The Python/C API

發Ⓕ *3.9.13*

**Guido van Rossum
and the Python development team**

9 月 06, 2022

Contents

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

简介

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⊞: 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 。查看语句解释及变量编译 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⊞: 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。

头文件通常会与 Python 一起安装。在 Unix 上，它们位于以下目录: *prefix*/include/pythonversion/ 和 *exec_prefix*/include/pythonversion/，其中 prefix 和 exec_prefix 是由向 Python 的 **configure** 脚本传入的对应形参所定义，而 *version* 则为 '%d.%d' % sys.version_info[:2]。在 Windows 上，头文件安装于 *prefix*/include，其中 prefix 是向安装程序指定的安装目录。

要包含头文件，请将两个目录（如果不同）都放到你所用编译器的包含搜索路径中。请 不要将父目录放入搜索路径然后使用 #include <pythonX.Y/Python.h>；这将使得多平台编译不可用，因为 prefix 下平台无关的头文件需要包含来自 exec_prefix 下特定平台的头文件。

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)
> 返回 x 的绝对值。

> 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。

示例：

```
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。

示例：

```
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。

示例：

```
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

多数 Python/C API 有一个或多个参数，以及一个 *PyObject** 类型的返回值。这种类型是指向任意 Python 对象的不透明数据类型的指针。所有 Python 对象类型在大多数情况下都被 Python 语言由相同的方式处理（例如，赋值，作用域规则，和参数传递），因此将它们由单个 C 类型表示才合适。几乎所有 Python 对象存放在堆中：你不能声明一个类型为 *PyObject* 的自动或静态的变量，只能声明类型为 *PyObject** 的指针。type 对象是唯一的例外，因为它们永远不能被释放，所以它们通常是静态的 *PyTypeObject* 对象。

所有 Python 对象（其至 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

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

Conversely, when a calling function passes in a reference to an object, there are two possibilities: the function *steals* a reference to the object, or it does not. *Stealing a reference* means that when you pass a reference to a function, that function assumes that it now owns that reference, and you are not responsible for it any longer.

Few functions steal references; the two notable exceptions are *PyList_SetItem()* and *PyTuple_SetItem()*, which steal a reference to the item (but not to the tuple or list into which the item is put!). These functions were designed to steal a reference because of a common idiom for populating a tuple or list with newly created objects; for example, the code to create the tuple (1, 2, "three") could look like this (forgetting about error handling for the moment; a better way to code this is shown below):

```
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"));
```

Here, *PyLong_FromLong()* returns a new reference which is immediately stolen by *PyTuple_SetItem()*. When you want to keep using an object although the reference to it will be stolen, use *Py_INCREF()* to grab another reference before calling the reference-stealing function.

Incidentally, *PyTuple_SetItem()* is the *only* way to set tuple items; *PySequence_SetItem()* and *PyObject_SetItem()* refuse to do this since tuples are an immutable data type. You should only use *PyTuple_SetItem()* for tuples that you are creating yourself.

Equivalent code for populating a list can be written using *PyList_New()* and *PyList_SetItem()*.

However, in practice, you will rarely use these ways of creating and populating a tuple or list. There's a generic function, *Py_BuildValue()*, that can create most common objects from C values, directed by a *format string*. For example, the above two blocks of code could be replaced by the following (which also takes care of the error checking):

```
PyObject *tuple, *list;

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

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

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

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

The situation is slightly different for function return values. While passing a reference to most functions does not change your ownership responsibilities for that reference, many functions that return a reference to an object give you ownership of the reference. The reason is simple: in many cases, the returned object is created on the fly, and the reference you get is the only reference to the object. Therefore, the generic functions that return object references, like *PyObject_GetItem()* and *PySequence_GetItem()*, always return a new reference (the caller becomes the owner of the reference).

It is important to realize that whether you own a reference returned by a function depends on which function you call only --- *the plumage* (the type of the object passed as an argument to the function) *doesn't enter into it!* Thus, if you extract an item from a list using *PyList_GetItem()*, you don't own the reference --- but if you obtain the same item from the same list using *PySequence_GetItem()* (which happens to take exactly the same arguments), you do own a reference to the returned object.

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()*。

```c
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;
}
```

```c
long
sum_sequence(PyObject *sequence)
{
```

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

1.4.2 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.

Py_ssize_t

A signed integral type such that `sizeof(Py_ssize_t) == sizeof(size_t)`. C99 doesn't define such a thing directly (size_t is an unsigned integral type). See **PEP 353** for details. `PY_SSIZE_T_MAX` is the largest positive value of type *Py_ssize_t*.

1.5 异常

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。

For C programmers, however, error checking always has to be explicit. All functions in the Python/C API can raise exceptions, unless an explicit claim is made otherwise in a function's documentation. In general, when a function encounters an error, it sets an exception, discards any object references that it owns, and returns an error indicator. If not documented otherwise, this indicator is either `NULL` or `-1`, depending on the function's return type. A few functions return a Boolean true/false result, with false indicating an error. Very few functions return no explicit error indicator or have an ambiguous return value, and require explicit testing for errors with *PyErr_Occurred()*. These exceptions are always explicitly documented.

Exception state is maintained in per-thread storage (this is equivalent to using global storage in an unthreaded application). A thread can be in one of two states: an exception has occurred, or not. The function *PyErr_Occurred()* can be used to check for this: it returns a borrowed reference to the exception type object when an exception has occurred, and `NULL`

otherwise. There are a number of functions to set the exception state: *PyErr_SetString()* is the most common (though not the most general) function to set the exception state, and *PyErr_Clear()* clears the exception state.

The full exception state consists of three objects (all of which can be NULL): the exception type, the corresponding exception value, and the traceback. These have the same meanings as the Python result of sys.exc_info(); however, they are not the same: the Python objects represent the last exception being handled by a Python try ... except statement, while the C level exception state only exists while an exception is being passed on between C functions until it reaches the Python bytecode interpreter's main loop, which takes care of transferring it to sys.exc_info() and friends.

Note that starting with Python 1.5, the preferred, thread-safe way to access the exception state from Python code is to call the function sys.exc_info(), which returns the per-thread exception state for Python code. Also, the semantics of both ways to access the exception state have changed so that a function which catches an exception will save and restore its thread's exception state so as to preserve the exception state of its caller. This prevents common bugs in exception handling code caused by an innocent-looking function overwriting the exception being handled; it also reduces the often unwanted lifetime extension for objects that are referenced by the stack frames in the traceback.

As a general principle, a function that calls another function to perform some task should check whether the called function raised an exception, and if so, pass the exception state on to its caller. It should discard any object references that it owns, and return an error indicator, but it should *not* set another exception --- that would overwrite the exception that was just raised, and lose important information about the exact cause of the error.

