有效的 Python 对象。

PyObject **_Py_NoneStruct**

这个对象是像 None 一样的 Python 对象。它可以使用Py_None 宏访问，该宏的拿到指向该对象的指针。

也参考:

PyModule_Create() 分配内存和创建扩展模块

12.2 通用物件結構

大量的结构体被用于定义 Python 的对象类型。这一节描述了这些的结构体和它们的使用方法。

12.2.1 基本的对象类型和宏

所有的 Python 对象都在对象的内存表示的开始部分共享少量的字段。这些字段用PyObject 或PyVarObject 类型来表示，这些类型又由一些宏定义，这些宏也直接或间接地用于所有其他 Python 对象的定义。

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;
```

参见上面PyVarObject 的文档。

int **Py_Is**(**const** *PyObject* *x, **const** *PyObject* *y)

Part of the Stable ABI *since version 3.10.* 测试 *x* 是否为 *y* 对象，与 Python 中的 x is y 相同。

3.10 版新加入.

int **Py_IsNone**(**const** *PyObject* *x)

Part of the Stable ABI *since version 3.10.* 测试一个对象是否为 None 单例，与 Python 中的 x is None 相同。

3.10 版新加入.

int **Py_IsTrue**(const *PyObject* *x)
Part of the Stable ABI *since version 3.10.* 测试一个对象是否为 `True` 单例，与 Python 中的 `x is True` 相同。

3.10 版新加入.

int **Py_IsFalse**(const *PyObject* *x)
Part of the Stable ABI *since version 3.10.* 测试一个对象是否为 `False` 单例，与 Python 中的 `x is False` 相同。

3.10 版新加入.

PyTypeObject ***Py_TYPE**(const *PyObject* *o)
获取 Python 对象 *o* 的类型。

返回一个*borrowed reference*。

Use the `Py_SET_TYPE()` function to set an object type.

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`.

3.9 版新加入.

void **Py_SET_TYPE**(*PyObject* *o, *PyTypeObject* *type)
Set the object *o* type to *type*.

3.9 版新加入.

Py_ssize_t **Py_REFCNT**(const *PyObject* *o)
Get the reference count of the Python object *o*.

3.10 版更變: `Py_REFCNT()` is changed to the inline static function. Use `Py_SET_REFCNT()` to set an object reference count.

void **Py_SET_REFCNT**(*PyObject* *o, *Py_ssize_t* refcnt)
Set the object *o* reference counter to *refcnt*.

3.9 版新加入.

Py_ssize_t **Py_SIZE**(const *PyVarObject* *o)
Get the size of the Python object *o*.

Use the `Py_SET_SIZE()` function to set an object size.

void **Py_SET_SIZE**(*PyVarObject* *o, *Py_ssize_t* size)
Set the object *o* size to *size*.

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,
                    Py_ssize_t nargs,
                    PyObject *kwnames)
```

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_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).

3.7 版新加入.

3.10 版更變: 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.

3.7 版新加入.

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.

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.

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.

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:

域	C Type	含意
`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 类型
T_SHORT	short
T_INT	int
T_LONG	long
T_FLOAT	float
T_DOUBLE	double
T_STRING	const char *
T_OBJECT	PyObject *
T_OBJECT_EX	PyObject *
T_CHAR	char
T_BYTE	char
T_UBYTE	unsigned char
T_UINT	unsigned int
T_USHORT	unsigned short
T_ULONG	unsigned long
T_BOOL	char
T_LONGLONG	long long
T_ULONGLONG	unsigned long long
T_PYSSIZET	Py_ssize_t

`T_OBJECT` 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.

域	C Type	含意
名称	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 function pointer, providing additional data for getter and setter

The `get` function takes one *PyObject** parameter (the instance) and a function 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 function 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 类型对象

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.

与大多数标准类型相比，类型对象相当大。这么大的原因是每个类型对象存储了大量的值，大部分是 C 函数指针，每个指针实现了类型功能的一小部分。本节将详细描述类型对象的字段。这些字段将按照它们在结构中出现的顺序进行描述。

除了下面的快速参考，範例 小节提供了快速了解*PyTypeObject* 的含义和用法的例子。

12.3.1 快速参考

"tp_ 方法槽"

PyTypeObject 槽[1]	*Type*	特殊方法/属性	信息[2]			
			O	T	D	I
<R> *tp_name*	const char *	`__name__`	X	X		
tp_basicsize	*Py_ssize_t*		X	X		X
tp_itemsize	*Py_ssize_t*			X		X
tp_dealloc	*destructor*		X	X		X
tp_vectorcall_offset	*Py_ssize_t*			X		X
(*tp_getattr*)	*getattrfunc*	`__getattribute__`, `__getattr__`				G
(*tp_setattr*)	*setattrfunc*	`__setattr__`, `__delattr__`				G
tp_as_async	*PyAsyncMethods* *	子方法槽（方法域）				%

下页继续

表 1 – 繼續上一頁

PyTypeObject 槽[1]	*Type*	特殊方法/属性	信息[2]			
			O	T	D	I
tp_repr	*reprfunc*	`__repr__`	X	X		X
tp_as_number	*PyNumberMethods* *	子方法槽 (方法域)				%
tp_as_sequence	*PySequenceMethods* *	子方法槽 (方法域)				%
tp_as_mapping	*PyMappingMethods* *	子方法槽 (方法域)				%
tp_hash	*hashfunc*	`__hash__`	X			G
tp_call	*ternaryfunc*	`__call__`		X		X
tp_str	*reprfunc*	`__str__`	X			X
tp_getattro	*getattrofunc*	`__getattribute__`, `__getattr__`	X	X		G
tp_setattro	*setattrofunc*	`__setattr__`, `__delattr__`	X	X		G
tp_as_buffer	*PyBufferProcs* *					%
tp_flags	unsigned long		X	X		?
tp_doc	const char *	`__doc__`	X	X		
tp_traverse	*traverseproc*			X		G
tp_clear	*inquiry*			X		G
tp_richcompare	*richcmpfunc*	`__lt__`, `__le__`, `__eq__`, `__ne__`, `__gt__`, `__ge__`	X			G
tp_weaklistoffset	*Py_ssize_t*			X		?
tp_iter	*getiterfunc*	`__iter__`				X
tp_iternext	*iternextfunc*	`__next__`				X
tp_methods	*PyMethodDef* []		X	X		
tp_members	*PyMemberDef* []			X		
tp_getset	*PyGetSetDef* []		X	X		
tp_base	*PyTypeObject* *	`__base__`			X	
tp_dict	*PyObject* *	`__dict__`			?	
tp_descr_get	*descrgetfunc*	`__get__`				X
tp_descr_set	*descrsetfunc*	`__set__`, `__delete__`				X
tp_dictoffset	*Py_ssize_t*			X		?
tp_init	*initproc*	`__init__`	X	X		X
tp_alloc	*allocfunc*		X		?	?
tp_new	*newfunc*	`__new__`	X	X	?	?
tp_free	*freefunc*		X	X	?	?
tp_is_gc	*inquiry*			X		X
<*tp_bases*>	*PyObject* *	`__bases__`			~	
<*tp_mro*>	*PyObject* *	`__mro__`			~	
[*tp_cache*]	*PyObject* *					
[*tp_subclasses*]	*PyObject* *	`__subclasses__`				
[*tp_weaklist*]	*PyObject* *					
(*tp_del*)	*destructor*					
[*tp_version_tag*]	unsigned int					
tp_finalize	*destructor*	`__del__`				X
tp_vectorcall	*vectorcallfunc*					

[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] 列:
"O": `PyBaseObject_Type` 必须设置
"T": *PyType_Type* 必须设置
"D": 默认设置 (如果方法槽被设置为 NULL)

子方法槽（方法域）

方法槽	Type	特殊方法
am_await	unaryfunc	__await__
am_aiter	unaryfunc	__aiter__
am_anext	unaryfunc	__anext__
am_send	sendfunc	
nb_add	binaryfunc	__add__ __radd__
nb_inplace_add	binaryfunc	__iadd__
nb_subtract	binaryfunc	__sub__ __rsub__
nb_inplace_subtract	binaryfunc	__isub__
nb_multiply	binaryfunc	__mul__ __rmul__
nb_inplace_multiply	binaryfunc	__imul__
nb_remainder	binaryfunc	__mod__ __rmod__
nb_inplace_remainder	binaryfunc	__imod__
nb_divmod	binaryfunc	__divmod__ __rdivmod__
nb_power	ternaryfunc	__pow__ __rpow__
nb_inplace_power	ternaryfunc	__ipow__
nb_negative	unaryfunc	__neg__
nb_positive	unaryfunc	__pos__
nb_absolute	unaryfunc	__abs__
nb_bool	inquiry	__bool__
nb_invert	unaryfunc	__invert__
nb_lshift	binaryfunc	__lshift__ __rlshift__
nb_inplace_lshift	binaryfunc	__ilshift__
nb_rshift	binaryfunc	__rshift__ __rrshift__
nb_inplace_rshift	binaryfunc	__irshift__
nb_and	binaryfunc	__and__ __rand__
nb_inplace_and	binaryfunc	__iand__
nb_xor	binaryfunc	__xor__ __rxor__
nb_inplace_xor	binaryfunc	__ixor__
nb_or	binaryfunc	__or__ __ror__
nb_inplace_or	binaryfunc	__ior__
nb_int	unaryfunc	__int__
nb_reserved	void *	
nb_float	unaryfunc	__float__
nb_floor_divide	binaryfunc	__floordiv__
nb_inplace_floor_divide	binaryfunc	__ifloordiv__
nb_true_divide	binaryfunc	__truediv__

```
X - PyType_Ready sets this value if it is NULL
~ - PyType_Ready always sets this value (it should be NULL)
? - PyType_Ready may set this value depending on other slots

Also see the inheritance column ("I").
```

