The Python/C API

Release 2.7.18

Guido van Rossum
and the Python development team

maio 07, 2020

Python Software Foundation
Email: docs@python.org

Sumário

Este manual documenta a API usada por programadores C e C++ que desejam escrever módulos de extensões ou embutir Python. É um complemento para extending-index, que descreve os princípios gerais da escrita de extensões mas não documenta as funções da API em detalhes.

CAPÍTULO 1

Introdução

The Application Programmer's Interface to Python gives C and C++ programmers access to the Python interpreter at a variety of levels. The API is equally usable from C++, but for brevity it is generally referred to as the Python/C API. There are two fundamentally different reasons for using the Python/C API. The first reason is to write *extension modules* for specific purposes; these are C modules that extend the Python interpreter. This is probably the most common use. The second reason is to use Python as a component in a larger application; this technique is generally referred to as *embedding* Python in an application.

Writing an extension module is a relatively well-understood process, where a "cookbook" approach works well. There are several tools that automate the process to some extent. While people have embedded Python in other applications since its early existence, the process of embedding Python is less straightforward than writing an extension.

Many API functions are useful independent of whether you're embedding or extending Python; moreover, most applications that embed Python will need to provide a custom extension as well, so it's probably a good idea to become familiar with writing an extension before attempting to embed Python in a real application.

1.1 Incluir Arquivos

All function, type and macro definitions needed to use the Python/C API are included in your code by the following line:

```
#include "Python.h"
```

This implies inclusion of the following standard headers: `<stdio.h>`, `<string.h>`, `<errno.h>`, `<limits.h>`, `<assert.h>` and `<stdlib.h>` (if available).

Nota: Since Python may define some pre-processor definitions which affect the standard headers on some systems, you *must* include `Python.h` before any standard headers are included.

All user visible names defined by Python.h (except those defined by the included standard headers) have one of the prefixes `Py` or `_Py`. Names beginning with `_Py` are for internal use by the Python implementation and should not be used by extension writers. Structure member names do not have a reserved prefix.

Important: user code should never define names that begin with `Py` or `_Py`. This confuses the reader, and jeopardizes the portability of the user code to future Python versions, which may define additional names beginning with one of these prefixes.

The header files are typically installed with Python. On Unix, these are located in the directories `prefix/include/pythonversion/` and `exec_prefix/include/pythonversion/`, where `prefix` and `exec_prefix` are defined by the corresponding parameters to Python's **configure** script and *version* is `sys.version[:3]`. On Windows, the headers are installed in `prefix/include`, where `prefix` is the installation directory specified to the installer.

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

C++ users should note that though the API is defined entirely using C, the header files do properly declare the entry points to be `extern "C"`, so there is no need to do anything special to use the API from C++.

1.2 Objetos, tipos e contagens de referência

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

All Python objects (even Python integers) have a *type* and a *reference count*. An object's type determines what kind of object it is (e.g., an integer, a list, or a user-defined function; there are many more as explained in types). For each of the well-known types there is a macro to check whether an object is of that type; for instance, `PyList_Check(a)` is true if (and only if) the object pointed to by *a* is a Python list.

1.2.1 Contagens de referência

A contagem de referência é importante porque os computadores de hoje têm um tamanho de memória finito (e geralmente muito limitado); Conta quantos lugares diferentes existem que têm uma referência a um objeto. Esse local poderia ser outro objeto, uma variável C global (ou estática) ou uma variável local em alguma função C. Quando a contagem de referência de um objeto se torna zero, o objeto é desalocado. Se contiver referências a outros objetos, sua contagem de referência será diminuída. Esses outros objetos podem ser desalocados, por sua vez, se esse decremento fizer com que sua contagem de referência se torne zero e assim por diante. (Há um problema óbvio com objetos que fazem referência um ao outro aqui; por enquanto, a solução é "não faça isso").

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

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

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

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

Detalhes da contagem de referência

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, PyInt_FromLong(1L));
PyTuple_SetItem(t, 1, PyInt_FromLong(2L));
PyTuple_SetItem(t, 2, PyString_FromString("three"));
```

Here, `PyInt_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)
{
    int i, n;

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

Here is an example of how you could write a function that computes the sum of the items in a list of integers; once using `PyList_GetItem()`, and once using `PySequence_GetItem()`.

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

(continua na próxima página)

(continuação da página anterior)

```
    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 (!PyInt_Check(item)) continue; /* Skip non-integers */
        total += PyInt_AsLong(item);
    }
    return total;
}
```

```
long
sum_sequence(PyObject *sequence)
{
    int i, n;
    long total = 0;
    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 (PyInt_Check(item))
            total += PyInt_AsLong(item);
        Py_DECREF(item); /* Discard reference ownership */
    }
    return total;
}
```

1.2.2 Tipos

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.

1.3 Exceções

The Python programmer only needs to deal with exceptions if specific error handling is required; unhandled exceptions are automatically propagated to the caller, then to the caller's caller, and so on, until they reach the top-level interpreter, where they are reported to the user accompanied by a stack traceback.

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 objects `sys.exc_type`, `sys.exc_value`, and `sys.exc_traceback`; 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_type` 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:

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

Here is the corresponding C code, in all its glory:

```
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 = PyInt_FromLong(0L);
        if (item == NULL)
            goto error;
    }
    const_one = PyInt_FromLong(1L);
```

(continua na próxima página)

(continuação da página anterior)

```
    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.

1.4 Incorporando 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 `__builtin__`, `__main__`, `sys`, and `exceptions`. 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/pythonX.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/pythonX.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_Finalize()`. 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_Finalize()` does *not* free all memory allocated by the Python interpreter, e.g. memory allocated by extension modules currently cannot be released.

1.5 Construções de Depuração

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.

Além da depuração de contagem de referência descrita abaixo, as seguintes verificações extras são executadas:

- Verificações extras são adicionadas ao alocador de objeto.
- Verificações extras são adicionadas ao analisador e ao compilador.
- Downcasts de tipos amplos para tipos restritos são verificados quanto à perda de informações.
- A number of assertions are added to the dictionary and set implementations. In addition, the set object acquires a `test_c_api()` method.
- As verificações de integridade dos argumentos de entrada são adicionadas à criação de quadros.
- The storage for long ints is initialized with a known invalid pattern to catch reference to uninitialized digits.
- O rastreamento de baixo nível e a verificação de exceções extras são adicionados à máquina virtual de tempo de execução.
- Verificações extras são adicionadas à implementação da arena de memória.
- Depuração extra é adicionada ao módulo de thread.

Pode haver verificações adicionais não mencionadas aqui.

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`.

Please refer to `Misc/SpecialBuilds.txt` in the Python source distribution for more detailed information.

CAPÍTULO 2

A camada de Mais Alto Nível

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

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

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

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

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

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

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

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

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

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

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

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

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

If *fp* refers to a file associated with an interactive device (console or terminal input or Unix pseudo-terminal), return the value of `PyRun_InteractiveLoop()`, otherwise return the result of `PyRun_SimpleFile()`. If *filename* is *NULL*, this function uses `"???"` as the filename.

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

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

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

int **PyRun_SimpleFile**(FILE **fp*, const char **filename*)
This is a simplified interface to `PyRun_SimpleFileExFlags()` below, leaving *closeit* set to `0` and *flags* set to *NULL*.

int **PyRun_SimpleFileFlags**(FILE **fp*, const char **filename*, PyCompilerFlags **flags*)
This is a simplified interface to `PyRun_SimpleFileExFlags()` below, leaving *closeit* set to `0`.

int **PyRun_SimpleFileEx**(FILE **fp*, const char **filename*, int *closeit*)
This is a simplified interface to `PyRun_SimpleFileExFlags()` below, leaving *flags* set to *NULL*.

int **PyRun_SimpleFileExFlags**(FILE **fp*, const char **filename*, int *closeit*, PyCompilerFlags **flags*)
Similar to `PyRun_SimpleStringFlags()`, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file. If *closeit* is true, the file is closed before PyRun_SimpleFileExFlags returns.

int **PyRun_InteractiveOne**(FILE **fp*, const char **filename*)
This is a simplified interface to `PyRun_InteractiveOneFlags()` below, leaving *flags* set to *NULL*.

int **PyRun_InteractiveOneFlags**(FILE **fp*, const char **filename*, PyCompilerFlags **flags*)
Read and execute a single statement from a file associated with an interactive device according to the *flags* argument. The user will be prompted using `sys.ps1` and `sys.ps2`. Returns `0` when the input was executed successfully, `-1` if there was an exception, or an error code from the `errcode.h` include file distributed as part of Python if there was a parse error. (Note that `errcode.h` is not included by `Python.h`, so must be included specifically if needed.)

int **PyRun_InteractiveLoop**(FILE **fp*, const char **filename*)
This is a simplified interface to `PyRun_InteractiveLoopFlags()` below, leaving *flags* set to *NULL*.

int **PyRun_InteractiveLoopFlags**(FILE **fp*, const char **filename*, PyCompilerFlags **flags*)
Read and execute statements from a file associated with an interactive device until EOF is reached. The user will be prompted using `sys.ps1` and `sys.ps2`. Returns `0` at EOF.

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`.

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*.

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.

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`.

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.

*PyObject** **PyRun_String** (const char *str, int start, PyObject *globals, PyObject *locals)
Return value: New reference. This is a simplified interface to `PyRun_StringFlags()` below, leaving *flags* set to *NULL*.

*PyObject** **PyRun_StringFlags** (const char *str, int start, PyObject *globals, PyObject *locals, PyCompilerFlags *flags)
Return value: New reference. Execute Python source code from *str* in the context specified by the dictionaries *globals* and *locals* with the compiler flags specified by *flags*. The parameter *start* specifies the start token that should be used to parse the source code.

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

*PyObject** **PyRun_File** (FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals)
Return value: New reference. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *closeit* set to `0` and *flags* set to *NULL*.

*PyObject** **PyRun_FileEx** (FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, int closeit)
Return value: New reference. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *flags* set to *NULL*.

*PyObject** **PyRun_FileFlags** (FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, PyCompilerFlags *flags)
Return value: New reference. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *closeit* set to `0`.

*PyObject** **PyRun_FileExFlags** (FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, int closeit, PyCompilerFlags *flags)
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. 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. This is a simplified interface to `Py_CompileStringFlags()` below, leaving *flags* set to *NULL*.

*PyObject** **Py_CompileStringFlags** (const char *str, const char *filename, int start, PyCompilerFlags *flags)
Return value: New reference. Parse and compile the Python source code in *str*, returning the resulting code object. The start token is given by *start*; this can be used to constrain the code which can be compiled and should be `Py_eval_input`, `Py_file_input`, or `Py_single_input`. The filename specified by *filename* is used to construct the code object and may appear in tracebacks or `SyntaxError` exception messages. This returns *NULL* if the code cannot be parsed or compiled.

*PyObject** **PyEval_EvalCode** (PyCodeObject *co, PyObject *globals, PyObject *locals)
Return value: New reference. This is a simplified interface to `PyEval_EvalCodeEx()`, with just the code object, and the dictionaries of global and local variables. The other arguments are set to *NULL*.

*PyObject** **PyEval_EvalCodeEx** (PyCodeObject *co, PyObject *globals, PyObject *locals, PyObject **args, int argcount, PyObject **kws, int kwcount, PyObject **defs, int defcount, PyObject *closure)
Evaluate a precompiled code object, given a particular environment for its evaluation. This environment consists of dictionaries of global and local variables, arrays of arguments, keywords and defaults, and a closure tuple of cells.

*PyObject** **PyEval_EvalFrame** (PyFrameObject **f*)
Evaluate an execution frame. This is a simplified interface to PyEval_EvalFrameEx, for backward compatibility.

*PyObject** **PyEval_EvalFrameEx** (PyFrameObject **f*, int *throwflag*)
This is the main, unvarnished function of Python interpretation. It is literally 2000 lines long. The code object associated with the execution frame *f* is executed, interpreting bytecode and executing calls as needed. The additional *throwflag* parameter can mostly be ignored - if true, then it causes an exception to immediately be thrown; this is used for the `throw()` methods of generator objects.

int **PyEval_MergeCompilerFlags** (*PyCompilerFlags* **cf*)
This function changes the flags of the current evaluation frame, and returns true on success, false on failure.

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

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

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

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

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

```
struct PyCompilerFlags {
    int cf_flags;
}
```

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

CAPÍTULO 3

Contagem de Referência

The macros in this section are used for managing reference counts of Python objects.

void **Py_INCREF**(*PyObject *o*)
: Increment the reference count for object *o*. The object must not be *NULL*; if you aren't sure that it isn't *NULL*, use `Py_XINCREF()`.

void **Py_XINCREF**(*PyObject *o*)
: Increment the reference count for object *o*. The object may be *NULL*, in which case the macro has no effect.

void **Py_DECREF**(*PyObject *o*)
: Decrement the reference count for object *o*. The object must not be *NULL*; if you aren't sure that it isn't *NULL*, use `Py_XDECREF()`. If the reference count reaches zero, the object's type's deallocation function (which must not be *NULL*) is invoked.

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

void **Py_XDECREF**(*PyObject *o*)
: Decrement the reference count for object *o*. The object may be *NULL*, in which case the macro has no effect; otherwise the effect is the same as for `Py_DECREF()`, and the same warning applies.

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

It is a good idea to use this macro whenever decrementing the value of a variable that might be traversed during garbage collection.

Novo na versão 2.4.

The following functions are for runtime dynamic embedding of Python: `Py_IncRef(PyObject *o)`, `Py_DecRef(PyObject *o)`. They are simply exported function versions of *Py_XINCREF()* and *Py_XDECREF()*, respectively.

The following functions or macros are only for use within the interpreter core: `_Py_Dealloc()`, `_Py_ForgetReference()`, `_Py_NewReference()`, as well as the global variable `_Py_RefTotal`.

CAPÍTULO 4

Manipulando Exceções

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

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

The error indicator consists of three Python objects corresponding to the Python variables `sys.exc_type`, `sys.exc_value` and `sys.exc_traceback`. API functions exist to interact with the error indicator in various ways. There is a separate error indicator for each thread.

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

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

Chame esta função **apenas** quando o indicador de erro estiver definido. Caso contrário, causará um erro fatal!

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

void **PyErr_Print**()

Alias para"PyErr_PrintEx(1)".

*PyObject** **PyErr_Occurred**()

Return value: Borrowed reference. Test whether the error indicator is set. If set, return the exception *type* (the first argument to the last call to one of the `PyErr_Set*()` functions or to `PyErr_Restore()`). If not set, return *NULL*. You do not own a reference to the return value, so you do not need to `Py_DECREF()` it.

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

int **PyErr_GivenExceptionMatches**(*PyObject* *given*, *PyObject* *exc*)
Return true if the *given* exception matches the exception 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 exceptions in the tuple (and recursively in subtuples) are searched for a match.

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.

void **PyErr_Clear**()
Limpe o indicador de erro. Se o indicador de erro não estiver definido, não haverá efeito.

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.

Nota: This function is normally only used by code that needs to handle exceptions or by code that needs to save and restore the error indicator temporarily.

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.)

Nota: 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 exception state.

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 converted to a string object.

void **PyErr_SetObject**(*PyObject* **type*, *PyObject* **value*)
Essa função é semelhante à `PyErr_SetString()` mas permite especificar um objeto Python arbitrário para o valor da exceção.

*PyObject** **PyErr_Format**(*PyObject* **exception*, const char **format*, ...)
Return value: Always NULL. This function sets the error indicator and returns *NULL*. *exception* should be a Python

exception class. The *format* and subsequent parameters help format the error message; they have the same meaning and values as in `PyString_FromFormat()`.

void **PyErr_SetNone**(*PyObject *type*)
Isso é uma abreviação para `PyErr_SetObject(type, Py_None)`.

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

*PyObject** **PyErr_NoMemory**()
Return value: Always NULL. This is a shorthand for `PyErr_SetNone(PyExc_MemoryError)`; it returns *NULL* so an object allocation function can write `return PyErr_NoMemory();` when it runs out of memory.

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

*PyObject** **PyErr_SetFromErrnoWithFilenameObject**(*PyObject *type*, *PyObject *filenameObject*)
Similar to `PyErr_SetFromErrno()`, with the additional behavior that if *filenameObject* is not *NULL*, it is passed to the constructor of *type* as a third parameter. In the case of exceptions such as `IOError` and `OSError`, this is used to define the `filename` attribute of the exception instance.

*PyObject** **PyErr_SetFromErrnoWithFilename**(*PyObject *type*, const char **filename*)
Return value: Always NULL. Similar to `PyErr_SetFromErrnoWithFilenameObject()`, but the filename is given as a C string.

*PyObject** **PyErr_SetFromWindowsErr**(int *ierr*)
Return value: Always NULL. This is a convenience function to raise `WindowsError`. If called with *ierr* of `0`, the error code returned by a call to `GetLastError()` is used instead. It calls the Win32 function `FormatMessage()` to retrieve the Windows description of error code given by *ierr* or `GetLastError()`, then it constructs a tuple object whose first item is the *ierr* value and whose second item is the corresponding error message (gotten from `FormatMessage()`), and then calls `PyErr_SetObject(PyExc_WindowsError, object)`. This function always returns *NULL*. Availability: Windows.

*PyObject** **PyErr_SetExcFromWindowsErr**(*PyObject *type*, int *ierr*)
Return value: Always NULL. Similar to `PyErr_SetFromWindowsErr()`, with an additional parameter specifying the exception type to be raised. Availability: Windows.

Novo na versão 2.3.

*PyObject** **PyErr_SetFromWindowsErrWithFilenameObject**(int *ierr*, *PyObject *filenameObject*)
Similar to `PyErr_SetFromWindowsErr()`, with the additional behavior that if *filenameObject* is not *NULL*, it is passed to the constructor of `WindowsError` as a third parameter. Availability: Windows.

*PyObject** **PyErr_SetFromWindowsErrWithFilename**(int *ierr*, const char **filename*)
Return value: Always NULL. Similar to `PyErr_SetFromWindowsErrWithFilenameObject()`, but the filename is given as a C string. Availability: Windows.

*PyObject** **PyErr_SetExcFromWindowsErrWithFilenameObject**(*PyObject *type*, int *ierr*, *PyObject *filename*)
Similar to `PyErr_SetFromWindowsErrWithFilenameObject()`, with an additional parameter specifying the exception type to be raised. Availability: Windows.

Novo na versão 2.3.

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

Return value: Always NULL. Similar to `PyErr_SetFromWindowsErrWithFilename()`, with an additional parameter specifying the exception type to be raised. Availability: Windows.

Novo na versão 2.3.

void **PyErr_BadInternalCall**()

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

int **PyErr_WarnEx**(*PyObject* **category*, char **message*, int *stacklevel*)

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

This function normally prints a warning message to *sys.stderr*; however, it is also possible that the user has specified that warnings are to be turned into errors, and in that case this will raise an exception. It is also possible that the function raises an exception because of a problem with the warning machinery (the implementation imports the `warnings` module to do the heavy lifting). The return value is `0` if no exception is raised, or `-1` if an exception is raised. (It is not possible to determine whether a warning message is actually printed, nor what the reason is for the exception; this is intentional.) If an exception is raised, the caller should do its normal exception handling (for example, `Py_DECREF()` owned references and return an error value).

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

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

int **PyErr_Warn**(*PyObject* **category*, char **message*)

Issue a warning message. The *category* argument is a warning category (see below) or *NULL*; the *message* argument is a message string. The warning will appear to be issued from the function calling `PyErr_Warn()`, equivalent to calling `PyErr_WarnEx()` with a *stacklevel* of 1.

Deprecated; use `PyErr_WarnEx()` instead.

int **PyErr_WarnExplicit**(*PyObject* **category*, const char **message*, const char **filename*, int *lineno*, const char **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.

int **PyErr_WarnPy3k**(char **message*, int *stacklevel*)

Issue a `DeprecationWarning` with the given *message* and *stacklevel* if the `Py_Py3kWarningFlag` flag is enabled.

Novo na versão 2.6.

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**()

This function simulates the effect of a `SIGINT` signal arriving — the next time `PyErr_CheckSignals()` is called, `KeyboardInterrupt` will be raised. It may be called without holding the interpreter lock.

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

This utility function specifies a file descriptor to which a `'\0'` byte will be written whenever a signal is received. It returns the previous such file descriptor. The value `-1` disables the feature; this is the initial state. This is equivalent to `signal.set_wakeup_fd()` in Python, but without any error checking. *fd* should be a valid file descriptor. The function should only be called from the main thread.

Novo na versão 2.6.