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

```python
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：

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

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

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

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

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

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

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

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

This example represents an endorsed use of the `goto` statement in C! It illustrates the use of `PyErr_ExceptionMatches()` and `PyErr_Clear()` to handle specific exceptions, and the use of `Py_XDECREF()` to dispose of owned references that may be `NULL` (note the `'X'` in the name; `Py_DECREF()` would crash when confronted with a `NULL` reference). It is important that the variables used to hold owned references are initialized to `NULL` for this to work; likewise, the proposed return value is initialized to `-1` (failure) and only set to success after the final call made is successful.

1.6 嵌入式 Python

The one important task that only embedders (as opposed to extension writers) of the Python interpreter have to worry about is the initialization, and possibly the finalization, of the Python interpreter. Most functionality of the interpreter can only be used after the interpreter has been initialized.

The basic initialization function is `Py_Initialize()`. This initializes the table of loaded modules, and creates the fundamental modules `builtins`, `__main__`, and `sys`. It also initializes the module search path (`sys.path`).

`Py_Initialize()` does not set the "script argument list" (`sys.argv`). If this variable is needed by Python code that will be executed later, it must be set explicitly with a call to `PySys_SetArgvEx(argc, argv, updatepath)` after the call to `Py_Initialize()`.

On most systems (in particular, on Unix and Windows, although the details are slightly different), `Py_Initialize()` calculates the module search path based upon its best guess for the location of the standard Python interpreter executable, assuming that the Python library is found in a fixed location relative to the Python interpreter executable. In particular, it looks for a directory named `lib/python`*X.Y* relative to the parent directory where the executable named `python` is found on the shell command search path (the environment variable `PATH`).

For instance, if the Python executable is found in `/usr/local/bin/python`, it will assume that the libraries are in `/usr/local/lib/python`*X.Y*. (In fact, this particular path is also the "fallback" location, used when no executable file named `python` is found along `PATH`.) The user can override this behavior by setting the environment variable `PYTHONHOME`, or insert additional directories in front of the standard path by setting `PYTHONPATH`.

The embedding application can steer the search by calling `Py_SetProgramName(file)` *before* calling `Py_Initialize()`. Note that `PYTHONHOME` still overrides this and `PYTHONPATH` is still inserted in front of the standard path. An application that requires total control has to provide its own implementation of `Py_GetPath()`,

Py_GetPrefix(), *Py_GetExecPrefix()*, and *Py_GetProgramFullPath()* (all defined in `Modules/getpath.c`).

Sometimes, it is desirable to "uninitialize" Python. For instance, the application may want to start over (make another call to *Py_Initialize()*) or the application is simply done with its use of Python and wants to free memory allocated by Python. This can be accomplished by calling *Py_FinalizeEx()*. The function *Py_IsInitialized()* returns true if Python is currently in the initialized state. More information about these functions is given in a later chapter. Notice that *Py_FinalizeEx()* does *not* free all memory allocated by the Python interpreter, e.g. memory allocated by extension modules currently cannot be released.

1.7 调试构建

Python can be built with several macros to enable extra checks of the interpreter and extension modules. These checks tend to add a large amount of overhead to the runtime so they are not enabled by default.

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.

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

除了前面描述的引用计数调试之外，还执行以下额外检查:

- 额外检查将添加到对象分配器。
- 额外的检查将添加到解析器和编译器中。
- 检查从宽类型向窄类型的向下强转是否损失了信息。
- 许多断言被添加到字典和集合实现中。另外，集合对象需要 `test_c_api()` 方法。
- 输入参数的完整性检查被添加到框架创建中。
- 使用已知的无效模式初始化整型的存储，以捕获对未初始化数字的引用。
- 添加底层跟踪和额外的异常检查到虚拟机的运行时中。
- 添加额外的检查到 arena 内存实现。
- 添加额外调试到线程模块。

这里可能没有提到的额外的检查。

Defining `Py_TRACE_REFS` enables reference tracing. When defined, a circular doubly linked list of active objects is maintained by adding two extra fields to every *PyObject*. Total allocations are tracked as well. Upon exit, all existing references are printed. (In interactive mode this happens after every statement run by the interpreter.) Implied by `Py_DEBUG`.

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

稳定的应用程序二进制接口

传统上，Python 的 C API 将随每个版本而变化。大多数更改都与源代码兼容，通常只添加 API，而不是更改现有 API 或删除 API（尽管某些接口会首先弃用然后再删除）。

不幸的是，API 兼容性没有扩展到二进制兼容性（ABI）。原因主要是结构定义的演变，在这里添加新字段或更改字段类型可能不会破坏 API，但可能会破坏 ABI。因此，每个 Python 版本都需要重新编译扩展模块（即使在未使用任何受影响的接口的情况下，Unix 上也可能会出现异常）。此外，在 Windows 上，扩展模块与特定的 pythonXY.dll 链接，需要重新编译才能与新的 pythonXY.dll 链接。

从 Python3.2 起，已经声明了一个 API 的子集，以确保稳定的 ABI。如果使用此 API（也被称为"受限 API"）的扩展模块需要定义 "Py_LIMITED_API"。许多解释器细节将从扩展模块中隐藏; 反过来，在任何 3.x 版本（x>=2）上构建的模块都不需要重新编译

在某些情况下，需要添加新函数来扩展稳定版 ABI。希望使用这些新 API 的扩展模块需要将 `Py_LIMITED_API` 设置为他们想要支持的最低 Python 版本的 PY_VERSION_HEX 值（例如: Python 3.3 为 `0x03030000`）（参见*API 和 ABI 版本管理*）。此类模块将适用于所有后续 Python 版本，但无法在旧版本上加载（因为缺少符号）。

从 Python 3.2 开始，受限 API 可用的函数集记录在 **PEP 384** 。在 C API 文档中，不属于受限 API 的 API 元素标记为"不属于受限 API"。

CHAPTER 3

极高层级

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。

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

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

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*)

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*)

If *fp* refers to a file associated with an interactive device (console or terminal input or Unix pseudo-terminal), re-

turn the value of *PyRun_InteractiveLoop()*, otherwise return the result of *PyRun_SimpleFile()*. *filename* is decoded from the filesystem encoding (sys.getfilesystemencoding()). If *filename* is NULL, this function uses "???" as the filename. If *closeit* is true, the file is closed before PyRun_SimpleFileExFlags() returns.

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

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

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

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

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

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

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

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

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

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

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

备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*)

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

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*)

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

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

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 版更变: 结果必须由 *PyMem_RawMalloc()* 或 *PyMem_RawRealloc()* 分配，而不是由 *PyMem_Malloc()* 或 *PyMem_Realloc()* 分配。

struct _node* **PyParser_SimpleParseString** (const char *str*, int *start*)
> This is a simplified interface to *PyParser_SimpleParseStringFlagsFilename()* below, leaving *filename* set to NULL and *flags* set to 0.

> Deprecated since version 3.9, will be removed in version 3.10.

struct _node* **PyParser_SimpleParseStringFlags** (const char *str*, int *start*, int *flags*)
> This is a simplified interface to *PyParser_SimpleParseStringFlagsFilename()* below, leaving *filename* set to NULL.

> Deprecated since version 3.9, will be removed in version 3.10.

struct _node* **PyParser_SimpleParseStringFlagsFilename** (const char *str*, const char *filename*, int *start*, int *flags*)
> Parse Python source code from *str* using the start token *start* according to the *flags* argument. The result can be used to create a code object which can be evaluated efficiently. This is useful if a code fragment must be evaluated many times. *filename* is decoded from the filesystem encoding (sys.getfilesystemencoding()).

> Deprecated since version 3.9, will be removed in version 3.10.

struct _node* **PyParser_SimpleParseFile** (FILE *fp*, const char *filename*, int *start*)
> This is a simplified interface to *PyParser_SimpleParseFileFlags()* below, leaving *flags* set to 0.

> Deprecated since version 3.9, will be removed in version 3.10.

struct _node* **PyParser_SimpleParseFileFlags** (FILE *fp*, const char *filename*, int *start*, int *flags*)
> Similar to *PyParser_SimpleParseStringFlagsFilename()*, but the Python source code is read from *fp* instead of an in-memory string.

> Deprecated since version 3.9, will be removed in version 3.10.

*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. Similar to *PyRun_StringFlags()*, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file, it is decoded from the filesystem encoding (sys.getfilesystemencoding()). If *closeit* is true, the file is closed before *PyRun_FileExFlags()* returns.

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

 Return value: New reference. 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. Like *Py_CompileStringObject()*, but *filename* is a byte string decoded from the filesystem encoding (os.fsdecode()).

 3.2 版新加入.

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

 Return value: New reference. 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. 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。

PyFrameObject

 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. 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. 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开始符号；应当与*Py_CompileString()* 一起使用。

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: 增加了 *cf_feature_version* 字段。

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"。

参照計数

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。对象必须不为 NULL；如果你不确定它不为 NULL，可使用 *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。

void **Py_DECREF** (*PyObject* *o*)
> 减少对象 *o* 的引用计数。对象必须不为 NULL；如果你不确定它不为 NULL，可使用 *Py_XDECREF()*。
> 如果引用计数降为零，将发起调用对象所属类型的释放函数 (它必须不为 NULL)。

> > **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。

以下函数适用于 Python 的运行时动态嵌入: Py_IncRef(PyObject *o), Py_DecRef(PyObject *o)。
它们分别只是 *Py_XINCREF()* 和 *Py_XDECREF()* 的简单导出函数版本。

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。

例外處理

本章描述的函数将让你处理和触发 Python 异常。了解一些 Python 异常处理的基本知识是很重要的。它的工作原理有点像 POSIX 的 errno 变量: (每个线程) 有一个全局指示器显示最近发生的错误。大多数 C API 函数不会在成功时理会它，但会在失败时设置它来指示错误的原因。多数 C API 函数也返回一个错误指示器，如果它们应该返回一个指针，通常返回 NULL，如果返回一个整数，则返回 -1(例外: PyArg_*() 函数成功时返回 1 而失败时返回 0)。

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

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

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

5.1 打印和清理

void **PyErr_Clear**()
    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*)
    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**()
> PyErr_PrintEx(1) 的别名。

void **PyErr_WriteUnraisable**(*PyObject *obj*)
> 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*)
> This is the most common way to set the error indicator. The first argument specifies the exception type; it is normally one of the standard exceptions, e.g. PyExc_RuntimeError. You need not increment its reference count. The second argument is an error message; it is decoded from 'utf-8'.

void **PyErr_SetObject**(*PyObject *type*, *PyObject *value*)
> 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.* 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.* 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*)
> 这是 PyErr_SetObject(type, Py_None) 的简写。

int **PyErr_BadArgument**()
> 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.* 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.* 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. 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. 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. 类似于 `PyErr_SetFromErrnoWithFilenameObject()`，但文件名以 C 字符串形式给出。*filename* 是从文件系统编码（`os.fsdecode()`）解码出来的。

PyObject* **PyErr_SetFromWindowsErr** (int *ierr*)
　　Return value: Always NULL. 这是触发 WindowsError 的便捷函数。如果 *ierr* 为 `0`，则改用调用 GetLastError() 返回的错误代码。它调用 Win32 函数 FormatMessage() 来检索 *ierr* 或 GetLastError() 给定的错误代码的 Windows 描述，然后构造一个元组对象，其第一项是 *ierr* 值，第二项是相应的错误信息（从 FormatMessage() 获取），然后调用 `PyErr_SetObject(PyExc_WindowsError, object)`。该函数永远返回 NULL。

　　可用性: Windows。

PyObject* **PyErr_SetExcFromWindowsErr** (*PyObject* *type, int *ierr*)
　　Return value: Always NULL. 类似于 `PyErr_SetFromWindowsErr()`，额外的参数指定要触发的异常类型。

　　可用性: Windows。

PyObject* **PyErr_SetFromWindowsErrWithFilename** (int *ierr*, const char *filename)
　　Return value: Always NULL. 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。

　　可用性: Windows。

PyObject* **PyErr_SetExcFromWindowsErrWithFilenameObject** (*PyObject* *type, int *ierr*, *PyObject* *filename)
　　Return value: Always NULL. 类似于 PyErr_SetFromWindowsErrWithFilenameObject()，额外参数指定要触发的异常类型。

　　可用性: Windows。

PyObject* **PyErr_SetExcFromWindowsErrWithFilenameObjects** (*PyObject* *type, int *ierr*, *PyObject* *filename, *PyObject* *filename2)
　　Return value: Always NULL. 类似于 `PyErr_SetExcFromWindowsErrWithFilenameObject()`，但是接受第二个 filename 对象。

　　可用性: Windows。

　　3.4 版新加入.

PyObject* **PyErr_SetExcFromWindowsErrWithFilename** (*PyObject* *type, int *ierr*, const char *filename)
　　Return value: Always NULL. 类似于 `PyErr_SetFromWindowsErrWithFilename()`，额外参数指定要触发的异常类型。

　　可用性: Windows。

PyObject* **PyErr_SetImportError** (*PyObject *msg*, *PyObject *name*, *PyObject *path*)
> *Return value: Always NULL.* 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.* 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*)
> 与 *PyErr_SyntaxLocationObject()* 类似，只是 *filename* 是从文件系统编码（os.fsdecode()）解码出的一个字节字符串。

> 3.2 版新加入.

void **PyErr_SyntaxLocation** (const char **filename*, int *lineno*)
> Like *PyErr_SyntaxLocationEx()*, but the *col_offset* parameter is omitted.

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

> 3.4 版新加入.

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

Similar to *PyErr_WarnExplicitObject()* except that *message* and *module* are UTF-8 encoded strings, and *filename* is decoded from the filesystem encoding (os.fsdecode()).

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

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*, ...)

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. 检测是否设置了错误指示器。如已设置，则返回异常 *type* (传给最近一次对某个 PyErr_Set*() 函数或*PyErr_Restore()* 的调用的第一个参数)。如未设置，则返回 NULL。你并不拥有对返回值的引用，因此你不需要对它执行*Py_DECREF()*。

呼叫者必须持有 GIL。

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

---

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

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*)

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

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

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

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

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

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

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

---

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

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

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

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

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

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

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

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

Retrieve the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. Returns new references for the three objects, any of which may be NULL. Does not modify the exception info state.

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

3.3 版新加入.

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

Set the exception info, as known from `sys.exc_info()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. This function steals the references of the arguments. To clear the exception state, pass NULL for all three arguments. For general rules about the three arguments, see *PyErr_Restore()*.

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

3.3 版新加入.

5.5 信号处理

int **PyErr_CheckSignals**()
> This function interacts with Python's signal handling. It checks whether a signal has been sent to the processes and if so, invokes the corresponding signal handler. If the signal module is supported, this can invoke a signal handler written in Python. In all cases, the default effect for SIGINT is to raise the KeyboardInterrupt exception. If an exception is raised the error indicator is set and the function returns −1; otherwise the function returns 0. The error indicator may or may not be cleared if it was previously set.

void **PyErr_SetInterrupt**()
> Simulate the effect of a SIGINT signal arriving. The next time *PyErr_CheckSignals()* is called, the Python signal handler for SIGINT will be called.

> 如果 Python 没有处理 signal.SIGINT (将它设为 signal.SIG_DFL 或 signal.SIG_IGN)，此函数将不做任何事。

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

> 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 Exception 类

*PyObject** **PyErr_NewException**(const char **name*, *PyObject **base*, *PyObject **dict*)
> *Return value: New reference.* This utility function creates and returns a new exception class. The *name* argument must be the name of the new exception, a C string of the form module.classname. The *base* and *dict* arguments are normally NULL. This creates a class object derived from Exception (accessible in C as PyExc_Exception).

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

*PyObject** **PyErr_NewExceptionWithDoc**(const char **name*, const char **doc*, *PyObject **base*, *PyObject **dict*)
> *Return value: New reference.* 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 异常对象

PyObject* **PyException_GetTraceback** (*PyObject* **ex*)
> *Return value: New reference.* Return the traceback associated with the exception as a new reference, as accessible from Python through `__traceback__`. If there is no traceback associated, this returns `NULL`.

int **PyException_SetTraceback** (*PyObject* **ex*, *PyObject* **tb*)
> 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.* Return the context (another exception instance during whose handling *ex* was raised) associated with the exception as a new reference, as accessible from Python through `__context__`. If there is no context associated, this returns `NULL`.

void **PyException_SetContext** (*PyObject* **ex*, *PyObject* **ctx*)
> 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.* Return the cause (either an exception instance, or `None`, set by `raise ... from ...`) associated with the exception as a new reference, as accessible from Python through `__cause__`.

void **PyException_SetCause** (*PyObject* **ex*, *PyObject* **cause*)
> Set the cause associated with the exception to *cause*. Use `NULL` to clear it. There is no type check to make sure that *cause* is either an exception instance or `None`. This steals a reference to *cause*.
>
> `__suppress_context__` 会被此函数隐式地设为 `True`。

5.8 Unicode 异常对象

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

PyObject* **PyUnicodeDecodeError_Create** (const char **encoding*, const char **object*, *Py_ssize_t* *length*, *Py_ssize_t* *start*, *Py_ssize_t* *end*, const char **reason*)
> *Return value: New reference.* Create a `UnicodeDecodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.

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.* Create a `UnicodeEncodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.
>
> 3.3 版後已弃用: 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.* Create a `UnicodeTranslateError` object with the attributes *object*, *length*, *start*, *end* and *reason*. *reason* is a UTF-8 encoded string.
>
> 3.3 版後已弃用: 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.* 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.* 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)

> Get the *start* attribute of the given exception object and place it into *start. *start* must not be NULL. Return 0 on success, −1 on failure.

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

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

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

> Set the *start* attribute of the given exception object to *start*. Return 0 on success, −1 on failure.

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

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

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

> 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*)

> 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.* 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)

> 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

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

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

> 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。
>
> The function then checks if the recursion limit is reached. If this is the case, a RecursionError is set and a nonzero value is returned. Otherwise, zero is returned.
>
> *where* 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: This function is now also available in the limited API.

void **Py_LeaveRecursiveCall** (void)

Ends a *Py_EnterRecursiveCall()*. Must be called once for each *successful* invocation of *Py_EnterRecursiveCall()*.

3.9 版更變: This function is now also available in the limited API.

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

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

Called at the beginning of the *tp_repr* implementation to detect cycles.

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

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

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

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

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

5.10 标准异常

所有的 Python 标准异常都可用作全局变量, 其名称为 PyExc_ 跟上 Python 异常名称。这些变量是*PyObject** 类型；都是类对象。下面列出了全部这些用作标准异常的变量：

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	

表 1 – 繼續上一頁

C 名稱	Python 名稱	⬚解
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	
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 and PyExc_TimeoutError 介绍如下 **PEP 3151**.

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	⬚解
PyExc_EnvironmentError	
PyExc_IOError	
PyExc_WindowsError	[2]

---

[1] 这是其他标准异常的基类。

3.3 版更變: 这些别名曾经是单独的异常类型。

⊞解:

5.11 标准警告类别

所有的标准 Python 警告类别都可以用作全局变量，其名称为 "PyExc_" 跟上 Python 异常名称。这些变量是 *PyObject** 类型；都是类对象。以下列出了所有用作警告的变量：

C 名称	Python 名称	⊞解
PyExc_Warning	Warning	[3]
PyExc_BytesWarning	BytesWarning	
PyExc_DeprecationWarning	DeprecationWarning	
PyExc_FutureWarning	FutureWarning	
PyExc_ImportWarning	ImportWarning	
PyExc_PendingDeprecationWarning	PendingDeprecationWarning	
PyExc_ResourceWarning	ResourceWarning	
PyExc_RuntimeWarning	RuntimeWarning	
PyExc_SyntaxWarning	SyntaxWarning	
PyExc_UnicodeWarning	UnicodeWarning	
PyExc_UserWarning	UserWarning	

3.2 版新加入: PyExc_ResourceWarning.

⊞解:

---

[2] 仅在 Windows 中定义；检测是否定义了预处理程序宏 MS_WINDOWS ，以便保护用到它的代码。
[3] 这是其他标准警告类别的基类。

工具

本章中的函数执行各种实用工具任务，包括帮助 C 代码提升跨平台可移植性，在 C 中使用 Python 模块，以及解析函数参数并根据 C 中的值构建 Python 中的值等等。

6.1 作業系統工具

PyObject* **PyOS_FSPath** (*PyObject *path*)
Return value: New reference. Return the file system representation for *path*. If the object is a str or bytes object, then its reference count is incremented. If the object implements the os.PathLike interface, then __fspath__() is returned as long as it is a str or bytes object. Otherwise TypeError is raised and NULL is returned.

3.6 版新加入.

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

void **PyOS_BeforeFork** ()
Function to prepare some internal state before a process fork. This should be called before calling fork() or any similar function that clones the current process. Only available on systems where fork() is defined.

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

3.7 版新加入.

void **PyOS_AfterFork_Parent** ()
Function to update some internal state after a process fork. This should be called from the parent process after

calling fork() or any similar function that clones the current process, regardless of whether process cloning was successful. Only available on systems where fork() is defined.

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

3.7 版新加入.

void **PyOS_AfterFork_Child**()
> Function to update internal interpreter state after a process fork. This must be called from the child process after calling fork(), or any similar function that clones the current process, if there is any chance the process will call back into the Python interpreter. Only available on systems where fork() is defined.

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

> 3.7 版新加入.

> 也參考:

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

void **PyOS_AfterFork**()
> Function to update some internal state after a process fork; this should be called in the new process if the Python interpreter will continue to be used. If a new executable is loaded into the new process, this function does not need to be called.

> 3.7 版後已⊞用: This function is superseded by *PyOS_AfterFork_Child()*.

int **PyOS_CheckStack**()
> Return true when the interpreter runs out of stack space. This is a reliable check, but is only available when USE_STACKCHECK is defined (currently on Windows using the Microsoft Visual C++ compiler). USE_STACKCHECK will be defined automatically; you should never change the definition in your own code.

PyOS_sighandler_t **PyOS_getsig**(int *i*)
> 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*)
> 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*)
> Decode a byte string from the locale encoding with the surrogateescape error handler: undecodable bytes are decoded as characters in range U+DC80..U+DCFF. If a byte sequence can be decoded as a surrogate character, escape the bytes using the surrogateescape error handler instead of decoding them.

> Encoding, highest priority to lowest priority:

> - UTF-8 on macOS, Android, and VxWorks;

> - UTF-8 on Windows if *Py_LegacyWindowsFSEncodingFlag* is zero;

> - UTF-8 if the Python UTF-8 mode is enabled;

- ASCII if the LC_CTYPE locale is "C", nl_langinfo(CODESET) returns the ASCII encoding (or an alias), and mbstowcs() and wcstombs() functions uses the ISO-8859-1 encoding.

- the current locale encoding.

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

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

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

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

也參考:

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

3.5 版新加入.

3.7 版更變: The function now uses the UTF-8 encoding in the UTF-8 mode.

3.8 版 更 變: The function now uses the UTF-8 encoding on Windows if *Py_LegacyWindowsFSEncodingFlag* is zero;

char* **Py_EncodeLocale** (const wchar_t *text*, size_t *error_pos*)
Encode a wide character string to the locale encoding with the surrogateescape error handler: surrogate characters in the range U+DC80..U+DCFF are converted to bytes 0x80..0xFF.

Encoding, highest priority to lowest priority:

- UTF-8 on macOS, Android, and VxWorks;

- UTF-8 on Windows if *Py_LegacyWindowsFSEncodingFlag* is zero;

- UTF-8 if the Python UTF-8 mode is enabled;

- ASCII if the LC_CTYPE locale is "C", nl_langinfo(CODESET) returns the ASCII encoding (or an alias), and mbstowcs() and wcstombs() functions uses the ISO-8859-1 encoding.

- the current locale encoding.

The function uses the UTF-8 encoding in the Python UTF-8 mode.

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

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

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

也參考:

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

3.5 版新加入.

3.7 版更變: The function now uses the UTF-8 encoding in the UTF-8 mode.

3.8 版 更 變: The function now uses the UTF-8 encoding on Windows if *Py_LegacyWindowsFSEncodingFlag* is zero.

6.2 系統函式

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

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

> *Return value: Borrowed reference.* Return the object *name* from the `sys` module or `NULL` if it does not exist, without setting an exception.

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

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

void **PySys_ResetWarnOptions** ()

> Reset `sys.warnoptions` to an empty list. This function may be called prior to *Py_Initialize()*.

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

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

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

> Append *unicode* to `sys.warnoptions`.

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

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

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

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

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

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

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

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

> As *PySys_WriteStdout()*, but write to `sys.stderr` or *stderr* instead.

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

> Function similar to PySys_WriteStdout() but format the message using *PyUnicode_FromFormatV()* and don't truncate the message to an arbitrary length.

> 3.2 版新加入.

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

> As *PySys_FormatStdout()*, but write to `sys.stderr` or *stderr* instead.

> 3.2 版新加入.

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

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

> 3.2 版新加入.

PyObject ***PySys_GetXOptions**()

> *Return value: Borrowed reference.* Return the current dictionary of -X options, similarly to sys._xoptions. On error, NULL is returned and an exception is set.

> 3.2 版新加入.

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

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

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

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

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

> 3.8 版新加入.

> 3.8.2 版更變: Require *Py_ssize_t* for # format characters. Previously, an unavoidable deprecation warning was raised.

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

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

> *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。

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

> The hook function is 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.

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

> 引发一个 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*)

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

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

> 3.9 版更變: Log the function name automatically.

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

> Exit the current process. This calls *Py_FinalizeEx()* and then calls the standard C library function exit(status). If *Py_FinalizeEx()* indicates an error, the exit status is set to 120.

> 3.6 版更變: Errors from finalization no longer ignored.

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

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

6.4 匯入模組

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

> *Return value: New reference.* This is a simplified interface to *PyImport_ImportModuleEx()* below, leaving the *globals* and *locals* arguments set to NULL and *level* set to 0. When the *name* argument contains a dot (when it specifies a submodule of a package), the *fromlist* argument is set to the list ['*'] so that the return value is the named module rather than the top-level package containing it as would otherwise be the case. (Unfortunately, this has an additional side effect when *name* in fact specifies a subpackage instead of a submodule: the submodules specified in the package's __all__ variable are loaded.) Return a new reference to the imported module, or NULL with an exception set on failure. A failing import of a module doesn't leave the module in sys.modules.

> 该函数总是使用绝对路径导入。

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

> *Return value: New reference.* 该函数是*PyImport_ImportModule()* 的一个被弃用的别名。

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

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

> *Return value: New reference.* Import a module. This is best described by referring to the built-in Python function __import__().

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

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

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

> *Return value: New reference.* Import a module. This is best described by referring to the built-in Python function __import__(), as the standard __import__() function calls this function directly.

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

> 3.3 版新加入.

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

> *Return value: New reference.* Similar to *PyImport_ImportModuleLevelObject()*, but the name is a UTF-8 encoded string instead of a Unicode object.

3.3 版更變: Negative values for *level* are no longer accepted.

PyObject* **PyImport_Import** (*PyObject* **name*)
> *Return value: New reference.* This is a higher-level interface that calls the current "import hook function" (with an explicit *level* of 0, meaning absolute import). It invokes the __import__() function from the __builtins__ of the current globals. This means that the import is done using whatever import hooks are installed in the current environment.

> 该函数总是使用绝对路径导入。

PyObject* **PyImport_ReloadModule** (*PyObject* **m*)
> *Return value: New reference.* Reload a module. Return a new reference to the reloaded module, or NULL with an exception set on failure (the module still exists in this case).

PyObject* **PyImport_AddModuleObject** (*PyObject* **name*)
> *Return value: Borrowed reference.* Return the module object corresponding to a module name. The *name* argument may be of the form package.module. First check the modules dictionary if there's one there, and if not, create a new one and insert it in the modules dictionary. Return NULL with an exception set on failure.

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

> 3.3 版新加入.

PyObject* **PyImport_AddModule** (const char **name*)
> *Return value: Borrowed reference.* Similar to *PyImport_AddModuleObject()*, but the name is a UTF-8 encoded string instead of a Unicode object.

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

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

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

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

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

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

PyObject* **PyImport_ExecCodeModuleEx** (const char **name*, *PyObject* **co*, const char **pathname*)
> *Return value: New reference.* Like *PyImport_ExecCodeModule()*, but the __file__ attribute of the module object is set to *pathname* if it is non-NULL.

> 参见*PyImport_ExecCodeModuleWithPathnames()*。

PyObject* **PyImport_ExecCodeModuleObject** (*PyObject *name*, *PyObject *co*, *PyObject *pathname*, *Py-Object *cpathname*)

> *Return value: New reference.* Like `PyImport_ExecCodeModuleEx()`, but the `__cached__` attribute of the module object is set to *cpathname* if it is non-`NULL`. Of the three functions, this is the preferred one to use.

> 3.3 版新加入.

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

> *Return value: New reference.* Like `PyImport_ExecCodeModuleObject()`, but *name*, *pathname* and *cpath-name* are UTF-8 encoded strings. Attempts are also made to figure out what the value for *pathname* should be from *cpathname* if the former is set to `NULL`.

> 3.2 版新加入.

> 3.3 版更變: Uses `imp.source_from_cache()` in calculating the source path if only the bytecode path is provided.

long **PyImport_GetMagicNumber** ()

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

> 3.3 版更變: 失敗时返回值 `-1`。

const char * **PyImport_GetMagicTag** ()

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

> 3.2 版新加入.

PyObject* **PyImport_GetModuleDict** ()

> *Return value: Borrowed reference.* Return the dictionary used for the module administration (a.k.a. `sys.modules`). Note that this is a per-interpreter variable.

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

> *Return value: New reference.* Return the already imported module with the given name. If the module has not been imported yet then returns `NULL` but does not set an error. Returns `NULL` and sets an error if the lookup failed.

> 3.7 版新加入.

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

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

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

> *Return value: New reference.* Load a frozen module named *name*. Return `1` for success, `0` if the module is not found, and `-1` with an exception set if the initialization failed. To access the imported module on a successful load, use `PyImport_ImportModule()`. (Note the misnomer --- this function would reload the module if it was already imported.)

> 3.3 版新加入.

> 3.4 版更變: The `__file__` attribute is no longer set on the module.

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

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

struct **_frozen**

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

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

const struct *_frozen** **PyImport_FrozenModules**

该指针被初始化为指向 `struct _frozen` 数组，以 `NULL` 或者 0 作为结束标记。当一个冻结模块被导入，首先要在这个表中搜索。第三方库可以以此来提供动态创建的冻结模块集合。

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

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

struct **_inittab**

Structure describing a single entry in the list of built-in modules. Each of these structures gives the name and initialization function for a module built into the interpreter. The name is an ASCII encoded string. Programs which embed Python may use an array of these structures in conjunction with `PyImport_ExtendInittab()` to provide additional built-in modules. The structure is defined in `Include/import.h` as:

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

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

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

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

6.5 数据 marshal 操作支持

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

数字值在存储时会将最低位字节放在开头。

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

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

将一个 `long` 整数 *value* 以 marshal 格式写入 *file*。这将只写入 *value* 最低的 32 位；无论本机 `long` 类型的长度如何。*version* 指明文件格式的版本。

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*)

从打开用于读取的 `FILE*` 的对应数据流返回一个 C `long`。使用只函数只能读取 32 位的值，无论本机 `long` 类型的长度为何。

发生错误时，将设置适当的异常 (`EOFError`) 并返回 `-1`。

int **PyMarshal_ReadShortFromFile**(FILE **file*)

从打开用于读取的 `FILE*` 的对应数据流返回一个 C `short`。使用此函数只能读取 16 位的值，无论本机 `short` 类型的长度为何。

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. 从打开用于读取的 `FILE*` 的对应数据流返回一个 Python 对象。

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

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

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

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

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

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 语句解释及变量编译

这些函数在创建你自己的函数时帮助很大。更多说明以及实例可参考说明文档中的 extending-index 小节。

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。

一般的，当一个表达式设置一个指针指向一个缓冲区，这个缓冲区可以被相应的 Python 对象管理，并且这个缓冲区共享这个对象的生存周期。你不需要人为的释放任何内存空间。除了这些 es, es#, et and et#.

然而，当一个 *Py_buffer* 结构被赋值，其包含的缓冲区被锁住，所以调用者在随后使用这个缓冲区，即使在 *Py_BEGIN_ALLOW_THREADS* 块中，可以避免可变数据因为调整大小或者被销毁所带来的风险。因此，**你不得不调用** *PyBuffer_Release()* 在你结束数据的处理时 (或者在之前任何中断事件中)

除非另有说明，缓冲区是不会以空终止的。

某些格式需要只读的 *bytes-like object*，并设置指针而不是缓冲区结构。他们通过检查对象的 *PyBufferProcs.bf_releasebuffer* 字段是否为 NULL 来发挥作用，该字段不允许为 bytearray 这样的可变对象。

备⊞: 所有 # 表达式的变式 (s#，y#，等等)，长度参数的类型 (整型或者 *Py_ssize_t*) 在包含 Python.h 头文件之前由 PY_SSIZE_T_CLEAN 宏的定义控制。如果这个宏被定义，长度是一个 *Py_ssize_t* Python 元大小类型而不是一个 int 整型。在未来的 Python 版本中将会改变，只支持 *Py_ssize_t* 而放弃支持 int 整型。最好一直定义 PY_SSIZE_T_CLEAN 这个宏。

---

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。

备⊞: 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 *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 or *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, 只读 *bytes-like object*) [const char *, int or *Py_ssize_t*] 像 s*，除了它不接受易变的对象。结果存储在两个 C 变量中，第一个是指向 C 字符串的指针，第二个是它的长度。字符串可能包含嵌入的 null 字节。Unicode 对象都被通过 'utf-8' 编码转化成 C 字符串。

z (str or 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* or 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, 只读 *bytes-like object* 或 None) [const char *, int 或 *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 (read-only *bytes-like object*) **[const char *]** 这个表达式将一个类字节类型对象转化成一个指向字符串的 C 指针；它不接受 Unicode 对象。字节缓存区必须不包含嵌入的 null 字节；如果包含了 null 字节，会引发一个 `ValueError` 异常。

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

S (bytes) **[PyBytesObject *]** 要求 Python 对象为 `bytes` 对象，不尝试进行任何转换。如果该对象不为 `bytes` 对象则会引发 `TypeError`。C 变量也可被声明为 *PyObject ** 类型。

Y (bytearray) **[PyByteArrayObject *]** 要求 Python 对象为 `bytearray` 对象，不尝试进行任何转换。如果该对象不为 `bytearray` 对象则会引发 `TypeError`。C 变量也可被声明为 *PyObject ** 类型。

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` 。

Deprecated since version 3.3, will be removed in version 3.12: 这是旧版样式*Py_UNICODE* API; 请迁移至*PyUnicode_AsWideCharString()*.

u# (str) **[const Py_UNICODE *, int 或 *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。

Deprecated since version 3.3, will be removed in version 3.12: 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`。

Deprecated since version 3.3, will be removed in version 3.12: 这是旧版样式*Py_UNICODE* API; 请迁移至*PyUnicode_AsWideCharString()*.

Z# (str 或 None) **[const Py_UNICODE *, int 或 *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`。

Deprecated since version 3.3, will be removed in version 3.12: 这是旧版样式*Py_UNICODE* API; 请迁移至*PyUnicode_AsWideCharString()*.

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

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

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

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

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

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

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

它需要三个参数。第一个仅用作输入，并且必须为 const char*，它指向一个编码格式名称，形式为以 NUL 结束的字符串或 NULL，在后一种情况下将使用 'utf-8' 编码格式。如果编码格式名称无法被 Python 识别则会引发异常。第二个参数必须为 char**；它所引用的指针值将被设为带有参数文本内容的缓冲区。文本将以第一个参数所指定的编码格式进行编码。第三个参数必须为指向一个整数的指针；被引用的整数将被设为输出缓冲区中的字节数。

有两种操作方式：

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

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

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

et# (str, bytes 或 bytearray) [const char *encoding, char **buffer, int 或 *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。

数字

b (int) [unsigned char] 将一个非负的 Python 整型转化成一个无符号的微整型，存储在一个 C unsigned char 类型中。

B (int) [unsigned char] 将一个 Python 整型转化成一个微整型并不检查溢出问题，存储在一个 C unsigned char 类型中。

h (int) [short int] 将一个 Python 整型转化成一个 C short int 短整型。

H (int) [unsigned short int] 将一个 Python 整型转化成一个 C unsigned short int 无符号短整型，并不检查溢出问题。

i (int) [int] 将一个 Python 整型转化成一个 C int 整型。

I (int) [unsigned int] 将一个 Python 整型转化成一个 C unsigned int 无符号整型，并不检查溢出问题。

l (int) [long int] 将一个 Python 整型转化成一个 C long int 长整型。

k (int) [unsigned long] 将一个 Python 整型转化成一个 C unsigned long int 无符号长整型，并不检查溢出问题。

L (int) [long long] 将一个 Python 整型转化成一个 C long long 长长整型。

K (int) [unsigned long long] 将一个 Python 整型转化成一个 C unsigned long long 无符号长长整型，并不检查溢出问题。

n (int) [*Py_ssize_t*] 将一个 Python 整型转化成一个 C *Py_ssize_t* Python 元大小类型。

c (bytes 或者 bytearray 长度为 1) [char] 将一个 Python 字节类型，如一个长度为 1 的 bytes 或者 bytearray 对象，转化成一个 C char 字符类型。

3.3 版更變: 允许 bytearray 类型的对象。

C (str 长度为 1) [int] 将一个 Python 字符，如一个长度为 1 的 `str` 字符串对象，转化成一个 C `int` 整型类型。

f (float) [float] 将一个 Python 浮点数转化成一个 C `float` 浮点数。

d (float) [double] 将一个 Python 浮点数转化成一个 C `double` 双精度浮点数。

D (complex) [Py_complex] 将一个 Python 复数类型转化成一个 C *Py_complex* Python 复数类型。

其他对象

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

O& (object) [*converter, anything*] 通过一个 *converter* 函数将一个 Python 对象转换为一个 C 变量。此函数接受两个参数：第一个是函数，第二个是 C 变量 (类型任意) 的地址，转换为 `void *` 类型。*converter* 函数将以如下方式被调用：

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

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

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

3.1 版更變: Py_CLEANUP_SUPPORTED 被添加。

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。

为了转换成功，*arg* 对象必须匹配格式并且格式必须用尽。成功的话，`PyArg_Parse*()` 函数返回 true，反之它们返回 false 并且引发一个合适的异常。当`PyArg_Parse*()` 函数因为某一个格式单元转化失败而失败时，对应的以及后续的格式单元地址内的变量都不会被使用。

API 函数

int **PyArg_ParseTuple**(*PyObject *args*, const char **format*, ...)
> 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*)
> 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[]*, ...)
> 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*)
> 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* *)
> 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*, ...)
> 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*, ...)
> 一个更简单的参数提取方式，它不使用格式字符串来指定参数类型。使用这种方法来提取参数的函数应当在函数或方法表中被声明为*METH_VARARGS*。包含实际参数的元组应当作为 *args* 传入；它必须确实是一个元组。元组的长度必须至少为 *min* 并且不超过 *max*；*min* 和 *max* 可能相等。额外的参数必须被传入函数，每个参数必须是一个指向*PyObject ** 变量的指针；它们将以来自 *args* 的值填充；它们将包含暂借的引用。对应于可选参数的变量不会由 *args* 给出的值填充；它们将由调用者来初始化。此函数执行成功时返回真值，如果 *args* 不是元组或者包含错误数量的元素则返回假值；如果执行失败则将设置一个异常。

> 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;
```

```
    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.* 基于类似于*PyArg_Parse*()* 函数系列和一系列值的格式字符串创建新值。
> 在出现错误时返回值或 NULL；如果返回 NULL，将引发异常。
>
> *Py_BuildValue()* 并不一直创建一个元组。只有当它的格式化字符串包含两个或更多的格式单元才
> 会创建一个元组。如果格式化字符串是空，它返回 None；如果它包含一个格式单元，它返回由格式单
> 元描述的的任一对象。用圆括号包裹格式化字符串可以强制它返回一个大小为 0 或者 1 的元组。
>
> 当内存缓存区的数据以参数形式传递用来构建对象时，如 s 和 s# 格式单元，会拷贝需要的数
> 据。调用者提供的缓冲区从来都不会被由*Py_BuildValue()* 创建的对象来引用。换句话说，如
> 果你的代码调用 malloc() 并且将分配的内存空间传递给*Py_BuildValue()*，你的代码就有责任
> 在*Py_BuildValue()* 返回时调用 free() 。
>
> 在下面的描述中，双引号的表达式使格式单元；圆括号 () 内的是格式单元将要返回的 Python 对象类型；
> 方括号 [] 内的是传递的 C 变量 (变量集) 的类型。
>
> 字符例如空格，制表符，冒号和逗号在格式化字符串中会被忽略 (但是不包括格式单元，如 s#)。这可
> 以使很长的格式化字符串具有更好的可读性。
>
> **s (str 或 None) [const char *]** 使用 'utf-8' 编码将空终止的 C 字符串转换为 Python str 对象。如
> 果 C 字符串指针为 NULL，则使用 None。
>
> **s# (str 或 None) [const char *, int 或*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 *, int 或*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 or None) [const char *]** 和 s 一样。
>
> **z# (str 或 None) [const char *, int 或*Py_ssize_t*]** 和 s# 一样。
>
> **u (str) [const wchar_t *]** 将空终止的 wchar_t 的 Unicode （UTF-16 或 UCS-4）数据缓冲区转换为
> Python Unicode 对象。如果 Unicode 缓冲区指针为 NULL，则返回 None。
>
> **u# (str) [const wchar_t *, int 或*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 一样。
>
> **U# (str 或 None) [const char *, int 或*Py_ssize_t*]** 和 s# 一样。
>
> **i (int) [int]** 将一个 C int 整型转化成 Python 整型对象。

b (int) [char] 将一个 C char 字符型转化成 Python 整型对象。

h (int) [short int] 将一个 C short int 短整型转化成 Python 整型对象。

l (int) [long int] 将一个 C long int 长整型转化成 Python 整型对象。

B (int) [unsigned char] 将一个 C unsigned char 无符号字符型转化成 Python 整型对象。

H (int) [unsigned short int] 将一个 C unsigned long 无符号短整型转化成 Python 整型对象。

I (int) [unsigned int] 将一个 C unsigned long 无符号整型转化成 Python 整型对象。

k (int) [unsigned long] 将一个 C unsigned long 无符号长整型转化成 Python 整型对象。

L (int) [long long] 将一个 C long long 长长整形转化成 Python 整形对象。

K (int) [unsigned long long] 将一个 C unsigned long long 无符号长长整型转化成 Python 整型对象。

n (int) [*Py_ssize_t*] 将一个 C *Py_ssize_t* 类型转化为 Python 整型。

c (bytes 长度为 1) [char] 将一个 C int 整型代表的字符转化为 Python bytes 长度为 1 的字节对象。

C (str 长度为 1) [int] 将一个 C int 整型代表的字符转化为 Python str 长度为 1 的字符串对象。

d (float) [double] 将一个 C double 双精度浮点数转化为 Python 浮点数类型数字。

f (float) [float] 将一个 C float 单精度浮点数转化为 Python 浮点数类型数字。

D (complex) [*Py_complex* *] 将一个 C *Py_complex* 类型的结构转化为 Python 复数类型。

O (object) [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 (object) [PyObject *] 和 O 相同。

N (object) [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& (object) [*converter, anything*] 通过 *converter* 函数将 *anything* 转换为 Python 对象。该函数调用时会传入 *anything* (应与 void* 兼容) 作为参数并且应当返回一个 "新的" Python 对象，或者当发生错误时返回 NULL。

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

[items] (list) [相关的元素] 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) [相关的元素] 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. 和 *Py_BuildValue()* 相同，然而它接受一个 va_list 类型的参数而不是可变数量的参数集。

6.7 字串轉⊞與格式化

數字轉⊞函數和被格式化的字串輸出。

int **PyOS_snprintf**(char *str, size_t size, const char *format, ...)
> 根據格式字符串 *format* 和額外參數，輸出不超過 *size* 個字节到 *str*。參見 Unix 手册頁面 *snprintf(3)*。

int **PyOS_vsnprintf**(char *str, size_t size, const char *format, va_list va)
> 根據格式字符串 *format* 和變量參數列表 *va*，輸出不超過 *size* 個字节到 *str*。參見 Unix 手册頁面 *vsnprintf(3)*。

PyOS_snprintf() 和 *PyOS_vsnprintf()* 包裝 C 標準庫函數 snprintf() 和 vsnprintf()。它們的目的是保證在極端情況下的一致行為，而標準 C 的函數則不然。

包裝器會確保 str[size-1] 在返回時始終為 '\0'。它們從不寫入超過 *size* 個字节 (包括末尾的 '\0') 到字符串。兩個函數都要求 str != NULL, size > 0 和 format != NULL。

如果平台沒有 vsnprintf() 而且緩衝區大小需要避免截斷超出 *size* 512 字节以上，Python 會以一個 *Py_FatalError()* 來中止。

當回傳值 (*rv*) 給這些函數應該被編譯如下：

- 當 0 <= rv < size 時，輸出轉換即成功並將 *rv* 個字符寫入到 *str* (不包括末尾 str[rv] 位置的 '\0' 字节)。

- 當 rv >= size 時，輸出轉換會被截斷並且需要一個具有 rv + 1 字节的緩衝區才能成功執行。在此情況下 str[size-1] 為 '\0'。

- 當 rv < 0 時，"會發生不好的事情。" 在此情況下 str[size-1] 也為 '\0'，但 *str* 的其余部分是未定義的。錯誤的確切原因取決於底層平台。

以下函數提供與語言環境無關的字符串到數字轉換。

double **PyOS_string_to_double**(const char *s, char **endptr, *PyObject* *overflow_exception)
> 將字符串 s 轉換為 double 類型，失敗時引發 Python 異常。接受的字符串的集合對應於被 Python 的 float() 構造函數接受的字符串的集合，除了 s 必須沒有前導或尾隨空格。轉換必須獨立於當前的區域。
>
> 如果 endptr 是 NULL，轉換整個字符串。引發 ValueError 并且返回 -1.0 如果字符串不是浮点数的有效的表達方式。
>
> 如果 endptr 不是 NULL，尽可能多的轉換字符串并将 *endptr 設置為指向第一個未轉換的字符。如果字符串的初始段不是浮点数的有效的表達方式，将 *endptr 設置為指向字符串的開頭，引發 ValueError 異常，并且返回 -1.0。
>
> 如果 s 表示一個太大而不能存儲在一個浮点数中的值（比方說，"1e500" 在許多平台上是一個字符串）然后如果 overflow_exception 是 NULL 返回 Py_HUGE_VAL（用適當的符號）并且不設置任何異常。在其他方面，overflow_exception 必須指向一個 Python 異常對象；引發異常並返回 -1.0。在這兩種情況下，設置 *endptr 指向轉換值之后的第一個字符。
>
> 如果在轉換期間發生任何其他錯誤（比如一個內存不足的錯誤），設置適當的 Python 異常并且返回 -1.0。
>
> 3.1 版新加入.

char* **PyOS_double_to_string**(double val, char format_code, int precision, int flags, int *ptype)
> 轉換 double *val* 為一個使用 *format_code*，*precision* 和 *flags* 的字符串
>
> 格式碼必須是以下其中之一，'e', 'E', 'f', 'F', 'g', 'G' 或者 'r'。對於 'r'，提供的 精度必須是 0。'r' 格式碼指定了標準函數 repr() 格式。
>
> *flags* 可以為零或者其他值 Py_DTSF_SIGN, Py_DTSF_ADD_DOT_0 或 Py_DTSF_ALT 或其組合：

- `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. 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. 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. 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. 返回当前线程状态的帧，如果没有当前执行的帧则返回 NULL。

另请参阅*PyThreadState_GetFrame()*。

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

返回一个强引用，如果 *frame* 没有外部帧则返回 NULL。

frame 必须不为 NULL。

3.9 版新加入.

PyCodeObject* **PyFrame_GetCode** (*PyFrameObject *frame*)
获取 *frame* 的代码。

返回一个强引用。

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*)
返回 *frame* 当前正在执行的行号。

frame 必须不为 NULL。

const char* **PyEval_GetFuncName**(*PyObject *func*)
> 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*)
> 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*)
> 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_KnownEncoding**(const char *encoding*)
> 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.* 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.* 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.* 为给定的 *encoding* 获取一个编码器函数。

*PyObject** **PyCodec_Decoder**(const char *encoding*)
> *Return value: New reference.* 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.* 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.* 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.* 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.* 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*)

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. 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. 引发 *exc* 作为异常。

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

Return value: New reference. 忽略 unicode 错误，跳过错误的输入。

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

Return value: New reference. 使用 ? 或 U+FFFD 替换 unicode 编码错误。

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

Return value: New reference. 使用 XML 字符引用替换 unicode 编码错误。

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

Return value: New reference. 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. 使用 \N{...} 转义符替换 unicode 编码错误。

3.5 版新加入.

抽象物件層

本章中的函数与 Python 对象交互，无论其类型，或具有广泛类的对象类型（例如，所有数值类型，或所有序列类型）。当使用对象类型并不适用时，他们会产生一个 Python 异常。

这些函数是不可能用于未正确初始化的对象的，如一个列表对象被 *PyList_New()* 创建，但其中的项目没有被设置为一些非 "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*)
    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*)
    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.* 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.* 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.* 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*)
> 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。
>
> If *v* is NULL, the attribute is deleted. This behaviour is deprecated in favour of using *PyObject_DelAttr()*,
> but there are currently no plans to remove it.

int **PyObject_SetAttrString**(*PyObject *o*, const char *attr_name*, *PyObject *v*)
> 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。
>
> If *v* is NULL, the attribute is deleted, but this feature is deprecated in favour of using
> *PyObject_DelAttrString()*.

int **PyObject_GenericSetAttr**(*PyObject *o*, *PyObject *name*, *PyObject *value*)
> 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.* __dict__ 描述符的获取函数的一种通用实现。必要时会创建字典。
>
> 3.3 版新加入.

int **PyObject_GenericSetDict**(*PyObject *o*, *PyObject *value*, void *context*)
> __dict__ 描述符设置函数的一种通用实现。这里不允许删除字典。
>
> 3.3 版新加入.

PyObject* **PyObject_RichCompare**(*PyObject *o1*, *PyObject *o2*, int *opid*)
> *Return value: New reference.* 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*)
> 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。

備注：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_Repr**(*PyObject *o*)

> *Return value: New reference.* 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.* 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.* 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.* 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*)

> 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*)

> 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* 可以通过 __bases__ 属性（必须是基类的元组）来覆盖它是否被认作类的状态，及其基类。

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

> 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: The return type is now Py_hash_t. This is a signed integer the same size as *Py_ssize_t*.

Py_hash_t **PyObject_HashNotImplemented**(*PyObject *o*)

> 设置一个 TypeError 表示 type(o) 是不可哈希的，并返回 -1。该函数保存在 tp_hash 槽中时会受到特别对待，允许某个类型向解释器显式表明它不可散列。

int **PyObject_IsTrue**(*PyObject *o*)

> 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*)

> 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.* 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* 为 *type* 类型或 *type* 的子类型则返回真值。两个参数都必须非 NULL。

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

Py_ssize_t **PyObject_Length** (*PyObject *o*)

> 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.* 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*)

> 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。

int **PyObject_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]。

PyObject* **PyObject_Dir** (*PyObject *o*)

> *Return value: New reference.* 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.* 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。

7.2 调用协议

CPython 支持两种不同的调用协议：*tp_call* 和矢量调用。

7.2.1 *tp_call* 协议

设置*tp_call* 的类的实例都是可调用的。槽位的签名为:

```
PyObject *tp_call(PyObject *callable, PyObject *args, PyObject *kwargs);
```

一个调用是用一个元组表示位置参数，用一个 dict 表示关键字参数，类似于 Python 代码中的 "callable(*args, **kwargs)"。*args* 必须是非空的（如果没有参数，会使用一个空元组），但如果没有关键字参数，*kwargs* 可以是 *NULL*。

这个约定不仅被 *tp_call* 使用：*tp_new* 和*tp_init* 也这样传递参数。

To call an object, use *PyObject_Call()* or another *call API*.

7.2.2 Vectorcall 协议

3.9 版新加入.

vectorcall 协议是在 **PEP 590** 被引入的，它是使调用函数更加有效的附加协议。

作为经验法则，如果可调用程序支持 vectorcall，CPython 会更倾向于内联调用。然而，这并不是一个硬性规定。此外，一些第三方扩展直接使用 *tp_call*（而不是使用 *PyObject_Call()*）。因此，一个支持 vectorcall 的类也必须实现 *tp_call*。此外，无论使用哪种协议，可调对象的行为都必须是相同的。推荐的方法是将 *tp_call* 设置为 *PyVectorcall_Call()*。值得一提的是:

> **警告:** 一个支持 Vectorcall 的类 **必须** 也实现具有相同语义的 *tp_call*。

如果一个类的 vectorcall 比 *tp_call* 慢，就不应该实现 vectorcall。例如，如果被调用者需要将参数转换为 args 元组和 kwargs dict，那么实现 vectorcall 就没有意义。

类可以通过启用 *Py_TPFLAGS_HAVE_VECTORCALL* 标志并将 *tp_vectorcall_offset* 设置为对象结构中的 *vectorcallfunc* 的 offset 来实现 vectorcall 协议。这是一个指向具有以下签名的函数的指针:

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

- *callable* 是指被调用的对象。

- *args* **是一个 C 语言数组，由位置参数和后面的** 关键字参数的值。如果没有参数，这个值可以是 *NULL*。

- *nargsf* **是位置参数的数量加上可能的** PY_VECTORCALL_ARGUMENTS_OFFSET 标志。要从 *nargsf* 获得实际的位置参数数，请使用 *PyVectorcall_NARGS()*。

- *kwnames* **是一包含所有关键字名称的元组。** 换句话说，就是 kwargs 字典的键。这些名字必须是字符串 (str 或其子类的实例)，并且它们必须是唯一的。如果没有关键字参数，那么 *kwnames* 可以用 *NULL* 代替。

PY_VECTORCALL_ARGUMENTS_OFFSET
> 如果在 vectorcall 的 *nargsf* 参数中设置了此标志，则允许被调用者临时更改 args[-1] 的值。换句话说，*args* 指向分配向量中的参数 1（不是 0）。被调用方必须在返回之前还原 args[-1] 的值。
>
> 对于 *PyObject_VectorcallMethod()* ，这个标志的改变意味着 "args[0]" 可能改变了。
>
> 当调用方可以以几乎无代价的方式（无额外的内存申请），那么调用者被推荐适用: PY_VECTORCALL_ARGUMENTS_OFFSET。这样做将允许诸如绑定方法之类的可调用函数非常有效地进行向前调用（其中包括一个带前缀的 *self* 参数）。

要调用一个实现了 vectorcall 的对象，请使用某个 *call API* 函数，就像其他可调对象一样。*PyObject_Vectorcall()* 通常是最有效的。

> **备⒠:** 在 CPython 3.8 中，vectorcall API 和相关的函数暂定以带开头下划线的名称提供: _PyObject_Vectorcall, _Py_TPFLAGS_HAVE_VECTORCALL, _PyObject_VectorcallMethod, _PyVectorcall_Function, _PyObject_CallOneArg, _PyObject_CallMethodNoArgs, _PyObject_CallMethodOneArg。此外，PyObject_VectorcallDict 以 _PyObject_FastCallDict 的名称提供。旧名称仍然被定义为不带下划线的新名称的别名。

递归控制

在 使 用 *tp_call* 时，被 调 用 者 不 必 担 心 递 归： CPython 对 于 使 用 *tp_call* 进 行 的 调 用 会 使 用 `Py_EnterRecursiveCall()` 和 `Py_LeaveRecursiveCall()`。

为保证效率，这不适用于使用 vectorcall 的调用：被调用方在需要时应当使用 *Py_EnterRecursiveCall* 和 *Py_LeaveRecursiveCall*。

Vectorcall 支持 API

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

给定一个 vectorcall *nargsf* 实参，返回参数的实际数量。目前等同于：

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

然而，应使用 `PyVectorcall_NARGS` 函数以便将来扩展。

这个函数不是 *limited API* 的一部分。

3.8 版新加入.

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

如果 *op* 不支持 vectorcall 协议（要么是因为类型不支持，要么是因为具体实例不支持），返回 *NULL*。否则，返回存储在 *op* 中的 vectorcall 函数指针。这个函数从不触发异常。

这在检查 *op* 是否支持 vectorcall 时最有用处，可以通过检查 `PyVectorcall_Function(op) != NULL` 来实现。

这个函数不是 *limited API* 的一部分。

3.8 版新加入.

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

调用 * 可调对象 * 的 `vectorcallfunc`，其位置参数和关键字参数分别以元组和 dict 形式给出。

这 是 一 个 专 门 函 数， 其 目 的 是 被 放 入 `tp_call` 槽 位 或 是 用 于 `tp_call` 的 实 现。 它 不 会 检 查 `Py_TPFLAGS_HAVE_VECTORCALL` 旗标并且它不会回退到 `tp_call`。

这个函数不是 *limited API* 的一部分。

3.8 版新加入.

7.2.3 调用对象的 API

Various functions are available for calling a Python object. Each converts its arguments to a convention supported by the called object –either *tp_call* or vectorcall. In order to do as little conversion as possible, pick one that best fits the format of data you have available.

下表总结了可用的功能；请参阅各个文档以了解详细信息。

函数	可调用对象（Callable）	args	kwargs
PyObject_Call()	PyObject *	元组	dict/NULL
PyObject_CallNoArgs()	PyObject *	---	---
PyObject_CallOneArg()	PyObject *	1 个对象	---
PyObject_CallObject()	PyObject *	元组/NULL	---
PyObject_CallFunction()	PyObject *	format	---
PyObject_CallMethod()	对象 + char*	format	---
PyObject_CallFunctionObjArgs()	PyObject *	可变参数	---
PyObject_CallMethodObjArgs()	对象 + 名称	可变参数	---
PyObject_CallMethodNoArgs()	对象 + 名称	---	---
PyObject_CallMethodOneArg()	对象 + 名称	1 个对象	---
PyObject_Vectorcall()	PyObject *	vectorcall	vectorcall
PyObject_VectorcallDict()	PyObject *	vectorcall	dict/NULL
PyObject_VectorcallMethod()	参数 + 名称	vectorcall	vectorcall

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

> *Return value: New reference.* 调用一个可调用的 Python 对象 *callable*，附带由元组 *args* 所给出的参数，以及由字典 *kwargs* 所给出的关键字参数。
>
> *args* 必须不为 *NULL*；如果不想要参数请使用一个空元组。如果不想要关键字参数，则 *kwargs* 可以为 *NULL*。
>
> 成功时返回结果，在失败时抛出一个异常并返回 *NULL*。
>
> 这等价于 Python 表达式 callable(*args, **kwargs)。

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

> 调用一个可调用的 Python 对象 *callable* 并不附带任何参数。这是不带参数调用 Python 可调用对象的最有效方式。
>
> 成功时返回结果，在失败时抛出一个异常并返回 *NULL*。
>
> 3.9 版新加入.

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

> 调用一个可调用的 Python 对象 *callable* 并附带恰好 1 个位置参数 *arg* 而没有关键字参数。
>
> 成功时返回结果，在失败时抛出一个异常并返回 *NULL*。
>
> 这个函数不是*limited API* 的一部分。
>
> 3.9 版新加入.

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

> *Return value: New reference.* 调用一个可调用的 Python 对象 *callable*，附带由元组 *args* 所给出的参数。如果不想要传入参数，则 *args* 可以为 *NULL*。
>
> 成功时返回结果，在失败时抛出一个异常并返回 *NULL*。
>
> 这等价于 Python 表达式 callable(*args)。

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

> *Return value: New reference.* 调用一个可调用的 Python 对象 *callable*，附带可变数量的 C 参数。这些 C 参数使用*Py_BuildValue()* 风格的格式化字符串来描述。format 可以为 *NULL*，表示没有提供任何参数。
>
> 成功时返回结果，在失败时抛出一个异常并返回 *NULL*。
>
> 这等价于 Python 表达式 callable(*args)。
>
> 请注意如果你只传入*PyObject ** 参数，则*PyObject_CallFunctionObjArgs()* 是更快速的选择。

3.4 版更變: 这个 *format* 类型已从 char * 更改。

PyObject* **PyObject_CallMethod**(*PyObject* *obj, const char *name, const char *format, ...)
　　Return value: New reference. 调用 *obj* 对象中名为 *name* 的方法并附带可变数量的 C 参数。这些 C 参数由 `Py_BuildValue()` 格式字符串来描述并应当生成一个元组。

　　格式可以为 *NULL* ，表示未提供任何参数。

　　成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

　　这和 Python 表达式 "obj.name(arg1, arg2, ...)" 是一样的。

　　请注意如果你只传入 `PyObject *` 参数，则 `PyObject_CallMethodObjArgs()` 是更快速的选择。

　　3.4 版更變: The types of *name* and *format* were changed from char *.

PyObject* **PyObject_CallFunctionObjArgs**(*PyObject* *callable, ...)
　　Return value: New reference. 调用一个可调用的 Python 对象 *callable*，附带可变数量的 `PyObject *` 参数。这些参数是以 *NULL* 之后可变数量的形参的形式提供的。

　　成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

　　这和 Python 表达式 "callable(arg1, arg2, ...)" 是一样的。

PyObject* **PyObject_CallMethodObjArgs**(*PyObject* *obj, *PyObject* *name, ...)
　　Return value: New reference. 调用 Python 对象 *obj* 中的一个方法，其中方法名称由 *name* 中的 Python 字符串对象给出。它将附带可变数量的 `PyObject *` 参数被调用。这些参数是以 *NULL* 之后可变数量的形参的形式提供的。

　　成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

PyObject* **PyObject_CallMethodNoArgs**(*PyObject* *obj, *PyObject* *name)
　　调用 Python 对象 *obj* 中的一个方法并不附带任何参数，其中方法名称由 *name* 中的 Python 字符串对象给出。

　　成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

　　这个函数不是 *limited API* 的一部分。

　　3.9 版新加入.

PyObject* **PyObject_CallMethodOneArg**(*PyObject* *obj, *PyObject* *name, *PyObject* *arg)
　　调用 Call a method of the Python 对象 *obj* 中的一个方法并附带单个位置参数 *arg*，其中方法名称由 *name* 中的 Python 字符串对象给出。

　　成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

　　这个函数不是 *limited API* 的一部分。

　　3.9 版新加入.

PyObject* **PyObject_Vectorcall**(*PyObject* *callable, *PyObject* *const *args, size_t *nargsf*, *PyObject* *kwnames)
　　调用一个可调用的 Python 对象 *callable*。附带的参数与 `vectorcallfunc` 相同。如果 *callable* 支持 *vectorcall*，则它会直接调用存放在 *callable* 中的 vectorcall 函数。

　　成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

　　这个函数不是 *limited API* 的一部分。

　　3.9 版新加入.

PyObject* **PyObject_VectorcallDict**(*PyObject* *callable, *PyObject* *const *args, size_t *nargsf*, *PyObject* *kwdict)
　　调用 *callable* 并附带与在 *vectorcall* 协议中传入的完全相同的位置参数，但会加上以字典 *kwdict* 形式传入的关键字参数。*args* 数组将只包含位置参数。

无论在内部使用哪种协议，都需要进行参数的转换。因此，此函数应当仅在调用方已经拥有作为关键字参数的字典，但没有作为位置参数的元组时才被使用。

这个函数不是 *limited API* 的一部分。

3.9 版新加入.

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

使用 vectorcall 调用惯例来调用一个方法。方法的名称以 Python 字符串 *name* 的形式给出。调用方法的对象为 *args[0]*，而 *args* 数组从 *args[1]* 开始的部分则代表调用的参数。必须传入至少一个位置参数。*nargsf* 为包括 *args[0]* 在内的位置参数的数量，如果 args[0] 的值可能被临时改变则要再加上 PY_VECTORCALL_ARGUMENTS_OFFSET。关键字参数可以像在 *PyObject_Vectorcall()* 中一样被传入。

如果对象具有 *Py_TPFLAGS_METHOD_DESCRIPTOR* 特性，此函数将调用调用未绑定的方法对象并附带完整的 *args* vector 作为参数。

成功时返回结果，在失败时抛出一个异常并返回 *NULL*。

这个函数不是 *limited API* 的一部分。

3.9 版新加入.

7.2.4 调用支持 API

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

确定对象 *o* 是可调对象。如果对象是可调对象则返回 1，其他情况返回 0。这个函数不会调用失败。

7.3 数字协议

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

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. 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. 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. 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. 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. Return the floor of *o1* divided by *o2*, or NULL on failure. This is the equivalent of the Python expression o1 // o2.

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

> *Return value: New reference.* Return a reasonable approximation for the mathematical value of *o1* divided by *o2*, or NULL on failure. The return value is "approximate" because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers. This is the equivalent of the Python expression o1 / o2.

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

> *Return value: New reference.* 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.* 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.* 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.* 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.* 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.* 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.* 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.* 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.* 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.* 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.* 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.* 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.* 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.* 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.* 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.* 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.* 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.* Return a reasonable approximation for the mathematical value of *o1* divided by *o2*,
> or NULL on failure. The return value is "approximate" because binary floating point numbers are approximate; it is
> not possible to represent all real numbers in base two. This function can return a floating point value when passed
> two integers. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement
> o1 /= o2.

PyObject* **PyNumber_InPlaceRemainder**(*PyObject *o1*, *PyObject *o2*)
> *Return value: New reference.* 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.* 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.* 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.* 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.* 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.* 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.* 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.* 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.* 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.* 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。

PyObject* **PyNumber_ToBase**(*PyObject* *n, int *base*)
> *Return value: New reference.* 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)
> Returns *o* converted to a *Py_ssize_t* value if *o* can be interpreted as an integer. If the call fails, an exception is raised and −1 is returned.
>
> If *o* can be converted to a Python int but the attempt to convert to a *Py_ssize_t* value would raise an OverflowError, then the *exc* argument is the type of exception that will be raised (usually IndexError or OverflowError). If *exc* is NULL, then the exception is cleared and the value is clipped to PY_SSIZE_T_MIN for a negative integer or PY_SSIZE_T_MAX for a positive integer.

int **PyIndex_Check**(*PyObject* *o)
> Returns 1 if *o* is an index integer (has the nb_index slot of the tp_as_number structure filled in), and 0 otherwise. This function always succeeds.

7.4 序列协议

int **PySequence_Check**(*PyObject* *o)
> Return 1 if the object provides the sequence protocol, and 0 otherwise. Note that it returns 1 for Python classes with a __getitem__() method, unless they are dict subclasses, since in general it is impossible to determine what type of keys the class supports. This function always succeeds.

Py_ssize_t **PySequence_Size**(*PyObject* *o)
Py_ssize_t **PySequence_Length**(*PyObject* *o)
> 到哪里积分返回序列 *o* 中对象的数量，失败时返回 −1。这相当于 Python 表达式 len(o)。

PyObject* **PySequence_Concat**(*PyObject* *o1, *PyObject* *o2)
> *Return value: New reference.* 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.* 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.* 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.* 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.* 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.* 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*)
> 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。

> If *v* is NULL, the element is deleted, but this feature is deprecated in favour of using *PySequence_DelItem()*.

int **PySequence_DelItem** (*PyObject *o*, *Py_ssize_t i*)
> 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*)
> 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*)
> 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*)
> 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*)
> 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*)
> 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.* 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.* 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.* 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*)
> Returns the length of *o*, assuming that *o* was returned by *PySequence_Fast()* and that *o* is not NULL. The size can also be retrieved by calling *PySequence_Size()* on *o*, but *PySequence_Fast_GET_SIZE()* is faster because it can assume *o* is a list or tuple.

PyObject* **PySequence_Fast_GET_ITEM** (*PyObject *o*, *Py_ssize_t i*)
> *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()` 理馈，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*)

> Return 1 if the object provides the mapping protocol or supports slicing, and 0 otherwise. Note that it returns 1 for Python classes with a `__getitem__()` method, since in general it is impossible to determine what type of keys the class supports. This function always succeeds.

Py_ssize_t **PyMapping_Size**(*PyObject *o*)
Py_ssize_t **PyMapping_Length**(*PyObject *o*)

> 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.* 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*)

> 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*)

> 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*)

> 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.* 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.* 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.* 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)

> 如果对象 *o* 支持迭代器协议则返回真值。此函数总是会成功执行。

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

> *Return value: New reference.* 返回迭代 *o* 的下一个值。对象必须是一个迭代器（这应由调用者来判断）。如果没有余下的值，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

```c
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 */
}
```

7.7 缓冲协议

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。

对于缓冲接口的消费者而言，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()* 。

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。
>
> void *****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。比如可能被导出器重组为一个整数，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。缓冲区用户 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	格式
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。

```python
def verify_structure(memlen, itemsize, ndim, shape, strides, offset):
    """Verify that the parameters represent a valid array within
       the bounds of the allocated memory:
           char *mem: start of the physical memory block
           memlen: length of the physical memory block
           offset: (char *)buf - mem
    """
    if offset % itemsize:
        return False
    if offset < 0 or offset+itemsize > memlen:
        return False
    if any(v % itemsize for v in strides):
        return False

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

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

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

PIL-风格：形状，步幅和子偏移量

除了常规项之外，PIL 风格的数组还可以包含指针，必须跟随这些指针才能到达维度的下一个元素。例如，常规的三维 C 语言数组 char v[2][2][3] 可以看作是一个指向 2 个二维数组的 2 个指针：char (*v[2])[2][3]。在子偏移表示中，这两个指针可以嵌入在*buf* 的开头，指向两个可以位于内存任何位置的 char x[2][3] 数组。