"I": 继承

```
X - type slot is inherited via *PyType_Ready* if defined with a *NULL* value
% - the slots of the sub-struct are inherited individually
G - inherited, but only in combination with other slots; see the slot's description
? - it's complicated; see the slot's description
```

注意，有些方法槽是通过普通属性查找链有效继承的。

表 2 – 繼續上一頁

方法槽	*Type*	特殊方法
nb_matrix_multiply	*binaryfunc*	__matmul__ __rmatmul__
nb_inplace_matrix_multiply	*binaryfunc*	__imatmul__
mp_length	*lenfunc*	__len__
mp_subscript	*binaryfunc*	__getitem__
mp_ass_subscript	*objobjargproc*	__setitem__, __delitem__
sq_length	*lenfunc*	__len__
sq_concat	*binaryfunc*	__add__
sq_repeat	*ssizeargfunc*	__mul__
sq_item	*ssizeargfunc*	__getitem__
sq_ass_item	*ssizeobjargproc*	__setitem__ __delitem__
sq_contains	*objobjproc*	__contains__
sq_inplace_concat	*binaryfunc*	__iadd__
sq_inplace_repeat	*ssizeargfunc*	__imul__
bf_getbuffer	*getbufferproc()*	
bf_releasebuffer	*releasebufferproc()*	

槽位 typedef

typedef	参数类型	返回类型
allocfunc	*PyTypeObject* * *Py_ssize_t*	*PyObject* *
destructor	void *	void
freefunc	void *	void
traverseproc	void * *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* *	int

更多細節請見下方的*Slot Type typedefs*。

12.3.2 PyTypeObject 定义

PyTypeObject 的结构定义可以在 Include/object.h 中找到。为了方便参考，此处复述了其中的定义:

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;
```

(下页继续)

(繼續上一頁)

```
    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    // Strong reference on a heap type, borrowed reference on a static type
    struct _typeobject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
    freefunc tp_free; /* Low-level free-memory routine */
    inquiry tp_is_gc; /* For PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_cache;
    PyObject *tp_subclasses;
    PyObject *tp_weaklist;
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6 */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;
} PyTypeObject;
```

12.3.3 PyObject 槽位

类型对象结构扩展了 *PyVarObject* 结构。`ob_size` 字段用于动态类型 (由 `type_new()` 创建，通常通过类语句来调用)。注意 *PyType_Type* (元类型) 会初始化 *tp_itemsize*，这意味着它的实例 (即类型对象) 必须具有 `ob_size` 字段。

Py_ssize_t PyObject.**ob_refcnt**

Part of the Stable ABI. 这是类型对象的引用计数，由 `PyObject_HEAD_INIT` 宏初始化为 `1`。请注意对于静态分配的类型对象，类型的实例 (对象的 `ob_type` 指回该类型) 不会被加入引用计数。但对于动态分配的类型对象，实例 确实会被算作引用。

继承：

子类型不继承此字段。

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.

继承：

此字段会被子类型继承。

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.

继承：

这些字段不会被子类型继承。

12.3.4 PyVarObject 槽位

Py_ssize_t PyVarObject.**ob_size**

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.

继承：

子类型不继承此字段。

12.3.5 PyTypeObject 槽

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*).

继承：

子类型不继承此字段。

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.

继承：

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_VarNew(), 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 decrement the reference count for its type object 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);
}
```

继承：

此字段会被子类型继承。

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 *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.

3.8 版更變: 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.

继承：

This field is always inherited. However, the *Py_TPFLAGS_HAVE_VECTORCALL* flag is not always inherited. If it's not, then the subclass won't use *vectorcall*, except when *PyVectorcall_Call()* is explicitly called. This is in particular the case for *heap types* (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.

继承：

分组: 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.

继承:

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.

3.5 版新加入: Formerly known as tp_compare and tp_reserved.

继承:

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.

继承:

此字段会被子类型继承。

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*.

继承:

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*.

继承:

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*.

继承:

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()`.

继承:

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);
```

继承:

此字段会被子类型继承。

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.

继承:

此字段会被子类型继承。

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.

继承：

分组: 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.

继承：

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 使用*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*.

继承：

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.

继承：

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type's value of the flag bit is copied into the subtype together with a pointer to the extension structure. The *Py_TPFLAGS_HAVE_GC* flag bit is inherited together with the tp_traverse and tp_clear fields, i.e. if the Py_TPFLAGS_HAVE_GC flag bit is clear in the subtype and the tp_traverse and tp_clear fields in the subtype exist and have NULL values.

預設：

PyBaseObject_Type 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).

继承：

???

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).

继承：

???

Py_TPFLAGS_READY

This bit is set when the type object has been fully initialized by PyType_Ready().

继承：

???

Py_TPFLAGS_READYING

This bit is set while PyType_Ready() is in the process of initializing the type object.

继承：

???

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 使对象类型支持循环垃圾回收. This bit also implies that the GC-related fields tp_traverse and tp_clear are present in the type object.

继承：

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.

继承：

???

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`.

3.8 版新加入.

继承：

This flag is never inherited by *heap types*. 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.

3.4 版新加入.

3.8 版後已棄用: 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.

继承：

This bit is inherited for *static subtypes* if `tp_call` is also inherited. *Heap types* do not inherit `Py_TPFLAGS_HAVE_VECTORCALL`.

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*.

继承：

This flag is not inherited.

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.

继承：

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F： 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.

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.

继承：

This flag is inherited by types that do not already set *Py_TPFLAGS_SEQUENCE*.

也参考：

PEP 634 ——结构化模式匹配：规范

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.

继承：

This flag is inherited by types that do not already set *Py_TPFLAGS_MAPPING*.

也参考：

PEP 634 ——结构化模式匹配：规范

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.

继承：

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 使对象类型支持循环垃圾回收.

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);
    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.

3.9 版更變: 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.

继承：

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);
    return 0;
}
```

The *Py_CLEAR()* macro should be used, because clearing references is delicate: the reference to the contained object must not be decremented until after the pointer to the contained object is set to NULL. This is because decrementing the reference count 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 使对象类型支持循环垃圾回收.

继承：

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	对照
`Py_LT`	`<`
`Py_LE`	`<=`
`Py_EQ`	`==`
`Py_NE`	`!=`
`Py_GT`	`>`
`Py_GE`	`>=`

定义以下宏是为了简化编写丰富的比较函数:

Py_RETURN_RICHCOMPARE(*VAL_A*, *VAL_B*, *op*)

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 return value's reference count is properly incremented.

On error, sets an exception and returns `NULL` from the function.

3.7 版新加入.

继承:

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.

继承:

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype 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);
```

继承：

此字段会被子类型继承。

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()*.

继承：

此字段会被子类型继承。

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.

继承：

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.

继承：

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.

继承:

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.

继承:

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__()`).

继承:

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);
```

继承:

此字段会被子类型继承。

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.

继承：

此字段会被子类型继承。

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 real dictionary offset in an instance can be computed from a negative *tp_dictoffset* as follows:

```
dictoffset = tp_basicsize + abs(ob_size)*tp_itemsize + tp_dictoffset
if dictoffset is not aligned on sizeof(void*):
    round up to sizeof(void*)
```

where *tp_basicsize*, *tp_itemsize* and *tp_dictoffset* are taken from the type object, and `ob_size` is taken from the instance. The absolute value is taken because ints use the sign of `ob_size` to store the sign of the number. (There's never a need to do this calculation yourself; it is done for you by `_PyObject_GetDictPtr()`.)

继承：

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype instances store the dictionary at a difference offset than the base type. Since the dictionary is always found via *tp_dictoffset*, this should not be a problem.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types has an instance variable dictionary, a dictionary slot is added to the instance layout and the *tp_dictoffset* is set to that slot's offset.

When a type defined by a class statement has a `__slots__` declaration, the type inherits its *tp_dictoffset* from its base type.

(Adding a slot named `__dict__` to the `__slots__` declaration does not have the expected effect, it just causes confusion. Maybe this should be added as a feature just like `__weakref__` though.)

預設：

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.