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

Return value: New reference. Same as `PyErr_NewException()`, except that the new exception class can easily be given a docstring: If *doc* is non-*NULL*, it will be used as the docstring for the exception class.

Novo na versão 2.7.

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

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

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

4.1 Objetos de exceção Unicode

As seguintes funções são usadas para criar e modificar exceções Unicode de 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*)

Create a `UnicodeDecodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*.

*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*)

Create a `UnicodeEncodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*.

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

Create a `UnicodeTranslateError` object with the attributes *object*, *length*, *start*, *end* and *reason*.

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

Retorna o atributo * encoding* dado no objeto da exceção.

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

*PyObject** **PyUnicodeTranslateError_GetObject** (*PyObject *exc*)
Retorna o atributo *object* dado no objeto da exceção.

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*)
Define o atributo *start* dado no objeto de exceção *start*. Em caso de sucesso, retorna `0`, em caso de falha, retorna `-1`.

int **PyUnicodeDecodeError_GetEnd** (*PyObject *exc*, Py_ssize_t **end*)
int **PyUnicodeEncodeError_GetEnd** (*PyObject *exc*, Py_ssize_t **end*)
int **PyUnicodeTranslateError_GetEnd** (*PyObject *exc*, Py_ssize_t **end*)
Get the *end* attribute of the given exception object and place it into **end*. *end* must not be *NULL*. Return `0` on success, `-1` on failure.

int **PyUnicodeDecodeError_SetEnd** (*PyObject *exc*, Py_ssize_t *end*)
int **PyUnicodeEncodeError_SetEnd** (*PyObject *exc*, Py_ssize_t *end*)
int **PyUnicodeTranslateError_SetEnd** (*PyObject *exc*, Py_ssize_t *end*)
Set the *end* attribute of the given exception object to *end*. Return `0` on success, `-1` on failure.

*PyObject** **PyUnicodeDecodeError_GetReason** (*PyObject *exc*)
*PyObject** **PyUnicodeEncodeError_GetReason** (*PyObject *exc*)
*PyObject** **PyUnicodeTranslateError_GetReason** (*PyObject *exc*)
Retorna o atributo *reason* dado no objeto da exceção.

int **PyUnicodeDecodeError_SetReason** (*PyObject *exc*, const char **reason*)
int **PyUnicodeEncodeError_SetReason** (*PyObject *exc*, const char **reason*)
int **PyUnicodeTranslateError_SetReason** (*PyObject *exc*, const char **reason*)
Set the *reason* attribute of the given exception object to *reason*. Return `0` on success, `-1` on failure.

4.2 Controle de recursão

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).

int **Py_EnterRecursiveCall** (const char **where*)
Marca um ponto em que a chamada recursiva em nível C está prestes a ser executada.

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

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

where should be a string such as `" in instance check"` to be concatenated to the `RuntimeError` message caused by the recursion depth limit.

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

4.3 Exceções Padrão

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

Nome C	Nome Python	Notas
PyExc_BaseException	BaseException	(1), (4)
PyExc_Exception	Exception	(1)
PyExc_StandardError	StandardError	(1)
PyExc_ArithmeticError	ArithmeticError	(1)
PyExc_AssertionError	AssertionError	
PyExc_AttributeError	AttributeError	
PyExc_BufferError	BufferError	
PyExc_EnvironmentError	EnvironmentError	(1)
PyExc_EOFError	EOFError	
PyExc_FloatingPointError	FloatingPointError	
PyExc_GeneratorExit	GeneratorExit	
PyExc_ImportError	ImportError	
PyExc_IndentationError	IndentationError	
PyExc_IndexError	IndexError	
PyExc_IOError	IOError	
PyExc_KeyError	KeyError	
PyExc_KeyboardInterrupt	KeyboardInterrupt	
PyExc_LookupError	LookupError	(1)
PyExc_MemoryError	MemoryError	
PyExc_NameError	NameError	
PyExc_NotImplementedError	NotImplementedError	
PyExc_OSError	OSError	
PyExc_OverflowError	OverflowError	
PyExc_ReferenceError	ReferenceError	(2)
PyExc_RuntimeError	RuntimeError	
PyExc_StopIteration	StopIteration	
PyExc_SyntaxError	SyntaxError	
PyExc_SystemError	SystemError	
PyExc_SystemExit	SystemExit	
PyExc_TabError	TabError	
PyExc_TypeError	TypeError	
PyExc_UnboundLocalError	UnboundLocalError	
PyExc_UnicodeDecodeError	UnicodeDecodeError	
PyExc_UnicodeEncodeError	UnicodeEncodeError	
PyExc_UnicodeError	UnicodeError	
PyExc_UnicodeTranslateError	UnicodeTranslateError	
PyExc_VMSError	VMSError	(5)
PyExc_ValueError	ValueError	
PyExc_WindowsError	WindowsError	(3)
PyExc_ZeroDivisionError	ZeroDivisionError	

Notas:

(1) Esta é uma classe base para outras exceções padrão.

(2) This is the same as `weakref.ReferenceError`.

(3) Defina apenas no Windows; proteja o código que usa isso testando se a macro do pré-processador MS_WINDOWS está definida.

(4) Novo na versão 2.5.

(5) Only defined on VMS; protect code that uses this by testing that the preprocessor macro __VMS is defined.

4.4 Categorias de aviso padrão

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

Nome C	Nome Python	Notas
PyExc_Warning	Warning	(1)
PyExc_BytesWarning	BytesWarning	
PyExc_DeprecationWarning	DeprecationWarning	
PyExc_FutureWarning	FutureWarning	
PyExc_ImportWarning	ImportWarning	
PyExc_PendingDeprecationWarning	PendingDeprecationWarning	
PyExc_RuntimeWarning	RuntimeWarning	
PyExc_SyntaxWarning	SyntaxWarning	
PyExc_UnicodeWarning	UnicodeWarning	
PyExc_UserWarning	UserWarning	

Notas:

(1) Esta é uma classe base para outras categorias de aviso padrão.

4.5 String Exceptions

Alterado na versão 2.6: All exceptions to be raised or caught must be derived from BaseException. Trying to raise a string exception now raises TypeError.

CAPÍTULO 5

Utilitários

As funções neste capítulo executam várias tarefas de utilidade pública, desde ajudar o código C a ser mais portátil em plataformas, usando módulos Python de C, como também, a analise de argumentos de função e a construção de valores Python a partir de valores C.

5.1 Utilitários do sistema operacional

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_AfterFork**()

Função para atualizar algum estado interno após um processo de garfo; Isso deve ser chamado no novo processo se o intérprete do Python continuar a ser usado. Se um novo executável é carregado no novo processo, esta função não precisa ser chamada.

int **PyOS_CheckStack**()

Retornar verdadeiro quando o intérprete ficar sem espaço de pilha. Esta é uma verificação confiável, mas só está disponível quando `USE_STACKCHECK` está definido (atualmente no Windows usando o compilador Microsoft Visual C++). `USE_STACKCHECK` será definido automaticamente; Você nunca deve mudar a definição em seu próprio código.

PyOS_sighandler_t **PyOS_getsig**(int *i*)

Retorna o manipulador de sinal atual para o sinal *i*. Este é um invólucro fino em torno de `sigaction()` ou `signal()`. Não ligue para essas funções diretamente! `PyOS_sighandler_t` é um alias de typedef para `void (*)(int)`.

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

Defina o manipulador de sinal para que o sinal *i* seja *h*; Devolva o antigo manipulador de sinal. Este é um invólucro fino em torno de `sigaction()` ou `signal()`. Não ligue para essas funções diretamente! `PyOS_sighandler_t` é um alias de typedef para `void (*)(int)`.

5.2 System Functions

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*** (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.

FILE ***PySys_GetFile*** (char **name*, FILE **def*)
Return the `FILE*` associated with the object *name* in the `sys` module, or *def* if *name* is not in the module or is not associated with a `FILE*`.

int **PySys_SetObject** (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.

void **PySys_AddWarnOption** (char **s*)
Append *s* to `sys.warnoptions`.

void **PySys_SetPath** (char **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 above, but write to `sys.stderr` or *stderr* instead.

5.3 Process Control

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.

void **Py_Exit** (int *status*)
Exit the current process. This calls `Py_Finalize()` and then calls the standard C library function `exit(status)`.

int **Py_AtExit** (void (**func*)())
Register a cleanup function to be called by `Py_Finalize()`. 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*.

5.4 Importando Módulos

*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. Before Python 2.4, the module may still be created in the failure case — examine `sys.modules` to find out. Starting with Python 2.4, a failing import of a module no longer leaves the module in `sys.modules`.

Alterado na versão 2.4: Failing imports remove incomplete module objects.

Alterado na versão 2.6: Always uses absolute imports.

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

This version of `PyImport_ImportModule()` does not block. It's intended to be used in C functions that import other modules to execute a function. The import may block if another thread holds the import lock. The function `PyImport_ImportModuleNoBlock()` never blocks. It first tries to fetch the module from sys.modules and falls back to `PyImport_ImportModule()` unless the lock is held, in which case the function will raise an `ImportError`.

Novo na versão 2.6.

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

Return value: New reference. Importa um módulo. Isso é melhor descrito referindo-se à função interna do Python `__import__()`, já que a função padrão `__import__()` chama essa função diretamente.

The return value is a new reference to the imported module or top-level package, or *NULL* with an exception set on failure (before Python 2.4, the module may still be created in this case). 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.

Alterado na versão 2.4: Failing imports remove incomplete module objects.

Alterado na versão 2.6: The function is an alias for `PyImport_ImportModuleLevel()` with `-1` as level, meaning relative import.

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

Return value: New reference. Importa um módulo. Isso é melhor descrito referindo-se à função interna do Python `__import__()`, já que a função padrão `__import__()` chama essa função diretamente.

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.

Novo na versão 2.5.

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

Return value: New reference. This is a higher-level interface that calls the current "import hook function". 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, e.g. by `rexec` or `ihooks`.

Alterado na versão 2.6: Always uses absolute imports.

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

Return value: New reference. Reload a module. This is best described by referring to the built-in Python function `reload()`, as the standard `reload()` function calls this function directly. 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_AddModule**(const char **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.

Nota: Esta função não carrega ou importa o módulo; se o módulo não foi carregado, você receberá um objeto de módulo vazio. utilize `PyImport_ImportModule()` ou uma de suas variações para importar um módulo. Estruturas de pacotes implícitos através de um ponto no nome para a *name* não são criados se não estiverem presentes.

*PyObject** **PyImport_ExecCodeModule**(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. Before Python 2.4, the module could still be created in error cases. Starting with Python 2.4, *name* is removed from `sys.modules` in error cases, and 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 `__file__` attribute will be set to the code object's `co_filename`.

Esta função poderá recarregar o módulo se este já foi importado. Veja `PyImport_ReloadModule()` para forma desejada de recarregar um módulo.

Se *name* apontar para um nome com ponto no formato de `package.module`, qualquer estruturas de pacotes que não tenha sido criada não será mais criada.

Alterado na versão 2.4: *name* is removed from `sys.modules` in error cases.

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

Return value: New reference. Como `PyImport_ExecCodeModule()`, mas o atributo `__file__` do objeto módulo é definido como *pathname* se não for `NULL`.

long **PyImport_GetMagicNumber**()

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

*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_GetImporter**(*PyObject* **path*)

Return an importer 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 it should fall back to the built-in import mechanism. Cache the result in `sys.path_importer_cache`. Return a new reference to the importer object.

Novo na versão 2.6.

void **_PyImport_Init**()

Inicia o mecanismo de importação. Apenas para uso interno.

void **PyImport_Cleanup**()
Esvazia a tabela do módulo. Apenas para uso interno.

void **_PyImport_Fini**()
Finaliza o mecanismo de importação. Apenas para uso interno.

*PyObject** **_PyImport_FindExtension**(char *, char *)
For internal use only.

*PyObject** **_PyImport_FixupExtension**(char *, char *)
For internal use only.

int **PyImport_ImportFrozenModule**(char **name*)
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.)

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 {
    char *name;
    unsigned char *code;
    int size;
};
```

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

int **PyImport_AppendInittab**(const char **name*, void (**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. 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 {
    char *name;
    void (*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 should be called before `Py_Initialize()`.

5.5 Suporte a *marshalling* de dados

Essas rotinas permitem que o código C trabalhe com objetos serializados usando o mesmo formato de dados que o módulo `marshal`. Existem funções para gravar dados no formato de serialização e funções adicionais que podem ser usadas para ler os dados novamente. Os arquivos usados para armazenar dados empacotados devem ser abertos no modo binário.

Os valores numéricos são armazenados primeiro com o byte menos significativo.

The module supports two versions of the data format: version `0` is the historical version, version `1` (new in Python 2.4) shares interned strings in the file, and upon unmarshalling. Version 2 (new in Python 2.5) uses a binary format for floating point numbers. *Py_MARSHAL_VERSION* indicates the current file format (currently 2).

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

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

Alterado na versão 2.4: *version* indicates the file format.

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

Marshal a Python object, *value*, to *file*.

Alterado na versão 2.4: *version* indicates the file format.

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

Return value: New reference. Return a string object containing the marshalled representation of *value*.

Alterado na versão 2.4: *version* indicates the file format.

As seguintes funções permitem que os valores pós-*marshalling* sejam lidos novamente.

XXX What about error detection? It appears that reading past the end of the file will always result in a negative numeric value (where that's relevant), but it's not clear that negative values won't be handled properly when there's no error. What's the right way to tell? Should only non-negative values be written using these routines?

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

Retorna um `long` C do fluxo de dados em um `FILE*` aberto para leitura. Somente um valor de 32 bits pode ser lido usando essa função, independentemente do tamanho nativo de `long`.

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

Retorna um `short` C do fluxo de dados em um `FILE*` aberto para leitura. Somente um valor de 16 bits pode ser lido usando essa função, independentemente do tamanho nativo de `short`.

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

Return value: New reference. Return a Python object from the data stream in a `FILE*` opened for reading. On error, sets the appropriate exception (`EOFError` or `TypeError`) and returns *NULL*.

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

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

*PyObject** **PyMarshal_ReadObjectFromString**(char **string*, Py_ssize_t *len*)

Return value: New reference. Return a Python object from the data stream in a character buffer containing *len* bytes pointed to by *string*. On error, sets the appropriate exception (`EOFError` or `TypeError`) and returns *NULL*.

Alterado na versão 2.5: This function used an `int` type for *len*. This might require changes in your code for properly supporting 64-bit systems.

5.6 Análise de argumentos e construção de valores

Essas funções são úteis ao criar suas próprias funções e métodos. Informações adicionais e exemplos estão disponíveis em extending-index.

As três primeiras funções descritas, `PyArg_ParseTuple()`, `PyArg_ParseTupleAndKeywords()`, e `PyArg_Parse()`, fazem uso do *formato string* que são usados para informar a função sobre os argumentos esperados. O formato string faz uso de uma mesma sintaxe para cada uma dessas funções.

Uma string consiste em zero ou mais "unidades de formato". Uma unidade de formato descreve um objeto Python; geralmente é um único caractere ou uma sequência entre parênteses de unidades de formato. Com algumas exceções, uma unidade de formato que não é uma sequência entre parênteses normalmente corresponde a um único argumento de endereço para essas funções. Na descrição a seguir, o formulário citado é a unidade de formato; A entrada em parênteses ( ) é o tipo de objeto Python que corresponde à unidade de formato; E a entrada em colchetes [ ] é o tipo da variável(s) C cujo endereço deve ser passado.

These formats allow accessing an object as a contiguous chunk of memory. You don't have to provide raw storage for the returned unicode or bytes area. Also, you won't have to release any memory yourself, except with the `es`, `es#`, `et` and `et#` formats.

`s` (string or Unicode) [const char *] Convert a Python string or Unicode object to a C pointer to a character string. You must not provide storage for the string itself; a pointer to an existing string is stored into the character pointer variable whose address you pass. The C string is NUL-terminated. The Python string must not contain embedded NUL bytes; if it does, a `TypeError` exception is raised. Unicode objects are converted to C strings using the default encoding. If this conversion fails, a `UnicodeError` is raised.

`s#` (string, Unicode or any read buffer compatible object) [const char *, int (or `Py_ssize_t`, see below)] This variant on `s` stores into two C variables, the first one a pointer to a character string, the second one its length. In this case the Python string may contain embedded null bytes. Unicode objects pass back a pointer to the default encoded string version of the object if such a conversion is possible. All other read-buffer compatible objects pass back a reference to the raw internal data representation.

Starting with Python 2.5 the type of the length argument can be controlled by defining the macro `PY_SSIZE_T_CLEAN` before including `Python.h`. If the macro is defined, length is a `Py_ssize_t` rather than an int.

`s*` (string, Unicode, or any buffer compatible object) [Py_buffer] Similar to `s#`, this code fills a Py_buffer structure provided by the caller. The buffer gets locked, so that the caller can subsequently use the buffer even inside a `Py_BEGIN_ALLOW_THREADS` block; the caller is responsible for calling `PyBuffer_Release` with the structure after it has processed the data.

Novo na versão 2.6.

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

`z#` (string, Unicode, `None` or any read buffer compatible object) [const char *, int] This is to `s#` as `z` is to `s`.

`z*` (string, Unicode, `None` or any buffer compatible object) [Py_buffer] This is to `s*` as `z` is to `s`.

Novo na versão 2.6.

`u` (Unicode) [Py_UNICODE *] Convert a Python Unicode object to a C pointer to a NUL-terminated buffer of 16-bit Unicode (UTF-16) data. As with `s`, there is no need to provide storage for the Unicode data buffer; a pointer to the existing Unicode data is stored into the `Py_UNICODE` pointer variable whose address you pass.

`u#` (Unicode) [Py_UNICODE *, int] This variant on `u` stores into two C variables, the first one a pointer to a Unicode data buffer, the second one its length. Non-Unicode objects are handled by interpreting their read-buffer pointer as pointer to a `Py_UNICODE` array.

es (string, Unicode or character buffer compatible object) [const char *encoding, char **buffer] This variant on `s` is used for encoding Unicode and objects convertible to Unicode into a character buffer. It only works for encoded data without embedded NUL bytes.

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

`PyArg_ParseTuple()` alocará um buffer do tamanho necessário, copiará os dados codificados nesse buffer e ajustará **buffer* para referenciar o armazenamento recém-alocado. O chamador é responsável por chamar `PyMem_Free()` para liberar o buffer alocado após o uso.

et (string, Unicode or character buffer compatible object) [const char *encoding, char **buffer] Same as `es` except that 8-bit string objects are passed through without recoding them. Instead, the implementation assumes that the string object uses the encoding passed in as parameter.

es# (string, Unicode or character buffer compatible object) [const char *encoding, char **buffer, int *buffer_length] This variant on `s#` is used for encoding Unicode and objects convertible to Unicode into a character buffer. Unlike the `es` format, this variant allows input data which contains NUL characters.

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

Há dois modos de operação:

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

If **buffer* points to a non-*NULL* pointer (an already allocated buffer), `PyArg_ParseTuple()` will use this location as the buffer and interpret the initial value of **buffer_length* as the buffer size. It will then copy the encoded data into the buffer and NUL-terminate it. If the buffer is not large enough, a `TypeError` will be set. Note: starting from Python 3.6 a `ValueError` will be set.

Em ambos os casos, o **buffer_length* é definido como o comprimento dos dados codificados sem o byte NUL à direita.

et# (string, Unicode or character buffer compatible object) [const char *encoding, char **buffer, int *buffer_length] Same as `es#` except that string objects are passed through without recoding them. Instead, the implementation assumes that the string object uses the encoding passed in as parameter.

b (integer) [unsigned char] Converte um inteiro Python não negativo em um inteiro pequeno não assinado (unsigned tiny int), armazenado em um `unsigned char` do C

B (integer) [unsigned char] Converte um inteiro Python para um pequeno inteiro (tiny int) sem verificação de estouro, armazenado em um `unsigned char` do C.

Novo na versão 2.3.

h (integer) [short int] Converte um inteiro Python para um `short int` do C.

H (integer) [unsigned short int] Converte um inteiro Python para um `unsigned short int` do C, sem verificação de estouro.

Novo na versão 2.3.

i (integer) [int] Converte um inteiro Python para um `int` simples do C.

I (integer) [unsigned int] Converte um inteiro Python para um `unsigned int` do C, sem verificação de estouro.

Novo na versão 2.3.

l (integer) [long int] Converte um inteiro Python para um `long int` do C.

k (integer) [unsigned long] Convert a Python integer or long integer to a C `unsigned long` without overflow checking.

Novo na versão 2.3.

L (integer) [PY_LONG_LONG] Convert a Python integer to a C `long long`. This format is only available on platforms that support `long long` (or `_int64` on Windows).

K (integer) [unsigned PY_LONG_LONG] Convert a Python integer or long integer to a C `unsigned long long` without overflow checking. This format is only available on platforms that support `unsigned long long` (or `unsigned _int64` on Windows).

Novo na versão 2.3.

n (integer) [Py_ssize_t] Convert a Python integer or long integer to a C `Py_ssize_t`.

Novo na versão 2.5.

c (string of length 1) [char] Convert a Python character, represented as a string of length 1, to a C `char`.

f (float) [float] Converte um número de ponto flutuante Python para um `float` do C.

d (float) [double] Converte um número de ponto flutuante Python para um `double` do C.

D (complex) [Py_complex] Converte um número complexo Python para uma estrutura C `Py_complex`

O (objeto) [PyObject*] Store a Python object (without any conversion) in a C object pointer. The C program thus receives the actual object that was passed. The object's reference count is not increased. The pointer stored is not *NULL*.

O! (objeto) [*typeobject*, PyObject *] Armazena um objeto Python em um ponteiro de objeto C. Isso é similar `O`, mas usa dois argumentos C: o primeiro é o endereço de um objeto do tipo Python, o segundo é um endereço da variável C (de tipo `PyObject*`) no qual o ponteiro do objeto está armazenado. Se o objeto Python não tiver o tipo necessário, `TypeError` é levantado.

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

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

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

S (string) [PyStringObject *] Like `O` but requires that the Python object is a string object. Raises `TypeError` if the object is not a string object. The C variable may also be declared as `PyObject*`.

U (Unicode string) [PyUnicodeObject *] Like `O` but requires that the Python object is a Unicode object. Raises `TypeError` if the object is not a Unicode object. The C variable may also be declared as `PyObject*`.

t# (read-only character buffer) [char *, int] Like `s#`, but accepts any object which implements the read-only buffer interface. The `char*` variable is set to point to the first byte of the buffer, and the `int` is set to the length of the buffer. Only single-segment buffer objects are accepted; `TypeError` is raised for all others.

w (read-write character buffer) [char *] Similar to s, but accepts any object which implements the read-write buffer interface. The caller must determine the length of the buffer by other means, or use w# instead. Only single-segment buffer objects are accepted; TypeError is raised for all others.

w# (read-write character buffer) [char *, Py_ssize_t] Like s#, but accepts any object which implements the read-write buffer interface. The char * variable is set to point to the first byte of the buffer, and the Py_ssize_t is set to the length of the buffer. Only single-segment buffer objects are accepted; TypeError is raised for all others.

w* (read-write byte-oriented buffer) [Py_buffer] This is to w what s* is to s.

Novo na versão 2.6.

(items) (tuple) [*matching-items*] O objeto deve ser uma sequência Python cujo comprimento seja o número de unidades de formato em *items*. Os argumentos C devem corresponder às unidades de formato individuais em *items*. As unidades de formato para sequências podem ser aninhadas.

Nota: Prior to Python version 1.5.2, this format specifier only accepted a tuple containing the individual parameters, not an arbitrary sequence. Code which previously caused TypeError to be raised here may now proceed without an exception. This is not expected to be a problem for existing code.

It is possible to pass Python long integers where integers are requested; however no proper range checking is done — the most significant bits are silently truncated when the receiving field is too small to receive the value (actually, the semantics are inherited from downcasts in C — your mileage may vary).

Alguns outros caracteres possuem significados na string de formatação. Isso pode não ocorrer dentro de parênteses aninhados. Eles são:

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

: A lista de unidades de formatação acaba aqui; a string após os dois pontos é usada como o nome da função nas mensagens de erro (o "valor associado" da exceção que *PyArg_ParseTuple()* levanta).

; A lista de unidades de formatação acaba aqui; a string após o ponto e vírgula é usada como a mensagem de erro *ao invés* da mensagem de erro padrão. **:** e **;** se excluem mutuamente.

Note que quaisquer referências a objeto Python que são fornecidas ao chamador são referências *emprestadas*; não decremente a contagem de referências delas!

Argumentos adicionais passados para essas funções devem ser endereços de variáveis cujo tipo é determinado pela string de formatação; estes são usados para armazenar valores vindos da tupla de entrada. Existem alguns casos, como descrito na lista de unidades de formatação acima, onde esses parâmetros são usados como valores de entrada; eles devem concordar com o que é especificado para a unidade de formatação correspondente nesse caso.

Para a conversão funcionar, o objeto *arg* deve corresponder ao formato e o formato deve estar completo. Em caso de sucesso, as funções *PyArg_Parse*()* retornam verdadeiro, caso contrário retornam falso e levantam uma exceção apropriada. Quando as funções *PyArg_Parse*()* falham devido a uma falha de conversão em uma das unidades de formatação, as variáveis nos endereços correspondentes àquela unidade e às unidades de formatação seguintes são deixadas intocadas.

int **PyArg_ParseTuple**(*PyObject* **args*, const char **format*, ...)
Analisa os parâmetros de uma função que recebe apenas parâmetros posicionais em variáveis locais. Retorna verdadeiro em caso de sucesso; em caso de falha, retorna falso e levanta a exceção apropriada.

int **PyArg_VaParse**(*PyObject* **args*, const char **format*, va_list *vargs*)
Idêntico a *PyArg_ParseTuple()*, exceto que aceita uma va_list ao invés de um número variável de argumentos.

int **PyArg_ParseTupleAndKeywords** (*PyObject* **args*, *PyObject* **kw*, const char **format*, char **keywords[]*, ...)
Parse the parameters of a function that takes both positional and keyword parameters into local variables. Returns true on success; on failure, it returns false and raises the appropriate exception.

int **PyArg_VaParseTupleAndKeywords** (*PyObject* **args*, *PyObject* **kw*, const char **format*, char **keywords[]*, va_list *vargs*)
Idêntico a `PyArg_ParseTupleAndKeywords()`, exceto que aceita uma va_list ao invés de um número variável de argumentos.

int **PyArg_Parse** (*PyObject* **args*, const char **format*, ...)
Function used to deconstruct the argument lists of "old-style" functions — these are functions which use the `METH_OLDARGS` parameter parsing method. This is not recommended for use in parameter parsing in new code, and most code in the standard interpreter has been modified to no longer use this for that purpose. It does remain a convenient way to decompose other tuples, however, and may continue to be used for that purpose.

int **PyArg_UnpackTuple** (*PyObject* **args*, const char **name*, Py_ssize_t *min*, Py_ssize_t *max*, ...)
Uma forma mais simples de recuperar parâmetros que não usa a string de formatação para especificar os tipos dos argumentos. Funções que usam esse método para recuperar seus parâmetros devem ser declaradas como `METH_VARARGS` em tabelas de função ou método. A tupla contendo os parâmetros reais deve ser passada como *args*; ela deve ser realmente uma tupla. O comprimento da tupla deve ser pelo menos *min* e não mais que *max*; *min* e *max* podem ser iguais. Argumentos adicionais devem ser passados para a função, cada qual deve ser um ponteiro para uma variável `PyObject*`; estes serão preenchidos com os valores vindos de *args*; eles vão conter referências emprestadas. As variáveis que corresponderem a parâmetros opcionais não dados por *args* não serão preenchidas; estas devem ser inicializadas pelo chamador. Essa função retorna verdadeiro em caso de sucesso e falso se *args* não é uma tupla ou contém o número errado de elementos; uma exceção será definida se houve uma falha.

Este é um exemplo do uso dessa função, tirado das fontes do módulo auxiliar para referências fracas `_weakref`:

```
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;
}
```

A chamada à `PyArg_UnpackTuple()` neste exemplo é inteiramente equivalente à chamada para `PyArg_ParseTuple()`:

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

Novo na versão 2.2.

Alterado na versão 2.5: This function used an `int` type for *min* and *max*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **Py_BuildValue** (const char **format*, ...)
Return value: New reference. Create a new value based on a format string similar to those accepted by the `PyArg_Parse*()` family of functions and a sequence of values. Returns the value or *NULL* in the case of an error; an exception will be raised if *NULL* is returned.

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

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

Na descrição a seguir, a forma entre aspas é a unidade de formatação; a entrada em parênteses (arredondado) é o tipo do objeto Python que a unidade de formatação irá retornar; e a entrada em colchetes [quadrado] é o tipo do(s) valor(es) C a ser(em) passado(s).

Os caracteres de espaço, tab, dois pontos e vírgula são ignorados em strings de formatação (mas não dentro de unidades de formatação como s#). Isso pode ser usado para tornar strings de formatação longas um pouco mais legíveis.

s (string) [char *] Convert a null-terminated C string to a Python object. If the C string pointer is *NULL*, None is used.

s# (string) [char *, int] Convert a C string and its length to a Python object. If the C string pointer is *NULL*, the length is ignored and None is returned.

z (string or None) [char *] O mesmo de s.

z# (string or None) [char *, int] O mesmo de s#.

u (Unicode string) [Py_UNICODE *] Convert a null-terminated buffer of Unicode (UCS-2 or UCS-4) data to a Python Unicode object. If the Unicode buffer pointer is *NULL*, None is returned.

u# (Unicode string) [Py_UNICODE *, int] Convert a Unicode (UCS-2 or UCS-4) data buffer and its length to a Python Unicode object. If the Unicode buffer pointer is *NULL*, the length is ignored and None is returned.

i (integer) [int] Converte um simples int do C em um objeto inteiro do Python.

b (integer) [char] Converte um simples char do C em um objeto inteiro do Python.

h (integer) [short int] Converte um simples short int do C em um objeto inteiro do Python.

l (integer) [long int] Converte um long int do C em um objeto inteiro do Python.

B (integer) [unsigned char] Converte um unsigned char do C em um objeto inteiro do Python.

H (integer) [unsigned short int] Converte um unsigned short int do C em um objeto inteiro do Python.

I (integer/long) [unsigned int] Convert a C unsigned int to a Python integer object or a Python long integer object, if it is larger than sys.maxint.

k (integer/long) [unsigned long] Convert a C unsigned long to a Python integer object or a Python long integer object, if it is larger than sys.maxint.

L (long) [PY_LONG_LONG] Convert a C long long to a Python long integer object. Only available on platforms that support long long.

K (long) [unsigned PY_LONG_LONG] Convert a C unsigned long long to a Python long integer object. Only available on platforms that support unsigned long long.

n (int) [Py_ssize_t] Convert a C Py_ssize_t to a Python integer or long integer.

Novo na versão 2.5.

c (string of length 1) [char] Convert a C int representing a character to a Python string of length 1.

d (float) [double] Converte um double do C em um número ponto flutuante do Python.

f (float) [float] Same as d.

D (complex) [Py_complex *] Converte uma estrutura *Py_complex* do C em um número complexo do Python.

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

S (objeto) [PyObject *] O mesmo que O.

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

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

(items) (tuple) [*matching-items*] Converte uma sequência de valores C para uma tupla Python com o mesmo número de itens.

[items] (list) [*matching-items*] Converte uma sequência de valores C para uma lista Python com o mesmo número de itens.

{items} (dictionary) [*matching-items*] Converte uma sequência de valores C para um dicionário Python. Cada par de valores consecutivos do C adiciona um item ao dicionário, servindo como chave e valor, respectivamente.

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

*PyObject** **Py_VaBuildValue**(const char **format*, va_list *vargs*)
Idêntico a *Py_BuildValue()*, exceto que aceita uma va_list ao invés de um número variável de argumentos.

5.7 Conversão de Strings e Formação

Funções para conversão de números e saída formatada de Strings.

int **PyOS_snprintf**(char **str*, size_t *size*, const char **format*, ...)
Output not more than *size* bytes to *str* according to the format string *format* and the extra arguments. See the Unix man page *snprintf(2)*.

int **PyOS_vsnprintf**(char **str*, size_t *size*, const char **format*, va_list *va*)
Output not more than *size* bytes to *str* according to the format string *format* and the variable argument list *va*. Unix man page *vsnprintf(2)*.

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

The wrappers ensure that *str*[*size*-1] is always '\0' upon return. They never write more than *size* bytes (including the trailing '\0' into str. Both functions require that str != NULL, size > 0 and format != NULL.

If the platform doesn't have vsnprintf() and the buffer size needed to avoid truncation exceeds *size* by more than 512 bytes, Python aborts with a *Py_FatalError*.

The return value (*rv*) for these functions should be interpreted as follows:

- When 0 <= rv < size, the output conversion was successful and *rv* characters were written to *str* (excluding the trailing '\0' byte at *str*[*rv*]).
- When rv >= size, the output conversion was truncated and a buffer with rv + 1 bytes would have been needed to succeed. *str*[*size*-1] is '\0' in this case.

- When `rv < 0`, "something bad happened." *str*[*size*-1] is `'\0'` in this case too, but the rest of *str* is undefined. The exact cause of the error depends on the underlying platform.

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

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

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

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

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

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

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

Novo na versão 2.7.

double **PyOS_ascii_strtod**(const char **nptr*, char ***endptr*)

Convert a string to a `double`. This function behaves like the Standard C function `strtod()` does in the C locale. It does this without changing the current locale, since that would not be thread-safe.

`PyOS_ascii_strtod()` should typically be used for reading configuration files or other non-user input that should be locale independent.

See the Unix man page *`strtod(2)`* for details.

Novo na versão 2.4.

Obsoleto desde a versão 2.7: Use *`PyOS_string_to_double()`* instead.

char* **PyOS_ascii_formatd**(char **buffer*, size_t *buf_len*, const char **format*, double *d*)

Convert a `double` to a string using the `'.'` as the decimal separator. *format* is a `printf()`-style format string specifying the number format. Allowed conversion characters are `'e'`, `'E'`, `'f'`, `'F'`, `'g'` and `'G'`.

The return value is a pointer to *buffer* with the converted string or NULL if the conversion failed.

Novo na versão 2.4.

Obsoleto desde a versão 2.7: This function is removed in Python 2.7 and 3.1. Use `PyOS_double_to_string()` instead.

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

Convert a `double` *val* to a string using supplied *format_code*, *precision*, and *flags*.

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

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

- *Py_DTSF_SIGN* means to always precede the returned string with a sign character, even if *val* is non-negative.
- *Py_DTSF_ADD_DOT_0* means to ensure that the returned string will not look like an integer.

- *Py_DTSF_ALT* means to apply "alternate" formatting rules. See the documentation for the `PyOS_snprintf()` `'#'` specifier for details.

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

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

Novo na versão 2.7.

double **PyOS_ascii_atof**(const char **nptr*)

Convert a string to a `double` in a locale-independent way.

See the Unix man page *atof(2)* for details.

Novo na versão 2.4.

Obsoleto desde a versão 3.1: Use `PyOS_string_to_double()` instead.

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

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

Novo na versão 2.6.

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

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

Novo na versão 2.6.

5.8 Reflexão

*PyObject** **PyEval_GetBuiltins**()

Return value: Borrowed reference. Retorna um dicionário dos componentes internos no quadro de execução atual ou o interpretador do estado do encadeamento, se nenhum quadro estiver em execução no momento.

*PyObject** **PyEval_GetLocals**()

Return value: Borrowed reference. Return a dictionary of the local variables in the current execution frame, or *NULL* if no frame is currently executing.

*PyObject** **PyEval_GetGlobals**()

Return value: Borrowed reference. Return a dictionary of the global variables in the current execution frame, or *NULL* if no frame is currently executing.

PyFrameObject* **PyEval_GetFrame**()

Return value: Borrowed reference. Return the current thread state's frame, which is *NULL* if no frame is currently executing.

int **PyFrame_GetLineNumber**(PyFrameObject **frame*)

Retorna o número da linha do *frame* atualmente em execução.

int **PyEval_GetRestricted**()

If there is a current frame and it is executing in restricted mode, return true, otherwise false.

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

Retorna o nome de *func* se for uma função, classe ou objeto de instância, senão o nome do tipo da *func*.

const char* **PyEval_GetFuncDesc**(*PyObject* *func*)
Retorna uma sequência de caracteres de descrição, dependendo do tipo de *func*. Os valores de retorno incluem "()" para funções e métodos, "construtor", "instância" e "objeto". Concatenado com o resultado de `PyEval_GetFuncName()`, o resultado será uma descrição de *func*.

5.9 Registro de codec e funções de suporte

int **PyCodec_Register**(*PyObject* *search_function*)
Registra uma nova função de busca de codec.

Como efeito colateral, tenta carregar o pacote `encodings`, se isso ainda não tiver sido feito, com o propósito de garantir que ele sempre seja o primeiro na lista de funções de busca.

int **PyCodec_KnownEncoding**(const char *encoding*)
Return `1` or `0` depending on whether there is a registered codec for the given *encoding*.

*PyObject** **PyCodec_Encode**(*PyObject* *object*, const char *encoding*, const char *errors*)
API de codificação baseada em codec genérico.

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

*PyObject** **PyCodec_Decode**(*PyObject* *object*, const char *encoding*, const char *errors*)
API de decodificação baseada em decodificador genérico.

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

5.9.1 API de pesquisa de codec

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

*PyObject** **PyCodec_Encoder**(const char *encoding*)
Obtém uma função de codificação para o *encoding* dado.

*PyObject** **PyCodec_Decoder**(const char *encoding*)
Obtém uma função de decodificação para o *encoding* dado.

*PyObject** **PyCodec_IncrementalEncoder**(const char *encoding*, const char *errors*)
Obtém um objeto `IncrementalEncoder` para o *encoding* dado.

*PyObject** **PyCodec_IncrementalDecoder**(const char *encoding*, const char *errors*)
Obtém um objeto `IncrementalDecoder` para o *encoding* dado.

*PyObject** **PyCodec_StreamReader**(const char *encoding*, *PyObject* *stream*, const char *errors*)
Obtém uma função de fábrica `StreamReader` para o *encoding* dado.

*PyObject** **PyCodec_StreamWriter**(const char *encoding*, *PyObject* *stream*, const char *errors*)
Obtém uma função de fábrica `StreamWriter` para o *encoding* dado.

5.9.2 API de registro de tratamentos de erros de decodificação Unicode

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

Registra a função de retorno de chamada de tratamento de *erro* para o *nome* fornecido. Esta chamada de função é invocada por um codificador quando encontra caracteres/bytes indecodificáveis e *nome* é especificado como o parâmetro de erro na chamada da função de codificação/decodificação.

O retorno de chamada obtém um único argumento, uma instância de `UnicodeEncodeError`, `UnicodeDecodeError` ou `UnicodeTranslateError` que contém informações sobre a sequencia problemática de caracteres ou bytes e seu deslocamento na string original (consulte *Objetos de exceção Unicode* para funções que extraem essa informação). A função de retorno de chamada deve levantar a exceção dada, ou retornar uma tupla de dois itens contendo a substituição para a sequência problemática, e um inteiro fornecendo o deslocamento na string original na qual a codificação/decodificação deve ser retomada.

Retorna“0“ em caso de sucesso, `-1` em caso de erro.

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

Lookup the error handling callback function registered under *name*. As a special case *NULL* can be passed, in which case the error handling callback for "strict" will be returned.

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

Levanta *exc* como uma exceção.

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

Ignora o erro de unicode, ignorando a entrada que causou o erro.

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

Substitui o erro de unicode por `?` ou `U+FFFD`.

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

Substitui o erro de unicode por caracteres da referência XML.

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

Substitui o erro de unicode com escapes de barra invertida (`\x`, `\u` e `\U`).

CAPÍTULO 6

Camada de Abstração de Objetos

As funções deste capítulo interagem com objetos Python de acordo com a sua tipagem, ou de acordo com as classes dos tipos de objetos (Exemplo: todos os tipos numéricos, todos os tipos de sequência). Quando usado em um objeto de um tipo que não se aplica será levantada uma exceção do Python.

Não é possível usar essas funções em objetos que não estão propriamente inicializados, tal como uma objeto de lista que foi criado por `PyList_New()`, mas do qual os itens não foram definidos como algum valor não `NULL` ainda.

6.1 Protocolo de Objeto

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

Print an object *o*, on file *fp*. Returns `-1` on error. The flags argument is used to enable certain printing options. The only option currently supported is `Py_PRINT_RAW`; if given, the `str()` of the object is written instead of the `repr()`.

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

Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This is equivalent to the Python expression `hasattr(o, attr_name)`. This function always succeeds.

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

Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This is equivalent to the Python expression `hasattr(o, attr_name)`. This function always succeeds.

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

Return value: New reference. Retrieve an attribute named *attr_name* from object *o*. Returns the attribute value on success, or *NULL* on failure. This is the equivalent of the Python expression `o.attr_name`.

*PyObject** **PyObject_GetAttrString**(*PyObject *o*, const char **attr_name*)

Return value: New reference. Retrieve an attribute named *attr_name* from object *o*. Returns the attribute value on success, or *NULL* on failure. This is the equivalent of the Python expression `o.attr_name`.

*PyObject** **PyObject_GenericGetAttr**(*PyObject *o*, *PyObject *name*)

Generic attribute getter function that is meant to be put into a type object's `tp_getattro` slot. It looks for a descriptor in the dictionary of classes in the object's MRO as well as an attribute in the object's `__dict__` (if present). As outlined in descriptors, data descriptors take preference over instance attributes, while non-data descriptors don't. Otherwise, an `AttributeError` is raised.

int **PyObject_SetAttr** (*PyObject* **o*, *PyObject* **attr_name*, *PyObject* **v*)
Set the value of the attribute named *attr_name*, for object *o*, to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o.attr_name = v`.

If *v* is *NULL*, the attribute is deleted, however this feature is deprecated in favour of using `PyObject_DelAttr()`.

int **PyObject_SetAttrString** (*PyObject* **o*, const char **attr_name*, *PyObject* **v*)
Set the value of the attribute named *attr_name*, for object *o*, to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o.attr_name = v`.

If *v* is *NULL*, the attribute is deleted, however this feature is deprecated in favour of using `PyObject_DelAttrString()`.

int **PyObject_GenericSetAttr** (*PyObject* **o*, *PyObject* **name*, *PyObject* **value*)
Generic attribute setter and deleter function that is meant to be put into a type object's `tp_setattro` slot. It looks for a data descriptor in the dictionary of classes in the object's MRO, and if found it takes preference over setting or deleting the attribute in the instance dictionary. Otherwise, the attribute is set or deleted in the object's `__dict__` (if present). On success, `0` is returned, otherwise an `AttributeError` is raised and `-1` is returned.

int **PyObject_DelAttr** (*PyObject* **o*, *PyObject* **attr_name*)
Delete attribute named *attr_name*, for object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o.attr_name`.

int **PyObject_DelAttrString** (*PyObject* **o*, const char **attr_name*)
Delete attribute named *attr_name*, for object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o.attr_name`.

*PyObject** **PyObject_RichCompare** (*PyObject* **o1*, *PyObject* **o2*, int *opid*)
Return value: New reference. Compare the values of *o1* and *o2* using the operation specified by *opid*, which must be one of `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, or `Py_GE`, corresponding to <, <=, ==, !=, >, or >= respectively. This is the equivalent of the Python expression `o1 op o2`, where `op` is the operator corresponding to *opid*. Returns the value of the comparison on success, or *NULL* on failure.

int **PyObject_RichCompareBool** (*PyObject* **o1*, *PyObject* **o2*, int *opid*)
Compare the values of *o1* and *o2* using the operation specified by *opid*, which must be one of `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, or `Py_GE`, corresponding to <, <=, ==, !=, >, or >= respectively. Returns `-1` on error, `0` if the result is false, `1` otherwise. This is the equivalent of the Python expression `o1 op o2`, where `op` is the operator corresponding to *opid*.

Nota: If *o1* and *o2* are the same object, `PyObject_RichCompareBool()` will always return `1` for `Py_EQ` and `0` for `Py_NE`.

int **PyObject_Cmp** (*PyObject* **o1*, *PyObject* **o2*, int **result*)
Compare the values of *o1* and *o2* using a routine provided by *o1*, if one exists, otherwise with a routine provided by *o2*. The result of the comparison is returned in *result*. Returns `-1` on failure. This is the equivalent of the Python statement `result = cmp(o1, o2)`.

int **PyObject_Compare** (*PyObject* **o1*, *PyObject* **o2*)
Compare the values of *o1* and *o2* using a routine provided by *o1*, if one exists, otherwise with a routine provided by *o2*. Returns the result of the comparison on success. On error, the value returned is undefined; use `PyErr_Occurred()` to detect an error. This is equivalent to the Python expression `cmp(o1, o2)`.

*PyObject** **PyObject_Repr** (*PyObject* **o*)
Return value: New reference. Compute a string representation of object *o*. Returns the string representation on

success, *NULL* on failure. This is the equivalent of the Python expression `repr(o)`. Called by the `repr()` built-in function and by reverse quotes.

*PyObject** **PyObject_Str**(*PyObject *o*)
Return value: New reference. Compute a string representation of object *o*. Returns the string representation on success, *NULL* on failure. This is the equivalent of the Python expression `str(o)`. Called by the `str()` built-in function and by the `print` statement.

*PyObject** **PyObject_Bytes**(*PyObject *o*)
Compute a bytes representation of object *o*. In 2.x, this is just an alias for `PyObject_Str()`.

*PyObject** **PyObject_Unicode**(*PyObject *o*)
Return value: New reference. Compute a Unicode string representation of object *o*. Returns the Unicode string representation on success, *NULL* on failure. This is the equivalent of the Python expression `unicode(o)`. Called by the `unicode()` built-in function.

int **PyObject_IsInstance**(*PyObject *inst*, *PyObject *cls*)
Returns `1` if *inst* is an instance of the class *cls* or a subclass of *cls*, or `0` if not. On error, returns `-1` and sets an exception. If *cls* is a type object rather than a class object, `PyObject_IsInstance()` returns `1` if *inst* is of type *cls*. If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `0`. If *inst* is not a class instance and *cls* is neither a type object, nor a class object, nor a tuple, *inst* must have a `__class__` attribute — the class relationship of the value of that attribute with *cls* will be used to determine the result of this function.

Novo na versão 2.1.

Alterado na versão 2.2: Support for a tuple as the second argument added.

Subclass determination is done in a fairly straightforward way, but includes a wrinkle that implementors of extensions to the class system may want to be aware of. If `A` and `B` are class objects, `B` is a subclass of `A` if it inherits from `A` either directly or indirectly. If either is not a class object, a more general mechanism is used to determine the class relationship of the two objects. When testing if *B* is a subclass of *A*, if *A* is *B*, `PyObject_IsSubclass()` returns true. If *A* and *B* are different objects, *B*'s `__bases__` attribute is searched in a depth-first fashion for *A* — the presence of the `__bases__` attribute is considered sufficient for this determination.

int **PyObject_IsSubclass**(*PyObject *derived*, *PyObject *cls*)
Returns `1` if the class *derived* is identical to or derived from the class *cls*, otherwise returns `0`. In case of an error, returns `-1`. If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `0`. If either *derived* or *cls* is not an actual class object (or tuple), this function uses the generic algorithm described above.

Novo na versão 2.1.

Alterado na versão 2.3: Older versions of Python did not support a tuple as the second argument.

int **PyCallable_Check**(*PyObject *o*)
Determine if the object *o* is callable. Return `1` if the object is callable and `0` otherwise. This function always succeeds.

*PyObject** **PyObject_Call**(*PyObject *callable_object*, *PyObject *args*, *PyObject *kw*)
Return value: New reference. Call a callable Python object *callable_object*, with arguments given by the tuple *args*, and named arguments given by the dictionary *kw*. If no named arguments are needed, *kw* may be *NULL*. *args* must not be *NULL*, use an empty tuple if no arguments are needed. Returns the result of the call on success, or *NULL* on failure. This is the equivalent of the Python expression `apply(callable_object, args, kw)` or `callable_object(*args, **kw)`.

Novo na versão 2.2.

*PyObject** **PyObject_CallObject**(*PyObject *callable_object*, *PyObject *args*)
Return value: New reference. Call a callable Python object *callable_object*, with arguments given by the tuple *args*. If no arguments are needed, then *args* may be *NULL*. Returns the result of the call on success, or

NULL on failure. This is the equivalent of the Python expression `apply(callable_object, args)` or `callable_object(*args)`.

*PyObject** **PyObject_CallFunction**(*PyObject* **callable*, char **format*, ...)
Return value: New reference. Call a callable Python object *callable*, with a variable number of C arguments. The C arguments are described using a `Py_BuildValue()` style format string. The format may be *NULL*, indicating that no arguments are provided. Returns the result of the call on success, or *NULL* on failure. This is the equivalent of the Python expression `apply(callable, args)` or `callable(*args)`. Note that if you only pass `PyObject *` args, `PyObject_CallFunctionObjArgs()` is a faster alternative.