这是一个函数，当 n 维索引所指向的 N-D 数组中有 "NULL"步长和子偏移量时，它返回一个指针

```c
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* ，成为连续字符串的形式。*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 旧缓冲协议

3.0 版後已⬚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*)
> 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*)
> 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*)
> 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*)
> 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。

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 基本对象

本节描述 Python 类型对象和单一实例对象 象 None。

8.1.1 Type 对象

PyTypeObject
    对象的 C 结构用于描述 built-in 类型。

PyTypeObject **PyType_Type**
    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*)
    Return non-zero if the object *o* is a type object, including instances of types derived from the standard type object. Return 0 in all other cases. This function always succeeds.

int **PyType_CheckExact** (*PyObject* *o*)
    Return non-zero if the object *o* is a type object, but not a subtype of the standard type object. Return 0 in all other cases. This function always succeeds.

unsigned int **PyType_ClearCache** ()
    Clear the internal lookup cache. Return the current version tag.

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

> 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: The return type is now `unsigned long` rather than `long`.

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

> Invalidate the internal lookup cache for the type and all of its subtypes. This function must be called after any manual modification of the attributes or base classes of the type.

int **PyType_HasFeature**(*PyTypeObject *o*, int *feature*)

> Return non-zero if the type object *o* sets the feature *feature*. Type features are denoted by single bit flags.

int **PyType_IS_GC**(*PyTypeObject *o*)

> Return true if the type object includes support for the cycle detector; this tests the type flag `Py_TPFLAGS_HAVE_GC`.

int **PyType_IsSubtype**(*PyTypeObject *a*, *PyTypeObject *b*)

> Return true if *a* is a subtype of *b*.

> This function only checks for actual subtypes, which means that `__subclasscheck__()` is not called on *b*. Call `PyObject_IsSubclass()` to do the same check that `issubclass()` would do.

PyObject* **PyType_GenericAlloc**(*PyTypeObject *type*, *Py_ssize_t nitems*)

> *Return value: New reference.* Generic handler for the `tp_alloc` slot of a type object. Use Python's default memory allocation mechanism to allocate a new instance and initialize all its contents to `NULL`.

PyObject* **PyType_GenericNew**(*PyTypeObject *type*, *PyObject *args*, *PyObject *kwds*)

> *Return value: New reference.* Generic handler for the `tp_new` slot of a type object. Create a new instance using the type's `tp_alloc` slot.

int **PyType_Ready**(*PyTypeObject *type*)

> Finalize a type object. This should be called on all type objects to finish their initialization. This function is responsible for adding inherited slots from a type's base class. Return `0` on success, or return `-1` and sets an exception on error.

> ---
> 備⊡: If some of the base classes implements the GC protocol and the provided type does not include the `Py_TPFLAGS_HAVE_GC` in its flags, then the GC protocol will be automatically implemented from its parents. On the contrary, if the type being created does include `Py_TPFLAGS_HAVE_GC` in its flags then it **must** implement the GC protocol itself by at least implementing the `tp_traverse` handle.
> ---

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

> Return the function pointer stored in the given slot. If the result is `NULL`, this indicates that either the slot is `NULL`, or that the function was called with invalid parameters. Callers will typically cast the result pointer into the appropriate function type.

> See `PyType_Slot.slot` for possible values of the *slot* argument.

> An exception is raised if *type* is not a heap type.

> 3.4 版新加入.

PyObject* **PyType_GetModule**(*PyTypeObject *type*)

> Return the module object associated with the given type when the type was created using `PyType_FromModuleAndSpec()`.

> If no module is associated with the given type, sets `TypeError` and returns `NULL`.

This function is usually used to get the module in which a method is defined. Note that in such a method, `PyType_GetModule(Py_TYPE(self))` may not return the intended result. `Py_TYPE(self)` may be a *subclass* of the intended class, and subclasses are not necessarily defined in the same module as their superclass. See *PyCMethod* to get the class that defines the method.

3.9 版新加入.

void* **PyType_GetModuleState**(*PyTypeObject *type*)
> Return the state of the module object associated with the given type. This is a shortcut for calling *PyModule_GetState()* on the result of *PyType_GetModule()*.
>
> If no module is associated with the given type, sets `TypeError` and returns `NULL`.
>
> If the *type* has an associated module but its state is `NULL`, returns `NULL` without setting an exception.
>
> 3.9 版新加入.

Creating Heap-Allocated Types

The following functions and structs are used to create *heap types*.

*PyObject** **PyType_FromModuleAndSpec**(*PyObject *module*, *PyType_Spec *spec*, *PyObject *bases*)
> *Return value: New reference.* Creates and returns a heap type object from the *spec* (*Py_TPFLAGS_HEAPTYPE*).
>
> If *bases* is a tuple, the created heap type contains all types contained in it as base types.
>
> If *bases* is `NULL`, the *Py_tp_bases* slot is used instead. If that also is `NULL`, the *Py_tp_base* slot is used instead. If that also is `NULL`, the new type derives from `object`.
>
> The *module* argument can be used to record the module in which the new class is defined. It must be a module object or `NULL`. If not `NULL`, the module is associated with the new type and can later be retrieved with *PyType_GetModule()*. The associated module is not inherited by subclasses; it must be specified for each class individually.
>
> This function calls *PyType_Ready()* on the new type.
>
> 3.9 版新加入.

*PyObject** **PyType_FromSpecWithBases**(*PyType_Spec *spec*, *PyObject *bases*)
> *Return value: New reference.* Equivalent to `PyType_FromModuleAndSpec(NULL, spec, bases)`.
>
> 3.3 版新加入.

*PyObject** **PyType_FromSpec**(*PyType_Spec *spec*)
> *Return value: New reference.* Equivalent to `PyType_FromSpecWithBases(spec, NULL)`.

PyType_Spec
> Structure defining a type's behavior.
>
> const char* **PyType_Spec.name**
> > Name of the type, used to set *PyTypeObject.tp_name*.
>
> int **PyType_Spec.basicsize**
>
> int **PyType_Spec.itemsize**
> > Size of the instance in bytes, used to set *PyTypeObject.tp_basicsize* and *PyTypeObject.tp_itemsize*.
>
> int **PyType_Spec.flags**
> > Type flags, used to set *PyTypeObject.tp_flags*.
> >
> > If the `Py_TPFLAGS_HEAPTYPE` flag is not set, *PyType_FromSpecWithBases()* sets it automatically.

PyType_Slot ***PyType_Spec.slots**
> Array of *PyType_Slot* structures. Terminated by the special slot value {0, NULL}.

PyType_Slot
> Structure defining optional functionality of a type, containing a slot ID and a value pointer.

> int **PyType_Slot.slot**

>> A slot ID.

>> Slot IDs are named like the field names of the structures *PyTypeObject*, *PyNumberMethods*, *PySequenceMethods*, *PyMappingMethods* and *PyAsyncMethods* with an added Py_ prefix. For example, use:

>> - Py_tp_dealloc to set *PyTypeObject.tp_dealloc*
>> - Py_nb_add to set *PyNumberMethods.nb_add*
>> - Py_sq_length to set *PySequenceMethods.sq_length*

>> The following fields cannot be set at all using *PyType_Spec* and *PyType_Slot*:

>>> - *tp_dict*
>>> - *tp_mro*
>>> - *tp_cache*
>>> - *tp_subclasses*
>>> - *tp_weaklist*
>>> - *tp_vectorcall*
>>> - *tp_weaklistoffset* (see *PyMemberDef*)
>>> - *tp_dictoffset* (see *PyMemberDef*)
>>> - *tp_vectorcall_offset* (see *PyMemberDef*)

>> The following fields cannot be set using *PyType_Spec* and *PyType_Slot* under the limited API:

>>> - *bf_getbuffer*
>>> - *bf_releasebuffer*

>> Setting Py_tp_bases or Py_tp_base may be problematic on some platforms. To avoid issues, use the *bases* argument of PyType_FromSpecWithBases() instead.

>> 3.9 版更變: Slots in *PyBufferProcs* may be set in the unlimited API.

> void ***PyType_Slot.pfunc**
> The desired value of the slot. In most cases, this is a pointer to a function.

> May not be NULL.

8.1.2 **None 对象**

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 数值对象

8.2.1 整數物件

所有整数都使用以任意大小的长整数对象表示。

在出错时，大多数 PyLong_As* API 返回 (返回类型)-1 ，无法与一般的数字区分开来。请使用*PyErr_Occurred()* 来区分。

PyLongObject

表示 Python 整数对象的*PyObject* 子类型。

PyTypeObject **PyLong_Type**

这个*PyTypeObject* 的实例表示 Python 的整数类型。与 Python 层中的 int 相同。

int **PyLong_Check** (*PyObject* *p)

如果它的参数是*PyLongObject* 或*PyLongObject* 的子类型则返回真值。此函数总是会成功执行。

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. 由 *v* 返回一个新的*PyLongObject* 对象，失败时返回 NULL 。

The current implementation keeps an array of integer objects for all integers between -5 and 256. When you create an int in that range you actually just get back a reference to the existing object.

PyObject* **PyLong_FromUnsignedLong** (unsigned long *v*)

Return value: New reference. 由 C unsigned long 类型返回一个新的*PyLongObject* 对象，失败时返回 NULL 。

PyObject* **PyLong_FromSsize_t** (*Py_ssize_t v*)

Return value: New reference. 从 C *Py_ssize_t* 类型返回一个新的*PyLongObject* 对象，如果失败则返回 NULL 。

PyObject* **PyLong_FromSize_t** (size_t *v*)

Return value: New reference. 从 C size_t 返回一个新的*PyLongObject* 对象，如果失败则返回 NULL 。

PyObject* **PyLong_FromLongLong** (long long *v*)

Return value: New reference. 从 C long long 返回一个新的*PyLongObject* 对象，失败时返回 NULL 。

PyObject* **PyLong_FromUnsignedLongLong** (unsigned long long *v*)

Return value: New reference. 从 C unsigned long long 返回一个新的*PyLongObject* 对象，失败时返回 NULL 。

PyObject* **PyLong_FromDouble** (double *v*)

> *Return value: New reference.* 从 *v* 的整数部分返回一个新的 `PyLongObject` 对象，如果失败则返回 NULL
> 。

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

> *Return value: New reference.* 根据 *str* 字符串值返回一个新的 `PyLongObject` ，*base* 指定基数。如果 *pend*
> 不是 NULL ，*/*pend* 将指向 *str* 中表示这个数字部分的后面的第一个字符。如果 *base* 是 0 ，*str* 将使用
> integers 定义来解释；在这种情况下，一个非零的十进制数中的前导零会引发一个 ValueError 。如果
> *base* 不是 0 ，它必须在 2 和 36 之间，包括 2 和 36。基数说明符后以及数字之间的前导空格、单下划
> 线将被忽略。如果没有数字，将引发 ValueError。

PyObject* **PyLong_FromUnicode** (*Py_UNICODE* *u, *Py_ssize_t length*, int *base*)

> *Return value: New reference.* 将 Unicode 数字序列转换为 Python 整数值。

> Deprecated since version 3.3, will be removed in version 3.10: 旧的 `Py_UNICODE` API 的一部分; 请迁移到使
> 用 `PyLong_FromUnicodeObject()` 。

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.* 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)

> 返回 *obj* 的 C long 表达方式。如果 *obj* 不是 `PyLongObject` 的实例，先调用它的 __index__() 或
> __int__() 方法 (如果有) 将其转换为 `PyLongObject` 。

> 如果 *obj* 的值溢出了 long 的范围，会引发 OverflowError。

> 发生错误时返回 -1 。使用 `PyErr_Occurred()` 来消歧义。

> 3.8 版更變: 如果可用将使用 __index__()。

> 3.8 版後已⧄用: __int__() 已被弃用。

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

> 返回 *obj* 的 C long 表达方式。如果 *obj* 不是 `PyLongObject` 的实例，先调用它的 __index__() 或
> __int__() 方法 (如果有) 将其转换为 `PyLongObject` 。

> 如果 *obj* 的值大于 LONG_MAX 或小于 LONG_MIN，则会把 *overflow 分别置为 "1" 或 -1，并返回 1；否
> 则，将 *overflow 置为 *0*。如果发生其他异常，则会按常规把 *overflow 置为 0，并返回 -1。

> 发生错误时返回 -1 。使用 `PyErr_Occurred()` 来消歧义。

> 3.8 版更變: 如果可用将使用 __index__()。

> 3.8 版後已⧄用: __int__() 已被弃用。

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

> Return a C long long representation of *obj*. If *obj* is not an instance of `PyLongObject`, first call its
> __index__() or __int__() method (if present) to convert it to a `PyLongObject`.

> 如果 *obj* 值超出 long long ，触发 OverflowError

> 发生错误时返回 -1 。使用 `PyErr_Occurred()` 来消歧义。

> 3.8 版更變: 如果可用将使用 __index__()。

> 3.8 版後已⧄用: __int__() 已被弃用。

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

> Return a C `long long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its
> __index__() or __int__() 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。
>
> 发生错误时返回 -1 。使用*PyErr_Occurred()* 来消歧义。
>
> 3.2 版新加入.
>
> 3.8 版更變: 如果可用将使用 __index__()。
>
> 3.8 版後已⊞用: __int__() 已被弃用。

Py_ssize_t **PyLong_AsSsize_t**(*PyObject* **pylong*)

> 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。
>
> 发生错误时返回 -1 。使用*PyErr_Occurred()* 来消歧义。

unsigned long **PyLong_AsUnsignedLong**(*PyObject* **pylong*)

> 返回 *pylong* 的 C unsigned long 形式。*pylong* 必须是*PyLongObject* 的实例。
>
> 如果 *pylong* 的值超出了 unsigned long 的取值范围则会引发 OverflowError。
>
> 出错时返回 (unsigned long)-1 ，请利用*PyErr_Occurred()* 获取详细信息。

size_t **PyLong_AsSize_t**(*PyObject* **pylong*)

> 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()* 获取详细信息。

unsigned long long **PyLong_AsUnsignedLongLong**(*PyObject* **pylong*)

> 返回 *pylong* 的 C 语言 unsigned long long 形式。*pylong* 必须是*PyLongObject* 的实例。
>
> 如果 *pylong* 的值超出了 unsigned long long 的取值范围则会引发 OverflowError。
>
> 出错时返回 (unsigned long long)-1，请利用*PyErr_Occurred()* 获取详细信息。
>
> 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*)

> 返回 *obj* 的 C unsigned long 表示形式。如果 *obj* 不是*PyLongObject* 的实例，则会先调用其
> __index__() 或 __int__() 方法（如果有的话）将其转为*PyLongObject*。
>
> 如果 *obj* 的值超出了 unsigned long 的范围，则返回该值对 ULONG_MAX + 1 求模的差值。
>
> 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__()。
>
> 3.8 版後已⊞用: __int__() 已被弃用。

unsigned long long **PyLong_AsUnsignedLongLongMask**(*PyObject* **obj*)

> 返回 *obj* 的 C unsigned long long 表示形式。如果 *obj* 不是*PyLongObject* 的实例，则会先调用
> 其 __index__() 或 __int__() 方法（如果有的话）将其转为*PyLongObject*。
>
> 如果 *obj* 的值超出了 unsigned long long 的范围，则返回该值对 ULLONG_MAX + 1 求模的差值。
>
> 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__()。
>
> 3.8 版後已⊞用: __int__() 已被弃用。

double **PyLong_AsDouble**(*PyObject* *pylong*)

返回 *pylong* 的 C 语言 double 形式。*pylong* 必须是*PyLongObject* 的实例。

如果 *pylong* 的值超出了 double 的取值范围则会引发 OverflowError。

出错时返回 -1.0，请利用*PyErr_Occurred()* 辨别具体问题。

void* **PyLong_AsVoidPtr**(*PyObject* *pylong*)

将一个 Python 整数 *pylong* 转换为 C 语言的 void 指针。如果 *pylong* 无法转换，则会触发 OverflowError。这只是保证为*PyLong_FromVoidPtr()* 创建的值产生一个合法的 void 指针。

出错时返回 NULL，请利用*PyErr_Occurred()* 辨别具体问题。

8.2.2 布林物件

Python 中的布尔值是作为整数的子类实现的。只有 Py_False 和 Py_True 两个布尔值。因此，正常的创建和删除功能不适用于布尔值。但是，下列宏可用。

int **PyBool_Check**(*PyObject* *o*)

如果 *o* 的类型为 PyBool_Type 则返回真值。此函数总是会成功执行。

*PyObject** **Py_False**

Python 的 False 对象没有任何方法，它需要和其他对象一样遵循引用计数。

*PyObject** **Py_True**

Python 的 True 对象没有任何方法，它需要和其他对象一样遵循引用计数。

Py_RETURN_FALSE

从函数返回 Py_False 时，需要增加它的引用计数。

Py_RETURN_TRUE

从函数返回 Py_True 时，需要增加它的引用计数。

*PyObject** **PyBool_FromLong**(long *v*)

Return value: New reference. 根据 *v* 的实际值，返回一个 Py_True 或者 Py_False 的新引用。

8.2.3 浮點數（**Floating Point**）物件

PyFloatObject

这个 C 类型*PyObject* 的子类型代表一个 Python 浮点数对象。

PyTypeObject **PyFloat_Type**

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. 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. 根据 *v* 创建一个*PyFloatObject* 对象，失败时返回 NULL。

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

> 返回一个 C double 代表 *pyfloat* 的内容。如果 *pyfloat* 不是一个 Python 浮点数对象但是具有 __float__() 方法,此方法将首先被调用,将 *pyfloat* 转换成一个数点数。如果 __float__() 未定义则 将回退至 __index__()。如果失败,此方法将返回 -1.0,因此开发者应当调用 *PyErr_Occurred()* 来检查错误。
>
> 3.8 版更變: 如果可用将使用 __index__()。

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

> 返回一个 *pyfloat* 内容的 C double 表示,但没有错误检查。

PyObject* **PyFloat_GetInfo**(void)

> *Return value: New reference.* 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**()

> 返回最大可表示的有限浮点数 *DBL_MAX* 为 C double 。

double **PyFloat_GetMin**()

> 返回最小可表示归一化正浮点数 *DBL_MIN* 为 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。

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:

```c
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

PyComplexObject
> 这个 C 类型*PyObject* 的子类型代表一个 Python 复数对象。

PyTypeObject **PyComplex_Type**
> 这是个属于 C 类型*PyTypeObject* 的代表 Python 复数类型的实例。和 Python 层面的类 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.* 根据 *real* 和 *imag* 返回一个新的 C 类型*PyComplexObject* 对象。

double **PyComplex_RealAsDouble**(*PyObject *op*)
> 以 C 类型 double 返回 *op* 的实部。

double **PyComplex_ImagAsDouble**(*PyObject *op*)
> 以 C 类型 double 返回 *op* 的虚部。

Py_complex **PyComplex_AsCComplex**(*PyObject *op*)
> 返回复数 *op* 的 C 类型*Py_complex* 值。

> 如果 *op* 不是一个 Python 复数对象，但是具有 __complex__() 方法，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 序列对象

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 字节对象

These functions raise `TypeError` when expecting a bytes parameter and called with a non-bytes parameter.

PyBytesObject
> 这种*PyObject* 的子类型表示一个 Python 字节对象。

PyTypeObject **PyBytes_Type**
> *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.* 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.* 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.* 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
%%	不适用	文字% 字符。
%c	int	一个字节，被表示为一个 C 语言的整型
%d	int	相当于 printf("%d").[1]
%u	unsigned int	相当于 printf("%u").[1]
%ld	长整型	相当于 printf("%ld").[1]
%lu	unsigned long	相当于 printf("%lu").[1]
%zd	*Py_ssize_t*	相当于 printf("%zd").[1]
%zu	size_t	相当于 printf("%zu").[1]
%i	int	相当于 printf("%i").[1]
%x	int	相当于 printf("%x").[1]
%s	const char*	以 null 为终止符的 C 字符数组。
%p	const void*	一个 C 指针的十六进制表示形式。基本等价于 printf("%p") 但它会确保以字面值 0x 开头，不论系统平台上 printf 的输出是什么。

> 无法识别的格式字符会导致将格式字符串的其余所有内容原样复制到结果对象，并丢弃所有多余的参数。

PyObject* **PyBytes_FromFormatV** (const char **format*, va_list *vargs*)
> *Return value: New reference.* 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.* 返回字节表示实现缓冲区协议的对象 *o*。

Py_ssize_t **PyBytes_Size** (*PyObject* *o*)
> 返回字节对象 *o* 中字节的长度。

---

[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。

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*)
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*)
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*)
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*)
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

PyByteArrayObject
这个 *PyObject* 的子类型表示一个 Python 字节数组对象。

PyTypeObject **PyByteArray_Type**
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.* 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.* 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.* 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*)
> 在检查 NULL 指针后返回 *bytearray* 的大小。

char* **PyByteArray_AsString** (*PyObject* *bytearray*)
> 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*)
> 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 物件與编码

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⊞: 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:

Py_UCS4
Py_UCS2
Py_UCS1
> 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.

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.

PyASCIIObject
PyCompactUnicodeObject
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**
> 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.
>
> Deprecated since version 3.10, will be removed in version 3.12: 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

> Return values of the *PyUnicode_KIND()* macro.

> 3.3 版新加入.

> Deprecated since version 3.10, will be removed in version 3.12: PyUnicode_WCHAR_KIND is deprecated.

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).

> Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using *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).

> Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using *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()*.

> Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using the PyUnicode_nBYTE_DATA() family of macros.

int **PyUnicode_IsIdentifier**(*PyObject *o*)

> 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*)

> Return 1 or 0 depending on whether *ch* is a whitespace character.

int **Py_UNICODE_ISLOWER**(*Py_UCS4 ch*)

> Return 1 or 0 depending on whether *ch* is a lowercase character.

int **Py_UNICODE_ISUPPER**(*Py_UCS4 ch*)

> Return 1 or 0 depending on whether *ch* is an uppercase character.

int **Py_UNICODE_ISTITLE**(*Py_UCS4 ch*)

> Return 1 or 0 depending on whether *ch* is a titlecase character.

int **Py_UNICODE_ISLINEBREAK**(*Py_UCS4 ch*)

> Return 1 or 0 depending on whether *ch* is a linebreak character.

int **Py_UNICODE_ISDECIMAL**(*Py_UCS4 ch*)

> Return 1 or 0 depending on whether *ch* is a decimal character.

int **Py_UNICODE_ISDIGIT**(*Py_UCS4 ch*)

> Return 1 or 0 depending on whether *ch* is a digit character.

int **Py_UNICODE_ISNUMERIC**(*Py_UCS4 ch*)

> Return 1 or 0 depending on whether *ch* is a numeric character.

int **Py_UNICODE_ISALPHA**(*Py_UCS4 ch*)
> Return `1` or `0` depending on whether *ch* is an alphabetic character.

int **Py_UNICODE_ISALNUM**(*Py_UCS4 ch*)
> Return `1` or `0` depending on whether *ch* is an alphanumeric character.

int **Py_UNICODE_ISPRINTABLE**(*Py_UCS4 ch*)
> Return `1` or `0` depending on whether *ch* is a printable character. Nonprintable characters are those characters defined in the Unicode character database as "Other" or "Separator", excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when `repr()` is invoked on a string. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.)

These APIs can be used for fast direct character conversions:

Py_UCS4 **Py_UNICODE_TOLOWER**(*Py_UCS4 ch*)
> Return the character *ch* converted to lower case.
>
> 3.3 版後已⬚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⬚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⬚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.* 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.* Create a Unicode object from a UTF-8 encoded null-terminated char buffer *u*.

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

> *Return value: New reference.* 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
%%	不适用	文字% 字符。
%c	int	单个字符，表示为 C 语言的整型。
%d	int	相当于 `printf("%d")`.[1]
%u	unsigned int	相当于 `printf("%u")`.[1]
%ld	长整型	相当于 `printf("%ld")`.[1]
%li	长整型	相当于 `printf("%li")`.[1]
%lu	unsigned long	相当于 `printf("%lu")`.[1]
%lld	long long	相当于 `printf("%lld")`.[1]
%lli	long long	相当于 `printf("%lli")`.[1]
%llu	unsigned long long	相当于 `printf("%llu")`.[1]
%zd	*Py_ssize_t*	相当于 `printf("%zd")`.[1]
%zi	*Py_ssize_t*	相当于 `printf("%zi")`.[1]
%zu	size_t	相当于 `printf("%zu")`.[1]
%i	int	相当于 `printf("%i")`.[1]
%x	int	相当于 `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*	A Unicode object.
%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⬚: 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. 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. Decode an encoded object *obj* to a Unicode object.

   `bytes`, `bytearray` and other *bytes-like objects* are decoded according to the given *encoding* and using the error handling defined by *errors*. Both can be `NULL` to have the interface use the default values (see *Built-in Codecs* for details).

   All other objects, including Unicode objects, cause a `TypeError` to be set.

---

[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.

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*)
> 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*)
> 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*)
> 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.* 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*)
> 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*)
> 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

Deprecated since version 3.3, will be removed in version 3.12.

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()*.
>
> Deprecated since version 3.3, will be removed in version 3.12: 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.
>
> Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using *PyUnicode_AsUCS4()*, *PyUnicode_AsWideChar()*, *PyUnicode_ReadChar()* or similar new APIs.
>
> Deprecated since version 3.3, will be removed in version 3.10.

*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.
>
> Deprecated since version 3.3, will be removed in version 3.11: 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.
>
> Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using *PyUnicode_AsUCS4()*, *PyUnicode_AsWideChar()*, *PyUnicode_ReadChar()* or similar new APIs.

*Py_UNICODE** **PyUnicode_AsUnicodeCopy** (*PyObject *unicode*)
> Create a copy of a Unicode string ending with a null code point. Return NULL and raise a MemoryError exception on memory allocation failure, otherwise return a new allocated buffer (use *PyMem_Free()* to free the buffer). 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.
>
> 3.2 版新加入.
>
> Please migrate to using *PyUnicode_AsUCS4Copy()* or similar new APIs.

Py_ssize_t **PyUnicode_GetSize**(*PyObject* *unicode*)

> Return the size of the deprecated *Py_UNICODE* representation, in code units (this includes surrogate pairs as 2 units).

> Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using *PyUnicode_GET_LENGTH()*.

PyObject* **PyUnicode_FromObject**(*PyObject* **obj*)

> *Return value: New reference.* 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.* 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:**

> The *Py_DecodeLocale()* 函数。

> 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.* 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.* 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:**

> The *Py_EncodeLocale()* function.

> 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`.

文件系统编码格式

To encode and decode file names and other environment strings, `Py_FileSystemDefaultEncoding` should be used as the encoding, and `Py_FileSystemDefaultEncodeErrors` should be used as the error handler (**PEP 383** and **PEP 529**). To encode file names to `bytes` during argument parsing, the `"O&"` converter should be used, passing `PyUnicode_FSConverter()` as the conversion function:

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

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: 接受一个类路径对象。

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*)

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: 接受一个类路径对象。

PyObject* **PyUnicode_DecodeFSDefaultAndSize** (const char **s*, *Py_ssize_t size*)

Return value: New reference. Decode a string using `Py_FileSystemDefaultEncoding` and the `Py_FileSystemDefaultEncodeErrors` 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:

The `Py_DecodeLocale()` 函数。

3.6 版更變: Use `Py_FileSystemDefaultEncodeErrors` error handler.

PyObject* **PyUnicode_DecodeFSDefault** (const char **s*)

Return value: New reference. Decode a null-terminated string using `Py_FileSystemDefaultEncoding` and the `Py_FileSystemDefaultEncodeErrors` 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. 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:

The *`Py_EncodeLocale()`* function.

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. 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*)

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*)

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.* 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.* 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.

> Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsEncodedString()*.

UTF-8 Codecs

These are the UTF-8 codec APIs:

PyObject* **PyUnicode_DecodeUTF8** (const char **s*, *Py_ssize_t size*, const char **errors*)
> *Return value: New reference.* 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.* 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.* 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*)
> 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 *`.

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.
>
> Deprecated since version 3.3, will be removed in version 3.11: 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.* 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.* 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.* 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.

Deprecated since version 3.3, will be removed in version 3.11: 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. Decode *size* bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. *errors* (if non-`NULL`) defines the error handling. It defaults to "strict".

    If *byteorder* is non-`NULL`, the decoder starts decoding using the given byte order:

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

    If `*byteorder` is zero, and the first two bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is -1 or 1, any byte order mark is copied to the output (where it will result in either a `\ufeff` or a `\ufffe` character).

    After completion, `*byteorder` is set to the current byte order at the end of input data.

    If *byteorder* is `NULL`, the codec starts in native order mode.

    Return `NULL` if an exception was raised by the codec.

PyObject* **PyUnicode_DecodeUTF16Stateful** (const char **s*, *Py_ssize_t size*, const char **errors*, int **byteorder*, *Py_ssize_t *consumed*)
    Return value: New reference. 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. 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.

Deprecated since version 3.3, will be removed in version 3.11: 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.* 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.* 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.

> Deprecated since version 3.3, will be removed in version 3.11: 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.* 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.* 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.

> Deprecated since version 3.3, will be removed in version 3.11: 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.* 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.* 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.

> Deprecated since version 3.3, will be removed in version 3.11: 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.* 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.* 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.

> Deprecated since version 3.3, will be removed in version 3.11: 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.* 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.* 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.

> Deprecated since version 3.3, will be removed in version 3.11: 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. 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. 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.

    Deprecated since version 3.3, will be removed in version 3.11: 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. 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.

    Deprecated since version 3.3, will be removed in version 3.11: 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.* 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.* 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.* 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.* 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.
>
> Deprecated since version 3.3, will be removed in version 4.0: 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.* Concat two strings giving a new Unicode string.

PyObject* **PyUnicode_Split** (*PyObject *s*, *PyObject *sep*, *Py_ssize_t maxsplit*)
> *Return value: New reference.* 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.* 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.* 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*)

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*)

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*)

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*)

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. 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*)

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*)

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. 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. Return a new string object from *format* and *args*; this is analogous to format % args.

int **PyUnicode_Contains** (*PyObject *container*, *PyObject *element*)

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*)

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.* 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 元组（**Tuple**）物件

PyTupleObject
> 这个*PyObject* 的子类型代表一个 Python 的元组对象。

PyTypeObject **PyTuple_Type**
> *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.* 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.* 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)
> 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.* 返回 *p* 所指向的元组中，位于 *pos* 处的对象。如果 *pos* 超出界限，返回 NULL，并抛出一个 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.* 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)
> 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。

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

---

void **PyTuple_SET_ITEM**(*PyObject* *p, *Py_ssize_t pos*, *PyObject* *o)
> 类似于*PyTuple_SetItem()*，但不进行错误检查，并且应该 只是被用来填充全新的元组。

備㊟： 这个宏会 "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. 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.