继承：

此字段会被子类型继承。

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);
```

继承：

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.

继承：

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()*.

继承：

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*.)

继承：

此字段会被子类型继承。

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.

继承：

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*.

继承：

This field is not inherited; it is calculated fresh by *PyType_Ready()*.

PyObject **PyTypeObject*.**tp_cache**

Unused. Internal use only.

继承：

This field is not inherited.

PyObject **PyTypeObject*.**tp_subclasses**

List of weak references to subclasses. Internal use only.

继承：

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.

继承：

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.

继承：

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. */
```

(下页继续)

(繼續上一頁)

```
    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.

继承：

此字段会被子类型继承。

3.4 版新加入.

3.8 版更變: 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.

继承：

This field is never inherited.

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 数字协议 section.

Here is the structure definition:

```
typedef struct {
     binaryfunc nb_add;
     binaryfunc nb_subtract;
     binaryfunc nb_multiply;
     binaryfunc nb_remainder;
     binaryfunc nb_divmod;
     ternaryfunc nb_power;
     unaryfunc nb_negative;
     unaryfunc nb_positive;
     unaryfunc nb_absolute;
     inquiry nb_bool;
     unaryfunc nb_invert;
     binaryfunc nb_lshift;
     binaryfunc nb_rshift;
     binaryfunc nb_and;
     binaryfunc nb_xor;
     binaryfunc nb_or;
     unaryfunc nb_int;
     void *nb_reserved;
     unaryfunc nb_float;

     binaryfunc nb_inplace_add;
     binaryfunc nb_inplace_subtract;
     binaryfunc nb_inplace_multiply;
     binaryfunc nb_inplace_remainder;
     ternaryfunc nb_inplace_power;
     binaryfunc nb_inplace_lshift;
     binaryfunc nb_inplace_rshift;
     binaryfunc nb_inplace_and;
     binaryfunc nb_inplace_xor;
     binaryfunc nb_inplace_or;

     binaryfunc nb_floor_divide;
     binaryfunc nb_true_divide;
     binaryfunc nb_inplace_floor_divide;
     binaryfunc nb_inplace_true_divide;

     unaryfunc nb_index;

     binaryfunc nb_matrix_multiply;
```

(下页继续)

(繼續上一頁)

```
    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()*, PyObject_SetSlice() and PyObject_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 PyExc_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

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);
```

必须返回一个*asynchronous iterator* 对象。请参阅 `__anext__()` 了解详情。

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`.

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 */
    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 */
```

(下页继续)

(繼續上一頁)

```
    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;

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:

```c
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:

```c
typedef struct {
    PyObject_VAR_HEAD
    const char *data[1];
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject) - sizeof(char *),
    .tp_itemsize = sizeof(char *),
};
```

12.11 使对象类型支持循环垃圾回收

Python 对循环引用的垃圾检测与回收需要“容器”对象类型的支持，此类型的容器对象中可能包含其它容器对象。不保存其它对象的引用的类型，或者只保存原子类型（如数字或字符串）的引用的类型，不需要显式提供垃圾回收的支持。

若要创建一个容器类，类型对象的`tp_flags` 字段必须包含`Py_TPFLAGS_HAVE_GC` 并提供一个`tp_traverse` 处理的实现。如果该类型的实例是可变的，还需要实现`tp_clear` 。

Py_TPFLAGS_HAVE_GC

设置了此标志位的类型的对象必须符合此处记录的规则。为方便起见，下文把这些对象称为容器对象。

容器类型的构造函数必须符合两个规则：

1. 必须使用`PyObject_GC_New()` 或`PyObject_GC_NewVar()` 为这些对象分配内存。
2. 初始化了所有可能包含其他容器的引用的字段后，它必须调用`PyObject_GC_Track()` 。

同样的，对象的释放器必须符合两个类似的规则：

1. 在引用其它容器的字段失效前，必须调用`PyObject_GC_UnTrack()` 。
2. 必须使用`PyObject_GC_Del()` 释放对象的内存。

> **警告：** 如果一个类型添加了 Py_TPFLAGS_HAVE_GC，则它 *必须* 实现至少一个`tp_traverse` 句柄或显式地使用来自其一个或多个子类的句柄。
>
> 当调用`PyType_Ready()` 或者 API 中某些间接调用它的函数例如`PyType_FromSpecWithBases()` 或`PyType_FromSpec()` 时解释器就自动填充`tp_flags`, `tp_traverse` 和`tp_clear` 字段，如果该类型是继承自实现了垃圾回收器协议的类并且该子类没有包括`Py_TPFLAGS_HAVE_GC` 旗标的话。

TYPE *`PyObject_GC_New`(TYPE, *PyTypeObject* *type*)

类似于`PyObject_New()`，适用于设置了`Py_TPFLAGS_HAVE_GC`标签的容器对象。

TYPE *`PyObject_GC_NewVar`(TYPE, *PyTypeObject* *type*, *Py_ssize_t* *size*)

类似于`PyObject_NewVar()`，适用于设置了`Py_TPFLAGS_HAVE_GC`标签的容器对象。

TYPE *`PyObject_GC_Resize`(TYPE, *PyVarObject* *op*, *Py_ssize_t* *newsize*)

为`PyObject_NewVar()`所分配对象重新调整大小。返回调整大小后的对象或在失败时返回 `NULL`。*op* 必须尚未被垃圾回收器追踪。

void `PyObject_GC_Track`(*PyObject* *op*)

Part of the Stable ABI. 把对象 *op* 加入到垃圾回收器跟踪的容器对象中。对象在被回收器跟踪时必须保持有效的，因为回收器可能在任何时候开始运行。在`tp_traverse`处理前的所有字段变为有效后，必须调用此函数，通常在靠近构造函数末尾的位置。

int `PyObject_IS_GC`(*PyObject* *obj*)

如果对象实现了垃圾回收器协议则返回非零值，否则返回 0。

如果此函数返回 0 则对象无法被垃圾回收器追踪。

int `PyObject_GC_IsTracked`(*PyObject* *op*)

Part of the Stable ABI *since version 3.9.* 如果 *op* 对象的类型实现了 GC 协议且 *op* 目前正被垃圾回收器追踪则返回 1，否则返回 0。

这类似于 Python 函数 `gc.is_tracked()`。

3.9 版新加入.

int `PyObject_GC_IsFinalized`(*PyObject* *op*)

Part of the Stable ABI *since version 3.9.* 如果 *op* 对象的类型实现了 GC 协议且 *op* 已经被垃圾回收器终结则返回 1，否则返回 0。

这类似于 Python 函数 `gc.is_finalized()`。

3.9 版新加入.

void `PyObject_GC_Del`(void *op*)

Part of the Stable ABI. 释放对象的内存，该对象初始化时由`PyObject_GC_New()`或`PyObject_GC_NewVar()`分配内存。

void `PyObject_GC_UnTrack`(void *op*)

Part of the Stable ABI. 从回收器跟踪的容器对象集合中移除 *op* 对象。请注意可以在此对象上再次调用`PyObject_GC_Track()`以将其加回到被跟踪对象集合。释放器 (`tp_dealloc` 句柄) 应当在`tp_traverse`句柄所使用的任何字段失效之前为对象调用此函数。

3.8 版更變: `_PyObject_GC_TRACK()` 和 `_PyObject_GC_UNTRACK()` 宏已从公有 C API 中移除。

`tp_traverse`处理接收以下类型的函数形参。

typedef int (***visitproc**)(*PyObject* *object, void *arg)

Part of the Stable ABI. 传给`tp_traverse`处理的访问函数的类型。*object* 是容器中需要被遍历的一个对象，第三个形参对应于`tp_traverse`处理的 *arg* 。Python 核心使用多个访问者函数实现循环引用的垃圾检测，不需要用户自行实现访问者函数。

`tp_traverse`处理必须是以下类型：

typedef int (***traverseproc**)(*PyObject* *self, *visitproc* visit, void *arg)

Part of the Stable ABI. 用于容器对象的遍历函数。它的实现必须对 *self* 所直接包含的每个对象调用 *visit* 函数，*visit* 的形参为所包含对象和传给处理程序的 *arg* 值。*visit* 函数调用不可附带 `NULL` 对象作为参数。如果 *visit* 返回非零值，则该值应当被立即返回。

为了简化`tp_traverse`处理的实现，Python 提供了一个`Py_VISIT()`宏。若要使用这个宏，必须把`tp_traverse`的参数命名为 *visit* 和 *arg* 。

void **Py_VISIT**(*PyObject* *o)

如果 *o* 不为 NULL，则调用 *visit* 回调函数，附带参数 *o* 和 *arg*。如果 *visit* 返回一个非零值，则返回该值。使用此宏之后，*tp_traverse* 处理程序的形式如下：

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

tp_clear 处理程序必须为*inquiry* 类型，如果对象不可变则为 NULL。

typedef int (***inquiry**)(*PyObject* *self)

Part of the Stable ABI. 丢弃产生循环引用的引用。不可变对象不需要声明此方法，因为他们不可能直接产生循环引用。需要注意的是，对象在调用此方法后必须仍是有效的（不能对引用只调用*Py_DECREF()* 方法）。当垃圾回收器检测到该对象在循环引用中时，此方法会被调用。

12.11.1 控制垃圾回收器状态

这个 C-API 提供了以下函数用于控制垃圾回收的运行。

Py_ssize_t **PyGC_Collect**(void)