*PyObject** **PyObject_CallMethod**(*PyObject* **o*, char **method*, char **format*, ...)
Return value: New reference. Call the method named *method* of object *o* with a variable number of C arguments. The C arguments are described by a `Py_BuildValue()` format string that should produce a tuple. The format may be *NULL*, indicating that no arguments are provided. Returns the result of the call on success, or *NULL* on failure. This is the equivalent of the Python expression `o.method(args)`. Note that if you only pass `PyObject *` args, `PyObject_CallMethodObjArgs()` is a faster alternative.

*PyObject** **PyObject_CallFunctionObjArgs**(*PyObject* **callable*, ..., NULL)
Return value: New reference. Call a callable Python object *callable*, with a variable number of `PyObject*` arguments. The arguments are provided as a variable number of parameters followed by *NULL*. Returns the result of the call on success, or *NULL* on failure.

Novo na versão 2.2.

*PyObject** **PyObject_CallMethodObjArgs**(*PyObject* **o*, *PyObject* **name*, ..., NULL)
Return value: New reference. Calls a method of the object *o*, where the name of the method is given as a Python string object in *name*. It is called with a variable number of `PyObject*` arguments. The arguments are provided as a variable number of parameters followed by *NULL*. Returns the result of the call on success, or *NULL* on failure.

Novo na versão 2.2.

long **PyObject_Hash**(*PyObject* **o*)
Compute and return the hash value of an object *o*. On failure, return `-1`. This is the equivalent of the Python expression `hash(o)`.

long **PyObject_HashNotImplemented**(*PyObject* **o*)
Set a `TypeError` indicating that `type(o)` is not hashable and return `-1`. This function receives special treatment when stored in a `tp_hash` slot, allowing a type to explicitly indicate to the interpreter that it is not hashable.

Novo na versão 2.6.

int **PyObject_IsTrue**(*PyObject* **o*)
Returns `1` if the object *o* is considered to be true, and `0` otherwise. This is equivalent to the Python expression `not not o`. On failure, return `-1`.

int **PyObject_Not**(*PyObject* **o*)
Returns `0` if the object *o* is considered to be true, and `1` otherwise. This is equivalent to the Python expression `not o`. On failure, return `-1`.

*PyObject** **PyObject_Type**(*PyObject* **o*)
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 common expression `o->ob_type`, which returns a pointer of type `PyTypeObject*`, except when the incremented reference count is needed.

int **PyObject_TypeCheck**(*PyObject* **o*, *PyTypeObject* **type*)
Return true if the object *o* is of type *type* or a subtype of *type*. Both parameters must be non-*NULL*.

Novo na versão 2.2.

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

Return the length of object *o*. If the object *o* provides either the sequence and mapping protocols, the sequence length is returned. On error, `-1` is returned. This is the equivalent to the Python expression `len(o)`.

Alterado na versão 2.5: These functions returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

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

Return value: New reference. Return element of *o* corresponding to the object *key* or *NULL* on failure. This is the equivalent of the Python expression `o[key]`.

int **PyObject_SetItem**(*PyObject *o*, *PyObject *key*, *PyObject *v*)

Map the object *key* to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o[key] = v`.

int **PyObject_DelItem**(*PyObject *o*, *PyObject *key*)

Delete the mapping for *key* from *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o[key]`.

int **PyObject_AsFileDescriptor**(*PyObject *o*)

Derives a file descriptor from a Python object. If the object is an integer or long integer, its value is returned. If not, the object's `fileno()` method is called if it exists; the method must return an integer or long integer, which is returned as the file descriptor value. Returns `-1` on failure.

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

Return value: New reference. This is equivalent to the Python expression `dir(o)`, returning a (possibly empty) list of strings appropriate for the object argument, or *NULL* if there was an error. If the argument is *NULL*, this is like the Python `dir()`, returning the names of the current locals; in this case, if no execution frame is active then *NULL* is returned but `PyErr_Occurred()` will return false.

*PyObject** **PyObject_GetIter**(*PyObject *o*)

Return value: New reference. This is equivalent to the Python expression `iter(o)`. It returns a new iterator for the object argument, or the object itself if the object is already an iterator. Raises `TypeError` and returns *NULL* if the object cannot be iterated.

6.2 Número de Protocolo

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

Retorna `1` se o objeto *o* prover um número de protocolo, caso contrário, retorna falso. Esta função sempre tem sucesso.

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

Return value: New reference. Returns the result of adding *o1* and *o2*, or *NULL* on failure. This is the equivalent of the Python expression `o1 + o2`.

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

Return value: New reference. Returns the result of subtracting *o2* from *o1*, or *NULL* on failure. This is the equivalent of the Python expression `o1 - o2`.

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

Return value: New reference. Returns the result of multiplying *o1* and *o2*, or *NULL* on failure. This is the equivalent of the Python expression `o1 * o2`.

*PyObject** **PyNumber_Divide**(*PyObject *o1*, *PyObject *o2*)

Return value: New reference. Returns the result of dividing *o1* by *o2*, or *NULL* on failure. This is the equivalent of the Python expression `o1 / o2`.

*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 equivalent to the "classic" division of integers.

Novo na versão 2.2.

*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.

Novo na versão 2.2.

*PyObject** **PyNumber_Remainder**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the remainder of dividing *o1* by *o2*, or *NULL* on failure. This is the equivalent of the Python expression `o1 % o2`.

*PyObject** **PyNumber_Divmod**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. See the built-in function `divmod()`. Returns *NULL* on failure. This is the equivalent of the Python expression `divmod(o1, o2)`.

*PyObject** **PyNumber_Power**(*PyObject *o1*, *PyObject *o2*, *PyObject *o3*)
Return value: New reference. See the built-in function `pow()`. Returns *NULL* on failure. This is the equivalent of the Python expression `pow(o1, o2, o3)`, where *o3* is optional. If *o3* is to be ignored, pass `Py_None` in its place (passing *NULL* for *o3* would cause an illegal memory access).

*PyObject** **PyNumber_Negative**(*PyObject *o*)
Return value: New reference. Returns the negation of *o* on success, or *NULL* on failure. This is the equivalent of the Python expression `-o`.

*PyObject** **PyNumber_Positive**(*PyObject *o*)
Return value: New reference. Returns *o* on success, or *NULL* on failure. This is the equivalent of the Python expression `+o`.

*PyObject** **PyNumber_Absolute**(*PyObject *o*)
Return value: New reference. Returns the absolute value of *o*, or *NULL* on failure. This is the equivalent of the Python expression `abs(o)`.

*PyObject** **PyNumber_Invert**(*PyObject *o*)
Return value: New reference. Returns the bitwise negation of *o* on success, or *NULL* on failure. This is the equivalent of the Python expression `~o`.

*PyObject** **PyNumber_Lshift**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of left shifting *o1* by *o2* on success, or *NULL* on failure. This is the equivalent of the Python expression `o1 << o2`.

*PyObject** **PyNumber_Rshift**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of right shifting *o1* by *o2* on success, or *NULL* on failure. This is the equivalent of the Python expression `o1 >> o2`.

*PyObject** **PyNumber_And**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the "bitwise and" of *o1* and *o2* on success and *NULL* on failure. This is the equivalent of the Python expression `o1 & o2`.

*PyObject** **PyNumber_Xor**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the "bitwise exclusive or" of *o1* by *o2* on success, or *NULL* on failure. This is the equivalent of the Python expression `o1 ^ o2`.

*PyObject** **PyNumber_Or**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the "bitwise or" of *o1* and *o2* on success, or *NULL* on failure. This is the

equivalent of the Python expression `o1 | o2`.

*PyObject** **PyNumber_InPlaceAdd**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of adding *o1* and *o2*, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 += o2`.

*PyObject** **PyNumber_InPlaceSubtract**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of subtracting *o2* from *o1*, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 -= o2`.

*PyObject** **PyNumber_InPlaceMultiply**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of multiplying *o1* and *o2*, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 *= o2`.

*PyObject** **PyNumber_InPlaceDivide**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of dividing *o1* by *o2*, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 /= o2`.

*PyObject** **PyNumber_InPlaceFloorDivide**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the mathematical floor of dividing *o1* by *o2*, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 //= o2`.

Novo na versão 2.2.

*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.

Novo na versão 2.2.

*PyObject** **PyNumber_InPlaceRemainder**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the remainder of dividing *o1* by *o2*, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 %= o2`.

*PyObject** **PyNumber_InPlacePower**(*PyObject *o1*, *PyObject *o2*, *PyObject *o3*)
Return value: New reference. See the built-in function `pow()`. Returns *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 **= o2` when o3 is `Py_None`, or an in-place variant of `pow(o1, o2, o3)` otherwise. If *o3* is to be ignored, pass `Py_None` in its place (passing *NULL* for *o3* would cause an illegal memory access).

*PyObject** **PyNumber_InPlaceLshift**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of left shifting *o1* by *o2* on success, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 <<= o2`.

*PyObject** **PyNumber_InPlaceRshift**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the result of right shifting *o1* by *o2* on success, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 >>= o2`.

*PyObject** **PyNumber_InPlaceAnd**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the "bitwise and" of *o1* and *o2* on success and *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 &= o2`.

*PyObject** **PyNumber_InPlaceXor**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the "bitwise exclusive or" of *o1* by *o2* on success, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 ^= o2`.

*PyObject** **PyNumber_InPlaceOr**(*PyObject *o1*, *PyObject *o2*)
Return value: New reference. Returns the "bitwise or" of *o1* and *o2* on success, or *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 |= o2`.

int **PyNumber_Coerce**(*PyObject* ***p1*, *PyObject* ***p2*)

This function takes the addresses of two variables of type `PyObject*`. If the objects pointed to by `*p1` and `*p2` have the same type, increment their reference count and return `0` (success). If the objects can be converted to a common numeric type, replace `*p1` and `*p2` by their converted value (with 'new' reference counts), and return `0`. If no conversion is possible, or if some other error occurs, return `-1` (failure) and don't increment the reference counts. The call `PyNumber_Coerce(&o1, &o2)` is equivalent to the Python statement `o1, o2 = coerce(o1, o2)`.

int **PyNumber_CoerceEx**(*PyObject* ***p1*, *PyObject* ***p2*)

This function is similar to `PyNumber_Coerce()`, except that it returns `1` when the conversion is not possible and when no error is raised. Reference counts are still not increased in this case.

*PyObject** **PyNumber_Int**(*PyObject* **o*)

Return value: New reference. Returns the *o* converted to an integer object on success, or *NULL* on failure. If the argument is outside the integer range a long object will be returned instead. This is the equivalent of the Python expression `int(o)`.

*PyObject** **PyNumber_Long**(*PyObject* **o*)

Return value: New reference. Returns the *o* converted to a long integer object on success, or *NULL* on failure. This is the equivalent of the Python expression `long(o)`.

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

Return value: New reference. Returns the *o* converted to a float object on success, or *NULL* on failure. This is the equivalent of the Python expression `float(o)`.

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

Returns the *o* converted to a Python int or long on success or *NULL* with a `TypeError` exception raised on failure.

Novo na versão 2.5.

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

Returns the integer *n* converted to *base* as a string with a base marker of `'0b'`, `'0o'`, or `'0x'` if applicable. When *base* is not 2, 8, 10, or 16, the format is `'x#num'` where x is the base. If *n* is not an int object, it is converted with `PyNumber_Index()` first.

Novo na versão 2.6.

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 *o* can be converted to a Python int or long 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.

Novo na versão 2.5.

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.

Novo na versão 2.5.

6.3 Protocolo de Sequência

int **PySequence_Check** (*PyObject* *o)
Return `1` if the object provides sequence protocol, and `0` otherwise. This function always succeeds.

Py_ssize_t **PySequence_Size** (*PyObject* *o)
Py_ssize_t **PySequence_Length** (*PyObject* *o)
Returns the number of objects in sequence *o* on success, and `-1` on failure. This is equivalent to the Python expression `len(o)`.

Alterado na versão 2.5: These functions returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PySequence_Concat** (*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Return the concatenation of *o1* and *o2* on success, and *NULL* on failure. This is the equivalent of the Python expression `o1 + o2`.

*PyObject** **PySequence_Repeat** (*PyObject* *o, Py_ssize_t *count*)
Return value: New reference. Return the result of repeating sequence object *o* *count* times, or *NULL* on failure. This is the equivalent of the Python expression `o * count`.

Alterado na versão 2.5: This function used an `int` type for *count*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PySequence_InPlaceConcat** (*PyObject* *o1, *PyObject* *o2)
Return value: New reference. Return the concatenation of *o1* and *o2* on success, and *NULL* on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python expression `o1 += o2`.

*PyObject** **PySequence_InPlaceRepeat** (*PyObject* *o, Py_ssize_t *count*)
Return value: New reference. Return the result of repeating sequence object *o* *count* times, or *NULL* on failure. The operation is done *in-place* when *o* supports it. This is the equivalent of the Python expression `o *= count`.

Alterado na versão 2.5: This function used an `int` type for *count*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PySequence_GetItem** (*PyObject* *o, Py_ssize_t *i*)
Return value: New reference. Return the *i*th element of *o*, or *NULL* on failure. This is the equivalent of the Python expression `o[i]`.

Alterado na versão 2.5: This function used an `int` type for *i*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PySequence_GetSlice** (*PyObject* *o, Py_ssize_t *i1*, Py_ssize_t *i2*)
Return value: New reference. Return the slice of sequence object *o* between *i1* and *i2*, or *NULL* on failure. This is the equivalent of the Python expression `o[i1:i2]`.

Alterado na versão 2.5: This function used an `int` type for *i1* and *i2*. This might require changes in your code for properly supporting 64-bit systems.

int **PySequence_SetItem** (*PyObject* *o, Py_ssize_t *i*, *PyObject* *v)
Assign object *v* to the *i*th element of *o*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o[i] = v`. This function *does not* steal a reference to *v*.

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

Alterado na versão 2.5: This function used an `int` type for *i*. This might require changes in your code for properly supporting 64-bit systems.

int **PySequence_DelItem**(*PyObject* *o, Py_ssize_t *i*)

Delete the *i*th element of object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `del o[i]`.

Alterado na versão 2.5: This function used an `int` type for *i*. This might require changes in your code for properly supporting 64-bit systems.

int **PySequence_SetSlice**(*PyObject* *o, Py_ssize_t *i1*, Py_ssize_t *i2*, *PyObject* *v)

Assign the sequence object *v* to the slice in sequence object *o* from *i1* to *i2*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o[i1:i2] = v`.

If *v* is *NULL*, the slice is deleted, however this feature is deprecated in favour of using `PySequence_DelSlice()`.

Alterado na versão 2.5: This function used an `int` type for *i1* and *i2*. This might require changes in your code for properly supporting 64-bit systems.

int **PySequence_DelSlice**(*PyObject* *o, Py_ssize_t *i1*, Py_ssize_t *i2*)

Delete the slice in sequence object *o* from *i1* to *i2*. Returns `-1` on failure. This is the equivalent of the Python statement `del o[i1:i2]`.

Alterado na versão 2.5: This function used an `int` type for *i1* and *i2*. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t **PySequence_Count**(*PyObject* *o, *PyObject* *value)

Return the number of occurrences of *value* in *o*, that is, return the number of keys for which `o[key] == value`. On failure, return `-1`. This is equivalent to the Python expression `o.count(value)`.

Alterado na versão 2.5: This function returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

int **PySequence_Contains**(*PyObject* *o, *PyObject* *value)

Determine if *o* contains *value*. If an item in *o* is equal to *value*, return `1`, otherwise return `0`. On error, return `-1`. This is equivalent to the Python expression `value in o`.

Py_ssize_t **PySequence_Index**(*PyObject* *o, *PyObject* *value)

Return the first index *i* for which `o[i] == value`. On error, return `-1`. This is equivalent to the Python expression `o.index(value)`.

Alterado na versão 2.5: This function returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PySequence_List**(*PyObject* *o)

Return value: New reference. Return a list object with the same contents as the arbitrary sequence *o*. The returned list is guaranteed to be new.

*PyObject** **PySequence_Tuple**(*PyObject* *o)

Return value: New reference. Return a tuple object with the same contents as the arbitrary sequence *o* or *NULL* on failure. If *o* is a tuple, a new reference will be returned, otherwise a tuple will be constructed with the appropriate contents. This is equivalent to the Python expression `tuple(o)`.

*PyObject** **PySequence_Fast**(*PyObject* *o, const char *m)

Return value: New reference. Return the sequence *o* as a list, unless it is already a tuple or list, in which case *o* is returned. Use `PySequence_Fast_GET_ITEM()` to access the members of the result. Returns *NULL* on failure. If the object is not a sequence, raises `TypeError` with *m* as the message text.

*PyObject** **PySequence_Fast_GET_ITEM**(*PyObject* *o, Py_ssize_t *i*)

Return value: Borrowed reference. Return the *i*th element of *o*, assuming that *o* was returned by `PySequence_Fast()`, *o* is not *NULL*, and that *i* is within bounds.

Alterado na versão 2.5: This function used an `int` type for *i*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject*** **PySequence_Fast_ITEMS**(*PyObject *o*)
Return the underlying array of PyObject pointers. Assumes that *o* was returned by `PySequence_Fast()` and *o* is not *NULL*.

Note, if a list gets resized, the reallocation may relocate the items array. So, only use the underlying array pointer in contexts where the sequence cannot change.

Novo na versão 2.4.

*PyObject** **PySequence_ITEM**(*PyObject *o*, Py_ssize_t *i*)
Return value: New reference. Return the *i*th element of *o* or *NULL* on failure. Macro form of `PySequence_GetItem()` but without checking that `PySequence_Check()` on *o* is true and without adjustment for negative indices.

Novo na versão 2.3.

Alterado na versão 2.5: This function used an `int` type for *i*. This might require changes in your code for properly supporting 64-bit systems.

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 gotten by calling `PySequence_Size()` on *o*, but `PySequence_Fast_GET_SIZE()` is faster because it can assume *o* is a list or tuple.

6.4 Protocolo de Mapeamento

int **PyMapping_Check**(*PyObject *o*)
Return `1` if the object provides mapping protocol, and `0` otherwise. This function always succeeds.

Py_ssize_t **PyMapping_Size**(*PyObject *o*)
Py_ssize_t **PyMapping_Length**(*PyObject *o*)
Returns the number of keys in object *o* on success, and `-1` on failure. For objects that do not provide mapping protocol, this is equivalent to the Python expression `len(o)`.

Alterado na versão 2.5: These functions returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

int **PyMapping_DelItemString**(*PyObject *o*, char **key*)
Remove the mapping for object *key* from the object *o*. Return `-1` on failure. This is equivalent to the Python statement `del o[key]`.

int **PyMapping_DelItem**(*PyObject *o*, *PyObject *key*)
Remove the mapping for object *key* from the object *o*. Return `-1` on failure. This is equivalent to the Python statement `del o[key]`.

int **PyMapping_HasKeyString**(*PyObject *o*, char **key*)
On success, return `1` if the mapping object has the key *key* and `0` otherwise. This is equivalent to `o[key]`, returning `True` on success and `False` on an exception. This function always succeeds.

int **PyMapping_HasKey**(*PyObject *o*, *PyObject *key*)
Return `1` if the mapping object has the key *key* and `0` otherwise. This is equivalent to `o[key]`, returning `True` on success and `False` on an exception. This function always succeeds.

*PyObject** **PyMapping_Keys**(*PyObject *o*)
Return value: New reference. On success, return a list of the keys in object *o*. On failure, return *NULL*. This is equivalent to the Python expression `o.keys()`.

*PyObject** **PyMapping_Values**(*PyObject *o*)
Return value: New reference. On success, return a list of the values in object *o*. On failure, return *NULL*. This is equivalent to the Python expression `o.values()`.

*PyObject** **PyMapping_Items**(*PyObject *o*)
Return value: New reference. On success, return a list of the items in object *o*, where each item is a tuple containing a key-value pair. On failure, return *NULL*. This is equivalent to the Python expression `o.items()`.

*PyObject** **PyMapping_GetItemString**(*PyObject *o*, char **key*)
Return value: New reference. Return element of *o* corresponding to the object *key* or *NULL* on failure. This is the equivalent of the Python expression `o[key]`.

int **PyMapping_SetItemString**(*PyObject *o*, char **key*, *PyObject *v*)
Map the object *key* to the value *v* in object *o*. Returns `-1` on failure. This is the equivalent of the Python statement `o[key] = v`.

6.5 Protocolo Iterador

Novo na versão 2.2.

Existem duas funções especificas para trabalhar com iteradores.

int **PyIter_Check**(*PyObject *o*)
Retorna verdadeiro se o objeto *o* suporta o protocolo do iterador.

This function can return a false positive in the case of old-style classes because those classes always define a `tp_iternext` slot with logic that either invokes a `next()` method or raises a `TypeError`.

*PyObject** **PyIter_Next**(*PyObject *o*)
Return value: New reference. Return the next value from the iteration *o*. The object must be an iterator (it is up to the caller to check this). If there are no remaining values, returns *NULL* with no exception set. If an error occurs while retrieving the item, returns *NULL* and passes along the exception.

Para escrever um loop que itere sobre um iterador, o código C deve ser algo como isto