PyStructSequence_Desc

包含要创建的结构序列类型的元信息。

域	C Type	意义
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）

PyStructSequence_Field

描述结构序列的一个字段。当结构序列被建模为元组时，所有字段的类型都是 *PyObject**。在 *PyStructSequence_Desc* 的 fields 数组中的索引确定了结构序列描述的是哪个字段。

域	C Type	意义
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. 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.* 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*)
> 将结构序列 *p* 的索引 *pos* 处的字段设置为值 *o*。与 `PyTuple_SET_ITEM()` 一样，它应该只用于填充全新的实例。

> ---
> 備⃞: 这个函数"窃取"了指向 *o* 的一个引用。
> ---

void **PyStructSequence_SET_ITEM**(*PyObject *p*, *Py_ssize_t *pos*, *PyObject *o*)
> `PyStructSequence_SetItem()` 的宏版本。

> ---
> 備⃞: 这个函数"窃取"了指向 *o* 的一个引用。
> ---

8.3.6 List（串列）物件

PyListObject
> 这个 C 类型 *PyObject* 的子类型代表一个 Python 列表对象。

PyTypeObject **PyList_Type**
> 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.* 成功时返回一个长度为 *len* 的新列表，失败时返回 NULL。

> ---
> 備⃞: 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*)
> 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.* 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*)

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*)

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*)

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. 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*)

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*)

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*)

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. 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 容器对象

8.4.1 字典物件

PyDictObject

PyObject 子型態代表一個 Python 字典物件。

PyTypeObject **PyDict_Type**

PyTypeObject 實例代表一個 Python 字典型態。此與 Python 層中的 dict 是同一個物件。

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.* 返回一个新的空字典，失败时返回 NULL。

PyObject* **PyDictProxy_New**(*PyObject *mapping*)
> *Return value: New reference.* 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*)
> 清空现有字典的所有键值对。

int **PyDict_Contains**(*PyObject *p*, *PyObject *key*)
> 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.* 返回与 *p* 包含相同键值对的新字典。

int **PyDict_SetItem**(*PyObject *p*, *PyObject *key*, *PyObject *val*)
> 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*)
> 使用 *key* 作为键将 *val* 插入到字典 *p*。*key* 应当为 const char*。键对象是使用 PyUnicode_FromString(key) 创建的。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_DelItem**(*PyObject *p*, *PyObject *key*)
> 移除字典 *p* 中键为 *key* 的条目。*key* 必须是可哈希的；如果不是，则会引发 TypeError。如果字典中没有 *key*，则会引发 KeyError。成功时返回 0，失败时返回 -1。

int **PyDict_DelItemString**(*PyObject *p*, const char **key*)
> 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.* 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。

PyObject* **PyDict_GetItemWithError**(*PyObject *p*, *PyObject *key*)
> *Return value: Borrowed reference.* *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.* 这与*PyDict_GetItem()*一样，但是 *key* 被指定为 const char*，而不是*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.* 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.* 返回一个包含字典中所有键 (keys) 的 `PyListObject`。

PyObject* **PyDict_Values**(*PyObject *p*)
> *Return value: New reference.* 返回一个包含字典中所有值 (values) 的 `PyListObject`。

Py_ssize_t **PyDict_Size**(*PyObject *p*)
> 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*)
> 迭代字典 *p* 中的所有键值对。在第一次调用此函数开始迭代之前，由 *ppos* 所引用的 `Py_ssize_t` 必须
> 被初始化为 `0`；该函数将为字典中的每个键值对返回真值，一旦所有键值对都报告完毕则返回假值。形
> 参 *pkey* 和 *pvalue* 应当指向 `PyObject *` 变量，它们将分别使用每个键和值来填充，或者也可以为 NULL。
> 通过它们返回的任何引用都是暂借的。*ppos* 在迭代期间不应被更改。它的值表示内部字典结构中的偏
> 移量，并且由于结构是稀疏的，因此偏移量并不连续。
>
> 例如:
>
> ```
> PyObject *key, *value;
> Py_ssize_t pos = 0;
>
> while (PyDict_Next(self->dict, &pos, &key, &value)) {
>     /* do something interesting with the values... */
>     ...
> }
> ```
>
> 字典 *p* 不应该在遍历期间发生改变。在遍历字典时，改变键中的值是安全的，但仅限于键的集合不发
> 生改变。例如:
>
> ```
> PyObject *key, *value;
> Py_ssize_t pos = 0;
>
> while (PyDict_Next(self->dict, &pos, &key, &value)) {
>     long i = PyLong_AsLong(value);
>     if (i == -1 && PyErr_Occurred()) {
>         return -1;
>     }
>     PyObject *o = PyLong_FromLong(i + 1);
>     if (o == NULL)
>         return -1;
>     if (PyDict_SetItem(self->dict, key, o) < 0) {
>         Py_DECREF(o);
>         return -1;
>     }
>     Py_DECREF(o);
> }
> ```

int **PyDict_Merge**(*PyObject *a*, *PyObject *b*, int *override*)
> 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*)
> 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*)

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）：

```python
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

This section details the public API for `set` and `frozenset` objects. Any functionality not listed below is best accessed using either the abstract object protocol (including *PyObject_CallMethod()*, *PyObject_RichCompareBool()*, *PyObject_Hash()*, *PyObject_Repr()*, *PyObject_IsTrue()*, *PyObject_Print()*, and *PyObject_GetIter()*) or the abstract number protocol (including *PyNumber_And()*, *PyNumber_Subtract()*, *PyNumber_Or()*, *PyNumber_Xor()*, *PyNumber_InPlaceAnd()*, *PyNumber_InPlaceSubtract()*, *PyNumber_InPlaceOr()*, and *PyNumber_InPlaceXor()*).

PySetObject

This subtype of *PyObject* is used to hold the internal data for both `set` and `frozenset` objects. It is like a *PyDictObject* in that it is a fixed size for small sets (much like tuple storage) and will point to a separate, variable sized block of memory for medium and large sized sets (much like list storage). None of the fields of this structure should be considered public and all are subject to change. All access should be done through the documented API rather than by manipulating the values in the structure.

PyTypeObject **PySet_Type**

这是一个 *PyTypeObject* 实例，表示 Python `set` 类型。

PyTypeObject **PyFrozenSet_Type**

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 **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. 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. 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*)
> 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*)
> 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*)
> Add *key* to a set instance. Also works with frozenset instances (like *PyTuple_SetItem()* it can be used to fill in the values of brand new frozensets before they are exposed to other code). Return 0 on success or -1 on failure. Raise a TypeError if the *key* is unhashable. Raise a MemoryError if there is no room to grow. Raise a SystemError if *set* is not an instance of set or its subtype.

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*)
> 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.* 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*)
> 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。

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. 返回函数对象 *op* 的 __module__ 属性，通常为一个包含了模块名称的字符串，但可以通过 Python 代码设为返回其他任意对象。

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 實體方法物件

实例方法是 *PyCFunction* 的包装器，也是将 *PyCFunction* 绑定到类对象的一种新方式。它替代了原先的调用 PyMethod_New(func, NULL, class)。

PyTypeObject **PyInstanceMethod_Type**
    这个 *PyTypeObject* 实例代表 Python 实例方法类型。它并不对 Python 程序公开。

int **PyInstanceMethod_Check**(*PyObject *o*)
    如果 *o* 是一个实例方法对象 (类型为 *PyInstanceMethod_Type*) 则返回真值。形参必须不为 NULL。此函数总是会成功执行。

PyObject* **PyInstanceMethod_New**(*PyObject *func*)
    Return value: New reference. Return a new instance method object, with *func* being any callable object. *func* is the function that will be called when the instance method is called.

PyObject* **PyInstanceMethod_Function**(*PyObject *im*)
    Return value: Borrowed reference. 返回关联到实例方法 *im* 的函数对象。

PyObject* **PyInstanceMethod_GET_FUNCTION**(*PyObject *im*)
> *Return value: Borrowed reference.* 宏版本的`PyInstanceMethod_Function()`，略去了错误检测。

8.5.3 方法对象

方法是绑定的函数对象。方法总是会被绑定到一个用户自定义类的实例。未绑定方法（绑定到一个类的方法）已不再可用。

PyTypeObject **PyMethod_Type**
> 这个`PyTypeObject`实例代表 Python 方法类型。它作为`types.MethodType`向 Python 程序公开。

int **PyMethod_Check**(*PyObject *o*)
> 如果 *o* 是一个方法对象 (类型为`PyMethod_Type`) 则返回真值。形参必须不为 NULL。此函数总是会成功执行。

PyObject* **PyMethod_New**(*PyObject *func*, *PyObject *self*)
> *Return value: New reference.* 返回一个新的方法对象，*func* 应为任意可调用对象，*self* 为该方法应绑定的实例。在方法被调用时 *func* 将作为函数被调用。*self* 必须不为 NULL。

PyObject* **PyMethod_Function**(*PyObject *meth*)
> *Return value: Borrowed reference.* 返回关联到方法 *meth* 的函数对象。

PyObject* **PyMethod_GET_FUNCTION**(*PyObject *meth*)
> *Return value: Borrowed reference.* 宏版本的`PyMethod_Function()`，略去了错误检测。

PyObject* **PyMethod_Self**(*PyObject *meth*)
> *Return value: Borrowed reference.* 返回关联到方法 *meth* 的实例。

PyObject* **PyMethod_GET_SELF**(*PyObject *meth*)
> *Return value: Borrowed reference.* 宏版本的`PyMethod_Self()`，略去了错误检测。

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。

PyCellObject
> C 結構的 cell 物件

PyTypeObject **PyCell_Type**
> 對應 cell 物件的物件型⽥。

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 ⽥容中的 *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。

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。

8.6 其他对象

8.6.1 档案（File）物件

这此 API 是对内置文件对象的 Python 2 C API 的最小仿真，它过去依赖于 C 标准库的缓冲 I/O (FILE*) 支持。在 Python 3 中，文件和流使用新的 io 模块，该模块在操作系统的低层级无缓冲 I/O 之上定义了几个层。下面所描述的函数是针对这些新 API 的便捷 C 包装器，主要用于解释器的内部错误报告；建议第三方代码改为访问 io API。

PyObject* **PyFile_FromFd**(int *fd*, const char *name, const char *mode, int *buffering*, const char *encoding,
const char *errors, const char *newline, int *closefd*)

Return value: New reference. 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)

将与 *p* 关联的文件描述器返回为 int 。如果对象是整数，则返回其值。如果没有，则调用对象的
fileno() 方法（如果存在）；该方法必须返回一个整数，该整数作为文件描述器值返回。设置异常并
在失败时返回 −1。

PyObject* **PyFile_GetLine**(*PyObject* *p, int *n*)

Return value: New reference. 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。

处理程序是一个类型为 PyObject *(*)(PyObject *path, void *userData) 的函数，其中
path 确保为 *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*)

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)

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 -1，⑤會設定合適的例外狀⑤。

8.6.2 模組物件模組

PyTypeObject **PyModule_Type**
> 这个*PyTypeObject* 的实例代表 Python 模块类型。它以 types.ModuleType 的形式暴露给 Python 程序。

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。该函数永远有返回。

int **PyModule_CheckExact** (*PyObject *p*)
> 当 *p* 为模块对象且不是*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.* 返回新的模块对象，其属性 __name__ 为 *name*。模块的这些属性 __name__,
> __doc__, __package__, and __loader__ （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: 属性 __package__ 和 __loader__ 被设为 "None"。

PyObject* **PyModule_New** (const char **name*)
> *Return value: New reference.* 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.* Return the dictionary object that implements *module*'s namespace; this object is the same as the __dict__ attribute of the module object. If *module* is not a module object (or a subtype of a module object), SystemError is raised and NULL is returned.
>
> It is recommended extensions use other PyModule_*() and PyObject_*() functions rather than directly manipulate a module's __dict__.

PyObject* **PyModule_GetNameObject** (*PyObject *module*)
> *Return value: New reference.* Return *module*'s __name__ value. If the module does not provide one, or if it is not a string, SystemError is raised and NULL is returned.
>
> 3.3 版新加入.

const char* **PyModule_GetName** (*PyObject *module*)
> Similar to *PyModule_GetNameObject()* but return the name encoded to 'utf-8'.

void* **PyModule_GetState** (*PyObject *module*)
> Return the "state" of the module, that is, a pointer to the block of memory allocated at module creation time, or NULL. See *PyModuleDef.m_size*.

PyModuleDef * **PyModule_GetDef** (*PyObject *module*)
> Return a pointer to the *PyModuleDef* struct from which the module was created, or NULL if the module wasn't created from a definition.

PyObject* **PyModule_GetFilenameObject** (*PyObject *module*)
> *Return value: New reference.* Return the name of the file from which *module* was loaded using *module*'s __file__ attribute. If this is not defined, or if it is not a unicode string, raise SystemError and return NULL; otherwise return a reference to a Unicode object.
>
> 3.2 版新加入.

const char* **PyModule_GetFilename** (*PyObject *module*)
> Similar to *PyModule_GetFilenameObject()* but return the filename encoded to 'utf-8'.
>
> 3.2 版後已⺊用: *PyModule_GetFilename()* raises UnicodeEncodeError on unencodable filenames, use *PyModule_GetFilenameObject()* instead.

Initializing C modules

Modules objects are usually created from extension modules (shared libraries which export an initialization function), or compiled-in modules (where the initialization function is added using *PyImport_AppendInittab()*). See building or extending-with-embedding for details.

The initialization function can either pass a module definition instance to *PyModule_Create()*, and return the resulting module object, or request "multi-phase initialization" by returning the definition struct itself.

PyModuleDef
  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.

    See **PEP 3121** for more details.

  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. 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.

備註: 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_AddObject()`.

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.* 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:

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 *module* 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.* Create a new module object, given the definition in *module* 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⬜: 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*)

> Process any execution slots (*Py_mod_exec*) given in *def*.

> 3.5 版新加入.

int **PyModule_SetDocString**(*PyObject *module*, const char **docstring*)

> 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*)

> 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_AddObject**(*PyObject *module*, const char **name*, *PyObject *value*)

> Add an object to *module* as *name*. This is a convenience function which can be used from the module's initialization function. This steals a reference to *value* on success. Return `-1` on error, `0` on success.

> ---
> 備⬜: 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. Example usage:

```
Py_INCREF(spam);
if (PyModule_AddObject(module, "spam", spam) < 0) {
    Py_DECREF(module);
    Py_DECREF(spam);
    return NULL;
}
```

int **PyModule_AddIntConstant** (*PyObject* **module*, const char **name*, long *value*)
> 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*)
> 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*)
> 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.* 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*)
> 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).

> 呼叫者必须持有 GIL。

> Return 0 on success or -1 on failure.

> 3.3 版新加入.

int **PyState_RemoveModule**(*PyModuleDef* **def*)
> Removes the module object created from *def* from the interpreter state. Return 0 on success or -1 on failure.

> 呼叫者必须持有 GIL。

> 3.3 版新加入.

8.6.3 迭代器（**Iterator**）物件

Python 提供了两个通用迭代器对象。第一个是序列迭代器，它使用支持 __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**
> *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.* 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**
> 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.* 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 修飾器物件

"描述符" 是描述对象的某些属性的对象。它们存在于类型对象的字典中。

PyTypeObject **PyProperty_Type**
> 内建描述符类型的类型对象。

PyObject* **PyDescr_NewGetSet**(*PyTypeObject* **type*, struct *PyGetSetDef* **getset*)
> *Return value: New reference.*

PyObject* **PyDescr_NewMember**(*PyTypeObject* **type*, struct *PyMemberDef* **meth*)
> *Return value: New reference.*

PyObject* **PyDescr_NewMethod**(*PyTypeObject* **type*, struct *PyMethodDef* **meth*)
> *Return value: New reference.*

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.*

int **PyDescr_IsData**(*PyObject* **descr*)
> 如果描述符对象 *descr* 描述数据属性，则返回 true；如果描述方法，则返回 false。*descr* 必须是描述符对象；没有错误检查。

PyObject* **PyWrapper_New**(*PyObject* *, *PyObject* *)
> *Return value: New reference.*

8.6.5 切片物件

PyTypeObject **PySlice_Type**
> 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.* 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)
> 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)
> *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 版後已弃用: 如果 Py_LIMITED_API 设置为小于 0x03050400 或 0x03060000 与 0x03060100 之间的值（不包括边界）则 PySlice_GetIndicesEx() 为已弃用的函数。

---

int **PySlice_Unpack**(*PyObject *slice*, *Py_ssize_t *start*, *Py_ssize_t *stop*, *Py_ssize_t *step*)

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*)

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 对象

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. 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. 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. 从定义缓冲区接口的对象创建一个 memoryview 对象*contiguous* 内存块（在'C'或'F'ortran *order* 中）。如果内存是连续的，则 memoryview 对象指向原始内存。否则，复制并且 memoryview 指向新的 bytes 对象。

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。

Py_buffer ****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 弱參照物件

Python 支持 "弱引用" 作为一类对象。具体来说，有两种直接实现弱引用的对象。第一种就是简单的引用对象，第二种尽可能地作用为一个原对象的代理。

int **PyWeakref_Check**(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.* 返回对象 *ob* 的一个弱引用对象。该函数总是会返回一个新引用，但不保证创建一个新的对象；它有可能返回一个现有的引用对象。第二个形参 *callback* 为一个可调用对象，它会在 *ob* 被作为垃圾回收时接收通知；它应该接受一个单独形参，即弱引用对象本身。*callback* 也可以为 None 或 NULL。如果 *ob* 不是一个弱引用对象，或者如果 *callback* 不是可调用对象，None 或 NULL，该函数将返回 NULL 并且引发 TypeError。

PyObject* **PyWeakref_NewProxy**(*PyObject* *ob, *PyObject* *callback)
> *Return value: New reference.* 返回对象 *ob* 的一个弱引用代理对象。该函数将总是返回一个新的引用，但不保证创建一个新的对象；它有可能返回一个现有的代理对象。第二个形参 *callback* 为一个可调用对象，它会在 *ob* 被作为垃圾回收时接收通知；它应该接受一个单独形参，即弱引用对象本身。*callback* 也可以为 None 或 NULL。如果 *ob* 不是一个弱引用对象，或者如果 *callback* 不是可调用对象，None 或 NULL，该函数将返回 NULL 并且引发 TypeError。

PyObject* **PyWeakref_GetObject**(*PyObject* *ref)
> *Return value: Borrowed reference.* 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。

> 備注: 该函数返回被引用对象的一个 ** 借来的引用 **。这意味着除非你很清楚在你使用期间这个对象不可能被销毁，否则你应该始终对该对象调用 *Py_INCREF()*。

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。

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.

PyCapsule
> 这个 *PyObject* 的子类型代表一个隐藏的值，适用于需要将隐藏值（作为 void* 指针）通过 Python 代码传递到其他 C 代码的 C 扩展模块。它常常被用来让在一个模块中定义的 C 函数指针在其他模块中可用，这样就可以使用常规导入机制来访问在动态加载的模块中定义的 C API。

PyCapsule_Destructor
> 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.* 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*)
> 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*)
> 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*)
> 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*)
> 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*)
> 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*)
> 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*)
> 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*)
> 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*)
> 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*)
> 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 ⏍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()*。

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 协程对象

3.5 版新加入.

协程对象是使用 async 关键字声明的函数返回的。

PyCoroObject
> 用于协程对象的 C 结构体。

PyTypeObject **PyCoro_Type**
> 与协程对象对应的类型对 象。

int **PyCoro_CheckExact** (*PyObject *ob*)
> 如果 *ob* 的类型是 *PyCoro_Type* 则返回真值；*ob* 必须不为 NULL。此函数总是会成功执行。

PyObject* **PyCoro_New** (*PyFrameObject *frame*, *PyObject *name*, *PyObject *qualname*)
> *Return value: New reference.* 基于 *frame* 对象创建并返回一个新的协程对象，其中 __name__ 和

__qualname__ 设为 *name* 和 *qualname*。此函数会取得一个对 *frame* 的引用。*frame* 参数必须不为 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);
```

请参阅 bpo-34762 了解详情。

3.7 版新加入.

本节深入介绍了 contextvars 模块的公用 C API。

PyContext
> 用于表示 contextvars.Context 对象的 C 结构体。

PyContextVar
> 用于表示 contextvars.ContextVar 对象的 C 结构体。

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.* 返回一個 datetime.timezone 對象，該對象具有以 *offset* 參數表示的固定時差和時區名稱 *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，為正整數。

int **PyDateTime_GET_MONTH** (PyDateTime_Date *o)
> 回傳月份，為正整數，從 1 到 12。

int **PyDateTime_GET_DAY** (PyDateTime_Date *o)
> 回傳日期，為正整數，從 1 到 31。

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，為正整數，從 0 到 23。

int **PyDateTime_DATE_GET_MINUTE** (PyDateTime_DateTime *o)
> 回傳分鐘，為正整數，從 0 到 59。

int **PyDateTime_DATE_GET_SECOND** (PyDateTime_DateTime *o)
> 回傳秒，為正整數，從 0 到 59。

int **PyDateTime_DATE_GET_MICROSECOND** (PyDateTime_DateTime *o)
> 回傳微秒，為正整數，從 0 到 999999。

int **PyDateTime_DATE_GET_FOLD** (PyDateTime_DateTime *o)
> Return the fold, as an int from 0 through 1.

> 3.6 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，為正整數，從 0 到 23。

int **PyDateTime_TIME_GET_MINUTE** (PyDateTime_Time *o)
> 回傳分鐘，為正整數，從 0 到 59。

int **PyDateTime_TIME_GET_SECOND** (PyDateTime_Time *o)
> 回傳秒，為正整數，從 0 到 59。

int **PyDateTime_TIME_GET_MICROSECOND** (PyDateTime_Time *o)
> 回傳微秒，為正整數，從 0 到 999999。

int **PyDateTime_TIME_GET_FOLD** (PyDateTime_Time *o)
> Return the fold, as an int from 0 through 1.

> 3.6 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

Python 提供了多种内置类型用于类型注解，但只有 GenericAlias 暴露给了 C。

PyObject* **Py_GenericAlias** (*PyObject* *origin*, *PyObject* *args*)
> 创建一个 GenericAlias 对象。相当于调用 Python 类 types.GenericAlias。参数 *origin* 和 *args* 分别设
> 置 GenericAlias's `__origin__` 属性和 `__args__` 属性。*origin* 应该是 *PyTypeObject** 类型，*args*
> 可以是 *PyTupleObject** 类型或者任意 PyObject* 类型。如果传递的 *args* 不是一个元组，则自动构
> 建一个 1 元元组，并将 `__args__` 设置为 (args,)。对参数进行了最小限度的检查，因此即使 *origin*
> 不是类型，函数也会成功。GenericAlias 的 `__parameters__` 属性是通过 `__args__` 懒加载的。
> 如果失败，则触发异常并返回 NULL。

> 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**
> 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 註: 以 下 函 数 **不 应 该** 在*Py_Initialize()*: *Py_EncodeLocale()*, *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()*, *Py_GetProgramFullPath()*, *Py_GetPythonHome()*, *Py_GetProgramName()* 和*PyEval_InitThreads()* 前调用。

9.2 全局配置变量

Python 有负责控制全局配置中不同特性和选项的变量。这些标志默认被 命令行选项。

当一个选项设置一个旗标时，该旗标的值将是设置选项的次数。例如，-b 会将*Py_BytesWarningFlag* 设为 1 而 -bb 会将*Py_BytesWarningFlag* 设为 2.

int **Py_BytesWarningFlag**
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，例如可能已设置的 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 instead of the UTF-8 encoding for the filesystem encoding.
>
> Set to 1 if the PYTHONLEGACYWINDOWSFSENCODING environment variable is set to a non-empty string.
>
> 有关更多详细信息，请参阅 **PEP 529**。
>
> 可用性: 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.
>
> 有关更多详细信息，请参阅 **PEP 528**。
>
> 可用性: 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**
> Force the stdout and stderr streams to be unbuffered.
>
> 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**()
> 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⬚: On Windows, changes the console mode from `O_TEXT` to `O_BINARY`, which will also affect non-Python uses of the console using the C Runtime.

---

void **Py_InitializeEx**(int *initsigs*)
> 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**()
> 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**()
> 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**()

> 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*)

> 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**()

> 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.

wchar_t* **Py_GetPrefix**()

> 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.

wchar_t* **Py_GetExecPrefix**()

> 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.

wchar_t* **Py_GetProgramFullPath**()
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.

wchar_t* **Py_GetPath**()
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.

void **Py_SetPath** (const wchar_t *)
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**()
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**()
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**()
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**()
> 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**()
> 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*)
> 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⬚: 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*)
> 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*)
> 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.

w_char* **Py_GetPythonHome**()
> 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.

9.5 Thread State and the Global Interpreter Lock

The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the *global interpreter lock* or *GIL*, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

Therefore, the rule exists that only the thread that has acquired the *GIL* may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads (see sys.setswitchinterval()). The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.

The Python interpreter keeps some thread-specific bookkeeping information inside a data structure called *PyThreadState*. There's also one global variable pointing to the current *PyThreadState*: it can be retrieved using *PyThreadState_Get()*.