Part of the Stable ABI. 执行完全的垃圾回收，如果垃圾回收器已启用的话。（请注意 gc.collect() 会无条件地执行它。）

返回已回收的 + 无法回收的不可获取对象的数量。如果垃圾回收器被禁用或已在执行回收，则立即返回 0。在垃圾回收期间发生的错误会被传给 sys.unraisablehook。此函数不会引发异常。

int **PyGC_Enable**(void)

Part of the Stable ABI *since version 3.10.* 启用垃圾回收器：类似于 gc.enable()。返回之前的状态，0 为禁用而 1 为启用。

3.10 版新加入.

int **PyGC_Disable**(void)

Part of the Stable ABI *since version 3.10.* 禁用垃圾回收器：类似于 gc.disable()。返回之前的状态，0 为禁用而 1 为启用。

3.10 版新加入.

int **PyGC_IsEnabled**(void)

Part of the Stable ABI *since version 3.10.* 查询垃圾回收器的状态：类似于 gc.isenabled()。返回当前的状态，0 为禁用而 1 为启用。

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**運行時期 (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 的穩定性*。

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`。

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）。

在運行時 (runtime)，區域變數的□釋無法被存取，但全域變數、class 屬性和函式的□解，會分□被儲存在模組、class 和函式的 `__annotations__` 特殊屬性中。

請參□*variable annotation*、*function annotation*、**PEP 484** 和 **PEP 526**，這些章節皆有此功能的□明。關於□釋的最佳實踐方法也請參□ annotations-howto。

argument（引數） 呼叫函式時被傳遞給*function*（或*method*）的值。引數有兩種：

- 關鍵字引數 *(keyword argument)*：在函式呼叫中，以識□字（identifier，例如 `name=`）開頭的引數，或是以 `**` 後面 dictionary（字典）□的值被傳遞的引數。例如，`3` 和 `5` 都是以下 `complex()` 呼叫中的關鍵字引數：

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- 位置引數 *(positional argument)*：不是關鍵字引數的引數。位置引數可在一個引數列表的起始處出現，和（或）作□ `*` 之後的*iterable*（可□代物件）中的元素被傳遞。例如，`3` 和 `5` 都是以下呼叫中的位置引數：

```
complex(3, 5)
complex(*(3, 5))
```

引數會被指定給函式主體中的附名區域變數。關於支配這個指定過程的規則，請參□ calls 章節。在語法上，任何運算式都可以被用來表示一個引數；其評估值會被指定給區域變數。

另請參□術語表的*parameter*（參數）條目、常見問題中的引數和參數之間的差□，以及 **PEP 362**。

asynchronous context manager（非同步情境管理器） 一個可以控制 `async with` 陳述式中所見環境的物件，而它是透過定義 `__aenter__()` 和 `__aexit__()` method（方法）來控制的。由 **PEP 492** 引入。

asynchronous generator（非同步□生器） 一個會回傳*asynchronous generator iterator*（非同步□生器□代器）的函式。它看起來像一個以 `async def` 定義的協程函式 (coroutine function)，但不同的是它包含了 `yield` 運算式，能生成一系列可用於 `async for` □圈的值。

這個術語通常用來表示一個非同步□生器函式，但在某些情境中，也可能是表示非同步□生器□代器 *(asynchronous generator iterator)*。萬一想表達的意思不□清楚，那就使用完整的術語，以避免歧義。

一個非同步□生器函式可能包含 `await` 運算式，以及 `async for` 和 `async with` 陳述式。

asynchronous generator iterator（非同步□生器□代器） 一個由*asynchronous generator*（非同步□生器）函式所建立的物件。

這是一個*asynchronous iterator*（非同步□代器），當它以 `__anext__()` method 被呼叫時，會回傳一個可等待物件 (awaitable object)，該物件將執行非同步□生器的函式主體，直到遇到下一個 `yield` 運算式。

每個 `yield` 會暫停處理程序，□記住位置執行狀態（包括區域變數及擱置中的 try 陳述式）。當非同步□生器□代器以另一個被 `__anext__()` 回傳的可等待物件有效地回復時，它會從停止的地方繼續執行。請參□ **PEP 492** 和 **PEP 525**。

asynchronous iterable（非同步可□代物件） 一個物件，它可以在 `async for` 陳述式中被使用。必須從它的 `__aiter__()` method 回傳一個*asynchronous iterator*（非同步□代器）。由 **PEP 492** 引入。

asynchronous iterator（非同步□代器） 一個實作 `__aiter__()` 和 `__anext__()` method 的物件。`__anext__` 必須回傳一個*awaitable*（可等待物件）。`async for` 會解析非同步□代器的 `__anext__()` method 所回傳的可等待物件，直到它引發 `StopAsyncIteration` 例外。由 **PEP 492** 引入。

attribute（屬性） 一個與某物件相關聯的值，該值大多能透過使用點分隔運算式 (dotted expression) 的名稱被參照。例如，如果物件 *o* 有一個屬性 *a*，則該屬性能以 *o.a* 被參照。

如果一個物件允許，給予該物件一個名稱不是由 identifiers 所定義之識□符 (identifier) 的屬性是有可能的，例如使用 `setattr()`。像這樣的屬性將無法使用點分隔運算式來存取，而是需要使用 `getattr()` 來取得它。

awaitable（可等待物件） 一個可以在 `await` 運算式中被使用的物件。它可以是一個*coroutine*（協程），或是一個有 `__await__()` method 的物件。另請參□ **PEP 492**。

BDFL Benevolent Dictator For Life（終身仁慈獨裁者），又名 Guido van Rossum，Python 的創造者。

binary file（二進制檔案） 一個能□讀取和寫入*bytes-like objects*（類位元組串物件）的*file object*（檔案物件）。二進制檔案的例子有：以二進制模式（`'rb'`、`'wb'` 或 `'rb+'`）開□的檔案、`sys.stdin.buffer`、`sys.stdout.buffer`，以及 `io.BytesIO` 和 `gzip.GzipFile` 實例。

另請參□*text file*（文字檔案），它是一個能□讀取和寫入 `str` 物件的檔案物件。

borrowed reference（借用參照） 在 Python 的 C API 中，借用參照是一個對物件的參照。它不會修改該物件的參照計數 (reference count)。如果該物件被銷□，它會成□一個迷途指標 (dangling pointer)。例如，一次垃圾回收 (garbage collection) 可以移除對物件的最後一個*strong reference*（□參照），而將該物件銷□。

對*borrowed reference* 呼叫`Py_INCREF()` 以將它原地 (in-place) 轉□□*strong reference* 是被建議的做法，除非該物件不能在最後一次使用借用參照之前被銷□。`Py_NewRef()` 函式可用於建立一個新的*strong reference*。

bytes-like object（類位元組串物件） 一個支援*緩衝協定 (Buffer Protocol)*且能□匯出 C-*contiguous* 緩衝區的物件。這包括所有的 `bytes`、`bytearray` 和 `array.array` 物件，以及許多常見的 `memoryview` 物件。類位元組串物件可用於處理二進制資料的各種運算；這些運算包括壓縮、儲存至二進制檔案和透過 socket（插座）發送。

有些運算需要二進制資料是可變的。□明文件通常會將這些物件稱□「可讀寫的類位元組串物件」。可變緩衝區的物件包括 `bytearray`，以及 `bytearray` 的 `memoryview`。其他的運算需要讓二進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□部表示法。該位元組碼也會被暫存在 `.pyc` 檔案中，以便第二次執行同一個檔案時能□更快速（可以不用從原始碼重新編譯□位元組碼）。這種「中間語言 (intermediate language)」據□是運行在一個*virtual machine*（□擬機器）上，該□擬機器會執行與每個位元組碼對應的機器碼 (machine code)。要注意的是，位元組碼理論上是無法在不同的 Python □擬機器之間運作的，也不能在不同版本的 Python 之間保持穩定。

位元組碼的指令列表可以在 dis 模組的□明文件中找到。

callable（可呼叫物件） 一個 callable 是可以被呼叫的物件，呼叫時可能以下列形式帶有一組引數（請見*argument*）：

```
callable(argument1, argument2, ...)
```

一個*function* 與其延伸的*method* 都是 callable。一個有實作 `__call__()` 方法的 class 之實例也是個 callable。

callback（回呼） 作□引數被傳遞的一個副程式 (subroutine) 函式，會在未來的某個時間點被執行。

class（類□） 一個用於建立使用者定義物件的模板。Class 的定義通常會包含 method 的定義，這些 method 可以在 class 的實例上進行操作。

class variable（類□變數） 一個在 class 中被定義，且應該只能在 class 層次（意即不是在 class 的實例中）被修改的變數。

coercion（□制轉型） 在涉及兩個不同型□引數的操作過程中，將某一種型□的實例□□另一種型□的隱式轉□ (implicit conversion) 過程。例如，`int(3.15)` 會將浮點數轉□□整數 `3`，但在 `3+4.5` 中，每個引數是不同的型□（一個 int，一個 float），而這兩個引數必須在被轉□□相同的型□之後才能相加，否則就會引發 `TypeError`。如果□有□制轉型，即使所有的引數型□皆相容，它們都必須要由程式設計師正規化 (normalize) □相同的值，例如，要用 `float(3)+4.5` 而不能只是 `3+4.5`。

complex number（□數） 一個我們熟悉的實數系統的擴充，在此所有數字都會被表示□一個實部和一個□部之和。□數就是□數單位（`-1` 的平方根）的實數倍，此單位通常在數學中被寫□ `i`，在工程學中被寫□ `j`。Python □建了對□數的支援，它是用後者的記法來表示□數；□部會帶著一個後綴的 `j` 被編寫，例如 `3+1j`。若要將 `math` 模組□的工具等效地用於□數，請使用 `cmath` 模組。□數的使用是一個相當進階的數學功能。如果你□有察覺到對它們的需求，那□幾乎能確定你可以安全地忽略它們。

context manager（情境管理器） 一個可以控制 `with` 陳述式中所見環境的物件，而它是透過定義 `__enter__()` 和 `__exit__()` method 來控制的。請參□ **PEP 343**。

context variable（情境變數） 一個變數，其值可以根據上下文的情境而有所不同。這類似執行緒區域儲存區 (Thread-Local Storage)，在其中，一個變數在每個執行緒可能具有不同的值。然而，關於情境變