```
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 */
}
```

6.6 Protocolo de Buffer Antigo

This section describes the legacy buffer protocol, which has been introduced in Python 1.6. It is still supported but deprecated in the Python 2.x series. Python 3 introduces a new buffer protocol which fixes weaknesses and shortcomings of the protocol, and has been backported to Python 2.6. See *Buffers and Memoryview Objects* for more information.

int **PyObject_AsCharBuffer**(*PyObject* **obj*, const char ***buffer*, Py_ssize_t **buffer_len*)

Retorna um ponteiro para um local de memória somente leitura utilizável como entrada baseada em caracteres. O argumento *obj* deve ter suporte a interface do buffer de caracteres de segmento único. Em caso de sucesso, retorna `0`, define *buffer* com o local da memória e *buffer_len* com o comprimento do buffer. Retorna `-1` e define a `TypeError` em caso de erro.

Novo na versão 1.6.

Alterado na versão 2.5: This function used an `int *` type for *buffer_len*. This might require changes in your code for properly supporting 64-bit systems.

int **PyObject_AsReadBuffer**(*PyObject* **obj*, const void ***buffer*, Py_ssize_t **buffer_len*)

Retorna um ponteiro para um local de memória somente leitura que contém dados arbitrários. O argumento *obj* deve ter suporte a interface de buffer legível de segmento único. Em caso de sucesso, retorna `0`, define *buffer* com o local da memória e *buffer_len* com o comprimento do buffer. Retorna `-1` e define a `TypeError` em caso de erro.

Novo na versão 1.6.

Alterado na versão 2.5: This function used an `int *` type for *buffer_len*. This might require changes in your code for properly supporting 64-bit systems.

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

Returns `1` if *o* supports the single-segment readable buffer interface. Otherwise returns `0`.

Novo na versão 2.2.

int **PyObject_AsWriteBuffer**(*PyObject* **obj*, void ***buffer*, Py_ssize_t **buffer_len*)

Returns a pointer to a writeable memory location. The *obj* argument must support the single-segment, character buffer interface. On success, returns `0`, sets *buffer* to the memory location and *buffer_len* to the buffer length. Returns `-1` and sets a `TypeError` on error.

Novo na versão 1.6.

Alterado na versão 2.5: This function used an `int *` type for *buffer_len*. This might require changes in your code for properly supporting 64-bit systems.

CAPÍTULO 7

Camada de Objetos Concretos

The functions in this chapter are specific to certain Python object types. Passing them an object of the wrong type is not a good idea; if you receive an object from a Python program and you are not sure that it has the right type, you must perform a type check first; for example, to check that an object is a dictionary, use `PyDict_Check()`. The chapter is structured like the "family tree" of Python object types.

> **Aviso:** While the functions described in this chapter carefully check the type of the objects which are passed in, many of them do not check for *NULL* being passed instead of a valid object. Allowing *NULL* to be passed in can cause memory access violations and immediate termination of the interpreter.

7.1 Fundamental Objects

This section describes Python type objects and the singleton object `None`.

7.1.1 Objetos de tipo

`PyTypeObject`

The C structure of the objects used to describe built-in types.

*PyObject** **`PyType_Type`**

This is the type object for type objects; it is the same object as `type` and `types.TypeType` in the Python layer.

int **`PyType_Check`**(*PyObject *o*)

Return true if the object *o* is a type object, including instances of types derived from the standard type object. Return false in all other cases.

int **`PyType_CheckExact`**(*PyObject *o*)

Return true if the object *o* is a type object, but not a subtype of the standard type object. Return false in all other cases.

Novo na versão 2.2.

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

Novo na versão 2.6.

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.

Novo na versão 2.6.

int **PyType_HasFeature**(*PyObject *o*, int *feature*)
Return true if the type object *o* sets the feature *feature*. Type features are denoted by single bit flags.

int **PyType_IS_GC**(*PyObject *o*)
Return true if the type object includes support for the cycle detector; this tests the type flag `Py_TPFLAGS_HAVE_GC`.

Novo na versão 2.0.

int **PyType_IsSubtype**(*PyTypeObject *a*, *PyTypeObject *b*)
Return true if *a* is a subtype of *b*.

Novo na versão 2.2.

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. Novo na versão 2.2.

Alterado na versão 2.5: This function used an `int` type for *nitems*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyType_GenericNew**(*PyTypeObject *type*, *PyObject *args*, *PyObject *kwds*)
Return value: New reference. Novo na versão 2.2.

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.

Novo na versão 2.2.

7.1.2 O Objeto `None`

Observe que o `PyTypeObject` para `None` não está diretamente exposto pela API Python/C. Como `None` é um singleton, é suficiente testar a identidade do objeto (usando == em C). Não há nenhuma função `PyNone_Check()` pela mesma razão.

*PyObject** **Py_None**
O objeto Python `None`, denota falta de valor. Este objeto não tem métodos. O mesmo precisa ser tratado como qualquer outro objeto com relação à contagem de referência.

Py_RETURN_NONE
Properly handle returning `Py_None` from within a C function.

Novo na versão 2.4.

7.2 Numeric Objects

7.2.1 Plain Integer Objects

PyIntObject
This subtype of *PyObject* represents a Python integer object.

PyTypeObject **PyInt_Type**
This instance of *PyTypeObject* represents the Python plain integer type. This is the same object as `int` and `types.IntType`.

int **PyInt_Check** (*PyObject *o*)
Return true if *o* is of type *PyInt_Type* or a subtype of *PyInt_Type*.

Alterado na versão 2.2: Allowed subtypes to be accepted.

int **PyInt_CheckExact** (*PyObject *o*)
Return true if *o* is of type *PyInt_Type*, but not a subtype of *PyInt_Type*.

Novo na versão 2.2.

*PyObject** **PyInt_FromString** (char **str*, char ***pend*, int *base*)
Return value: New reference. Return a new *PyIntObject* or *PyLongObject* based on the string value in *str*, which is interpreted according to the radix in *base*. If *pend* is non-*NULL*, `*pend` will point to the first character in *str* which follows the representation of the number. If *base* is `0`, the radix will be determined based on the leading characters of *str*: if *str* starts with `'0x'` or `'0X'`, radix 16 will be used; if *str* starts with `'0'`, radix 8 will be used; otherwise radix 10 will be used. If *base* is not `0`, it must be between `2` and `36`, inclusive. Leading spaces are ignored. If there are no digits, `ValueError` will be raised. If the string represents a number too large to be contained within the machine's `long int` type and overflow warnings are being suppressed, a *PyLongObject* will be returned. If overflow warnings are not being suppressed, *NULL* will be returned in this case.

*PyObject** **PyInt_FromLong** (long *ival*)
Return value: New reference. Create a new integer object with a value of *ival*.

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. So it should be possible to change the value of `1`. I suspect the behaviour of Python in this case is undefined. :-)

*PyObject** **PyInt_FromSsize_t** (Py_ssize_t *ival*)
Return value: New reference. Create a new integer object with a value of *ival*. If the value is larger than `LONG_MAX` or smaller than `LONG_MIN`, a long integer object is returned.

Novo na versão 2.5.

*PyObject** **PyInt_FromSize_t** (size_t *ival*)
Create a new integer object with a value of *ival*. If the value exceeds `LONG_MAX`, a long integer object is returned.

Novo na versão 2.5.

long **PyInt_AsLong** (*PyObject *io*)
Will first attempt to cast the object to a *PyIntObject*, if it is not already one, and then return its value. If there is an error, `-1` is returned, and the caller should check `PyErr_Occurred()` to find out whether there was an error, or whether the value just happened to be `-1`.

long **PyInt_AS_LONG** (*PyObject *io*)
Return the value of the object *io*. No error checking is performed.

unsigned long **PyInt_AsUnsignedLongMask** (*PyObject *io*)
Will first attempt to cast the object to a *PyIntObject* or *PyLongObject*, if it is not already one, and then return its value as unsigned long. This function does not check for overflow.

Novo na versão 2.3.

unsigned PY_LONG_LONG **PyInt_AsUnsignedLongLongMask** (*PyObject *io*)

Will first attempt to cast the object to a *PyIntObject* or *PyLongObject*, if it is not already one, and then return its value as unsigned long long, without checking for overflow.

Novo na versão 2.3.

Py_ssize_t **PyInt_AsSsize_t** (*PyObject *io*)

Will first attempt to cast the object to a *PyIntObject* or *PyLongObject*, if it is not already one, and then return its value as `Py_ssize_t`.

Novo na versão 2.5.

long **PyInt_GetMax** ()

Return the system's idea of the largest integer it can handle (`LONG_MAX`, as defined in the system header files).

int **PyInt_ClearFreeList** ()

Clear the integer free list. Return the number of items that could not be freed.

Novo na versão 2.6.

7.2.2 Objetos Booleanos

Booleano em Python é implementado como uma subclasse de inteiros. Existem apenas dois tipos de booleanos `Py_False` e `Py_True`. Como tal, as funções normais de criação e exclusão não se aplicam a booleanos. No entanto, as seguintes macros estão disponíveis.

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

Retorna verdadeiro se *o* for do tipo `PyBool_Type`.

Novo na versão 2.3.

*PyObject** **Py_False**

O objeto Python `False`. Este objeto não possui métodos. Ele precisa ser tratado como qualquer outro objeto em relação às contagens de referência.

*PyObject** **Py_True**

O objeto Python `True`. Este objeto não possui métodos. Ele precisa ser tratado como qualquer outro objeto em relação às contagens de referência.

Py_RETURN_FALSE

Retornar `Py_False` de uma função, incrementando adequadamente sua contagem de referência.

Novo na versão 2.4.

Py_RETURN_TRUE

Retorna `Py_True` de uma função, incrementando adequadamente sua contagem de referência.

Novo na versão 2.4.

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

Return value: New reference. Retorna uma nova referência para `Py_True` ou `Py_False` dependendo do valor de verdade de *v*.

Novo na versão 2.3.

7.2.3 Long Integer Objects

PyLongObject
This subtype of *PyObject* represents a Python long integer object.

PyTypeObject **PyLong_Type**
This instance of *PyTypeObject* represents the Python long integer type. This is the same object as `long` and `types.LongType`.

int **PyLong_Check** (*PyObject* **p*)
Return true if its argument is a *PyLongObject* or a subtype of *PyLongObject*.

Alterado na versão 2.2: Allowed subtypes to be accepted.

int **PyLong_CheckExact** (*PyObject* **p*)
Return true if its argument is a *PyLongObject*, but not a subtype of *PyLongObject*.

Novo na versão 2.2.

*PyObject** **PyLong_FromLong** (long *v*)
Return value: New reference. Return a new *PyLongObject* object from *v*, or *NULL* on failure.

*PyObject** **PyLong_FromUnsignedLong** (unsigned long *v*)
Return value: New reference. Return a new *PyLongObject* object from a C `unsigned long`, or *NULL* on failure.

*PyObject** **PyLong_FromSsize_t** (Py_ssize_t *v*)
Return value: New reference. Return a new *PyLongObject* object from a C `Py_ssize_t`, or *NULL* on failure.

Novo na versão 2.6.

*PyObject** **PyLong_FromSize_t** (size_t *v*)
Return value: New reference. Return a new *PyLongObject* object from a C `size_t`, or *NULL* on failure.

Novo na versão 2.6.

*PyObject** **PyLong_FromLongLong** (PY_LONG_LONG *v*)
Return value: New reference. Return a new *PyLongObject* object from a C `long long`, or *NULL* on failure.

*PyObject** **PyLong_FromUnsignedLongLong** (unsigned PY_LONG_LONG *v*)
Return value: New reference. Return a new *PyLongObject* object from a C `unsigned long long`, or *NULL* on failure.

*PyObject** **PyLong_FromDouble** (double *v*)
Return value: New reference. Return a new *PyLongObject* object from the integer part of *v*, or *NULL* on failure.

*PyObject** **PyLong_FromString** (char **str*, char ***pend*, int *base*)
Return value: New reference. Return a new *PyLongObject* based on the string value in *str*, which is interpreted according to the radix in *base*. If *pend* is non-*NULL*, **pend* will point to the first character in *str* which follows the representation of the number. If *base* is `0`, the radix will be determined based on the leading characters of *str*: if *str* starts with `'0x'` or `'0X'`, radix 16 will be used; if *str* starts with `'0'`, radix 8 will be used; otherwise radix 10 will be used. If *base* is not `0`, it must be between `2` and `36`, inclusive. Leading spaces are ignored. If there are no digits, `ValueError` will be raised.

*PyObject** **PyLong_FromUnicode** (*Py_UNICODE* **u*, Py_ssize_t *length*, int *base*)
Return value: New reference. Convert a sequence of Unicode digits to a Python long integer value. The first parameter, *u*, points to the first character of the Unicode string, *length* gives the number of characters, and *base* is the radix for the conversion. The radix must be in the range [2, 36]; if it is out of range, `ValueError` will be raised.

Novo na versão 1.6.

Alterado na versão 2.5: This function used an `int` for *length*. This might require changes in your code for properly supporting 64-bit systems.

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

Return value: New reference. Create a Python integer or long integer from the pointer *p*. The pointer value can be retrieved from the resulting value using `PyLong_AsVoidPtr()`.

Novo na versão 1.5.2.

Alterado na versão 2.5: If the integer is larger than LONG_MAX, a positive long integer is returned.

long **PyLong_AsLong**(*PyObject* **pylong*)

Return a C `long` representation of the contents of *pylong*. If *pylong* is greater than `LONG_MAX`, an `OverflowError` is raised and `-1` will be returned.

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

Return a C `long` representation of the contents of *pylong*. If *pylong* is greater than `LONG_MAX` or less than `LONG_MIN`, set **overflow* to `1` or `-1`, respectively, and return `-1`; otherwise, set **overflow* to `0`. If any other exception occurs (for example a TypeError or MemoryError), then `-1` will be returned and **overflow* will be `0`.

Novo na versão 2.7.

PY_LONG_LONG **PyLong_AsLongLongAndOverflow**(*PyObject* **pylong*, int **overflow*)

Return a C `long long` representation of the contents of *pylong*. If *pylong* is greater than `PY_LLONG_MAX` or less than `PY_LLONG_MIN`, set **overflow* to `1` or `-1`, respectively, and return `-1`; otherwise, set **overflow* to `0`. If any other exception occurs (for example a TypeError or MemoryError), then `-1` will be returned and **overflow* will be `0`.

Novo na versão 2.7.

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

Return a C `Py_ssize_t` representation of the contents of *pylong*. If *pylong* is greater than `PY_SSIZE_T_MAX`, an `OverflowError` is raised and `-1` will be returned.

Novo na versão 2.6.

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

Return a C `unsigned long` representation of the contents of *pylong*. If *pylong* is greater than `ULONG_MAX`, an `OverflowError` is raised.

PY_LONG_LONG **PyLong_AsLongLong**(*PyObject* **pylong*)

Return a C `long long` from a Python long integer. If *pylong* cannot be represented as a `long long`, an `OverflowError` is raised and `-1` is returned.

Novo na versão 2.2.

unsigned PY_LONG_LONG **PyLong_AsUnsignedLongLong**(*PyObject* **pylong*)

Return a C `unsigned long long` from a Python long integer. If *pylong* cannot be represented as an `unsigned long long`, an `OverflowError` is raised and `(unsigned long long)-1` is returned.

Novo na versão 2.2.

Alterado na versão 2.7: A negative *pylong* now raises `OverflowError`, not `TypeError`.

unsigned long **PyLong_AsUnsignedLongMask**(*PyObject* **io*)

Return a C `unsigned long` from a Python long integer, without checking for overflow.

Returns `(unsigned long)-1` on error. Use `PyErr_Occurred()` to disambiguate.

Novo na versão 2.3.

unsigned PY_LONG_LONG **PyLong_AsUnsignedLongLongMask**(*PyObject* **io*)

Return a C `unsigned long long` from a Python long integer, without checking for overflow.

Returns `(unsigned PY_LONG_LONG)-1` on error. Use *PyErr_Occurred()* to disambiguate.

Novo na versão 2.3.

double **PyLong_AsDouble**(*PyObject* *pylong)
Return a C `double` representation of the contents of *pylong*. If *pylong* cannot be approximately represented as a `double`, an `OverflowError` exception is raised and `-1.0` will be returned.

void* **PyLong_AsVoidPtr**(*PyObject* *pylong)
Convert a Python integer or long integer *pylong* to a C `void` pointer. If *pylong* cannot be converted, an `OverflowError` will be raised. This is only assured to produce a usable `void` pointer for values created with *PyLong_FromVoidPtr()*.

Novo na versão 1.5.2.

Alterado na versão 2.5: For values outside 0..LONG_MAX, both signed and unsigned integers are accepted.

7.2.4 Objetos de Ponto Flutuante

PyFloatObject
This subtype of *PyObject* represents a Python floating point object.

PyTypeObject **PyFloat_Type**
This instance of *PyTypeObject* represents the Python floating point type. This is the same object as `float` and `types.FloatType`.

int **PyFloat_Check**(*PyObject* *p)
Return true if its argument is a *PyFloatObject* or a subtype of *PyFloatObject*.

Alterado na versão 2.2: Allowed subtypes to be accepted.

int **PyFloat_CheckExact**(*PyObject* *p)
Return true if its argument is a *PyFloatObject*, but not a subtype of *PyFloatObject*.

Novo na versão 2.2.

*PyObject** **PyFloat_FromString**(*PyObject* *str, char **pend)
Return value: New reference. Create a *PyFloatObject* object based on the string value in *str*, or *NULL* on failure. The *pend* argument is ignored. It remains only for backward compatibility.

*PyObject** **PyFloat_FromDouble**(double *v*)
Return value: New reference. Create a *PyFloatObject* object from *v*, or *NULL* on failure.

double **PyFloat_AsDouble**(*PyObject* *pyfloat)
Return a C `double` representation of the contents of *pyfloat*. If *pyfloat* is not a Python floating point object but has a `__float__()` method, this method will first be called to convert *pyfloat* into a float. This method returns `-1.0` upon failure, so one should call *PyErr_Occurred()* to check for errors.

double **PyFloat_AS_DOUBLE**(*PyObject* *pyfloat)
Return a C `double` representation of the contents of *pyfloat*, but without error checking.

*PyObject** **PyFloat_GetInfo**(void)
Return a structseq instance which contains information about the precision, minimum and maximum values of a float. It's a thin wrapper around the header file `float.h`.

Novo na versão 2.6.

double **PyFloat_GetMax**()
Return the maximum representable finite float *DBL_MAX* as C `double`.

Novo na versão 2.6.

double **PyFloat_GetMin**()
Return the minimum normalized positive float *DBL_MIN* as C `double`.

Novo na versão 2.6.

int **PyFloat_ClearFreeList**()
Clear the float free list. Return the number of items that could not be freed.

Novo na versão 2.6.

void **PyFloat_AsString**(char **buf*, *PyFloatObject* **v*)
Convert the argument *v* to a string, using the same rules as `str()`. The length of *buf* should be at least 100.

This function is unsafe to call because it writes to a buffer whose length it does not know.

Obsoleto desde a versão 2.7: Use `PyObject_Str()` or `PyOS_double_to_string()` instead.

void **PyFloat_AsReprString**(char **buf*, *PyFloatObject* **v*)
Same as PyFloat_AsString, except uses the same rules as `repr()`. The length of *buf* should be at least 100.

This function is unsafe to call because it writes to a buffer whose length it does not know.

Obsoleto desde a versão 2.7: Use `PyObject_Repr()` or `PyOS_double_to_string()` instead.

7.2.5 Objetos de Números Complexos

Python's complex number objects are implemented as two distinct types when viewed from the C API: one is the Python object exposed to Python programs, and the other is a C structure which represents the actual complex number value. The API provides functions for working with both.

Números complexos como estruturas C.

Note that the functions which accept these structures as parameters and return them as results do so *by value* rather than dereferencing them through pointers. This is consistent throughout the API.

Py_complex
The C structure which corresponds to the value portion of a Python complex number object. Most of the functions for dealing with complex number objects use structures of this type as input or output values, as appropriate. It is defined as:

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

Py_complex **_Py_c_sum**(*Py_complex left*, *Py_complex right*)
Retorna a soma de dois números complexos, utilizando a representação C `Py_complex`.

Py_complex **_Py_c_diff**(*Py_complex left*, *Py_complex right*)
Retorna a diferença entre dois números complexos, utilizando a representação C `Py_complex`:

Py_complex **_Py_c_neg**(*Py_complex complex*)
Retorna a negação do número complexo *complex*, utilizando a representação C `Py_complex`.

Py_complex **_Py_c_prod**(*Py_complex left*, *Py_complex right*)
Retorna o produto de dois números complexos, utilizando a representação C `Py_complex`.

Py_complex **_Py_c_quot**(*Py_complex dividend*, *Py_complex divisor*)
Retorna o quociente de dois números complexos, utilizando a representação C `Py_complex`.

Se *divisor* é nulo, este método retorna zero e define `errno` para `EDOM`.

Py_complex **_Py_c_pow** (*Py_complex num*, *Py_complex exp*)
Retorna a exponenciação de *num* por *exp*, utilizando a representação C `Py_complex`

If *num* is null and *exp* is not a positive real number, this method returns zero and sets `errno` to `EDOM`.

Números complexos como objetos Python

PyComplexObject
Este subtipo de `PyObject` representa um objeto Python de número complexo.

PyTypeObject **PyComplex_Type**
This instance of `PyTypeObject` represents the Python complex number type. It is the same object as `complex` and `types.ComplexType`.

int **PyComplex_Check** (*PyObject *p*)
Return true if its argument is a `PyComplexObject` or a subtype of `PyComplexObject`.

Alterado na versão 2.2: Allowed subtypes to be accepted.

int **PyComplex_CheckExact** (*PyObject *p*)
Return true if its argument is a `PyComplexObject`, but not a subtype of `PyComplexObject`.

Novo na versão 2.2.

*PyObject** **PyComplex_FromCComplex** (*Py_complex v*)
Return value: New reference. Create a new Python complex number object from a C `Py_complex` value.

*PyObject** **PyComplex_FromDoubles** (double *real*, double *imag*)
Return value: New reference. Return a new `PyComplexObject` object from *real* and *imag*.

double **PyComplex_RealAsDouble** (*PyObject *op*)
Return the real part of *op* as a C `double`.

double **PyComplex_ImagAsDouble** (*PyObject *op*)
Return the imaginary part of *op* as a C `double`.

Py_complex **PyComplex_AsCComplex** (*PyObject *op*)
Return the `Py_complex` value of the complex number *op*. Upon failure, this method returns `-1.0` as a real value.

Alterado na versão 2.6: If *op* is not a Python complex number object but has a `__complex__()` method, this method will first be called to convert *op* to a Python complex number object.

7.3 Sequence Objects

Generic operations on sequence objects were discussed in the previous chapter; this section deals with the specific kinds of sequence objects that are intrinsic to the Python language.

7.3.1 Objetos Byte Array

Novo na versão 2.6.

PyByteArrayObject
Esse subtipo de `PyObject` representa um objeto Python bytearray.

PyTypeObject **PyByteArray_Type**
This instance of `PyTypeObject` represents the Python bytearray type; it is the same object as `bytearray` in the Python layer.

Macros para verificação de tipo

int **PyByteArray_Check**(*PyObject *o*)
Retorna verdadeiro se o objeto *o* for um objeto bytearray ou se for uma instância de um subtipo do tipo bytearray.

int **PyByteArray_CheckExact**(*PyObject *o*)
Retorna verdadeiro se o objeto *o* for um objeto bytearray, mas não for uma instância de um subtipo do tipo bytearray.

Funções diretas da API

*PyObject** **PyByteArray_FromObject**(*PyObject *o*)
Return a new bytearray object from any object, *o*, that implements the buffer protocol.

*PyObject** **PyByteArray_FromStringAndSize**(const char **string*, Py_ssize_t *len*)
Create a new bytearray object from *string* and its length, *len*. On failure, *NULL* is returned.

*PyObject** **PyByteArray_Concat**(*PyObject *a*, *PyObject *b*)
Concatena os bytearrays *a* e *b* e retorna um novo bytearray com o resultado.

Py_ssize_t **PyByteArray_Size**(*PyObject *bytearray*)
Return the size of *bytearray* after checking for a *NULL* pointer.

char* **PyByteArray_AsString**(*PyObject *bytearray*)
Return the contents of *bytearray* as a char array after checking for a *NULL* pointer.

int **PyByteArray_Resize**(*PyObject *bytearray*, Py_ssize_t *len*)
Redimensiona o buffer interno de *bytearray* para o tamanho *len*.

Macros

Estas macros trocam segurança por velocidade e não verificam os ponteiros.

char* **PyByteArray_AS_STRING**(*PyObject *bytearray*)
Versão macro de `PyByteArray_AsString()`.

Py_ssize_t **PyByteArray_GET_SIZE**(*PyObject *bytearray*)
Versão macro de `PyByteArray_Size()`.

7.3.2 String/Bytes Objects

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

Nota: These functions have been renamed to PyBytes_* in Python 3.x. Unless otherwise noted, the PyBytes functions available in 3.x are aliased to their PyString_* equivalents to help porting.

`PyStringObject`
This subtype of *`PyObject`* represents a Python string object.

PyTypeObject **`PyString_Type`**
This instance of *`PyTypeObject`* represents the Python string type; it is the same object as `str` and `types.StringType` in the Python layer. .

int **`PyString_Check`**(*PyObject *o*)
Return true if the object *o* is a string object or an instance of a subtype of the string type.

Alterado na versão 2.2: Allowed subtypes to be accepted.

int **`PyString_CheckExact`**(*PyObject *o*)
Return true if the object *o* is a string object, but not an instance of a subtype of the string type.

Novo na versão 2.2.

*PyObject** **`PyString_FromString`**(const char **v*)
Return value: New reference. Return a new string object with a copy of the string *v* as value on success, and *NULL* on failure. The parameter *v* must not be *NULL*; it will not be checked.

*PyObject** **`PyString_FromStringAndSize`**(const char **v*, Py_ssize_t *len*)
Return value: New reference. Return a new string object with a copy of the string *v* as value and length *len* on success, and *NULL* on failure. If *v* is *NULL*, the contents of the string are uninitialized.

Alterado na versão 2.5: This function used an `int` type for *len*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **`PyString_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 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* string. The following format characters are allowed:

Format Characters	Type	Comment
`%%`	*n/a*	The literal % character.
`%c`	int	A single character, represented as a C int.
`%d`	int	Exactly equivalent to `printf("%d")`.
`%u`	unsigned int	Exactly equivalent to `printf("%u")`.
`%ld`	long	Exactly equivalent to `printf("%ld")`.
`%lu`	unsigned long	Exactly equivalent to `printf("%lu")`.
`%lld`	long long	Exactly equivalent to `printf("%lld")`.
`%llu`	unsigned long long	Exactly equivalent to `printf("%llu")`.
`%zd`	Py_ssize_t	Exactly equivalent to `printf("%zd")`.
`%zu`	size_t	Exactly equivalent to `printf("%zu")`.
`%i`	int	Exactly equivalent to `printf("%i")`.
`%x`	int	Exactly equivalent to `printf("%x")`.
`%s`	char*	A null-terminated C character array.
`%p`	void*	The hex representation of a C pointer. Mostly equivalent to `printf("%p")` except that it is guaranteed to start with the literal `0x` regardless of what the platform's `printf` yields.

An unrecognized format character causes all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

Nota: The *"%lld"* and *"%llu"* format specifiers are only available when `HAVE_LONG_LONG` is defined.

Alterado na versão 2.7: Support for *"%lld"* and *"%llu"* added.

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

Return value: New reference. Identical to `PyString_FromFormat()` except that it takes exactly two arguments.

Py_ssize_t **PyString_Size**(*PyObject* **string*)

Return the length of the string in string object *string*.

Alterado na versão 2.5: This function returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t **PyString_GET_SIZE**(*PyObject* **string*)

Macro form of `PyString_Size()` but without error checking.

Alterado na versão 2.5: This macro returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

char* **PyString_AsString**(*PyObject* **string*)