9.5.1 Releasing the GIL from extension code

Most extension code manipulating the *GIL* has the following simple structure:

```
Save the thread state in a local variable.
Release the global interpreter lock.
... Do some blocking I/O operation ...
Reacquire the global interpreter lock.
Restore the thread state from the local variable.
```

This is so common that a pair of macros exists to simplify it:

```
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

The *Py_BEGIN_ALLOW_THREADS* macro opens a new block and declares a hidden local variable; the *Py_END_ALLOW_THREADS* macro closes the block.

The block above expands to the following code:

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

Here is how these functions work: the global interpreter lock is used to protect the pointer to the current thread state. When releasing the lock and saving the thread state, the current thread state pointer must be retrieved before the lock is

released (since another thread could immediately acquire the lock and store its own thread state in the global variable). Conversely, when acquiring the lock and restoring the thread state, the lock must be acquired before storing the thread state pointer.

備⬚: 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:

PyInterpreterState
> 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.

PyThreadState
> 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**()
> Deprecated function which does nothing.
>
> In Python 3.6 and older, this function created the GIL if it didn't exist.
>
> 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.
>
> Deprecated since version 3.9, will be removed in version 3.11.

int **PyEval_ThreadsInitialized**()
> 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()*.
>
> Deprecated since version 3.9, will be removed in version 3.11.

PyThreadState* **PyEval_SaveThread**()
> 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*)
> 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⚠: 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**()

> 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*)

> 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**()

> 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⬚: 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)

> 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**()

> 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

> 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

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

This macro expands to PyEval_RestoreThread(_save);: it is equivalent to *Py_END_ALLOW_THREADS* without the closing brace.

Py_UNBLOCK_THREADS

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**()

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*)

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*)

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*)

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*)

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*)

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*)

Get the current frame of the Python thread state *tstate*.

Return a strong reference. Return NULL if no frame is currently executing.

See also *PyEval_GetFrame()*.

tstate must not be NULL.

3.9 版新加入.

uint64_t **PyThreadState_GetID** (*PyThreadState *tstate*)
> Get the unique thread state identifier of the Python thread state *tstate*.
>
> *tstate* must not be NULL.
>
> 3.9 版新加入.

*PyInterpreterState** **PyThreadState_GetInterpreter** (*PyThreadState *tstate*)
> Get the interpreter of the Python thread state *tstate*.
>
> *tstate* must not be NULL.
>
> 3.9 版新加入.

*PyInterpreterState** **PyInterpreterState_Get** (void)
> Get the current interpreter.
>
> 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*)
> 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*)
> 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.

*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: The function now takes a *tstate* parameter.

_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.* Return a dictionary in which extensions can store thread-specific state informa-tion. 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)
> 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)
> 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⊞: 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)
> 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**()
> 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⊞用: This function does not update the current thread state. Please use *PyEval_RestoreThread()* or *PyEval_AcquireThread()* instead.

> 備⊞: 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**()
> Release the global interpreter lock. The lock must have been created earlier.

> 3.2 版後已⊞用: This function does not update the current thread state. Please use *PyEval_SaveThread()* or *PyEval_ReleaseThread()* instead.

9.6 Sub-interpreter support

While in most uses, you will only embed a single Python interpreter, there are cases where you need to create several independent interpreters in the same process and perhaps even in the same thread. Sub-interpreters allow you to do that.

The "main" interpreter is the first one created when the runtime initializes. It is usually the only Python interpreter in a process. Unlike sub-interpreters, the main interpreter has unique process-global responsibilities like signal handling. It is also responsible for execution during runtime initialization and is usually the active interpreter during runtime finalization. The *PyInterpreterState_Main()* function returns a pointer to its state.

You can switch between sub-interpreters using the *PyThreadState_Swap()* function. You can create and destroy them using the following functions:

*PyThreadState** **Py_NewInterpreter**()
> 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*)
> 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*)

> 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.

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	Value being returned to the caller, or NULL if caused by an exception.
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 op-code 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 Thread Local Storage Support

The Python interpreter provides low-level support for thread-local storage (TLS) which wraps the underlying native TLS implementation to support the Python-level thread local storage API (`threading.local`). The CPython C level APIs are similar to those offered by pthreads and Windows: use a thread key and functions to associate a `void*` value per thread.

The GIL does *not* need to be held when calling these functions; they supply their own locking.

Note that `Python.h` does not include the declaration of the TLS APIs, you need to include `pythread.h` to use thread-local storage.

備⬚: 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.

也參考:

"A New C-API for Thread-Local Storage in CPython" (**PEP 539**)

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**()
> 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*)
> Free the given *key* allocated by *PyThread_tss_alloc()*, after first calling *PyThread_tss_delete()* to ensure any associated thread locals have been unassigned. This is a no-op if the *key* argument is *NULL*.

> ---
> 備⊞: A freed key becomes a dangling pointer. You should reset the key to *NULL*.
> ---

方法

The parameter *key* of these functions must not be NULL. Moreover, the behaviors of *PyThread_tss_set()* and *PyThread_tss_get()* are undefined if the given *Py_tss_t* has not been initialized by *PyThread_tss_create()*.

int **PyThread_tss_is_created**(*Py_tss_t *key*)
> 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*)
> 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*)

> 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*)

> 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*)

> 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⬚用: This API is superseded by *Thread Specific Storage (TSS) API*.

備⬚: This version of the API does not support platforms where the native TLS key is defined in a way that cannot be safely cast to int. On such platforms, *PyThread_create_key()* will return immediately with a failure status, and the other TLS functions will all be no-ops on such platforms.

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**()

void **PyThread_delete_key**(int *key*)

int **PyThread_set_key_value**(int *key*, void **value*)

void* **PyThread_get_key_value**(int *key*)

void **PyThread_delete_key_value**(int *key*)

void **PyThread_ReInitTLS**()

Python 初始化配置

3.8 版新加入.

结构

- *PyConfig*
- *PyPreConfig*
- *PyStatus*
- *PyWideStringList*

函数

- *PyConfig_Clear()*
- *PyConfig_InitIsolatedConfig()*
- *PyConfig_InitPythonConfig()*
- *PyConfig_Read()*
- *PyConfig_SetArgv()*
- *PyConfig_SetBytesArgv()*
- *PyConfig_SetBytesString()*
- *PyConfig_SetString()*
- *PyConfig_SetWideStringList()*
- *PyPreConfig_InitIsolatedConfig()*
- *PyPreConfig_InitPythonConfig()*
- *PyStatus_Error()*
- *PyStatus_Exception()*
- *PyStatus_Exit()*
- *PyStatus_IsError()*

- *PyStatus_IsExit()*

- *PyStatus_NoMemory()*

- *PyStatus_Ok()*

- *PyWideStringList_Append()*

- *PyWideStringList_Insert()*

- *Py_ExitStatusException()*

- *Py_InitializeFromConfig()*

- *Py_PreInitialize()*

- *Py_PreInitializeFromArgs()*

- *Py_PreInitializeFromBytesArgs()*

- *Py_RunMain()*

- *Py_GetArgcArgv()*

The preconfiguration (`PyPreConfig` type) is stored in `_PyRuntime.preconfig` and the configuration (`PyConfig` type) is stored in `PyInterpreterState.config`.

参见*Initialization, Finalization, and Threads*.

也參考:

PEP 587 "Python 初始化配置".

10.1 PyWideStringList

PyWideStringList
List of `wchar_t*` strings.

If *length* is non-zero, *items* must be non-`NULL` and all strings must be non-`NULL`.

方法

PyStatus **PyWideStringList_Append**(*PyWideStringList *list*, const wchar_t *item*)
Append *item* to *list*.

Python must be preinitialized to call this function.

PyStatus **PyWideStringList_Insert**(*PyWideStringList *list*, *Py_ssize_t index*, const wchar_t *item*)
Insert *item* into *list* at *index*.

If *index* is greater than or equal to *list* length, append *item* to *list*.

index must be greater than or equal to 0.

Python must be preinitialized to call this function.

Structure fields:

Py_ssize_t **length**
List 长度。

wchar_t** **items**
列表项目。

10.2 PyStatus

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**
> > 错误信息
>
> 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.
>
> *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.

备[F]: Internally, Python uses macros which set PyStatus.func, whereas functions to create a status set func to
NULL.

示例:

```
PyStatus alloc(void **ptr, size_t size)
{
    *ptr = PyMem_RawMalloc(size);
    if (*ptr == NULL) {
        return PyStatus_NoMemory();
    }
    return PyStatus_Ok();
```

(下页继续)

```
}

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;
}
```

10.3 PyPreConfig

PyPreConfig
> Structure used to preinitialize Python:

> • Set the Python memory allocator

> • Configure the LC_CTYPE locale

> • 设置为 UTF-8 模式

> 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 memory allocator:

> • 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): force usage of malloc()

> • 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

> 参见*Memory Management*.

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 to 0.

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.

int **coerce_c_locale_warn**
> If non-zero, emit a warning if the C locale is coerced.

int **dev_mode**
> 参见 *PyConfig.dev_mode*.

int **isolated**
> 参见 *PyConfig.isolated*.

int **legacy_windows_fs_encoding** (Windows *only*)
> If non-zero, disable UTF-8 Mode, set the Python filesystem encoding to `mbcs`, set the filesystem error handler to `replace`.
>
> Only available on Windows. `#ifdef MS_WINDOWS` macro can be used for Windows specific code.

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.

int **use_environment**
> 参见 *PyConfig.use_environment*.

int **utf8_mode**
> If non-zero, enable the UTF-8 mode.

10.4 Preinitialization with PyPreConfig

Functions to preinitialize Python:

PyStatus **Py_PreInitialize** (const *PyPreConfig* *preconfig*)
> Preinitialize Python from *preconfig* preconfiguration.

PyStatus **Py_PreInitializeFromBytesArgs** (const *PyPreConfig* *preconfig*, int *argc*, char * const *argv*)
> Preinitialize Python from *preconfig* preconfiguration and command line arguments (bytes strings).

PyStatus **Py_PreInitializeFromArgs** (const *PyPreConfig* *preconfig*, int *argc*, wchar_t * const * *argv*)
> Preinitialize Python from *preconfig* preconfiguration and command line arguments (wide strings).

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 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 Python preinitialization, whereas calling directly `malloc()` and `free()` is always safe. *Py_DecodeLocale()* must not be called before the preinitialization.

Example using the preinitialization to enable the 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 will speak UTF-8 */

Py_Initialize();
/* ... use Python API here ... */
Py_Finalize();
```

10.5 PyConfig

PyConfig
>Structure containing most parameters to configure Python.

>Structure methods:

>void **PyConfig_InitPythonConfig**(*PyConfig *config*)
>>Initialize configuration with *Python Configuration*.

>void **PyConfig_InitIsolatedConfig**(*PyConfig *config*)
>>Initialize configuration with *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 from wide character strings.

>>Preinitialize Python if needed.

>*PyStatus* **PyConfig_SetBytesArgv**(*PyConfig *config*, int *argc*, char * const **argv*)
>>Set command line arguments: 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.

Preinitialize Python if needed.

void **PyConfig_Clear** (*PyConfig *config*)
    Release configuration memory.

Most `PyConfig` methods preinitialize Python if needed. In that case, the Python preinitialization configuration 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:

- *dev_mode*
- *isolated*
- *parse_argv*
- *use_environment*

Moreover, if *PyConfig_SetArgv()* or *PyConfig_SetBytesArgv()* is used, this method must be called first, 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`. See *parse_argv* to parse *argv* the same way the regular Python parses Python command line arguments. If *argv* is empty, an empty string is added to ensure that `sys.argv` always exists and is never empty.

wchar_t* **base_exec_prefix**
    sys.base_exec_prefix.

wchar_t* **base_executable**
    sys._base_executable: `__PYVENV_LAUNCHER__` environment variable value, or copy of *PyConfig.executable*.

wchar_t* **base_prefix**
    sys.base_prefix.

wchar_t* **platlibdir**
    sys.platlibdir: platform library directory name, set at configure time by `--with-platlibdir`, overrideable by the `PYTHONPLATLIBDIR` environment variable.

    3.9 版新加入.

int **buffered_stdio**
    If equals to 0, enable unbuffered mode, making the stdout and stderr streams unbuffered.

    stdin is always opened in buffered mode.

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.

wchar_t* **check_hash_pycs_mode**
    Control the validation behavior of hash-based `.pyc` files (see **PEP 552**): `--check-hash-based-pycs` command line option value.

    Valid values: `always`, `never` and `default`.

    默认值为: `default`.

int **configure_c_stdio**
> If non-zero, configure C standard streams (`stdio`, `stdout`, `stdout`). For example, set their mode to `O_BINARY` on Windows.

int **dev_mode**
> If non-zero, enable the Python Development Mode.

int **dump_refs**
> If non-zero, dump all objects which are still alive at exit.
>
> `Py_TRACE_REFS` macro must be defined in build.

wchar_t* **exec_prefix**
> `sys.exec_prefix`.

wchar_t* **executable**
> `sys.executable`.

int **faulthandler**
> If non-zero, call `faulthandler.enable()` at startup.

wchar_t* **filesystem_encoding**
> Filesystem encoding, `sys.getfilesystemencoding()`.

wchar_t* **filesystem_errors**
> Filesystem encoding errors, `sys.getfilesystemencodeerrors()`.

unsigned long **hash_seed**

int **use_hash_seed**
> Randomized hash function seed.
>
> If *use_hash_seed* is zero, a seed is chosen randomly at Pythonstartup, and *hash_seed* is ignored.

wchar_t* **home**
> Python home directory.
>
> Initialized from `PYTHONHOME` environment variable value by default.

int **import_time**
> If non-zero, profile import time.

int **inspect**
> Enter interactive mode after executing a script or a command.

int **install_signal_handlers**
> Install signal handlers?

int **interactive**
> 交互模式

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.

int **legacy_windows_stdio**
> If non-zero, use `io.FileIO` instead of `io.WindowsConsoleIO` for `sys.stdin`, `sys.stdout` and `sys.stderr`.

Only available on Windows. `#ifdef MS_WINDOWS` macro can be used for Windows specific code.

int **malloc_stats**

If non-zero, dump statistics on *Python pymalloc memory allocator* at exit.

The option is ignored if Python is built using `--without-pymalloc`.

wchar_t* **pythonpath_env**

Module search paths as a string separated by `DELIM` (`os.path.pathsep`).

Initialized from `PYTHONPATH` environment variable value by default.

PyWideStringList **module_search_paths**

int **module_search_paths_set**

`sys.path`. If *module_search_paths_set* is equal to 0, the *module_search_paths* is over-ridden by the function calculating the *Path Configuration*.

int **optimization_level**

Compilation optimization level:

- 0: Peephole optimizer (and `__debug__` is set to `True`)

- 1: Remove assertions, set `__debug__` to `False`

- 2: Strip docstrings

int **parse_argv**

If non-zero, parse *argv* the same way the regular Python command line arguments, and strip Python arguments from *argv*: see Command Line Arguments.

int **parser_debug**

If non-zero, turn on parser debugging output (for expert only, depending on compilation options).

int **pathconfig_warnings**

If equal to 0, suppress warnings when calculating the *Path Configuration* (Unix only, Windows does not log any warning). Otherwise, warnings are written into `stderr`.

wchar_t* **prefix**

`sys.prefix`.

wchar_t* **program_name**

Program name. Used to initialize *executable*, and in early error messages.

wchar_t* **pycache_prefix**

`sys.pycache_prefix`: `.pyc` cache prefix.

If `NULL`, `sys.pycache_prefix` is set to `None`.

int **quiet**

Quiet mode. For example, don't display the copyright and version messages in interactive mode.

wchar_t* **run_command**

`python3 -c COMMAND` argument. Used by *Py_RunMain()*.

wchar_t* **run_filename**

`python3 FILENAME` argument. Used by *Py_RunMain()*.

wchar_t* **run_module**

`python3 -m MODULE` argument. Used by *Py_RunMain()*.

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 (`Py_REF_DEBUG` macro must be defined).

int **site_import**
> Import the `site` module at startup?

int **skip_source_first_line**
> Skip the first line of the source?

wchar_t* **stdio_encoding**

wchar_t* **stdio_errors**
> Encoding and encoding errors of `sys.stdin`, `sys.stdout` and `sys.stderr`.

int **tracemalloc**
> If non-zero, call `tracemalloc.start()` at startup.

int **use_environment**
> If greater than 0, use environment variables.

int **user_site_directory**
> If non-zero, add user site directory to `sys.path`.

int **verbose**
> If non-zero, enable verbose mode.

PyWideStringList **warnoptions**
> `sys.warnoptions`: options of the `warnings` module to build warnings filters: lowest to highest priority.
>
> 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).

int **write_bytecode**
> If non-zero, write `.pyc` files.
>
> `sys.dont_write_bytecode` is initialized to the inverted value of *write_bytecode*.

PyWideStringList **xoptions**
> `sys._xoptions`.

int **_use_peg_parser**
> Enable PEG parser? Default: 1.
>
> Set to 0 by `-X oldparser` and `PYTHONOLDPARSER`.
>
> 参见 **PEP 617**。
>
> Deprecated since version 3.9, will be removed in version 3.10.

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`: see Command Line Arguments.

The `xoptions` options are parsed to set other options: see `-X` option.

3.9 版更變: The `show_alloc_count` field has been removed.

10.6 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.

Example setting the program name:

```c
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name. Implicitly preinitialize Python. */
    status = PyConfig_SetString(&config, &config.program_name,
                                L"/path/to/my_program");
    if (PyStatus_Exception(status)) {
        goto fail;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto fail;
    }
    PyConfig_Clear(&config);
    return;

fail:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}
```

More complete example modifying the default configuration, read the configuration, and then override some parameters:

```c
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);

done:
    PyConfig_Clear(&config);
    return status;
}
```

10.7 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 *Path Configuration* ("output fields") to ignore these configuration files and avoid the function computing the default path configuration.

10.8 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 UTF-8 Mode (**PEP 540**) depending on the LC_CTYPE locale, PYTHONUTF8 and PYTHONCOERCECLOCALE environment variables.

Example of customized Python always running in isolated mode:

```c
int main(int argc, char **argv)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config.isolated = 1;

    /* Decode command line arguments.
       Implicitly preinitialize Python (in isolated mode). */
    status = PyConfig_SetBytesArgv(&config, argc, argv);
    if (PyStatus_Exception(status)) {
        goto fail;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto fail;
    }
    PyConfig_Clear(&config);

    return Py_RunMain();

fail:
    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);
}
```

10.9 路径配置

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__ environment variable
    - (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*

10.10 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()*.

10.11 Py_GetArgcArgv()

void **Py_GetArgcArgv** (int **argc*, wchar_t ****argv*)

> Get the original command line arguments, before Python modified them.

10.12 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:

```c
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);
    }
}
```

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：

```c
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyBytes_FromString(buf);
free(buf); /* malloc'ed */
return res;
```

在这个例子中，I/O 缓冲区的内存请求是由 C 库分配器处理的。Python 内存管理器只参与了分配作为结果返回的字节对象。

然而，在大多数情况下，建议专门从 Python 堆中分配内存，因为后者由 Python 内存管理器控制。例如，当解释器扩展了用 C 写的新对象类型时，就必须这样做。使用 Python 堆的另一个原因是希望 * 通知 * Python 内存管理器关于扩展模块的内存需求。即使所请求的内存全部只用于内部的、高度特定的目的，将所有的内存请求交给 Python 内存管理器能让解释器对其内存占用的整体情况有更准确的了解。因此，在某些情况下，Python 内存管理器可能会触发或不触发适当的操作，如垃圾回收、内存压缩或其他预防性操作。请注意，通过使用前面例子中所示的 C 库分配器，为 I/O 缓冲区分配的内存会完全不受 Python 内存管理器管理。

也參考:

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 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*)

分配 *n* 个字节并返回一个指向分配的内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyMem_RawMalloc(1)` 一样。但是内存不会以任何方式被初始化。

void* **PyMem_RawCalloc**(size_t *nelem*, size_t *elsize*)

分配 *nelem* 个元素，每个元素的大小为 *elsize* 字节，并返回指向分配的内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。内存会被初始化为零。

请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyMem_RawCalloc(1, 1)` 一样。

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。

11.3 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*)
> 分配 *n* 个字节并返回一个指向分配的内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。
>
> 请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyMem_Malloc(1)` 一样。但是内存不会以任何方式被初始化。

void* **PyMem_Calloc** (size_t *nelem*, size_t *elsize*)
> 分配 *nelem* 个元素，每个元素的大小为 *elsize* 字节，并返回指向分配的内存的 `void*` 类型指针，如果请求失败则返回 `NULL`。内存会被初始化为零。
>
> 请求零字节可能返回一个独特的非 `NULL` 指针，就像调用了 `PyMem_Calloc(1, 1)` 一样。
>
> 3.5 版新加入.

void* **PyMem_Realloc** (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_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*)
> 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*)
> 与*PyMem_Malloc()* 相同，但会分配 (n * sizeof(TYPE)) 字节的内存。返回一个转换为 `TYPE*` 的指针。内存将不会以任何方式被初始化。

TYPE* **PyMem_Resize** (void **p*, TYPE, size_t *n*)
> 与*PyMem_Realloc()* 相同，但内存块的大小被调整为 (n * sizeof(TYPE)) 字节。返回一个转换为 `TYPE*` 类型的指针。返回时，*p* 将为指向新内存区域的指针，如果失败则返回 `NULL`。
>
> 这是一个 C 预处理宏，*p* 总是被重新赋值。请保存 *p* 的原始值，以避免在处理错误时丢失内存。

void **PyMem_Del** (void **p*)
> 与*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)`

11.4 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。

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*)

分配 *n* 个字节并返回一个指向分配的内存的 `void*` 类型指针，如果请求失败则返回 NULL。

请求零字节可能返回一个独特的非 NULL 指针，就像调用了 `PyObject_Malloc(1)` 一样。但是内存不会以任何方式被初始化。

void* **PyObject_Calloc**(size_t *nelem*, size_t *elsize*)

分配 *nelem* 个元素，每个元素的大小为 *elsize* 字节，并返回指向分配的内存的 `void*` 类型指针，如果请求失败则返回 NULL。内存会被初始化为零。

请求零字节可能返回一个独特的非 NULL 指针，就像调用了 `PyObject_Calloc(1, 1)` 一样。

3.5 版新加入.

void* **PyObject_Realloc**(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 `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*)

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。

11.5 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: 环境变量 `PYTHONMALLOC` 的值
- malloc: 来自 C 标准库的系统分配, C 函数 `malloc()`, `calloc()`, `realloc()` and `free()`
- pymalloc: *pymalloc 内存分配器*
- "+ debug": 带有 `PyMem_SetupDebugHooks()` 安装的调试钩子

11.6 自定义内存分配器

3.4 版新加入.

PyMemAllocatorEx
Structure used to describe a memory block allocator. The structure has the following fields:

域	含义
`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.

PyMemAllocatorDomain
用来识别分配器域的枚举类。域有:

PYMEM_DOMAIN_RAW
函数

- `PyMem_RawMalloc()`
- `PyMem_RawRealloc()`
- `PyMem_RawCalloc()`

- *PyMem_RawFree()*

PYMEM_DOMAIN_MEM
　函数

- *PyMem_Malloc()*,

- *PyMem_Realloc()*

- *PyMem_Calloc()*

- *PyMem_Free()*

PYMEM_DOMAIN_OBJ
　函数

- *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。

void **PyMem_SetupDebugHooks** (void)
　设置检测 Python 内存分配器函数中错误的钩子。

　新分配的内存由字节 0xCD（CLEANBYTE）填充，释放的内存由字节 0xDD（DEADBYTE）填充。内存块被"禁止字节"包围（FORBIDDENBYTE：字节 0xFD）。

　运行时检查：

- 检测对 API 的违反，例如：对用 *PyMem_Malloc()* 分配的缓冲区调用 *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* 已被保持。

　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。

　如果 Python 是在调试模式下编译的，这些钩子是 *installed by default* 。环境变量 PYTHONMALLOC 可以用来在发布模式编译的 Python 上安装调试钩子。

　3.6 版更變: 这个函数现在也适用于以发布模式编译的 Python。在出错时，调试钩子现在使用 tracemalloc 来回溯内存块被分配的位置。调试钩子现在也检查当 *PYMEM_DOMAIN_OBJ* 和 *PYMEM_DOMAIN_MEM* 域的函数被调用时，全局解释器锁是否被持有。

　3.8 版更變: 字节模式 0xCB（CLEANBYTE）、0xDB（DEADBYTE）和 0xFB（FORBIDDENBYTE）已被 0xCD、0xDD 和 0xFD 替代以使用与 Windows CRT 调试 malloc() 和 free() 相同的值。

11.7 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() 。

11.7.1 自定义 pymalloc Arena 分配器

3.4 版新加入.

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	含义
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

11.8 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。

11.9 示例

以下是来自總覽 小节的示例，经过重写以使 I/O 缓冲区是通过使用第一个函数集从 Python 堆中分配的:

```
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

使用面向类型函数集的相同代码:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

请注意在以上两个示例中，缓冲区总是通过归属于相同集的函数来操纵的。事实上，对于一个给定的内存块必须使用相同的内存 API 族，以便使得混合不同分配器的风险减至最低。以下代码序列包含两处错误，其中一个被标记为 *fatal* 因为它混合了两种在不同堆上操作的不同分配器。

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);    /* Wrong -- should be PyMem_Free() */
free(buf2);         /* Right -- allocated via malloc() */
free(buf1);         /* Fatal -- should be PyMem_Del()  */
```

除了旨在处理来自 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。

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 在堆中分配对象

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. 为新分配的对象 *op* 初始化它的类型和引用。返回初始化后的对象。如果 *type* 声明这个对象参与循环垃圾检测，那么这个对象会被添加进垃圾检测的对象集中。这个对象的其他字段不会被影响。

PyVarObject* **PyObject_InitVar** (*PyVarObject *op*, *PyTypeObject *type*, *Py_ssize_t size*)
    Return value: Borrowed reference. 它的功能和 *PyObject_Init()* 一样，并且初始化变量大小的对象的长度。

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**
> 像 None 一样的 Python 对象。这个对象仅可以使用 *Py_None* 宏访问，这个宏取得指向这个对象的指针。

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。

PyObject
> 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*.

PyVarObject
> This is an extension of *PyObject* that adds the ob_size field. This is only used for objects that have some notion of *length*. This type does not often appear in the Python/C API. Access to the members must be done by using the macros *Py_REFCNT*, *Py_TYPE*, and *Py_SIZE*.

PyObject_HEAD
> This is a macro used when declaring new types which represent objects without a varying length. The PyObject_HEAD macro expands to:

```
PyObject ob_base;
```

> See documentation of *PyObject* above.

PyObject_VAR_HEAD
> This is a macro used when declaring new types which represent objects with a length that varies from instance to instance. The PyObject_VAR_HEAD macro expands to:

```
PyVarObject ob_base;
```

> See documentation of *PyVarObject* above.

Py_TYPE(o)
> This macro is used to access the ob_type member of a Python object. It expands to:

```
(((PyObject*)(o))->ob_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_REFCNT (o)
> This macro is used to access the `ob_refcnt` member of a Python object. It expands to:

```
(((PyObject*)(o))->ob_refcnt)
```

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_SIZE (o)
> This macro is used to access the `ob_size` member of a Python object. It expands to:

```
(((PyVarObject*)(o))->ob_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

PyCFunction
> 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);
```

PyCFunctionWithKeywords
> 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);
```

_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);
```

_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);
```

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.

PyMethodDef

Structure used to describe a method of an extension type. This structure has four fields:

域	C Type	意义
ml_name	const char *	name of the method
ml_meth	PyCFunction	pointer to the C implementation
ml_flags	int	flag bits indicating how the call should be constructed
ml_doc	const char *	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).

This is not part of the *limited API*.

3.7 版新加入.

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 *vector-call 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.

This is not part of the *limited API*.

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

PyMemberDef

Structure which describes an attribute of a type which corresponds to a C struct member. Its fields are:

域	C Type	意义
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 数据类型
T_SHORT	short
T_INT	int
T_LONG	长整型
T_FLOAT	float
T_DOUBLE	double
T_STRING	const char *
T_OBJECT	PyObject *
T_OBJECT_EX	PyObject *
T_CHAR	字符
T_BYTE	字符
T_UBYTE	unsigned char
T_UINT	unsigned int
T_USHORT	unsigned short
T_ULONG	unsigned long
T_BOOL	字符
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.

PyGetSetDef

> Structure to define property-like access for a type. See also description of the *PyTypeObject.tp_getset* slot.