數，在一個執行緒中可能會有多個情境，而情境變數的主要用途，是在並行的非同步任務 (concurrent asynchronous task) 中，對於變數狀態的追蹤。請參閱 `contextvars`。

contiguous（連續的） 如果一個緩衝區是 *C-contiguous* 或是 *Fortran contiguous*，則它會確切地被視為是連續的。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) 的一種更為廣義的形式。副程式是在某個時間點被進入並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。它們能夠以 `async def` 陳述式被實作。另請參閱 **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，並可能會包含 `await`、`async for` 和 `async with` 關鍵字。這些關鍵字由 **PEP 492** 引入。

CPython Python 程式語言的標准實作 (canonical implementation)，被發布在 python.org 上。「CPython」這個術語在必要時被使用，以區分此實作與其它語言的實作，例如 Jython 或 IronPython。

decorator（裝飾器） 一個函式，它會回傳另一個函式，通常它會使用 `@wrapper` 語法，被應用為一種函式的變換 (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，但在那裡比較不常用。關於裝飾器的更多內容，請參閱函式定義和 class 定義的說明文件。

descriptor（描述器） 任何定義了 `__get__()`、`__set__()` 或 `__delete__()` method 的物件。當一個 class 屬性是一個描述器時，它的特殊連結行為會在屬性查找時被觸發。通常，使用 *a.b* 來取得、設定或刪除某個屬性時，會在 *a* 的 class 字典中查找名稱為 *b* 的物件，但如果 *b* 是一個描述器，則相對應的描述器 method 會被呼叫。對描述器的理解是深入理解 Python 的關鍵，因為它們是許多功能的基礎，這些功能包括函式、method、屬性 (property)、class method、靜態 method，以及對 super class（父類別）的參照。

關於描述器 method 的更多資訊，請參閱 descriptors 或描述器使用指南。

dictionary（字典） 一個關聯陣列 (associative array)，其中任意的鍵會被映射到值。鍵可以是任何帶有 `__hash__()` 和 `__eq__()` method 的物件。在 Perl 中被稱為雜湊 (hash)。

dictionary comprehension（字典綜合運算） 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） 一個在 class、函式或模組中，作為第一個運算式出現的字串文本。雖然它在套件執行時會被忽略，但它會被編譯器辨識，並被放入所屬 class、函式或模組的 `__doc__` 屬性中。由於說明字串可以透過內省 (introspection) 來瀏覽，因此它是物件的說明文件存放的標准位置。

duck-typing（鴨子型□） 一種程式設計風格，它不是藉由檢查一個物件的型□來確定它是否具有正確的介面；取而代之的是，method 或屬性會單純地被呼叫或使用。（「如果它看起來像一□鴨子而且叫起來像一□鴨子，那□它一定是一□鴨子。」）因□□調介面而非特定型□，精心設計的程式碼能讓多形替代 (polymorphic substitution) 來增進它的靈活性。鴨子型□要避免使用 `type()` 或 `isinstance()` 進行測試。（但是請注意，鴨子型□可以用*抽象基底類□ (abstract base class)* 來補充。）然而，它通常會□用 `hasattr()` 測試，或是*EAFP* 程式設計風格。

EAFP Easier to ask for forgiveness than permission.（請求寬恕比請求許可更容易。）這種常見的 Python 編碼風格會先假設有效的鍵或屬性的存在，□在該假設被推翻時再捕獲例外。這種乾□且快速的風格，其特色是存在許多的 `try` 和 `except` 陳述式。該技術與許多其他語言（例如 C）常見的*LBYL* 風格形成了對比。

expression（運算式） 一段可以被評估□求值的語法。□句話□，一個運算式就是文字、名稱、屬性存取、運算子或函式呼叫等運算式元件的累積，而這些元件都能回傳一個值。與許多其他語言不同的是，□非所有的 Python 語言構造都是運算式。另外有一些*statement*（陳述式）不能被用作運算式，例如 `while`。賦值 (assignment) 也是陳述式，而不是運算式。

extension module（擴充模組） 一個以 C 或 C++ 編寫的模組，它使用 Python 的 C API 來與核心及使用者程式碼進行互動。

f-string（f 字串） 以 `'f'` 或 `'F'` □前綴的字串文本通常被稱□「f 字串」，它是格式化的字串文本的縮寫。另請參□ **PEP 498**。

file object（檔案物件） 一個讓使用者透過檔案導向 (file-oriented) API（如 `read()` 或 `write()` 等 method）來操作底層資源的物件。根據檔案物件被建立的方式，它能□協調對真實磁碟檔案或是其他類型的儲存器或通訊裝置（例如標准輸入 / 輸出、記憶體□緩衝區、socket（插座）、管□ (pipe) 等）的存取。檔案物件也被稱□*類檔案物件 (file-like object)* 或*串流 (stream)*。

實際上，有三種檔案物件：原始的*二進制檔案*、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。建立檔案物件的標准方法是使用 `open()` 函式。

file-like object（類檔案物件） *file object*（檔案物件）的同義字。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式） Python 所使用的一種編碼和錯誤處理函式，用來解碼來自作業系統的位元組，以及將 Unicode 編碼到作業系統。

檔案系統編碼必須保證能成功解碼所有小於 128 的位元組。如果檔案系統編碼無法提供此保證，則 API 函式會引發 `UnicodeError`。

`sys.getfilesystemencoding()` 和 `sys.getfilesystemencodeerrors()` 函式可用於取得檔案系統編碼和錯誤處理函式。

filesystem encoding and error handler（檔案系統編碼和錯誤處理函式）會在 Python □動時由 `PyConfig_Read()` 函式來配置：請參□*`filesystem_encoding`*，以及*`PyConfig`* 的成員*`filesystem_errors`*。

另請參□*locale encoding*（區域編碼）。

finder（尋檢器） 一個物件，它會嘗試□正在被 import 的模組尋找*loader*（載入器）。

從 Python 3.3 開始，有兩種類型的尋檢器：*元路徑尋檢器 (meta path finder)* 會使用 `sys.meta_path`，而*路徑項目尋檢器 (path entry finder)* 會使用 `sys.path_hooks`。

請參□ **PEP 302**、**PEP 420** 和 **PEP 451** 以了解更多細節。

floor division（向下取整除法） 向下無條件舍去到最接近整數的數學除法。向下取整除法的運算子是 `//`。例如，運算式 `11 // 4` 的計算結果□ `2`，與 float（浮點數）真除法所回傳的 `2.75` 不同。請注意，`(-11) // 4` 的結果是 `-3`，因□是 `-2.75` 被向下無條件舍去。請參□ **PEP 238**。

function（函式） 一連串的陳述式，它能□向呼叫者回傳一些值。它也可以被傳遞零個或多個*引數*，這些引數可被使用於函式本體的執行。另請參□*parameter*（參數）、*method*（方法），以及 function 章節。

function annotation（函式□釋） 函式參數或回傳值的一個*annotation*（□釋）。

函式□釋通常被使用於型□提示：例如，這個函式預期會得到兩個 `int` 引數，□會有一個 `int` 回傳值：

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

函式□釋的語法在 function 章節有詳細解釋。

請參□*variable annotation* 和 **PEP 484**，皆有此功能的描述。關於□釋的最佳實踐方法，另請參□ annotations-howto。

__future__ future 陳述式：`from __future__ import <feature>`，會指示編譯器使用那些在 Python 未來的發布版本中將成□標准的語法或語義，來編譯當前的模組。而 `__future__` 模組則記□了 *feature*（功能）可能的值。透過 import 此模組□對其變數求值，你可以看見一個新的功能是何時首次被新增到此語言中，以及它何時將會（或已經）成□預設的功能：

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection（垃圾回收） 當記憶體不再被使用時，將其釋放的過程。Python 執行垃圾回收，是透過參照計數 (reference counting)，以及一個能□檢測和中斷參照循環 (reference cycle) 的循環垃圾回收器 (cyclic garbage collector) 來完成。垃圾回收器可以使用 `gc` 模組對其進行控制。

generator（□生器） 一個會回傳*generator iterator*（□生器□代器）的函式。它看起來像一個正常的函式，但不同的是它包含了 `yield` 運算式，能□生一系列的值，這些值可用於 for □圈，或是以 `next()` 函式，每次檢索其中的一個值。