Return a NUL-terminated representation of the contents of *string*. The pointer refers to the internal buffer of *string*, not a copy. The data must not be modified in any way, unless the string was just created using `PyString_FromStringAndSize(NULL, size)`. It must not be deallocated. If *string* is a Unicode object, this function computes the default encoding of *string* and operates on that. If *string* is not a string object at all, `PyString_AsString()` returns *NULL* and raises `TypeError`.

char* **PyString_AS_STRING**(*PyObject *string*)
Macro form of `PyString_AsString()` but without error checking. Only string objects are supported; no Unicode objects should be passed.

int **PyString_AsStringAndSize**(*PyObject *obj*, char ***buffer*, Py_ssize_t **length*)
Return a NUL-terminated representation of the contents of the object *obj* through the output variables *buffer* and *length*.

The function accepts both string and Unicode objects as input. For Unicode objects it returns the default encoded version of the object. If *length* is *NULL*, the resulting buffer may not contain NUL characters; if it does, the function returns `-1` and a `TypeError` is raised.

The buffer refers to an internal string buffer of *obj*, not a copy. The data must not be modified in any way, unless the string was just created using `PyString_FromStringAndSize(NULL, size)`. It must not be deallocated. If *string* is a Unicode object, this function computes the default encoding of *string* and operates on that. If *string* is not a string object at all, `PyString_AsStringAndSize()` returns `-1` and raises `TypeError`.

Alterado na versão 2.5: This function used an `int *` type for *length*. This might require changes in your code for properly supporting 64-bit systems.

void **PyString_Concat**(*PyObject **string*, *PyObject *newpart*)
Create a new string object in **string* containing the contents of *newpart* appended to *string*; the caller will own the new reference. The reference to the old value of *string* will be stolen. If the new string cannot be created, the old reference to *string* will still be discarded and the value of **string* will be set to *NULL*; the appropriate exception will be set.

void **PyString_ConcatAndDel**(*PyObject **string*, *PyObject *newpart*)
Create a new string object in **string* containing the contents of *newpart* appended to *string*. This version decrements the reference count of *newpart*.

int **_PyString_Resize**(*PyObject **string*, Py_ssize_t *newsize*)
A way to resize a string object even though it is "immutable". Only use this to build up a brand new string object; don't use this if the string may already be known in other parts of the code. It is an error to call this function if the refcount on the input string object is not one. Pass the address of an existing string object as an lvalue (it may be written into), and the new size desired. On success, **string* holds the resized string object and `0` is returned; the address in **string* may differ from its input value. If the reallocation fails, the original string object at **string* is deallocated, **string* is set to *NULL*, a memory exception is set, and `-1` is returned.

Alterado na versão 2.5: This function used an `int` type for *newsize*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyString_Format**(*PyObject *format*, *PyObject *args*)
Return value: New reference. Return a new string object from *format* and *args*. Analogous to `format % args`. The *args* argument must be a tuple or dict.

void **PyString_InternInPlace**(*PyObject **string*)
Intern the argument **string* in place. The argument must be the address of a pointer variable pointing to a Python 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.)

Nota: This function is not available in 3.x and does not have a PyBytes alias.

*PyObject** **PyString_InternFromString** (const char **v*)
Return value: New reference. A combination of `PyString_FromString()` and `PyString_InternInPlace()`, returning either a new string object that has been interned, or a new ("owned") reference to an earlier interned string object with the same value.

Nota: This function is not available in 3.x and does not have a PyBytes alias.

*PyObject** **PyString_Decode** (const char **s*, Py_ssize_t *size*, const char **encoding*, const char **errors*)
Return value: New reference. Create an object by decoding *size* bytes of the encoded buffer *s* using the codec registered for *encoding*. *encoding* and *errors* have the same meaning as the parameters of the same name in the `unicode()` 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.

Nota: This function is not available in 3.x and does not have a PyBytes alias.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyString_AsDecodedObject** (*PyObject* **str*, const char **encoding*, const char **errors*)
Return value: New reference. Decode a string object by passing it to the codec registered for *encoding* and return the result as Python object. *encoding* and *errors* have the same meaning as the parameters of the same name in the string `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.

Nota: This function is not available in 3.x and does not have a PyBytes alias.

*PyObject** **PyString_Encode** (const char **s*, Py_ssize_t *size*, const char **encoding*, const char **errors*)
Return value: New reference. Encode the `char` buffer of the given size by passing it to the codec registered for *encoding* and return a Python object. *encoding* and *errors* have the same meaning as the parameters of the same name in the string `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.

Nota: This function is not available in 3.x and does not have a PyBytes alias.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyString_AsEncodedObject** (*PyObject* **str*, const char **encoding*, const char **errors*)
Return value: New reference. Encode a string object using the codec registered for *encoding* and return the result as Python object. *encoding* and *errors* have the same meaning as the parameters of the same name in the string `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.

Nota: This function is not available in 3.x and does not have a PyBytes alias.

7.3.3 Objetos Unicode e Codecs

Unicode Objects

Unicode Type

These are the basic Unicode object types used for the Unicode implementation in Python:

Py_UNICODE

This type represents the storage type which is used by Python internally as basis for holding Unicode ordinals. Python's default builds use a 16-bit type for `Py_UNICODE` and store Unicode values internally as UCS2. It is also possible to build a UCS4 version of Python (most recent Linux distributions come with UCS4 builds of Python). These builds then use a 32-bit type for `Py_UNICODE` and store Unicode data internally as UCS4. On platforms where `wchar_t` is available and compatible with the chosen Python Unicode build variant, `Py_UNICODE` is a typedef alias for `wchar_t` to enhance native platform compatibility. On all other platforms, `Py_UNICODE` is a typedef alias for either `unsigned short` (UCS2) or `unsigned long` (UCS4).

Note that UCS2 and UCS4 Python builds are not binary compatible. Please keep this in mind when writing extensions or interfaces.

PyUnicodeObject

This subtype of `PyObject` represents a Python Unicode object.

PyTypeObject **PyUnicode_Type**

This instance of `PyTypeObject` represents the Python Unicode type. It is exposed to Python code as `unicode` and `types.UnicodeType`.

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.

Alterado na versão 2.2: Allowed subtypes to be accepted.

int **PyUnicode_CheckExact** (*PyObject* **o*)

Return true if the object *o* is a Unicode object, but not an instance of a subtype.

Novo na versão 2.2.

Py_ssize_t **PyUnicode_GET_SIZE** (*PyObject* **o*)

Return the size of the object. *o* has to be a `PyUnicodeObject` (not checked).

Alterado na versão 2.5: This function returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t **PyUnicode_GET_DATA_SIZE** (*PyObject* **o*)

Return the size of the object's internal buffer in bytes. *o* has to be a `PyUnicodeObject` (not checked).

Alterado na versão 2.5: This function returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

*Py_UNICODE** **PyUnicode_AS_UNICODE** (*PyObject* **o*)

Return a pointer to the internal `Py_UNICODE` buffer of the object. *o* has to be a `PyUnicodeObject` (not checked).

const char* **PyUnicode_AS_DATA** (*PyObject* **o*)

Return a pointer to the internal buffer of the object. *o* has to be a `PyUnicodeObject` (not checked).

int **PyUnicode_ClearFreeList** ()

Limpe a lista livre. Retorna o número total de itens liberados.

Novo na versão 2.6.

Unicode Character Properties

Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration.

int **Py_UNICODE_ISSPACE**(*Py_UNICODE ch*)
Return 1 or 0 depending on whether *ch* is a whitespace character.

int **Py_UNICODE_ISLOWER**(*Py_UNICODE ch*)
Return 1 or 0 depending on whether *ch* is a lowercase character.

int **Py_UNICODE_ISUPPER**(*Py_UNICODE ch*)
Return 1 or 0 depending on whether *ch* is an uppercase character.

int **Py_UNICODE_ISTITLE**(*Py_UNICODE ch*)
Return 1 or 0 depending on whether *ch* is a titlecase character.

int **Py_UNICODE_ISLINEBREAK**(*Py_UNICODE ch*)
Return 1 or 0 depending on whether *ch* is a linebreak character.

int **Py_UNICODE_ISDECIMAL**(*Py_UNICODE ch*)
Return 1 or 0 depending on whether *ch* is a decimal character.

int **Py_UNICODE_ISDIGIT**(*Py_UNICODE ch*)
Return 1 or 0 depending on whether *ch* is a digit character.

int **Py_UNICODE_ISNUMERIC**(*Py_UNICODE ch*)
Return 1 or 0 depending on whether *ch* is a numeric character.

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

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

These APIs can be used for fast direct character conversions:

Py_UNICODE **Py_UNICODE_TOLOWER**(*Py_UNICODE ch*)
Return the character *ch* converted to lower case.

Py_UNICODE **Py_UNICODE_TOUPPER**(*Py_UNICODE ch*)
Return the character *ch* converted to upper case.

Py_UNICODE **Py_UNICODE_TOTITLE**(*Py_UNICODE ch*)
Return the character *ch* converted to title case.

int **Py_UNICODE_TODECIMAL**(*Py_UNICODE 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_UNICODE 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_UNICODE ch*)
Return the character *ch* converted to a double. Return -1.0 if this is not possible. This macro does not raise exceptions.

Plain Py_UNICODE

To create Unicode objects and access their basic sequence properties, use these APIs:

*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*.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*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. *u* may also 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*.

Novo na versão 2.6.

PyObject ***PyUnicode_FromString**(const char **u*)
Return value: New reference. Create a Unicode object from a UTF-8 encoded null-terminated char buffer *u*.

Novo na versão 2.6.

*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* string. The following format characters are allowed:

Caracteres Formatados	Tipo	Comentário
`%%`	*n/d*	O caractere % literal.
`%c`	int	A single character, represented as a C int.
`%d`	int	Exactly equivalent to `printf("%d")`.
`%u`	unsigned int	Exactly equivalent to `printf("%u")`.
`%ld`	extenso, comprido	Exactly equivalent to `printf("%ld")`.
`%lu`	unsigned long	Exactly equivalent to `printf("%lu")`.
`%zd`	Py_stamanho_t	Exactly equivalent to `printf("%zd")`.
`%zu`	tamanho_t	Exactly equivalent to `printf("%zu")`.
`%i`	int	Exactly equivalent to `printf("%i")`.
`%x`	int	Exactly equivalent to `printf("%x")`.
`%s`	char*	Uma matriz de caracteres C com terminação nula.
`%p`	void*	A representação hexadecimal de um ponteiro C. Principalmente equivalente a " printf ("% p") " exceto que é garantido que comece com o literal " 0x" independentemente do que o " printf" da plataforma ceda.
`%U`	PyObject*	A unicode object.
`%V`	PyObject*, 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_Unicode()`.
`%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.

Novo na versão 2.6.

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

Return value: New reference. Identical to `PyUnicode_FromFormat()` except that it takes exactly two arguments.

Novo na versão 2.6.

*Py_UNICODE** **PyUnicode_AsUnicode** (*PyObject* **unicode*)

Return a read-only pointer to the Unicode object's internal `Py_UNICODE` buffer, *NULL* if *unicode* is not a Unicode object. Note that the resulting `Py_UNICODE*` string may contain embedded null characters, which would cause the string to be truncated when used in most C functions.

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

Return the length of the Unicode object.

Alterado na versão 2.5: This function returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyUnicode_FromEncodedObject** (*PyObject* **obj*, const char **encoding*, const char **errors*)

Return value: New reference. Coerce an encoded object *obj* to a Unicode object and return a reference with incremented refcount.

String and other char buffer compatible 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 the next section for details).

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

The API returns *NULL* if there was an error. The caller is responsible for decref'ing the returned objects.

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

Return value: New reference. Shortcut for `PyUnicode_FromEncodedObject(obj, NULL, "strict")` which is used throughout the interpreter whenever coercion to Unicode is needed.

If the platform supports `wchar_t` and provides a header file wchar.h, Python can interface directly to this type using the following functions. Support is optimized if Python's own `Py_UNICODE` type is identical to the system's `wchar_t`.

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*. Return *NULL* on failure.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t **PyUnicode_AsWideChar** (*PyUnicodeObject* **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 0-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 0-terminated. It is the responsibility of the caller to make sure that the `wchar_t` string is 0-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.

Alterado na versão 2.5: This function returned an `int` type and used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

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 `unicode()` Unicode object constructor.

Setting encoding to *NULL* causes the default encoding to be used which is ASCII. The file system calls should use `Py_FileSystemDefaultEncoding` as the encoding for file names. 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 deviation 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 `unicode()` 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.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*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 string 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.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*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 string object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode `encode()` method. The codec to be used is looked up using the Python codec registry. Return *NULL* if an exception was raised by the codec.

UTF-8 Codecs

These are the UTF-8 codec APIs:

*PyObject** **PyUnicode_DecodeUTF8**(const char **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.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*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*.

Novo na versão 2.4.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*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 string object. Return *NULL* if an exception was raised by the codec.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyUnicode_AsUTF8String**(*PyObject* *unicode)

Return value: New reference. Encode a Unicode object using UTF-8 and return the result as Python string object. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

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)

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.

In a narrow build code points outside the BMP will be decoded as surrogate pairs.

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

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

Novo na versão 2.6.

*PyObject** **PyUnicode_DecodeUTF32Stateful**(const char *s, Py_ssize_t size, const char *errors, int *byteorder, Py_ssize_t *consumed)

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*.

Novo na versão 2.6.

*PyObject** **PyUnicode_EncodeUTF32** (const *Py_UNICODE* **s*, Py_ssize_t *size*, const char **errors*, int *byteorder*)

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.

Novo na versão 2.6.

*PyObject** **PyUnicode_AsUTF32String** (*PyObject* **unicode*)

Return a Python 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.

Novo na versão 2.6.

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 `\uffe` character).

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

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

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

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*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*.

Novo na versão 2.4.

Alterado na versão 2.5: This function used an `int` type for *size* and an `int *` type for *consumed*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **`PyUnicode_EncodeUTF16`**(const *Py_UNICODE* **s*, Py_ssize_t *size*, const char **errors*, int *byteorder*)

Return value: New reference. Return a Python string 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.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **`PyUnicode_AsUTF16String`**(*PyObject* **unicode*)

Return value: New reference. Return a Python string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

UTF-7 Codecs

These are the UTF-7 codec APIs:

*PyObject** **`PyUnicode_DecodeUTF7`**(const char **s*, Py_ssize_t *size*, const char **errors*)

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*)

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*)

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.

Unicode-Escape Codecs

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.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*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 Python string object. Return *NULL* if an exception was raised by the codec.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyUnicode_AsUnicodeEscapeString**(*PyObject* **unicode*)
Return value: New reference. Encode a Unicode object using Unicode-Escape and return the result as Python string object. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

Raw-Unicode-Escape Codecs

These are the "Raw Unicode Escape" codec APIs:

*PyObject** **PyUnicode_DecodeRawUnicodeEscape**(const char **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.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyUnicode_EncodeRawUnicodeEscape**(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 Raw-Unicode-Escape and return a Python string object. Return *NULL* if an exception was raised by the codec.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyUnicode_AsRawUnicodeEscapeString**(*PyObject* **unicode*)
Return value: New reference. Encode a Unicode object using Raw-Unicode-Escape and return the result as Python string object. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

Latin-1 Codecs

These are the Latin-1 codec APIs: Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

*PyObject** **PyUnicode_DecodeLatin1**(const char **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.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*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 string object. Return *NULL* if an exception was raised by the codec.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyUnicode_AsLatin1String**(*PyObject* **unicode*)
Return value: New reference. Encode a Unicode object using Latin-1 and return the result as Python string object. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

ASCII Codecs

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

*PyObject** **PyUnicode_DecodeASCII**(const char **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.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*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 string object. Return *NULL* if an exception was raised by the codec.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyUnicode_AsASCIIString**(*PyObject* **unicode*)
Return value: New reference. Encode a Unicode object using ASCII and return the result as Python string object. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

Character Map Codecs

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the `encodings` package). The codec uses mapping to encode and decode characters.

Decoding mappings must map single string characters to single Unicode characters, integers (which are then interpreted as Unicode ordinals) or `None` (meaning "undefined mapping" and causing an error).

Encoding mappings must map single Unicode characters to single string characters, integers (which are then interpreted as Latin-1 ordinals) or `None` (meaning "undefined mapping" and causing an error).

The mapping objects provided must only support the __getitem__ mapping interface.

If a character lookup fails with a LookupError, the character is copied as-is meaning that its ordinal value will be interpreted as Unicode or Latin-1 ordinal resp. Because of this, mappings only need to contain those mappings which map characters to different code points.

These are the mapping codec APIs:

*PyObject** **PyUnicode_DecodeCharmap**(const char **s*, 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 done. Else it can be a dictionary mapping byte or a unicode string, which is treated as a lookup table. Byte values greater that the length of the string and U+FFFE "characters" are treated as "undefined mapping".

Alterado na versão 2.4: Allowed unicode string as mapping argument.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*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 a Python string object. Return *NULL* if an exception was raised by the codec.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*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 Python string object. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

The following codec API is special in that maps Unicode to Unicode.

*PyObject** **PyUnicode_TranslateCharmap** (const *Py_UNICODE* **s*, Py_ssize_t *size*, *PyObject* **table*, 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.

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.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

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.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyUnicode_DecodeMBCSStateful** (const char **s*, int *size*, const char **errors*, int **consumed*)

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*.

Novo na versão 2.5.

*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 string object. Return *NULL* if an exception was raised by the codec.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyUnicode_AsMBCSString** (*PyObject *unicode*)
Return value: New reference. Encode a Unicode object using MBCS and return the result as Python string object. Error handling is "strict". Return *NULL* if an exception was raised by the codec.

Methods & Slots

Methods and Slot Functions

The following APIs are capable of handling Unicode objects and strings on input (we refer to them as strings in the descriptions) and return Unicode objects or integers as appropriate.

They all return *NULL* or `-1` if an exception occurs.

*PyObject** **PyUnicode_Concat** (*PyObject *left*, *PyObject *right*)
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.

Alterado na versão 2.5: This function used an `int` type for *maxsplit*. This might require changes in your code for properly supporting 64-bit systems.

*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_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.

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_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.

Alterado na versão 2.5: This function used an `int` type for *start* and *end*. This might require changes in your code for properly supporting 64-bit systems.

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.

Alterado na versão 2.5: This function used an `int` type for *start* and *end*. This might require changes in your code for properly supporting 64-bit systems.

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.

Alterado na versão 2.5: This function returned an `int` type and used an `int` type for *start* and *end*. This might require changes in your code for properly supporting 64-bit systems.

*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.

Alterado na versão 2.5: This function used an `int` type for *maxcount*. This might require changes in your code for properly supporting 64-bit systems.

int **PyUnicode_Compare** (*PyObject* **left*, *PyObject* **right*)
Compare two strings and return `-1`, `0`, `1` for less than, equal, and greater than, respectively.

int **PyUnicode_RichCompare** (*PyObject* **left*, *PyObject* **right*, int *op*)
Rich compare two unicode strings and return one of the following:

- `NULL` in case an exception was raised
- `Py_True` or `Py_False` for successful comparisons
- `Py_NotImplemented` in case the type combination is unknown

Note that `Py_EQ` and `Py_NE` comparisons can cause a `UnicodeWarning` in case the conversion of the arguments to Unicode fails with a `UnicodeDecodeError`.

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.

7.3.4 Buffers and Memoryview Objects

Python objects implemented in C can export a group of functions called the "buffer interface." These functions can be used by an object to expose its data in a raw, byte-oriented format. Clients of the object can use the buffer interface to access the object data directly, without needing to copy it first.

Two examples of objects that support the buffer interface are strings and arrays. The string object exposes the character contents in the buffer interface's byte-oriented form. An array can only expose its contents via the old-style buffer interface. This limitation does not apply to Python 3, where `memoryview` objects can be constructed from arrays, too. Array elements may be multi-byte values.

An example user of the buffer interface is the file object's `write()` method. Any object that can export a series of bytes through the buffer interface can be written to a file. There are a number of format codes to `PyArg_ParseTuple()` that operate against an object's buffer interface, returning data from the target object.

Starting from version 1.6, Python has been providing Python-level buffer objects and a C-level buffer API so that any built-in or used-defined type can expose its characteristics. Both, however, have been deprecated because of various shortcomings, and have been officially removed in Python 3 in favour of a new C-level buffer API and a new Python-level object named `memoryview`.

The new buffer API has been backported to Python 2.6, and the `memoryview` object has been backported to Python 2.7. It is strongly advised to use them rather than the old APIs, unless you are blocked from doing so for compatibility reasons.

The new-style Py_buffer struct

Py_buffer

void ***buf**
A pointer to the start of the memory for the object.

Py_ssize_t **len**
The total length of the memory in bytes.

int **readonly**
An indicator of whether the buffer is read only.

const char ***format**
A *NULL* terminated string in `struct` module style syntax giving the contents of the elements available through the buffer. If this is *NULL*, `"B"` (unsigned bytes) is assumed.

int **ndim**
The number of dimensions the memory represents as a multi-dimensional array. If it is `0`, `strides` and `suboffsets` must be *NULL*.

Py_ssize_t ***shape**
An array of `Py_ssize_t`s the length of `ndim` giving the shape of the memory as a multi-dimensional array. Note that `((*shape)[0] * ... * (*shape)[ndims-1])*itemsize` should be equal to `len`.

Py_ssize_t ***strides**
An array of `Py_ssize_t`s the length of `ndim` giving the number of bytes to skip to get to a new element in each dimension.

Py_ssize_t ***suboffsets**
An array of `Py_ssize_t`s the length of `ndim`. If these suboffset numbers are greater than or equal to 0, then the value stored along the indicated dimension is a pointer and the suboffset value dictates how many bytes to add to the pointer after de-referencing. A suboffset value that it negative indicates that no de-referencing should occur (striding in a contiguous memory block).

If all suboffsets are negative (i.e. no de-referencing is needed), then this field must be NULL (the default value).

Here is a function that returns a pointer to the element in an N-D array pointed to by an N-dimensional index when there are both non-NULL strides and suboffsets:

```
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
    Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >=0 ) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

Py_ssize_t **itemsize**

This is a storage for the itemsize (in bytes) of each element of the shared memory. It is technically unnecessary as it can be obtained using `PyBuffer_SizeFromFormat()`, however an exporter may know this information without parsing the format string and it is necessary to know the itemsize for proper interpretation of striding. Therefore, storing it is more convenient and faster.

void ***internal**

This is for use internally by the exporting object. For example, this might be re-cast as an integer by the exporter and used to store flags about whether or not the shape, strides, and suboffsets arrays must be freed when the buffer is released. The consumer should never alter this value.

Buffer related functions

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

Return `1` if *obj* supports the buffer interface otherwise `0`.

int **PyObject_GetBuffer**(*PyObject *obj*, *Py_buffer *view*, int *flags*)

Export *obj* into a `Py_buffer`, *view*. These arguments must never be *NULL*. The *flags* argument is a bit field indicating what kind of buffer the caller is prepared to deal with and therefore what kind of buffer the exporter is allowed to return. The buffer interface allows for complicated memory sharing possibilities, but some caller may not be able to handle all the complexity but may want to see if the exporter will let them take a simpler view to its memory.

Some exporters may not be able to share memory in every possible way and may need to raise errors to signal to some consumers that something is just not possible. These errors should be a `BufferError` unless there is another error that is actually causing the problem. The exporter can use flags information to simplify how much of the `Py_buffer` structure is filled in with non-default values and/or raise an error if the object can't support a simpler view of its memory.

`0` is returned on success and `-1` on error.

The following table gives possible values to the *flags* arguments.