域	C Type	意义
name	const char *	attribute name
get	getter	C function to get the attribute
set	setter	optional C function to set or delete the attribute, if omitted the attribute is readonly
doc	const char *	optional docstring
closure	void *	optional 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 Type 对象

也许 Python 对象系统中最重要的结构之一就是定义新类型的结构: *PyTypeObject* 结构。类型对象可以使用任何 PyObject_*() 或 PyType_*() 函数来处理，但不能提供大多数 Python 应用程序所感兴趣的内容。这些对象是对象行为的基础，所以它们对解释器本身或任何实现新类型的扩展模块都非常重要。

与大多数标准类型相比，类型对象相当大。这么大的原因是每个类型对象存储了大量的值，大部分是 C 函数指针，每个指针实现了类型功能的一小部分。本节将详细描述类型对象的字段。这些字段将按照它们在结构中出现的顺序进行描述。

除了下面的快速参考，*例子* 小节提供了快速了解*PyTypeObject* 的含义和用法的例子。

12.3.1 快速参考

"tp 槽"

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）				%
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	无符号长整型		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* *					

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_subclasses*]	*PyObject* *	__subclasses__				
[*tp_weaklist*]	*PyObject* *					
(*tp_del*)	*destructor*					
[*tp_version_tag*]	无符号整型					
tp_finalize	*destructor*	__del__				X
tp_vectorcall	*vectorcallfunc*					

子方法槽（方法域）

槽	*Type*	特殊方法
am_await	*unaryfunc*	__await__
am_aiter	*unaryfunc*	__aiter__
am_anext	*unaryfunc*	__anext__
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__

繼續下一頁

---

[1] 小括号中的槽名表示它 (实际上) 已弃用。尖括号中的名称应该被视为只读的。方括号中的名称仅供内部使用。"<R>"(作为前缀) 表示该字段是必需的 (必须是非 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)

```
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。

<div align="center">表 2 – 繼續上一頁</div>

槽	*Type*	特殊方法
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__
nb_inplace_true_divide	*binaryfunc*	__itruediv__
nb_index	*unaryfunc*	__index__
nb_matrix_multiply	*binaryfunc*	__matmul__ __rmatmul__
nb_inplace_matrix_multiply	*binaryfunc*	__imatmul__
mp_length	*lenfunc*	__len__
mp_subscript	*binaryfunc*	__getitem__
mp_ass_subscript	*objobjargproc*	__setitem__, __delitem__
sq_length	*lenfunc*	__len__
sq_concat	*binaryfunc*	__add__
sq_repeat	*ssizeargfunc*	__mul__
sq_item	*ssizeargfunc*	__getitem__
sq_ass_item	*ssizeobjargproc*	__setitem__ __delitem__
sq_contains	*objobjproc*	__contains__
sq_inplace_concat	*binaryfunc*	__iadd__
sq_inplace_repeat	*ssizeargfunc*	__imul__
bf_getbuffer	*getbufferproc()*	
bf_releasebuffer	*releasebufferproc()*	

slot typedefs

typedef	参数类型	返回类型
allocfunc	*PyTypeObject* * *Py_ssize_t*	*PyObject* *
destructor	void *	void
freefunc	void *	void
traverseproc	void * *visitproc* void *	整型
newfunc	*PyObject* * *PyObject* * *PyObject* *	*PyObject* *
initproc	*PyObject* * *PyObject* * *PyObject* *	整型
reprfunc	*PyObject* *	*PyObject* *
getattrfunc	*PyObject* * const char *	*PyObject* *
setattrfunc	*PyObject* * const char * *PyObject* *	整型
getattrofunc	*PyObject* * *PyObject* *	*PyObject* *
setattrofunc	*PyObject* * *PyObject* * *PyObject* *	整型
descrgetfunc	*PyObject* * *PyObject* * *PyObject* *	*PyObject* *
descrsetfunc	*PyObject* * *PyObject* *	整型

请参阅*Slot Type typedefs* 里有更多详细信息。

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:

```c
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 */

    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;

    /* Iterators */
    getiterfunc tp_iter;
```

```
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    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;

} PyTypeObject;
```

12.3.3 PyObject Slots

The type object structure extends the *PyVarObject* structure. The `ob_size` field is used for dynamic types (created by `type_new()`, usually called from a class statement). Note that *PyType_Type* (the metatype) initializes *tp_itemsize*, which means that its instances (i.e. type objects) *must* have the `ob_size` field.

PyObject* **PyObject._ob_next**
PyObject* **PyObject._ob_prev**
> These fields are only present when the macro `Py_TRACE_REFS` is defined. 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 use is to print the objects that are still alive at the end of a run when the environment variable `PYTHONDUMPREFS` is set.
>
> 继承:
>
> These fields are not inherited by subtypes.

Py_ssize_t **PyObject.ob_refcnt**
> 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 (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**

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:

This field is inherited by subtypes.

12.3.4 PyVarObject Slots

Py_ssize_t **PyVarObject.ob_size**

For statically allocated type objects, this should be initialized to zero. For dynamically allocated type objects, this field has a special internal meaning.

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:

This field is inherited by subtypes.

Py_ssize_t **PyTypeObject.tp_vectorcall_offset**

An optional offset to a per-instance function that implements calling the object using the *vectorcall protocol*, a more efficient alternative of the simpler *tp_call*.

This field is only used if the flag *Py_TPFLAGS_HAVE_VECTORCALL* is set. If so, this must be a positive integer containing the offset in the instance of a *vectorcallfunc* pointer.

The *vectorcallfunc* pointer may be NULL, in which case the instance behaves as if *Py_TPFLAGS_HAVE_VECTORCALL* was not set: calling the instance falls back to *tp_call*.

Any class that sets Py_TPFLAGS_HAVE_VECTORCALL must also set *tp_call* and make sure its behaviour is consistent with the *vectorcallfunc* function. This can be done by setting *tp_call* to *PyVectorcall_Call()*.

> **警告:** It is not recommended for *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.

備⟨F⟩: The semantics of the tp_vectorcall_offset slot are provisional and expected to be finalized in Python 3.9. If you use vectorcall, plan for updating your code for Python 3.9.

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:

Group: tp_getattr, tp_getattro

This field is inherited by subtypes together with *tp_getattro*: a subtype inherits both *tp_getattr* and *tp_getattro* from its base type when the subtype's *tp_getattr* and *tp_getattro* are both NULL.

setattrfunc **PyTypeObject.tp_setattr**
> An optional pointer to the function for setting and deleting attributes.
>
> This field is deprecated. When it is defined, it should point to a function that acts the same as the *tp_setattro* function, but taking a C string instead of a Python string object to give the attribute name.
>
> 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:
>
> This field is inherited by subtypes.
>
> 默认:
>
> 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:

This field is inherited by subtypes.

reprfunc **PyTypeObject.tp_str**

An optional pointer to a function that implements the built-in operation str(). (Note that str is a type now, and str() calls the constructor for that type. This constructor calls *PyObject_Str()* to do the actual work, and *PyObject_Str()* will call this handler.)

The signature is the same as for *PyObject_Str()*:

```
PyObject *tp_str(PyObject *self);
```

The function must return a string or a Unicode object. It should be a "friendly" string representation of the object, as this is the representation that will be used, among other things, by the print() function.

继承:

This field is inherited by subtypes.

默认:

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：

    Group: tp_getattr, tp_getattro

    This field is inherited by subtypes together with *tp_getattr*: a subtype inherits both *tp_getattr* and *tp_getattro* from its base type when the subtype's *tp_getattr* and *tp_getattro* are both NULL.

    默认：

    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.

    默认：

    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.

默认：

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, Py_TPFLAGS_HAVE_VERSION_TAG.

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⬚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.

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.

> Heap-allocated types (*Py_TPFLAGS_HEAPTYPE*, such as those created with *PyType_FromSpec()* and sim-
> ilar APIs) 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()*:

常数	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`.

默认:

`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:**
>
> This field is inherited by subtypes.

iternextfunc **PyTypeObject.tp_iternext**
> An optional pointer to a function that returns the next item in an iterator. The signature is:

```
PyObject *tp_iternext(PyObject *self);
```

> When the iterator is exhausted, it must return NULL; a StopIteration exception may or may not be set. When another error occurs, it must return NULL too. Its presence signals that the instances of this type are iterators.
>
> Iterator types should also define the *tp_iter* function, and that function should return the iterator instance itself (not a new iterator instance).
>
> This function has the same signature as *PyIter_Next()*.
>
> **继承:**
>
> This field is inherited by subtypes.

struct *PyMethodDef* * **PyTypeObject.tp_methods**
> An optional pointer to a static NULL-terminated array of *PyMethodDef* structures, declaring regular methods of this type.
>
> For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a method descriptor.
>
> **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⊞: 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).

默认:

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).

默认:

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:

This field is inherited by subtypes.

descrsetfunc **PyTypeObject.tp_descr_set**
> An optional pointer to a function for setting and deleting a descriptor's value.
>
> The function signature is:

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

> The *value* argument is set to NULL to delete the value.
>
> **继承：**
>
> This field is inherited by subtypes.

Py_ssize_t **PyTypeObject.tp_dictoffset**
> If the instances of this type have a dictionary containing instance variables, this field is non-zero and contains the offset in the instances of the type of the instance variable dictionary; this offset is used by *PyObject_GenericGetAttr()*.
>
> Do not confuse this field with *tp_dict*; that is the dictionary for attributes of the type object itself.
>
> If the value of this field is greater than zero, it specifies the offset from the start of the instance structure. If the value is less than zero, it specifies the offset from the *end* of the instance structure. A negative offset is more expensive to use, and should only be used when the instance structure contains a variable-length part. This is used for example to add an instance variable dictionary to subtypes of str or tuple. Note that the *tp_basicsize* field should account for the dictionary added to the end in that case, even though the dictionary is not included in the basic object layout. On a system with a pointer size of 4 bytes, *tp_dictoffset* should be set to -4 to indicate that the dictionary is at the very end of the structure.
>
> The 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.)
>
> **默认：**
>
> 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：

This field is inherited by subtypes.

默认：

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).

默认：

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*.

继承：

This field is inherited by subtypes, except it is not inherited by static types whose *tp_base* is NULL or &PyBaseObject_Type.

默认:

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)

默认:

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:

This field is inherited by subtypes.

默认:

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 is set for types created by a class statement. It should be NULL for statically defined types.

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.

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. */
    PyErr_Fetch(&error_type, &error_value, &error_traceback);

    /* ... */

    /* Restore the saved exception. */
    PyErr_Restore(error_type, error_value, error_traceback);
}
```

> For this field to be taken into account (even through inheritance), you must also set the *Py_TPFLAGS_HAVE_FINALIZE* flags bit.

> 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：

This field is inherited by subtypes.

3.4 版新加入.

也参考：

"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 Heap 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 not part of the *stable ABI*, any extension modules using static types must be compiled for a specific Python minor version.

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.

This is done by filling a `PyType_Spec` structure and calling `PyType_FromSpecWithBases()`.

12.4 Number Object Structures

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;
       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

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

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

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.

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;
} PyAsyncMethods;
```

unaryfunc **PyAsyncMethods.am_await**
    The signature of this function is:

```
PyObject *am_await(PyObject *self);
```

The returned object must be an iterator, i.e. *PyIter_Check()* must return 1 for it.

This slot may be set to NULL if an object is not an *awaitable*.

unaryfunc **PyAsyncMethods.am_aiter**
    The signature of this function is:

```
PyObject *am_aiter(PyObject *self);
```

Must return an *asynchronous iterator* object. See __anext__() for details.

This slot may be set to NULL if an object does not implement asynchronous iteration protocol.

unaryfunc **PyAsyncMethods.am_anext**
    The signature of this function is:

```
PyObject *am_anext(PyObject *self);
```

Must return an *awaitable* object. See __anext__() for details. This slot may be set to NULL.

12.9 Slot Type typedefs

PyObject * **(*allocfunc)** (*PyTypeObject *cls*, *Py_ssize_t nitems*)

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*.

void **(*destructor)** (*PyObject* *)

void **(*freefunc)** (void *)

See *tp_free*.

PyObject * **(*newfunc)** (*PyObject* *, *PyObject* *, *PyObject* *)

See *tp_new*.

int **(*initproc)** (*PyObject* *, *PyObject* *, *PyObject* *)

See *tp_init*.

PyObject * **(*reprfunc)** (*PyObject* *)

See *tp_repr*.

PyObject * **(*getattrfunc)** (*PyObject *self*, char *attr*)

Return the value of the named attribute for the object.

int **(*setattrfunc)** (*PyObject *self*, char *attr*, *PyObject *value*)

Set the value of the named attribute for the object. The value argument is set to NULL to delete the attribute.

PyObject * **(*getattrofunc)** (*PyObject *self*, *PyObject *attr*)

Return the value of the named attribute for the object.

See *tp_getattro*.

int **(*setattrofunc)** (*PyObject *self*, *PyObject *attr*, *PyObject *value*)

Set the value of the named attribute for the object. The value argument is set to NULL to delete the attribute.

See *tp_setattro*.

PyObject * **(*descrgetfunc)** (*PyObject* *, *PyObject* *, *PyObject* *)

See tp_descrget.

int **(*descrsetfunc)** (*PyObject* *, *PyObject* *, *PyObject* *)

See tp_descrset.

Py_hash_t **(*hashfunc)** (*PyObject* *)

See *tp_hash*.

PyObject * **(*richcmpfunc)** (*PyObject* *, *PyObject* *, int)

See *tp_richcompare*.

PyObject * **(*getiterfunc)** (*PyObject* *)

See *tp_iter*.

PyObject * **(*iternextfunc)** (*PyObject* *)

See *tp_iternext*.

Py_ssize_t **(*lenfunc)** (*PyObject* *)

int **(*getbufferproc)** (*PyObject* *, *Py_buffer* *, int)

void **(*releasebufferproc)** (*PyObject* *, *Py_buffer* *)

PyObject ***(*unaryfunc)** (*PyObject* *)

PyObject ***(*binaryfunc)** (*PyObject* *, *PyObject* *)

PyObject ***(*ternaryfunc)** (*PyObject* *, *PyObject* *, *PyObject* *)

PyObject ***(*ssizeargfunc)** (*PyObject* *, *Py_ssize_t*)

int **(*ssizeobjargproc)** (*PyObject* *, *Py_ssize_t*)

int **(*objobjproc)** (*PyObject* *, *PyObject* *)

int **(*objobjargproc)** (*PyObject* *, *PyObject* *, *PyObject* *)

12.10 例子

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:

```c
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:

```c
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 */
```

<div align="right">（下页继续）</div>

```
    0,                                  /* tp_str */
    0,                                  /* tp_getattro */
    0,                                  /* tp_setattro */
    0,                                  /* tp_as_buffer */
    0,                                  /* tp_flags */
    PyDoc_STR("My objects"),            /* tp_doc */
    0,                                  /* tp_traverse */
    0,                                  /* tp_clear */
    0,                                  /* tp_richcompare */
    0,                                  /* tp_weaklistoffset */
    0,                                  /* tp_iter */
    0,                                  /* tp_iternext */
    0,                                  /* tp_methods */
    0,                                  /* tp_members */
    0,                                  /* tp_getset */
    0,                                  /* tp_base */
    0,                                  /* tp_dict */
    0,                                  /* tp_descr_get */
    0,                                  /* tp_descr_set */
    0,                                  /* tp_dictoffset */
    0,                                  /* tp_init */
    0,                                  /* tp_alloc */
    myobj_new,                          /* tp_new */
};
```

A type that supports weakrefs, instance dicts, and hashing:

```
typedef struct {
    PyObject_HEAD
    const char *data;
    PyObject *inst_dict;
    PyObject *weakreflist;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_weaklistoffset = offsetof(MyObject, weakreflist),
    .tp_dictoffset = offsetof(MyObject, inst_dict),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
    .tp_new = myobj_new,
    .tp_traverse = (traverseproc)myobj_traverse,
    .tp_clear = (inquiry)myobj_clear,
    .tp_alloc = PyType_GenericNew,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
    .tp_hash = (hashfunc)myobj_hash,
    .tp_richcompare = PyBaseObject_Type.tp_richcompare,
};
```

A str subclass that cannot be subclassed and cannot be called to create instances (e.g. uses a separate factory func):

```
typedef struct {
    PyUnicodeObject raw;
```

```c
    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,
    .tp_new = NULL,
    .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*)
> 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*)
> 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*)
> 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*)
> 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*)
> 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。

int **(*visitproc)** (*PyObject *object*, void *arg*)
> 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：

int **(*traverseproc)** (*PyObject *self*, *visitproc visit*, void **arg*)
  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：

```c
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。

int **(*inquiry)** (*PyObject *self*)
  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。

API 和 ABI 版本管理

PY_VERSION_HEX 是 Python 的版本号的整数形式。

例如，如果 PY_VERSION_HEX 被置为 0x030401a2,其包含的版本信息可以通过以下方式将其作为一个 32 位数字来处理:

字节	位数（大端字节序）	意义
1	1-8	PY_MAJOR_VERSION（3.4.1a2 中的 3）
2	9-16	PY_MINOR_VERSION（3.4.1a2 中的 4）
3	17-24	PY_MICRO_VERSION（3.4.1a2 中的 1）
4	25-28	PY_RELEASE_LEVEL (0xA 是 alpha 版本, 0xB 是 beta 版本, 0xC 发布的候选版本并且 0xF 是最终版本)，在这个例子中这个版本是 alpha 版本。
	29-32	PY_RELEASE_SERIAL（3.4.1a2 中的 2，最终版本用 0)

因此 3.4.1a2 的 16 进制版本号是 0x030401a2 。

所有提到的宏都定义在 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）内部，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。
- 内建常數 Ellipsis。

2to3 一個試著將 Python 2.x 程式碼轉換成 Python 3.x 程式碼的工具，它是透過處理大部分的不相容性來達成此目的，而這些不相容性能夠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。請參閱 2to3-reference。

abstract base class（抽象基底類別） 抽象基底類別（又稱為 ABC）提供了一種定義介面的方法，作為 *duck-typing*（鴨子型別）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 引用虛擬的 subclass（子類別），它們並不繼承自另一個 class（類別），但仍可被 isinstance() 及 issubclass() 辨識；請參閱 abc 模組的說明文件。Python 有許多內建的 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（註釋） 一個與變數、class 屬性、函式的參數或回傳值相關聯的標籤。照慣例，它被用來作為 *type hint*（型別提示）。

　　在運行時 (runtime)，區域變數的註釋無法被存取，但全域變數、class 屬性和函式的註解，會分別被儲存在模組、class 和函式的 __annotations__ 特殊屬性中。

　　請參閱 *variable annotation*、*function annotation*、**PEP 484** 和 **PEP 526**，這些章節皆有此功能的說明。

argument（引數） 呼叫函式時被傳遞給 *function*（或 *method*）的值。引數有兩種：

- 關鍵字引數 *(keyword argument)*：在函式呼叫中，以識別字（identifier，例如 name=）開頭的引數，或是以 ** 後面 dictionary（字典）内的值被傳遞的引數。例如，3 和 5 都是以下 complex() 呼叫中的關鍵字引數：

231

```
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（包括區域變數及攔置中的 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。

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。

bytes-like object（類位元組串物件） 一個支援緩冲協议且能⊡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。說明文件通常會將這些物件稱為「可讀寫的類位元組串物件」。可變緩衝區的物件包括 bytearray，以及 bytearray 的 memoryview。其他的運算需要讓二進制資料被儲存在不可變物件（「唯讀的類位元組串物件」）中；這些物件包括 bytes，以及 bytes 物件的 memoryview。

bytecode（位元組碼） Python 的原始碼會被編譯成位元組碼，它是 Python 程式在 CPython 直譯器中的內部表示法。該位元組碼也會被暫存在 .pyc 檔案中，以便第二次執行同一個檔案時能更快速（可以不用從原始碼重新編譯出位元組碼）。這種「中間語言 (intermediate language)」據說是運行在一個 *virtual machine*（虛擬機器）上，該虛擬機器會執行與每個位元組碼對應的機器碼 (machine code)。要注意的是，位元組碼理論上是無法在不同的 Python 虛擬機器之間運作的，也不能在不同版本的 Python 之間保持穩定。

位元組碼的指令列表可以在 dis 模組的說明文件中找到。

callback（回呼） 作為引數被傳遞的一個副程式 (subroutine) 函式，會在未來的某個時間點被執行。

class（類別） 一個用於建立使用者定義物件的模板。Class 的定義通常會包含 method 的定義，這些 method 可以在 class 的實例上進行操作。

class variable（類別變數） 一個在 class 中被定義，且應該只能在 class 層次（意即不是在 class 的實例中）被修改的變數。

coercion（強制轉型） 在涉及兩個不同型別引數的操作過程中，將某一種型別的實例轉為另一種型別的隱式轉型 (implicit conversion) 過程。例如，int(3.15) 會將浮點數轉換為整數 3，但在 3+4.5 中，每個引數是不同的型別（一個 int，一個 float），而這兩個引數必須在被轉換為相同的型別之後才能相加，否則就會引發 TypeError。如果沒有強制轉型，即使所有的引數型別皆相容，它們都必須要由程式設計師正規化 (normalize) 為相同的值，例如，要用 float(3)+4.5 而不能只是 3+4.5。

complex number（複數） 一個我們熟悉的實數系統的擴充，在此所有數字都會被表示為一個實部和一個虛部之和。虛數就是虛數單位（-1 的平方根）的實數倍，此單位通常在數學中被寫為 i，在工程學中被寫為 j。Python 內建了對複數的支援，它是用後者的記法來表示虛數；虛部會帶著一個後綴的 j 被編寫，例如 3+1j。若要將 math 模組內的工具等效地用於複數，請使用 cmath 模組。複數的使用是一個相當進階的數學功能。如果你沒有察覺到對它們的需求，那就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。請參閱 **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，是在平行的非同步任務 (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 程式語言的標準實作 (canonical implementation)，被發布在 python.org 上。「CPython」這個術語在必要時被使用，以區分此實作與其它語言的實作，例如 Jython 或 IronPython。

decorator（裝飾器） 一個函式，它會回傳另一個函式，通常它會使用 `@wrapper` 語法，被應用為一種函式的變換 (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：

```python
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準位置。

duck-typing（鴨子型別） 一種程式設計風格，它不是藉由檢查一個物件的型別來確定它是否具有正確的介面；取而代之的是，method 或屬性會單純地被呼叫或使用。（「如果它看起來像一隻鴨子而且叫起來像一隻鴨子，那麼它一定是一隻鴨子。」）因為強調介面而非特定型別，精心設計的程式碼能讓多形替代 (polymorphic substitution) 來增進它的靈活性。鴨子型別要避免使用 `type()` 或 `isinstance()` 進行測試。（但是請注意，鴨子型別可以用*抽象基底類別 (abstract base class)* 來補充。）然而，它通常會改用 `hasattr()` 測試，或是 *EAFP* 程式設計風格。

EAFP Easier to ask for forgiveness than permission.（請求寬恕比請求許可更容易。）這種常見的 Python 編碼風格會先假設有效的鍵或屬性的存在，並在該假設被推翻時再捕獲例外。這種乾淨且快速的風格，其特色是存在許多的 `try` 和 `except` 陳述式。該技術與許多其他語言（例如 C）常見的 *LBYL* 風格形成了對比。

expression（運算式） 一段可以被評估並求值的語法。換句話說，一個運算式就是文字、名稱、屬性存取、運算子或函式呼叫等運算式元件的累積，而這些元件都能回傳一個值。與許多其他語言不同的是，並非所有的 Python 語言構造都是運算式。另外有一些 *statement*（陳述式）不能被用作運算式，例如 `while`。賦值 (assignment) 也是陳述式，而不是運算式。

extension module（擴充模組） 一個以 C 或 C++ 編寫的模組，它使用 Python 的 C API 來與核心及使用者程式碼進行互動。

f-string（f 字串）　以 `'f'` 或 `'F'` 為前綴的字串文本通常被稱為「f 字串」，它是格式化的字串文本的縮寫。另請參閱 **PEP 498**。

file object（檔案物件）　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，它能夠協調對真實磁碟檔案或是其他類型的儲存器或通訊裝置（例如標準輸入 / 輸出、記憶體的緩衝區、socket（插座）、管線 (pipe) 等）的存取。檔案物件也被稱為類檔案物件 *(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。建立檔案物件的標準方法是使用 `open()` 函式。

file-like object（類檔案物件）　*file object*（檔案物件）的同義字。

finder（尋檢器）　一個物件，它會嘗試為正在被 import 的模組尋找 *loader*（載入器）。

從 Python 3.3 開始，有兩種類型的尋檢器：*元路徑尋檢器 (meta path finder)* 會使用 `sys.meta_path`，而*路徑項目尋檢器 (path entry finder)* 會使用 `sys.path_hooks`。

請參閱 **PEP 302**、**PEP 420** 和 **PEP 451** 以了解更多細節。

floor division（向下取整除法）　向下無條件捨去到最接近整數的數學除法。向下取整除法的運算子是 `//`。例如，運算式 `11 // 4` 的計算結果為 `2`，與 float（浮點數）真除法所回傳的 `2.75` 不同。請注意，`(-11) // 4` 的結果是 `-3`，因為是 `-2.75` 被向下無條件捨去。請參閱 **PEP 238**。

function（函式）　一連串的陳述式，它能夠向呼叫者回傳一些值。它也可以被傳遞零個或多個引數，這些引數可被使用於函式本體的執行。另請參閱 *parameter*（參數）、*method*（方法），以及 function 章節。

function annotation（函式註釋）　函式參數或回傳值的一個 *annotation*（註釋）。

函式註釋通常被使用於型別提示：例如，這個函式預期會得到兩個 int 引數，並會有一個 int 回傳值：

```python
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

函式註釋的語法在 function 章節有詳細解釋。

請參閱 *variable annotation* 和 **PEP 484**，皆有此功能的描述。

__future__　future 陳述式：`from __future__ import <feature>`，會指示編譯器使用那些在 Python 未來的發布版本中將成為標準的語法或語義，來編譯當前的模組。而 `__future__` 模組則記錄了 *feature*（功能）可能的值。透過 import 此模組並對其變數求值，你可以看見一個新的功能是何時首次被新增到此語言中，以及它何時將會（或已經）成為預設的功能：

```python
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection（垃圾回收）　當記憶體不再被使用時，將其釋放的過程。Python 執行垃圾回收，是透過參照計數 (reference counting)，以及一個能夠檢測和中斷參照循環 (reference cycle) 的循環垃圾回收器 (cyclic garbage collector) 來完成。垃圾回收器可以使用 gc 模組對其進行控制。

generator（產生器）　一個會回傳*generator iterator*（產生器疊代器）的函式。它看起來像一個正常的函式，但不同的是它包含了 `yield` 運算式，能夠產生一系列的值，這些值可用於 for 迴圈，或是以 `next()` 函式，每次檢索其中的一個值。