這個術語通常用來表示一個□生器函式，但在某些情境中，也可能是表示□生器□代器。萬一想表達的意思不□清楚，那就使用完整的術語，以避免歧義。

generator iterator（□生器□代器） 一個由*generator*（□生器）函式所建立的物件。

每個 `yield` 會暫停處理程序，□記住位置執行狀態（包括區域變數及擱置中的 try 陳述式）。當□生器□代器回復時，它會從停止的地方繼續執行（與那些每次調用時都要重新開始的函式有所不同）。

generator expression（□生器運算式） 一個會回傳□代器的運算式。它看起來像一個正常的運算式，後面接著一個 `for` 子句，該子句定義了□圈變數、範圍以及一個選擇性的 `if` 子句。該組合運算式會□外層函數□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） 如果一個物件有一個雜□值，該值在其生命□期中永不改變（它需要一個 `__hash__()` method），且可與其他物件互相比較（它需要一個 `__eq__()` method），那□它就是一個可雜□物件。比較結果□相等的多個可雜□物件，它們必須擁有相同的雜□值。

可雜□性 (hashability) 使一個物件可用作 dictionary（字典）的鍵和 set（集合）的成員，因□這些資料結構都在其□部使用了雜□值。

大多數的 Python 不可變□建物件都是可雜□的；可變的容器（例如 list 或 dictionary）□不是；而不可變的容器（例如 tuple（元組）和 frozenset），只有當它們的元素是可雜□的，它們本身才是可雜□的。若物件是使用者自定 class 的實例，則這些物件會被預設□可雜□的。它們在互相比較時都是不相等的（除非它們與自己比較），而它們的雜□值則是衍生自它們的 `id()`。

IDLE Python 的 Integrated Development and Learning Environment（整合開發與學習環境）。idle 是一個基本的編輯器和直譯器環境，它和 Python 的標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（匯入路徑） 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（匯入） 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（匯入器） 一個能□尋找及載入模組的物件；它既是 *finder*（尋檢器）也是 *loader*（載入器）物件。

interactive（互動的） Python 有一個互動式直譯器，這表示你可以在直譯器的提示字元輸入陳述式和運算式，立即執行它們□且看到它們的結果。只要□動 `python`，不需要任何引數（可能藉由從你的電腦的主選單選擇它）。這是測試新想法或檢查模塊和包的非常□大的方法（請記住 help(x)）。

interpreted（直譯的） Python 是一種直譯語言，而不是編譯語言，不過這個區分可能有些模糊，因□有位元組碼 (bytecode) 編譯器的存在。這表示原始檔案可以直接被運行，而不需明確地建立另一個執行檔，然後再執行它。直譯語言通常比編譯語言有更短的開發／除錯□期，不過它們的程式通常也運行得較慢。另請參□ *interactive*（互動的）。

interpreter shutdown（直譯器關閉） 當 Python 直譯器被要求關閉時，它會進入一個特殊階段，在此它逐漸釋放所有被配置的資源，例如模組和各種關鍵□部結構。它也會多次呼叫垃圾回收器 *(garbage collector)*。這能□觸發使用者自定的解構函式 (destructor) 或弱引用的回呼 (weakref callback)，□執行其中的程式碼。在關閉階段被執行的程式碼會遇到各種例外，因□它所依賴的資源可能不再有作用了（常見的例子是函式庫模組或是警告機制）。

直譯器關閉的主要原因，是 `__main__` 模組或正被運行的□本已經執行完成。

iterable（可□代物件） 一種能□一次回傳一個其中成員的物件。可□代物件的例子包括所有的序列型□（像是 `list`、`str` 和 `tuple`）和某些非序列型□，像是 `dict`、檔案物件，以及你所定義的任何 class 物件，只要那些 class 有 `__iter__()` method 或是實作 *Sequence*（序列）語意的 `__getitem__()` method，該物件就是可□代物件。

可□代物件可用於 `for` □圈和許多其他需要一個序列的地方（`zip()`、`map()` ...）。當一個可□代物件作□引數被傳遞給□建函式 `iter()` 時，它會□該物件回傳一個□代器。此□代器適用於針對一組值

進行一遍 (one pass) 運算。使用□代器時，通常不一定要呼叫 `iter()` 或自行處理□代器物件。`for` 陳述式會自動地□你處理這些事，它會建立一個暫時性的未命名變數，用於在□圈期間保有該□代器。另請參□*iterator*（□代器）、*sequence*（序列）和*generator*（□生器）。

iterator（□代器） 一個表示資料流的物件。重□地呼叫□代器的 `__next__()` method（或是將它傳遞給□建函式 `next()`）會依序回傳資料流中的各項目。當不再有資料時，則會引發 `StopIteration` 例外。此時，該□代器物件已被用盡，而任何對其 `__next__()` method 的進一步呼叫，都只會再次引發 `StopIteration`。□代器必須有一個 `__iter__()` method，它會回傳□代器物件本身，所以每個□代器也都是可□代物件，且可以用於大多數適用其他可□代物件的場合。一個明顯的例外，是嘗試多遍□代 (multiple iteration passes) 的程式碼。一個容器物件（像是 `list`）在每次你將它傳遞給 `iter()` 函式或在 `for` □圈中使用它時，都會□生一個全新的□代器。使用□代器嘗試此事（多遍□代）時，只會回傳在前一遍□代中被用過的、同一個已被用盡的□代器物件，使其看起來就像一個空的容器。

在 typeiter 文中可以找到更多資訊。

CPython 實作細節： CPython □不是始終如一地都會檢查「□代器有定義 `__iter__()`」這個規定。

key function（鍵函式） 鍵函式或理序函式 (collation function) 是一個可呼叫 (callable) 函式，它會回傳一個用於排序 (sorting) 或定序 (ordering) 的值。例如，`locale.strxfrm()` 被用來□生一個了解區域特定排序慣例的排序鍵。

Python 中的許多工具，都接受以鍵函式來控制元素被定序或分組的方式。它們包括 `min()`、`max()`、`sorted()`、`list.sort()`、`heapq.merge()`、`heapq.nsmallest()`、`heapq.nlargest()` 和 `itertools.groupby()`。

有幾種方法可以建立一個鍵函式。例如，`str.lower()` method 可以作□不分大小寫排序的鍵函式。或者，一個鍵函式也可以從 `lambda` 運算式被建造，例如 `lambda r: (r[0], r[2])`。另外，`operator` 模組提供了三個鍵函式的建構函式 (constructor)：`attrgetter()`、`itemgetter()` 和 `methodcaller()`。關於如何建立和使用鍵函式的範例，請參□如何排序。

keyword argument（關鍵字引數） 請參□*argument*（引數）。

lambda 由單一*expression*（運算式）所組成的一個匿名行□函式 (inline function)，於該函式被呼叫時求值。建立 lambda 函式的語法是 `lambda [parameters]: expression`

LBYL Look before you leap.（三思而後行。）這種編碼風格會在進行呼叫或查找之前，明確地測試先□條件。這種風格與*EAFP* 方式形成對比，且它的特色是會有許多 `if` 陳述式的存在。

在一個多執行緒環境中，LBYL 方式有在「三思」和「後行」之間引入了競□條件 (race condition) 的風險。例如以下程式碼 `if key in mapping: return mapping[key]`，如果另一個執行緒在測試之後但在查找之前，從 *mapping* 中移除了 *key*，則該程式碼就會失效。這個問題可以用鎖 (lock) 或使用 EAFP 編碼方式來解□。

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`）。

`locale.getpreferredencoding(False)` 可以用來取得區域編碼。

Python 使用*filesystem encoding and error handler*（檔案系統編碼和錯誤處理函式）在 Unicode 檔案名稱和位元組檔案名稱之間進行轉□。

list（串列） 一個 Python □建的*sequence*（序列）。□管它的名字是 list，它其實更類似其他語言中的一個陣列 (array) 而較不像一個鏈結串列 (linked list)，因□存取元素的時間□雜度是 O(1)。

list comprehension（串列綜合運算） 一種用來處理一個序列中的全部或部分元素，□將處理結果以一個 list 回傳的簡要方法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 會□生一個字串 list，其中包含 0 到 255 範圍□，所有偶數的十六進位數 (0x..)。`if` 子句是選擇性的。如果省略它，則 `range(256)` 中的所有元素都會被處理。

loader（載入器） 一個能□載入模組的物件。它必須定義一個名□ `load_module()` 的 method（方法）。載入器通常是被*finder*（尋檢器）回傳。更多細節請參□ **PEP 302**，關於*abstract base class*（抽象基底類□），請參□ `importlib.abc.Loader`。

magic method（魔術方法） *special method*（特殊方法）的一個非正式同義詞。

mapping（對映） 一個容器物件，它支援任意鍵的查找，且能實作 abstract base classes（抽象基底類□）中，`Mapping` 或 `MutableMapping` 所指定的 method。範例包括 `dict`、`collections.defaultdict`、`collections.OrderedDict` 和 `collections.Counter`。

meta path finder（元路徑尋檢器） 一種經由搜尋 `sys.meta_path` 而回傳的*finder*（尋檢器）。元路徑尋檢器與*路徑項目尋檢器 (path entry finder)* 相關但是不同。