這個術語通常用來表示一個產生器函式，但在某些情境中，也可能是表示產生器疊代器。萬一想表達的意思不夠清楚，那就使用完整的術語，以避免歧義。

generator iterator（產生器疊代器）　一個由*generator*（產生器）函式所建立的物件。

每個 `yield` 會暫停處理程序，並記住位置執行狀態（包括區域變數及攔置中的 try 陳述式）。當產生器疊代器回復時，它會從停止的地方繼續執行（與那些每次調用時都要重新開始的函式有所不同）。

generator expression（產生器運算式） 一個會回傳迭代器的運算式。它看起來像一個正常的運算式，後面接著一個 `for` 子句，該子句定義了迴圈變數、範圍以及一個選擇性的 `if` 子句。該組合運算式會為外層函數產生多個值：

```
>>> 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別實作相同的運算。呼叫期間應該使用哪種實作，是由調度演算法 (dispatch algorithm) 來決定。

另請參閱 *single dispatch*（單一調度）術語表條目、`functools.singledispatch()` 裝飾器和 **PEP 443**。

generic type（泛型型別） A *type* that can be parameterized; typically a container class such as `list` or `dict`. Used for *type hints* and *annotations*.

For more details, see generic alias types, **PEP 483**, **PEP 484**, **PEP 585**, and the `typing` module.

GIL 請參閱 *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（包括關鍵的內建型別，如 `dict`）自動地避免並行存取 (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為多執行緒 (multi-threaded)，但代價是會犧牲掉多處理器的機器能所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湊等計算密集 (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）的努力並未成功，因為在一般的單一處理器情況下，效能會有所損失。一般認為，若要克服這個效能問題，會使實作變得複雜許多，進而付出更高的維護成本。

hash-based pyc（雜湊架構的 pyc） 一個位元組碼 (bytecode) 暫存檔，它使用雜湊值而不是對應原始檔案的最後修改時間，來確定其有效性。請參閱 pyc-invalidation。

hashable（可雜湊的） 如果一個物件有一個雜湊值，該值在其生命週期中永不改變（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），那麼它就是一個可雜湊物件。比較結果為相等的多個可雜湊物件，它們必須擁有相同的雜湊值。

可雜湊性 (hashability) 使一個物件可用作 dictionary（字典）的鍵和 set（集合）的成員，因為這些資料結構都在其內部使用了雜湊值。

大多數的 Python 不可變內建物件都是可雜湊的；可變的容器（例如 list 或 dictionary）則不是；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湊的，它們本身才是可雜湊的。若物件是使用者自定 class 的實例，則這些物件會被預設為可雜湊的。它們在互相比較時都是不相等的（除非它們與自己比較），而它們的雜湊值則是衍生自它們的 `id()`。

IDLE Python 的 Integrated Development Environment（整合開發環境）。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。它們在需要固定雜湊值的地方，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） 一個能夠尋找及載入模組的物件；它既是 *finder*（尋檢器）也是 *loader*（載入器）物件。

interactive（互動的） Python 有一個互動式直譯器，這表示你可以在直譯器的提示字元輸入陳述式和運算式，立即執行它們並且看到它們的結果。只要啟動 python，不需要任何引數（可能藉由從你的電腦的主選單選擇它）。這是測試新想法或檢查模塊和包的非常強大的方法（請記住 help(x)）。

interpreted（直譯的） Python 是一種直譯語言，而不是編譯語言，不過這個區分可能有些模糊，因為有位元組碼 (bytecode) 編譯器的存在。這表示原始檔案可以直接被運行，而不需明確地建立另一個執行檔，然後再執行它。直譯語言通常比編譯語言有更短的開發／除錯週期，不過它們的程式通常也運行得較慢。另請參閱 *interactive*（互動的）。

interpreter shutdown（直譯器關閉） 當 Python 直譯器被要求關閉時，它會進入一個特殊階段，在此它逐漸釋放所有被配置的資源，例如模組和各種關鍵內部結構。它也會多次呼叫垃圾回收器 *(garbage collector)*。這能夠觸發使用者自定的解構函式 (destructor) 或弱引用的回呼 (weakref callback)，並執行其中的程式碼。在關閉階段被執行的程式碼會遇到各種例外，因為它所依賴的資源可能不再有作用了（常見的例子是函式庫模組或是警告機制）。

直譯器關閉的主要原因，是 __main__ 模組或正被運行的腳本已經執行完成。

iterable（可迭代物件） 一種能夠一次回傳一個其中成員的物件。可迭代物件的例子包括所有的序列型別（像是 list、str 和 tuple）和某些非序列型別，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，該物件就是可迭代物件。

可迭代物件可用於 for 迴圈和許多其他需要一個序列的地方 (zip()、map()...)。當一個可迭代物件作為引數被傳遞給內建函式 iter() 時，它會將該物件回傳一個迭代器。此迭代器適用於針對一組值進行一遍 (one pass) 運算。使用迭代器時，通常不一定要呼叫 iter() 或自行處理迭代器物件。for 陳述式會自動地為你處理這些事，它會建立一個暫時性的未命名變數，用於在迴圈期間保有該迭代器。另請參閱 *iterator*（迭代器）、*sequence*（序列）和 *generator*（產生器）。

iterator（迭代器） 一個表示資料流的物件。重複地呼叫迭代器的 __next__() method（或是將它傳遞給內建函式 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，該迭代器物件已被用盡，而任何對其 __next__() method 的進一步呼叫，都只會再次引發 StopIteration。迭代器必須有一個 __iter__() method，它會回傳迭代器物件本身，所以每個迭代器也都是可迭代物件，且可以用於大多數適用其他可迭代物件的場合。一個明顯的例外，是嘗試多遍迭代 (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 迴圈中使用它時，都會產生一個全新的迭代器。使用迭代器嘗試此事（多遍迭代）時，只會回傳在前一遍迭代中被用過的、同一個已被用盡的迭代器物件，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。

key function（鍵函式） 鍵函式或理序函式 (collation function) 是一個可呼叫 (callable) 函式，它會回傳一個用於排序 (sorting) 或定序 (ordering) 的值。例如，locale.strxfrm() 被用來產生一個了解區域特定排序慣例的排序鍵。

Python 中的許多工具，都接受以鍵函式來控制元素被定序或分組的方式。它們包括 min()、max()、sorted()、list.sort()、heapq.merge()、heapq.nsmallest()、heapq.nlargest() 和 itertools.groupby()。

有幾種方法可以建立一個鍵函式。例如，str.lower() method 可以作為不分大小寫排序的鍵函式。或者，一個鍵函式也可以從 lambda 運算式被建造，例如 lambda r: (r[0], r[2])。另外，operator 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，請參閱如何排序。

keyword argument（關鍵字引數） 請參閱 *argument*（引數）。

lambda 由單一 *expression*（運算式）所組成的一個匿名行內函式 (inline function)，於該函式被呼叫時求值。建立 lambda 函式的語法是 lambda [parameters]: expression。

LBYL Look before you leap.（三思而後行。）這種編碼風格會在進行呼叫或查找之前，明確地測試先決條件。這種風格與 *EAFP* 方式形成對比，且它的特色是會有許多 if 陳述式的存在。

在一個多執行緒環境中，LBYL 方式有在「三思」和「後行」之間引入了競爭條件 (race condition) 的風險。例如以下程式碼 `if key in mapping: return mapping[key]`，如果另一個執行緒在測試之後但在查找之前，從 *mapping* 中移除了 *key*，則該程式碼就會失效。這個問題可以用鎖 (lock) 或使用 EAFP 編碼方式來解決。

list（串列） 一個 Python 內建的 *sequence*（序列）。儘管它的名字是 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)，因為存取元素的時間複雜度是 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，並將處理結果以一個 list 回傳的簡要方法。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` 會產生一個字串 list，其中包含 0 到 255 範圍內，所有偶數的十六進位數 (0x..)。`if` 子句是選擇性的。如果省略它，則 `range(256)` 中的所有元素都會被處理。

loader（載入器） 一個能夠載入模組的物件。它必須定義一個名為 `load_module()` 的 method（方法）。載入器通常是被 *finder*（尋檢器）回傳。更多細節請參閱 **PEP 302**，關於 *abstract base class*（抽象基底類別），請參閱 `importlib.abc.Loader`。

magic method（魔術方法） *special method*（特殊方法）的一個非正式同義詞。

mapping（對映） 一個容器物件，它支援任意鍵的查找，且能實作 abstract base classes（抽象基底類別）中，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，請參閱 `importlib.abc.MetaPathFinder`。

metaclass（元類別） 一種 class 的 class。Class 定義過程會建立一個 class 名稱、一個 class dictionary（字典），以及一個 base class（基底類別）的列表。Metaclass 負責接受這三個引數，並建立該 class。大多數的物件導向程式語言會提供一個預設的實作。Python 的特別之處在於它能夠建立自訂的 metaclass。大部分的使用者從未需要此工具，但是當需要時，metaclass 可以提供強大且優雅的解決方案。它們已被用於記錄屬性存取、增加執行緒安全性、追蹤物件建立、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 本體內被定義的函式。如果 method 作為其 class 實例的一個屬性被呼叫，則它將會得到該實例物件成為它的第一個 *argument*（引數）（此引數通常被稱為 `self`）。請參閱 *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（基底類別）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，請參閱 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。

另請參閱 *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 請參閱 *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()`。另請參閱 *immutable*（不可變物件）。

named tuple（附名元組） 術語「named tuple（附名元組）」是指從 tuple 繼承的任何型別或 class，且它的可索引 (indexable) 元素也可以用附名屬性來存取。這些型別或 class 也可以具有其他的特性。

有些內建型別是 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⬚是由 random 和 itertools 模組在實作。

namespace package（命名空間套件）　一個 **PEP 420** *package*（套件），它只能作⬚子套件 (subpackage) 的一個容器。命名空間套件可能⬚有實體的表示法，而且具體來⬚它們不像是一個*regular package*（正規套件），因⬚它們⬚⬚有 __init__.py 這個檔案。

　　另請參⬚*module*（模組）。

nested scope（巢狀作用域）　能⬚參照外層定義 (enclosing definition) 中的變數的能力。舉例來⬚，一個函式如果是在另一個函式中被定義，則它便能⬚參照外層函式中的變數。請注意，在預設情⬚下，巢狀作用域僅適用於參照，而無法用於賦值。區域變數能在最⬚層作用域中讀取及寫入。同樣地，全域變數是在全域命名空間中讀取及寫入。nonlocal 容許對外層作用域進行寫入。

new-style class（新式類⬚）　一個舊名，它是指現在所有的 class 物件所使用的 class 風格。在早期的 Python 版本中，只有新式 class 才能使用 Python 較新的、多樣的功能，像是 __slots__、描述器 (descriptor)、屬性 (property)、__getattribute__()、class method（類⬚方法）和 static method（⬚態方法）。

object（物件）　具有狀態（屬性或值）及被定義的行⬚（method）的任何資料。它也是任何*new-style class*（新式類⬚）的最終 base class（基底類⬚）。

package（套件）　一個 Python 的*module*（模組），它可以包含子模組 (submodule) 或是遞⬚的子套件 (subpackage)。技術上而言，套件就是具有 __path__ 屬性的一個 Python 模組。

　　另請參⬚*regular package*（正規套件）和*namespace package*（命名空間套件）。

parameter（參數）　在*function*（函式）或 method 定義中的一個命名實體 (named entity)，它指明該函式能⬚接受的一個*argument*（引數），或在某些情⬚下指示多個引數。共有有五種不同的參數類型：

- *positional-or-keyword*（位置或關鍵字）：指明一個可以按照位置或是作⬚關鍵字引數被傳遞的引數。這是參數的預設類型，例如以下的 *foo* 和 *bar*：

```
def func(foo, bar=None): ...
```

- *positional-only*（僅限位置）：指明一個只能按照位置被提供的引數。在函式定義的參數列表中包含一個 / 字元，就可以在該字元前面定義僅限位置參數，例如以下的 *posonly1* 和 *posonly2*：

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*（僅限關鍵字）：指明一個只能以關鍵字被提供的引數。在函式定義的參數列表中，包含一個任意數量位置參數 (var-positional parameter) 或是單純的 * 字元，就可以在其後方定義僅限關鍵字參數，例如以下的 *kw_only1* 和 *kw_only2*：

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*（任意數量位置）：指明一串能以任意序列被提供的位置引數（在已被其他參數接受的任何位置引數之外）。這類參數是透過在其參數名稱字首加上 * 來定義的，例如以下的 *args*：

```python
def func(*args, **kwargs): ...
```

- *var-keyword*（任意數量關鍵字）：指明可被提供的任意數量關鍵字引數（在已被其他參數接受的任何關鍵字引數之外）。這類參數是透過在其參數名稱字首加上 ** 來定義的，例如上面範例中的 *kwargs*。

參數可以指明引數是選擇性的或必需的，也可以⊞一些選擇性的引數指定預設值。

另請參⊞術語表的*argument*（引數）條目、常見問題中的引數和參數之間的差⊞、inspect.Parameter class、function 章節，以及 **PEP 362**。

path entry（路徑項目）　在*import path*（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⊞ importlib.abc.PathEntryFinder。

path entry hook（路徑項目⊞）　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⊞⊞ str 或 bytes 檔案系統路徑；而 os.fsdecode() 及 os.fsencode() 則分⊞可用於確保 str 及 bytes 的結果。由 **PEP 519** 引入。

PEP　Python Enhancement Proposal（Python 增⊞提案）。PEP 是一份設計⊞明文件，它能⊞ Python 社群提供資訊，或是描述 Python 的一個新功能或該功能的程序和環境。PEP 應該要提供簡潔的技術規範以及被提案功能的運作原理。

PEP 的存在目的，是要成⊞重大新功能的提案、社群中關於某個問題的意見收集，以及已納入 Python 的設計⊞策的記⊞，這些過程的主要機制。PEP 的作者要負責在社群⊞建立共識⊞記⊞反對意見。

請參⊞ **PEP 1**。

portion（部分）　在單一目⊞中的一組檔案（也可能儲存在一個 zip 檔中），這些檔案能對一個命名空間套件 (namespace package) 有所貢獻，如同 **PEP 420** 中的定義。

positional argument（位置引數）　請參⊞*argument*（引數）。

provisional API（暫行 API）　暫行 API 是指，從標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⊞暫行的，理論上⊞不會有重大的變更，但如果核心開發人員認⊞有必要，也可能會出現向後不相容的變更（甚至包括移除該介面）。這種變更⊞不會無端地⊞生——只有 API 被納入之前未察覺的嚴重基本缺陷被揭露時，它們才會發生。

即使對於暫行 API，向後不相容的變更也會被視⊞「最後的解⊞方案」——對於任何被發現的問題，仍然會盡可能找出一個向後相容的解⊞方案。

這個過程使得標准函式庫能隨著時間不斷進化，而避免耗費過長的時間去鎖定有問題的設計錯誤。請參⊞ **PEP 411** 了解更多細節。

provisional package（暫行套件）　請參⊞*provisional API*（暫行 API）。

Python 3000　Python 3.x 系列版本的⊞稱（很久以前創造的，當時第 3 版的發布是在⊞遠的未來。）也可以縮寫⊞「Py3k」。

Pythonic（Python 風格的） 一個想法或一段程式碼，它應用了 Python 語言最常見的慣用語，而不是使用其他語言常見的概念來實作程式碼。例如，Python 中常見的一種習慣用法，是使用一個 for 陳述式，對一個可Ⓕ代物件的所有元素進行Ⓕ圈。許多其他語言ⒻⒻ有這種類型的架構，所以不熟悉 Python 的人有時會使用一個數值計數器來代替：

```
for i in range(len(food)):
    print(food[i])
```

相較之下，以下方法更簡潔、更具有 Python 風格：

```
for piece in food:
    print(piece)
```

qualified name（限定名稱） 一個「點分隔名稱」，它顯示從一個模組的全域作用域到該模組中定義的 class、函式或 method 的「路徑」，如 **PEP 3155** 中的定義。對於頂層的函式和 class 而言，限定名稱與其物件名稱相同：

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

當用於引用模組時，完全限定名憼 *(fully qualified name)* 是表示該模組的完整點分隔路徑，包括任何的父套件，例如 email.mime.text：

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count（參照計數） 對於一個物件的參照次數。當一個物件的參照計數下降到零時，它會被解除配置 (deallocated)。參照計數通常在 Python 程式碼中看不到，但它Ⓕ是*CPython* 實作的一個關鍵元素。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Ⓕ。

另請參Ⓕ*namespace package*（命名空間套件）。

__slots__ 在 class Ⓕ部的一個宣告，它藉由預先宣告實例屬性的空間，以及消除實例 dictionary（字典），來節省記憶體。雖然該技術很普遍，但它有點難以正確地使用，最好保留給那種在一個記憶體關鍵 (memory-critical) 的應用程式中存在大量實例的罕見情Ⓕ。

sequence（序列） 一個*iterable*（可Ⓕ代物件），它透過 __getitem__() special method（特殊方法），使用整數索引來支援高效率的元素存取，Ⓕ定義了一個 __len__() method 來回傳該序列的長度。一些Ⓕ建序列型Ⓕ包括 list、str、tuple 和 bytes。請注意，雖然 dict 也支援 __getitem__() 和 __len__()，但它被視Ⓕ對映 (mapping) 而不是序列，因Ⓕ其查找方式是使用任意的*immutable* 鍵，而不是整數。

抽象基底類Ⓕ (abstract base class) collections.abc.Sequence 定義了一個更加豐富的介面，Ⓕ不僅止於 __getitem__() 和 __len__()，還增加了 count()、index()、__contains__() 和 __reversed__()。實作此擴充介面的型Ⓕ，可以使用 register() 被明確地ⒻⒻ。

set comprehension（集合綜合運算） 一種緊密的方法，用來處理一個可迭代物件中的全部或部分元素，並將處理結果以一個 set 回傳。`results = {c for c in 'abracadabra' if c not in 'abc'}` 會產生一個字串 set: `{'r', 'd'}`。請參見 comprehensions。

single dispatch（單一調度） *generic function*（泛型函式）調度的一種形式，在此，實作的選擇是基於單一引數的型別。

slice（切片） 一個物件，它通常包含一段 *sequence*（序列）的某一部分。建立一段切片的方法是使用下標符號（subscript notation）"[]"，若要給出多個數字，則在數字之間使用冒號，例如 in `variable_name[1:3:5]`。在括號（下標）符號的內部，會使用 `slice` 物件。

special method（特殊方法） 一種會被 Python 自動呼叫的 method，用於對某種型別執行某種運算，例如加法。這種 method 的名稱會在開頭和結尾有兩個下底線。Special method 在 specialnames 中有詳細說明。

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。

text encoding（文字編碼） A string in Python is a sequence of Unicode code points (in range `U+0000--U+10FFFF`). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as "encoding", and recreating the string from the sequence of bytes is known as "decoding".

There are a variety of different text serialization codecs, which are collectively referred to as "text encodings".

text file（文字檔案） 一個能以讀取和寫入 `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) 並會自動處理 *text encoding*（文字編碼）。文字檔案的例子有：以文字模式（`'r'` 或 `'w'`）開啟的檔案、`sys.stdin`、`sys.stdout` 以及 `io.StringIO` 的實例。

另請參見 *binary file*（二進制檔案），它是一個能以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 (') 的作為邊界的一個字串。雖然它們並沒有提供優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脫 (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，這使得它們在編寫說明字串時特別有用。

type（型別） 一個 Python 物件的型別界定了它是什麼類型的物件；每個物件都有一個型別。一個物件的型別可以用它的 `__class__` 屬性來存取，或以 `type(obj)` 來檢索。

type alias（型別別名） 一個型別的同義詞，透過將型別指定給一個識別符 (identifier) 來建立。

型別別名對於簡化*型別提示 (type hint)* 很有用。例如：

```python
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：

```python
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

請參見 `typing` 和 **PEP 484**，有此功能的描述。

type hint（型別提示） 一種 *annotation*（註釋），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別。

型別提示是選擇性的，而不是被 Python 強制的，但它們對靜態型別分析工具很有用，也能協助 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）的型別提示，都可以使用 `typing.get_type_hints()` 來存取。

請參閱 `typing` 和 **PEP 484**，有此功能的描述。

universal newlines（通用換行字元） 一種解譯文字流 (text stream) 的方式，會將以下所有的情況識別為同一行的結束：Unix 行尾慣例 `'\n'`、Windows 慣例 `'\r\n'` 和舊的 Macintosh 慣例 `'\r'`。請參閱 **PEP 278** 和 **PEP 3116**，以及用於 `bytes.splitlines()` 的附加用途。

variable annotation（變數註釋） 一個變數或 class 屬性的*annotation*（註釋）。

註釋變數或 class 屬性時，賦值是選擇性的：

```python
class C:
    field: 'annotation'
```

變數註釋通常用於型別提示 *(type hint)*：例如，這個變數預期會取得 int（整數）值：

```python
count: int = 0
```

變數註釋的語法在 annassign 章節有詳細的解釋。

請參閱*function annotation*（函式註釋）、**PEP 484** 和 **PEP 526**，皆有此功能的描述。

virtual environment（虛擬環境） 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 發行套件，而不會對同一個系統上運行的其他 Python 應用程式的行為產生干擾。

另請參閱 `venv`。

virtual machine（虛擬機器） 一部完全由軟體所定義的電腦 (computer)。Python 的虛擬機器會執行由*bytecode*（位元組碼）編譯器所發出的位元組碼。

Zen of Python（Python 之禪） Python 設計原則與哲學的列表，其內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。

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 for his Alternative Python Reference project from which Sphinx got many good ideas.

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！

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F一種稱F ABC 語言的後繼者。F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，F在那F發F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 F成立了 BeOpen PythonLabs 團隊。同年十月，PythonLabs 團隊轉移到 Digital Creations（現F 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F擁有 Python 相關的智慧財F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F 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F。

發F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F： GPL 相容F不表示我們是在 GPL 下發F Python。不像 GPL，所有的 Python 授權都可以讓您發F修改

後的版本，但不一定要使您的變更成⊡開源。GPL 相容的授權使得 Python 可以結合其他在 GPL 下發⊡的軟體一起使用；但其它的授權則不行。

感謝許多的外部志工，在 Guido 指導下的付出，使得這些版本的發⊡成⊡可能。

C.2 關於存取或以其他方式使用 Python 的合約條款

Python 軟體和⊡明文件的授權是基於*PSF 授權合約*。

從 Python 3.8.6 開始，⊡明文件中的範例、程式庫和其他程式碼，是被雙重授權 (dual licensed) 於 PSF 授權合約以及*Zero-Clause BSD 授權*。

有些被納入 Python 中的軟體是基於不同的授權。這些授權將會與其授權之程式碼一起被列出。關於這些授權的不完整清單，請參⊡*被收⊡軟體的授權與致謝*。

C.2.1 用於 PYTHON 3.9.13 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
   3.9.13 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 3.9.13 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice␣
→of
   copyright, i.e., "Copyright © 2001-2022 Python Software Foundation; All␣
→Rights
   Reserved" are retained in Python 3.9.13 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 3.9.13 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
   3.9.13.

4. PSF is making Python 3.9.13 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 3.9.13 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.9.13
```

```
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT␣
↪OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.9.13, 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 3.9.13, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 用於 PYTHON 2.0 的 BEOPEN.COM 授權合約

BEOPEN PYTHON 開源授權合約第 1 版

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
```

```
    Nothing in this License Agreement shall be deemed to create any relationship of
    agency, partnership, or joint venture between BeOpen and Licensee.  This License
    Agreement does not grant permission to use BeOpen trademarks or trade names in a
    trademark sense to endorse or promote products or services of Licensee, or any
    third party.  As an exception, the "BeOpen Python" logos available at
    http://www.pythonlabs.com/logos.html may be used according to the permissions
    granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 用於 PYTHON 1.6.1 的 CNRI 授權合約

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   Internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
```

```
law of the United States, including without limitation the federal copyright
law, and, to the extent such U.S. federal law does not apply, by the law of the
Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
Notwithstanding the foregoing, with regard to derivative works based on Python
1.6.1 that incorporate non-separable material that was previously distributed
under the GNU General Public License (GPL), the law of the Commonwealth of
Virginia shall govern this License Agreement only as to issues arising under or
with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
this License Agreement shall be deemed to create any relationship of agency,
partnership, or joint venture between CNRI and Licensee.  This License Agreement
does not grant permission to use CNRI trademarks or trade name in a trademark
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 3.9.13 ⬚明文件⬚程式碼的 ZERO-CLAUSE BSD 授權

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 被收⊞軟體的授權與致謝

本節是一個不完整但持續增加的授權與致謝清單，對象是在 Python 發⊞版本中所收⊞的第三方軟體。

C.3.1 Mersenne Twister

_random 模組包含了以 http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html 的下載⊞
容⊞基礎的程式碼。以下是原始程式碼的完整聲明：

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

socket 模組使用 `getaddrinfo()` 和 `getnameinfo()` 函式，它們在 WIDE 專案（http://www.wide.ad.jp/）中，於不同的原始檔案中被編碼：

C.3.3 非同步 socket 服務

`asynchat` 和 `asyncore` 模組包含以下聲明：

C.3.4 Cookie 管理

`http.cookies` 模組包含以下聲明：

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

                All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 執行追⊞

`trace` 模組包含以下聲明：

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode 與 UUdecode 函式

uu 模組包含以下聲明：

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                    All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.


Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML 遠端程序呼叫

xmlrpc.client 模組包含以下聲明：

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
```

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

`test_epoll` 模組包含以下聲明:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

`select` 模組對於 kqueue 介面包含以下聲明:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
```

```
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

Python/pyhash.c 檔案包含 Marek Majkowski' 基於 Dan Bernstein 的 SipHash24 演算法的實作。它包含以下聲明:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 與 dtoa

Python/dtoa.c 檔案提供了 C 的 dtoa 和 strtod 函式,用於將 C 的雙精度浮點數和字串互相轉行。該檔案是衍生自 David M. Gay 建立的同名檔案,後者現在可以從 http://www.netlib.org/fp/ 下載。於 2009 年 3 月 16 日所檢索的原始檔案包含以下版權與授權聲明:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
```

```
 *
 ***************************************************************/
```

C.3.12 OpenSSL

如果 OpenSSL 函式庫可被作業系統使用，則 hashlib、posix、ssl、crypt 模組會使用它來提升效能。
此外，因F 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F
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/)"
```

```
     *
     * 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
```

```
 *      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⊞ `--with-system-expat`，否則該擴充會用一個⊞含 expat 原始碼的副本來建置：

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                         and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

除非在建置 `_ctypes` 擴充時設定⻆ `--with-system-libffi`，否則該擴充會用一個⻆含 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，則該擴充會用一個⻆含 zlib 原始碼的副
本來建置:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

`tracemalloc` 使用的雜⬜表 (hash table) 實作，是以 cfuhash 專案⬜基礎：

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

除非在建置 `_decimal` 模組時設定⬜ `--with-system-libmpdec`，否則該模組會用一個⬜含 libmpdec 函式庫的副本來建置：

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
```

```
THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N 測試套件

test 程式包中的 C14N 2.0 測試套件 (`Lib/test/xmltestdata/c14n-20/`) 是從 W3C 網站 https://www.w3.org/TR/xml-c14n2-testcases/ 被檢索，且是基於 3-clause BSD 授權被發行：

```
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.
```

APPENDIX D

版權宣告

Python 和這份聲明文件的版權：

完整的授權條款資訊請參見*沿革與授權*。

Z