關於元路徑尋檢器實作的 method，請參□ `importlib.abc.MetaPathFinder`。

metaclass（元類□） 一種 class 的 class。Class 定義過程會建立一個 class 名稱、一個 class dictionary（字典），以及一個 base class（基底類□）的列表。Metaclass 負責接受這三個引數，□建立該 class。大多數的物件導向程式語言會提供一個預設的實作。Python 的特□之處在於它能□建立自訂的 metaclass。大部分的使用者從未需要此工具，但是當需要時，metaclass 可以提供□大且優雅的解□方案。它們已被用於記□屬性存取、增加執行緒安全性、追□物件建立、實作單例模式 (singleton)，以及許多其他的任務。

更多資訊可以在 metaclasses 章節中找到。

method（方法） 一個在 class 本體□被定義的函式。如果 method 作□其 class 實例的一個屬性被呼叫，則它將會得到該實例物件成□它的第一個*argument*（引數）（此引數通常被稱□ `self`）。請參□*function*（函式）和*nested scope*（巢狀作用域）。

method resolution order（方法解析順序） 方法解析順序是在查找某個成員的過程中，base class（基底類□）被搜尋的順序。關於第 2.3 版至今，Python 直譯器所使用的演算法細節，請參□ Python 2.3 版方法解析順序。

module（模組） 一個擔任 Python 程式碼的組織單位 (organizational unit) 的物件。模組有一個命名空間，它包含任意的 Python 物件。模組是藉由*importing* 的過程，被載入至 Python。

另請參□*package*（套件）。

module spec（模組規格） 一個命名空間，它包含用於載入模組的 import 相關資訊。它是 `importlib.machinery.ModuleSpec` 的一個實例。

MRO 請參□*method resolution order*（方法解析順序）。

mutable（可變物件） 可變物件可以改變它們的值，但維持它們的 `id()`。另請參□*immutable*（不可變物件）。

named tuple（附名元組） 術語「named tuple（附名元組）」是指從 tuple 繼承的任何型□或 class，且它的可索引 (indexable) 元素也可以用附名屬性來存取。這些型□或 class 也可以具有其他的特性。

有些□建型□是 named tuple，包括由 `time.localtime()` 和 `os.stat()` 回傳的值。另一個例子是 `sys.float_info`：

```
>>> sys.float_info[1]                    # indexed access
1024
>>> sys.float_info.max_exp               # named field access
1024
>>> isinstance(sys.float_info, tuple)    # kind of tuple
True
```

有些 named tuple 是□建型□（如上例）。或者，一個 named tuple 也可以從一個正規的 class 定義來建立，只要該 class 是繼承自 `tuple`，且定義了附名欄位 (named field) 即可。這類的 class 可以手工編寫，也可以使用工廠函式 (factory function) `collections.namedtuple()` 來建立。後者技術也增加了一些額外的 method，這些 method 可能是在手寫或□建的 named tuple 中，無法找到的。

namespace（命名空間） 變數被儲存的地方。命名空間是以 dictionary（字典）被實作。有區域的、全域的及□建的命名空間，而在物件中（在 method 中）也有巢狀的命名空間。命名空間藉由防止命名衝突，來支援

模組化。例如，函式 builtins.open 和 os.open() 是透過它們的命名空間來區分彼此。命名空間也藉由明確地區分是哪個模組在實作一個函式，來增進可讀性及可維護性。例如，寫出 random.seed() 或 itertools.islice() 明確地表示，這些函式分別是由 random 和 itertools 模組在實作。

namespace package（命名空間套件） 一個 **PEP 420** *package*（套件），它只能作為子套件 (subpackage) 的一個容器。命名空間套件可能沒有實體的表示法，而且具體來說它們不像是一個*regular package*（正規套件），因為它們沒有 __init__.py 這個檔案。

另請參閱*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） 一個舊名，它是指現在所有的 class 物件所使用的 class 風格。在早期的 Python 版本中，只有新式 class 才能使用 Python 較新的、多樣的功能，像是 __slots__、描述器 (descriptor)、屬性 (property)、__getattribute__()、class method（類別方法）和 static method（靜態方法）。

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） A Python *module* which can contain submodules or recursively, subpackages. Technically, a package is a Python module with a __path__ attribute.

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*（匯入路徑）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） 在 `sys.path_hook` 列表中的一個可呼叫物件 (callable)，若它知道如何在一個特定的*path entry* 中尋找模組，則會回傳一個*path entry finder*（路徑項目尋檢器）。

path based finder（基於路徑的尋檢器） 預設的*元路徑尋檢器 (meta path finder)* 之一，它會在一個*import path* 中搜尋模組。

path-like object（類路徑物件） 一個表示檔案系統路徑的物件。類路徑物件可以是一個表示路徑的 `str` 或 `bytes` 物件，或是一個實作 `os.PathLike` 協定的物件。透過呼叫 `os.fspath()` 函式，一個支援 `os.PathLike` 協定的物件可以被轉換為 `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[F]是*CPython* 實作的一個關鍵元素。sys 模組定義了一個 getrefcount() 函式，程序設計師可以呼叫該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[F]。

另請參[F]*namespace package*（命名空間套件）。

__slots__ 在 class [F]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[F]。

sequence（序列） 一個*iterable*（可[F]代物件），它透過 __getitem__() special method（特殊方法），使用整數索引來支援高效率的元素存取，[F]定義了一個 __len__() method 來回傳該序列的長度。一些[F]建序列型[F]包括 list、str、tuple 和 bytes。請注意，雖然 dict 也支援 __getitem__() 和 __len__()，但它被視[F]對映 (mapping) 而不是序列，因[F]其查找方式是使用任意的*immutable* 鍵，而不是整數。

抽象基底類[F] (abstract base class) collections.abc.Sequence 定義了一個更加豐富的介面，[F]不僅止於 __getitem__() 和 __len__()，還增加了 count()、index()、__contains__() 和 __reversed__()。實作此擴充介面的型[F]，可以使用 register() 被明確地[F][F]。

set comprehension（集合綜合運算） 一種緊密的方法，用來處理一個可[F]代物件中的全部或部分元素，[F]將處理結果以一個 set 回傳。results = {c for c in 'abracadabra' if c not in 'abc'} 會[F]生一個字串 set：{'r', 'd'}。請參[F] 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[F]。

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[F]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[F]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[F]。Special method 在 specialnames 中有詳細[F]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。

strong reference（□參照） 在 Python 的 C API 中，□參照是一個對物件的參照，在它被建立時會增加該物件的參照計數 (reference count)，在它被□除時則會□少該物件的參照計數。

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□。

型□提示是選擇性的，而不是被 Python □制的，但它們對□態型□分析工具很有用，□能協助 IDE 完成程式碼的補全 (completion) 和重構 (refactoring)。

全域變數、class 屬性和函式（不含區域變數）的型□提示，都可以使用 `typing.get_type_hints()` 來存取。

請參□ `typing` 和 **PEP 484**，有此功能的描述。

universal newlines（通用□行字元） 一種解譯文字流 (text stream) 的方式，會將以下所有的情□識□□一行的結束：Unix 行尾慣例 `'\n'`、Windows 慣例 `'\r\n'` 和舊的 Macintosh 慣例 `'\r'`。請參□ **PEP 278** 和 **PEP 3116**，以及用於 `bytes.splitlines()` 的附加用途。

variable annotation（變數□釋） 一個變數或 class 屬性的*annotation*（□釋）。

□釋變數或 class 屬性時，賦值是選擇性的：

```
class C:
    field: 'annotation'
```

變數□釋通常用於型□提示 *(type hint)*：例如，這個變數預期會取得 `int`（整數）值：

```
count: int = 0
```

變數□釋的語法在 annassign 章節有詳細的解釋。

請參□*function annotation*（函式□釋）、**PEP 484** 和 **PEP 526**，皆有此功能的描述。關於□釋的最佳實踐方法，另請參□ annotations-howto。

virtual environment（□擬環境） 一個協作隔離 (cooperatively isolated) 的執行環境，能讓 Python 的使用者和應用程式得以安裝和升級 Python 發□套件，而不會對同一個系統上運行的其他 Python 應用程式的行□□生干擾。

另請參□ `venv`。

virtual machine（□擬機器） 一部完全由軟體所定義的電腦 (computer)。Python 的□擬機器會執行由*bytecode*（位元組碼）編譯器所發出的位元組碼。

Zen of Python（Python 之□） Python 設計原則與哲學的列表，其□容有助於理解和使用此語言。此列表可以透過在互動式提式字元後輸入「`import this`」來找到它。

APPENDIX B

關於這些□明文件

這些□明文件是透過 Sphinx （一個專□ Python □明文件所撰寫的文件處理器）將使用 reStructuredText 撰寫的原始檔轉□而成。

如同 Python 自身，透過自願者的努力下□出文件與封裝後自動化執行工具。若想要回報臭蟲，請見 reporting-bugs 頁面，□含相關資訊。我們永遠歡迎新的自願者加入！

致謝：

- Fred L. Drake, Jr.，原始 Python 文件工具集的創造者以及一大部份□容的作者；
- 創造 reStructuredText 和 Docutils 工具組的 Docutils 專案；
- Fredrik Lundh 先生，Sphinx 從他的 Alternative Python Reference 計劃中獲得許多的好主意。

B.1 Python 文件的貢獻者們

許多人都曾□ Python 這門語言、Python 標准函式庫和 Python □明文件貢獻過。Python 所發□的原始碼中含有部份貢獻者的清單，請見 Misc/ACKS 。

正因□ Python 社群的撰寫與貢獻才有這份這□棒的□明文件 -- 感謝所有貢獻過的人們！

APPENDIX C

沿革與授權

C.1 軟體沿革

Python 是由荷蘭數學和計算機科學研究學會（CWI，見 https://www.cwi.nl/）的 Guido van Rossum 於 1990 年代早期所創造，目的是作□一種稱□ ABC 語言的後繼者。□管 Python 包含了許多來自其他人的貢獻，Guido 仍是其主要作者。

1995 年，Guido 在維吉尼亞州雷斯頓的國家創新研究公司（CNRI，見 https://www.cnri.reston.va.us/）繼續他在 Python 的工作，□在那□發□了該軟體的多個版本。

2000 年五月，Guido 和 Python 核心開發團隊轉移到 BeOpen.com □成立了 BeOpen PythonLabs 團隊。同年十月，PythonLabs 團隊轉移到 Digital Creations（現□ Zope Corporation；見 https://www.zope.org/）。2001 年，Python 軟體基金會（PSF，見 https://www.python.org/psf/）成立，這是一個專□擁有 Python 相關的智慧財□權而創立的非營利組織。Zope Corporation 是 PSF 的一個贊助會員。

所有的 Python 版本都是開源的（有關開源的定義，參□ https://opensource.org/）。歷史上，大多數但非全部的 Python 版本，也是 GPL 相容的；以下表格總結各個版本的差□。

發□版本	源自	年份	擁有者	GPL 相容性？
0.9.0 至 1.2	不適用	1991-1995	CWI	是
1.3 至 1.5.2	1.2	1995-1999	CNRI	是
1.6	1.5.2	2000	CNRI	否
2.0	1.6	2000	BeOpen.com	否
1.6.1	1.6	2001	CNRI	否
2.1	2.0+1.6.1	2001	PSF	否
2.0.1	2.0+1.6.1	2001	PSF	是
2.1.1	2.1+2.0.1	2001	PSF	是
2.1.2	2.1.1	2002	PSF	是
2.1.3	2.1.2	2002	PSF	是
2.2 以上	2.1.1	2001 至今	PSF	是

備□： GPL 相容□不表示我們是在 GPL 下發□ Python。不像 GPL，所有的 Python 授權都可以讓您發□修改

後的版本，但不一定要使您的變更成為開源。GPL 相容的授權使得 Python 可以結合其他在 GPL 下發布的軟體一起使用；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，使得這些版本的發布成為可能。

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 3.10.11 的 PSF 授權合約

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),␣
↪and
   the Individual or Organization ("Licensee") accessing and otherwise using␣
↪Python
   3.10.11 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.10.11 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
↪of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.10.11 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.10.11 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python
   3.10.11.

4. PSF is making Python 3.10.11 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION␣
↪OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT␣
↪THE
   USE OF PYTHON 3.10.11 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.10.11
```

```
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.10.11, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach␣
↪of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This␣
↪License
   Agreement does not grant permission to use PSF trademarks or trade name in␣
↪a
   trademark sense to endorse or promote products or services of Licensee, or␣
↪any
   third party.

8. By copying, installing or otherwise using Python 3.10.11, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 用於 PYTHON 2.0 的 BEOPEN.COM 授權合約

BEOPEN PYTHON 開源授權合約第 1 版

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
```

(下页继续)

(繼續上一頁)

```
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 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
```

(下页继续)

(繼續上一頁)

```
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 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 3.10.11 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 被收錄軟體的授權與致謝

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 發行版本中所收錄的第三方軟體。

C.3.1 Mersenne Twister

_random 模組包含了以 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 的下載內容為基礎的程式碼。以下是原始程式碼的完整聲明：

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

socket 模組使用 getaddrinfo() 和 getnameinfo() 函式，它們在 WIDE 專案（https://www.wide.ad.jp/）F，於不同的原始檔案中被編碼：

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 非同步 socket 服務

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 執行追蹤

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
```

(下页继续)

(繼續上一頁)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

test_epoll 模組包含以下聲明:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

select 模組對於 kqueue 介面包含以下聲明:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

(下页继续)

(繼續上一頁)

```
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.
```

(下页继续)

(繼續上一頁)

```
*
****************************************************************/
```

C.3.12 OpenSSL

如果 OpenSSL 函式庫可被作業系統使用，則 `hashlib`、`posix`、`ssl`、`crypt` 模組會使用它來提升效能。此外，因為 Windows 和 macOS 的 Python 安裝程式可能包含 OpenSSL 函式庫的副本，所以我們也在此收錄 OpenSSL 授權的副本：

```
LICENSE ISSUES
==============

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
---------------

  /* ====================================================================
   * Copyright (c) 1998-2008 The OpenSSL 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. All advertising materials mentioning features or use of this
   *    software must display the following acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
   *
   * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
   *    endorse or promote products derived from this software without
   *    prior written permission. For written permission, please contact
   *    openssl-core@openssl.org.
   *
   * 5. Products derived from this software may not be called "OpenSSL"
   *    nor may "OpenSSL" appear in their names without prior written
   *    permission of the OpenSSL Project.
   *
   * 6. Redistributions of any form whatsoever must retain the following
   *    acknowledgment:
   *    "This product includes software developed by the OpenSSL Project
   *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
```

(下页继续)

(繼續上一頁)

```
 *
 * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
 * EXPRESSED 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 OpenSSL PROJECT OR
 * ITS 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.
 * ====================================================================
 *
 * This product includes cryptographic software written by Eric Young
 * (eay@cryptsoft.com).  This product includes software written by Tim
 * Hudson (tjh@cryptsoft.com).
 *
 */

Original SSLeay License
-----------------------

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
    * All rights reserved.
    *
    * This package is an SSL implementation written
    * by Eric Young (eay@cryptsoft.com).
    * The implementation was written so as to conform with Netscapes SSL.
    *
    * This library is free for commercial and non-commercial use as long as
    * the following conditions are aheared to.  The following conditions
    * apply to all code found in this distribution, be it the RC4, RSA,
    * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
    * included with this distribution is covered by the same copyright terms
    * except that the holder is Tim Hudson (tjh@cryptsoft.com).
    *
    * Copyright remains Eric Young's, and as such any Copyright notices in
    * the code are not to be removed.
    * If this package is used in a product, Eric Young should be given attribution
    * as the author of the parts of the library used.
    * This can be in the form of a textual message at program startup or
    * in documentation (online or textual) provided with the package.
    *
    * 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 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. All advertising materials mentioning features or use of this software
    *    must display the following acknowledgement:
    *    "This product includes cryptographic software written by
```

(下页继续)

(繼續上一頁)

```
 *     Eric Young (eay@cryptsoft.com)"
 *    The word 'cryptographic' can be left out if the rouines from the library
 *    being used are not cryptographic related :-).
 * 4. If you include any Windows specific code (or a derivative thereof) from
 *    the apps directory (application code) you must include an acknowledgement:
 *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
 *
 * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``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.
 *
 * The licence and distribution terms for any publically available version or
 * derivative of this code cannot be changed.  i.e. this code cannot simply be
 * copied and put under another distribution licence
 * [including the GNU Public Licence.]
 */
```

C.3.13 expat

除非在建置 `pyexpat` 擴充時設定了 `--with-system-expat`，否則該擴充會用一個包含 expat 原始碼的副本來建置：

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

除非在建置 _ctypes 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，則該擴充會用一個內含 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
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.
```

(下页继续)

(繼續上一頁)

```
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.

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:

```
Programming the AdLib/Sound Blaster
FM Music Chips
Version 2.0 (24 Feb 1992)
Copyright (c) 1991, 1992 by Jeffrey S. Lee
jlee@smylex.uucp
Warranty and Copyright Policy
This document is provided on an "as-is" basis, and its author makes
no warranty or representation, express or implied, with respect to
its quality performance or fitness for a particular purpose.  In no
event will the author of this document be liable for direct, indirect,
special, incidental, or consequential damages arising out of the use
or inability to use the information contained within.  Use of this
document is at your own risk.
This file may be used and copied freely so long as the applicable
copyright notices are retained, and no modifications are made to the
text of the document.  No money shall be charged for its distribution
beyond reasonable shipping, handling and duplication costs, nor shall
proprietary changes be made to this document so that it cannot be
distributed freely.  This document may not be included in published
material or commercial packages without the written consent of its
author.
```

APPENDIX D

版權宣告

Python 和這份說明文件的版權：

完整的授權條款資訊請參見沿革與授權。

索引

非字母

E

F

G

H

I

K

L

M