Flag	Description
PyBUF_SIMPLE	This is the default flag state. The returned buffer may or may not have writable memory. The format of the data will be assumed to be unsigned bytes. This is a "stand-alone" flag constant. It never needs to be 'l'd to the others. The exporter will raise an error if it cannot provide such a contiguous buffer of bytes.
PyBUF_WRITABLE	The returned buffer must be writable. If it is not writable, then raise an error.
PyBUF_STRIDES	This implies `PyBUF_ND`. The returned buffer must provide strides information (i.e. the strides cannot be NULL). This would be used when the consumer can handle strided, discontiguous arrays. Handling strides automatically assumes you can handle shape. The exporter can raise an error if a strided representation of the data is not possible (i.e. without the suboffsets).
PyBUF_ND	The returned buffer must provide shape information. The memory will be assumed C-style contiguous (last dimension varies the fastest). The exporter may raise an error if it cannot provide this kind of contiguous buffer. If this is not given then shape will be *NULL*.
PyBUF_C_CONTIGUOUS PyBUF_F_CONTIGUOUS PyBUF_ANY_CONTIGUOUS	These flags indicate that the contiguity returned buffer must be respectively, C-contiguous (last dimension varies the fastest), Fortran contiguous (first dimension varies the fastest) or either one. All of these flags imply `PyBUF_STRIDES` and guarantee that the strides buffer info structure will be filled in correctly.
PyBUF_INDIRECT	This flag indicates the returned buffer must have suboffsets information (which can be NULL if no suboffsets are needed). This can be used when the consumer can handle indirect array referencing implied by these suboffsets. This implies `PyBUF_STRIDES`.
PyBUF_FORMAT	The returned buffer must have true format information if this flag is provided. This would be used when the consumer is going to be checking for what 'kind' of data is actually stored. An exporter should always be able to provide this information if requested. If format is not explicitly requested then the format must be returned as *NULL* (which means `'B'`, or unsigned bytes)
PyBUF_STRIDED	This is equivalent to `(PyBUF_STRIDES | PyBUF_WRITABLE)`.
PyBUF_STRIDED_RO	This is equivalent to `(PyBUF_STRIDES)`.
PyBUF_RECORDS	This is equivalent to `(PyBUF_STRIDES | PyBUF_FORMAT | PyBUF_WRITABLE)`.
PyBUF_RECORDS_RO	This is equivalent to `(PyBUF_STRIDES | PyBUF_FORMAT)`.
PyBUF_FULL	This is equivalent to `(PyBUF_INDIRECT | PyBUF_FORMAT | PyBUF_WRITABLE)`.
PyBUF_FULL_RO	This is equivalent to `(PyBUF_INDIRECT | PyBUF_FORMAT)`.
PyBUF_CONTIG	This is equivalent to `(PyBUF_ND | PyBUF_WRITABLE)`.
PyBUF_CONTIG_RO	This is equivalent to `(PyBUF_ND)`.

void **PyBuffer_Release**(*Py_buffer *view*)

Release the buffer *view*. This should be called when the buffer is no longer being used as it may free memory from it.

Py_ssize_t **PyBuffer_SizeFromFormat**(const char *)

Return the implied *itemsize* from the struct-stype `format`.

int **PyBuffer_IsContiguous**(*Py_buffer *view*, char *fortran*)

Return `1` if the memory defined by the *view* is C-style (*fortran* is `'C'`) or Fortran-style (*fortran* is `'F'`) contiguous or either one (*fortran* is `'A'`). Return `0` otherwise.

void **PyBuffer_FillContiguousStrides**(int *ndims*, Py_ssize_t **shape*, Py_ssize_t **strides*, int *itemsize*, char *fortran*)

Fill the *strides* array with byte-strides of a contiguous (C-style if *fortran* is `'C'` or Fortran-style if *fortran* is `'F'`) array of the given shape with the given number of bytes per element.

int **PyBuffer_FillInfo**(*Py_buffer* **view*, *PyObject* **obj*, void **buf*, Py_ssize_t *len*, int *readonly*, int *infoflags*)
Fill in a buffer-info structure, *view*, correctly for an exporter that can only share a contiguous chunk of memory of "unsigned bytes" of the given length. Return `0` on success and `-1` (with raising an error) on error.

MemoryView objects

Novo na versão 2.7.

A `memoryview` object exposes the new C level buffer interface as a Python object which can then be passed around like any other object.

PyObject ***PyMemoryView_FromObject**(*PyObject* **obj*)
Create a memoryview object from an object that defines the new buffer interface.

PyObject ***PyMemoryView_FromBuffer**(*Py_buffer* **view*)
Create a memoryview object wrapping the given buffer-info structure *view*. The memoryview object then owns the buffer, which means you shouldn't try to release it yourself: it will be released on deallocation of the memoryview object.

PyObject ***PyMemoryView_GetContiguous**(*PyObject* **obj*, int *buffertype*, char *order*)
Create a memoryview object to a contiguous chunk of memory (in either 'C' or 'F'ortran *order*) from an object that defines the buffer interface. If memory is contiguous, the memoryview object points to the original memory. Otherwise copy is made and the memoryview points to a new bytes object.

int **PyMemoryView_Check**(*PyObject* **obj*)
Return true if the object *obj* is a memoryview object. It is not currently allowed to create subclasses of `memoryview`.

Py_buffer ***PyMemoryView_GET_BUFFER**(*PyObject* **obj*)
Return a pointer to the buffer-info structure wrapped by the given object. The object **must** be a memoryview instance; this macro doesn't check its type, you must do it yourself or you will risk crashes.

Old-style buffer objects

More information on the old buffer interface is provided in the section *Buffer Object Structures*, under the description for `PyBufferProcs`.

A "buffer object" is defined in the `bufferobject.h` header (included by `Python.h`). These objects look very similar to string objects at the Python programming level: they support slicing, indexing, concatenation, and some other standard string operations. However, their data can come from one of two sources: from a block of memory, or from another object which exports the buffer interface.

Buffer objects are useful as a way to expose the data from another object's buffer interface to the Python programmer. They can also be used as a zero-copy slicing mechanism. Using their ability to reference a block of memory, it is possible to expose any data to the Python programmer quite easily. The memory could be a large, constant array in a C extension, it could be a raw block of memory for manipulation before passing to an operating system library, or it could be used to pass around structured data in its native, in-memory format.

PyBufferObject
This subtype of `PyObject` represents a buffer object.

PyTypeObject **PyBuffer_Type**
The instance of `PyTypeObject` which represents the Python buffer type; it is the same object as `buffer` and `types.BufferType` in the Python layer. .

int **Py_END_OF_BUFFER**
This constant may be passed as the *size* parameter to `PyBuffer_FromObject()` or

PyBuffer_FromReadWriteObject(). It indicates that the new *PyBufferObject* should refer to *base* object from the specified *offset* to the end of its exported buffer. Using this enables the caller to avoid querying the *base* object for its length.

int **PyBuffer_Check** (*PyObject* **p*)
Return true if the argument has type *PyBuffer_Type*.

*PyObject** **PyBuffer_FromObject** (*PyObject* **base*, Py_ssize_t *offset*, Py_ssize_t *size*)
Return value: New reference. Return a new read-only buffer object. This raises `TypeError` if *base* doesn't support the read-only buffer protocol or doesn't provide exactly one buffer segment, or it raises `ValueError` if *offset* is less than zero. The buffer will hold a reference to the *base* object, and the buffer's contents will refer to the *base* object's buffer interface, starting as position *offset* and extending for *size* bytes. If *size* is `Py_END_OF_BUFFER`, then the new buffer's contents extend to the length of the *base* object's exported buffer data.

Alterado na versão 2.5: This function used an `int` type for *offset* and *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyBuffer_FromReadWriteObject** (*PyObject* **base*, Py_ssize_t *offset*, Py_ssize_t *size*)
Return value: New reference. Return a new writable buffer object. Parameters and exceptions are similar to those for *PyBuffer_FromObject()*. If the *base* object does not export the writeable buffer protocol, then `TypeError` is raised.

Alterado na versão 2.5: This function used an `int` type for *offset* and *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyBuffer_FromMemory** (void **ptr*, Py_ssize_t *size*)
Return value: New reference. Return a new read-only buffer object that reads from a specified location in memory, with a specified size. The caller is responsible for ensuring that the memory buffer, passed in as *ptr*, is not deallocated while the returned buffer object exists. Raises `ValueError` if *size* is less than zero. Note that `Py_END_OF_BUFFER` may *not* be passed for the *size* parameter; `ValueError` will be raised in that case.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyBuffer_FromReadWriteMemory** (void **ptr*, Py_ssize_t *size*)
Return value: New reference. Similar to *PyBuffer_FromMemory()*, but the returned buffer is writable.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyBuffer_New** (Py_ssize_t *size*)
Return value: New reference. Return a new writable buffer object that maintains its own memory buffer of *size* bytes. `ValueError` is returned if *size* is not zero or positive. Note that the memory buffer (as returned by *PyObject_AsWriteBuffer()*) is not specifically aligned.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

7.3.5 Objeto Tuple

PyTupleObject
Este subtipo de *PyObject* representa um objeto tupla em Python.

PyTypeObject **PyTuple_Type**
This instance of *PyTypeObject* represents the Python tuple type; it is the same object as `tuple` and `types.TupleType` in the Python layer..

int **PyTuple_Check** (*PyObject* **p*)
Devolve verdadeiro se *p* é um objeto tupla ou uma instância de um subtipo do tipo tupla.

Alterado na versão 2.2: Allowed subtypes to be accepted.

int **PyTuple_CheckExact**(*PyObject* *p*)
Devolve verdadeiro se *p* é um objeto tupla, mas não uma instância de um subtipo do tipo tupla.

Novo na versão 2.2.

*PyObject** **PyTuple_New**(Py_ssize_t *len*)
Return value: New reference. Return a new tuple object of size *len*, or *NULL* on failure.

Alterado na versão 2.5: This function used an `int` type for *len*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyTuple_Pack**(Py_ssize_t *n*, ...)
Return value: New reference. Return a new tuple object of size *n*, or *NULL* on failure. The tuple values are initialized to the subsequent *n* C arguments pointing to Python objects. `PyTuple_Pack(2, a, b)` is equivalent to `Py_BuildValue("(OO)", a, b)`.

Novo na versão 2.4.

Alterado na versão 2.5: This function used an `int` type for *n*. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t **PyTuple_Size**(*PyObject* *p*)
Take a pointer to a tuple object, and return the size of that tuple.

Alterado na versão 2.5: This function returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t **PyTuple_GET_SIZE**(*PyObject* *p*)
Return the size of the tuple *p*, which must be non-*NULL* and point to a tuple; no error checking is performed.

Alterado na versão 2.5: This function returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyTuple_GetItem**(*PyObject* *p*, Py_ssize_t *pos*)
Return value: Borrowed reference. Return the object at position *pos* in the tuple pointed to by *p*. If *pos* is out of bounds, return *NULL* and set an `IndexError` exception.

Alterado na versão 2.5: This function used an `int` type for *pos*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyTuple_GET_ITEM**(*PyObject* *p*, Py_ssize_t *pos*)
Return value: Borrowed reference. Like `PyTuple_GetItem()`, but does no checking of its arguments.

Alterado na versão 2.5: This function used an `int` type for *pos*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyTuple_GetSlice**(*PyObject* *p*, Py_ssize_t *low*, Py_ssize_t *high*)
Return value: New reference. Return the slice of the tuple pointed to by *p* between *low* and *high*, or *NULL* on failure. This is the equivalent of the Python expression `p[low:high]`. Indexing from the end of the list is not supported.

Alterado na versão 2.5: This function used an `int` type for *low* and *high*. This might require changes in your code for properly supporting 64-bit systems.

int **PyTuple_SetItem**(*PyObject* *p*, Py_ssize_t *pos*, *PyObject* *o*)
Insert a reference to object *o* at position *pos* of the tuple pointed to by *p*. Return `0` on success. If *pos* is out of bounds, return `-1` and set an `IndexError` exception.

Nota: This function "steals" a reference to *o* and discards a reference to an item already in the tuple at the affected position.

Alterado na versão 2.5: This function used an `int` type for *pos*. This might require changes in your code for properly supporting 64-bit systems.

void **PyTuple_SET_ITEM**(*PyObject *p*, Py_ssize_t *pos*, *PyObject *o*)
Like `PyTuple_SetItem()`, but does no error checking, and should *only* be used to fill in brand new tuples.

Nota: This macro "steals" a reference to *o*, and, unlike `PyTuple_SetItem()`, does *not* discard a reference to any item that is being replaced; any reference in the tuple at position *pos* will be leaked.

Alterado na versão 2.5: This function used an `int` type for *pos*. This might require changes in your code for properly supporting 64-bit systems.

int **_PyTuple_Resize**(*PyObject **p*, Py_ssize_t *newsize*)
Can be used to resize a tuple. *newsize* will be the new length of the tuple. Because tuples are *supposed* to be immutable, this should only be used if there is only one reference to the object. Do *not* use this if the tuple may already be known to some other part of the code. The tuple will always grow or shrink at the end. Think of this as destroying the old tuple and creating a new one, only more efficiently. Returns `0` on success. Client code should never assume that the resulting value of `*p` will be the same as before calling this function. If the object referenced by `*p` is replaced, the original `*p` is destroyed. On failure, returns `-1` and sets `*p` to *NULL*, and raises `MemoryError` or `SystemError`.

Alterado na versão 2.2: Removed unused third parameter, *last_is_sticky*.

Alterado na versão 2.5: This function used an `int` type for *newsize*. This might require changes in your code for properly supporting 64-bit systems.

int **PyTuple_ClearFreeList**()
Limpe a lista livre. Retorna o número total de itens liberados.

Novo na versão 2.6.

7.3.6 Objeto List

PyListObject
This subtype of `PyObject` represents a Python list object.

PyTypeObject **PyList_Type**
This instance of `PyTypeObject` represents the Python list type. This is the same object as `list` in the Python layer.

int **PyList_Check**(*PyObject *p*)
Return true if *p* is a list object or an instance of a subtype of the list type.

Alterado na versão 2.2: Allowed subtypes to be accepted.

int **PyList_CheckExact**(*PyObject *p*)
Return true if *p* is a list object, but not an instance of a subtype of the list type.

Novo na versão 2.2.

*PyObject** **PyList_New**(Py_ssize_t *len*)
Return value: New reference. Return a new list of length *len* on success, or *NULL* on failure.

Nota: If *len* is greater than zero, the returned list object's items are set to `NULL`. Thus you cannot use abstract API functions such as `PySequence_SetItem()` or expose the object to Python code before setting all items to a real object with `PyList_SetItem()`.

Alterado na versão 2.5: This function used an `int` for *size*. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t **PyList_Size**(*PyObject* **list*)

Return the length of the list object in *list*; this is equivalent to `len(list)` on a list object.

Alterado na versão 2.5: This function returned an `int`. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t **PyList_GET_SIZE**(*PyObject* **list*)

Macro form of `PyList_Size()` without error checking.

Alterado na versão 2.5: This macro returned an `int`. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyList_GetItem**(*PyObject* **list*, Py_ssize_t *index*)

Return value: Borrowed reference. Return the object at position *index* in the list pointed to by *list*. The position must be non-negative; indexing from the end of the list is not supported. If *index* is out of bounds (<0 or >=len(list)), return *NULL* and set an `IndexError` exception.

Alterado na versão 2.5: This function used an `int` for *index*. This might require changes in your code for properly supporting 64-bit systems.

*PyObject** **PyList_GET_ITEM**(*PyObject* **list*, Py_ssize_t *i*)

Return value: Borrowed reference. Macro form of `PyList_GetItem()` without error checking.

Alterado na versão 2.5: This macro used an `int` for *i*. This might require changes in your code for properly supporting 64-bit systems.

int **PyList_SetItem**(*PyObject* **list*, Py_ssize_t *index*, *PyObject* **item*)

Set the item at index *index* in list to *item*. Return `0` on success. If *index* is out of bounds, return `-1` and set an `IndexError` exception.

Nota: Esta função "rouba" uma referência para o *item* e descarta uma referência para um item já presente na lista na posição afetada.

Alterado na versão 2.5: This function used an `int` for *index*. This might require changes in your code for properly supporting 64-bit systems.

void **PyList_SET_ITEM**(*PyObject* **list*, Py_ssize_t *i*, *PyObject* **o*)

Forma macro de `PyList_SetItem()` sem checagem de erro. Este é normalmente usado apenas para preencher novas listas onde não há conteúdo anterior.

Nota: This macro "steals" a reference to *item*, and, unlike `PyList_SetItem()`, does *not* discard a reference to any item that it being replaced; any reference in *list* at position *i* will be leaked.

Alterado na versão 2.5: This macro used an `int` for *i*. This might require changes in your code for properly supporting 64-bit systems.

int **PyList_Insert**(*PyObject* **list*, Py_ssize_t *index*, *PyObject* **item*)

Insert the item *item* into list *list* in front of index *index*. Return `0` if successful; return `-1` and set an exception if unsuccessful. Analogous to `list.insert(index, item)`.

Alterado na versão 2.5: This function used an `int` for *index*. This might require changes in your code for properly supporting 64-bit systems.

int **PyList_Append** (*PyObject *list*, *PyObject *item*)

Append the object *item* at the end of list *list*. Return `0` if successful; return `-1` and set an exception if unsuccessful. Analogous to `list.append(item)`.

*PyObject** **PyList_GetSlice** (*PyObject *list*, Py_ssize_t *low*, Py_ssize_t *high*)

Return value: New reference. Return a list of the objects in *list* containing the objects *between low* and *high*. Return *NULL* and set an exception if unsuccessful. Analogous to `list[low:high]`. Indexing from the end of the list is not supported.

Alterado na versão 2.5: This function used an `int` for *low* and *high*. This might require changes in your code for properly supporting 64-bit systems.

int **PyList_SetSlice** (*PyObject *list*, Py_ssize_t *low*, Py_ssize_t *high*, *PyObject *itemlist*)

Set the slice of *list* between *low* and *high* to the contents of *itemlist*. Analogous to `list[low:high] = itemlist`. The *itemlist* may be *NULL*, indicating the assignment of an empty list (slice deletion). Return `0` on success, `-1` on failure. Indexing from the end of the list is not supported.

Alterado na versão 2.5: This function used an `int` for *low* and *high*. This might require changes in your code for properly supporting 64-bit systems.

int **PyList_Sort** (*PyObject *list*)

Sort the items of *list* in place. Return `0` on success, `-1` on failure. This is equivalent to `list.sort()`.

int **PyList_Reverse** (*PyObject *list*)

Inverte os termos de *list* no mesmo lugar. Retorna `0` em caso de sucesso, e `-1` em caso de falha. Isso é o equivalente de `list.reverse()`.

*PyObject** **PyList_AsTuple** (*PyObject *list*)

Return value: New reference. Retorna um novo objeto tupla contendo os conteúdos de *list*; equivale a"tuple(list)".

7.4 Mapping Objects

7.4.1 Objetos Dicionário

PyDictObject

Este subtipo do *PyObject* representa um objeto dicionário Python.

PyTypeObject **PyDict_Type**

This instance of *PyTypeObject* represents the Python dictionary type. This is exposed to Python programs as `dict` and `types.DictType`.

int **PyDict_Check** (*PyObject *p*)

Retorna verdadeiro se *p* é um objeto dict ou uma instância de um subtipo do tipo dict.

Alterado na versão 2.2: Allowed subtypes to be accepted.

int **PyDict_CheckExact** (*PyObject *p*)

Return true if *p* is a dict object, but not an instance of a subtype of the dict type.

Novo na versão 2.4.

*PyObject** **PyDict_New** ()

Return value: New reference. Return a new empty dictionary, or *NULL* on failure.

*PyObject** **PyDictProxy_New**(*PyObject *dict*)
Return value: New reference. Return a proxy object for a mapping which enforces read-only behavior. This is normally used to create a proxy to prevent modification of the dictionary for non-dynamic class types.

Novo na versão 2.2.

void **PyDict_Clear**(*PyObject *p*)
Esvazie um dicionário existente de todos os pares chave-valor.

int **PyDict_Contains**(*PyObject *p*, *PyObject *key*)
Determine if dictionary *p* contains *key*. If an item in *p* is matches *key*, return `1`, otherwise return `0`. On error, return `-1`. This is equivalent to the Python expression `key in p`.

Novo na versão 2.4.

*PyObject** **PyDict_Copy**(*PyObject *p*)
Return value: New reference. Retorna um novo dicionário que contém o mesmo chave-valor como *p*.

Novo na versão 1.6.

int **PyDict_SetItem**(*PyObject *p*, *PyObject *key*, *PyObject *val*)
Insert *value* into the dictionary *p* with a key of *key*. *key* must be *hashable*; if it isn't, `TypeError` will be raised. Return `0` on success or `-1` on failure.

int **PyDict_SetItemString**(*PyObject *p*, const char **key*, *PyObject *val*)
Insert *value* into the dictionary *p* using *key* as a key. *key* should be a `char*`. The key object is created using `PyString_FromString(key)`. Return `0` on success or `-1` on failure.

int **PyDict_DelItem**(*PyObject *p*, *PyObject *key*)
Remove the entry in dictionary *p* with key *key*. *key* must be hashable; if it isn't, `TypeError` is raised. Return `0` on success or `-1` on failure.

int **PyDict_DelItemString**(*PyObject *p*, char **key*)
Remove a entrada no dicionário *p*, que possui uma chave especificada pela string *chave*. Retorna `0` em caso de sucesso ou `-1` em caso de falha.

*PyObject** **PyDict_GetItem**(*PyObject *p*, *PyObject *key*)
Return value: Borrowed reference. Return the object from dictionary *p* which has a key *key*. Return *NULL* if the key *key* is not present, but *without* setting an exception.

*PyObject** **PyDict_GetItemString**(*PyObject *p*, const char **key*)
Return value: Borrowed reference. This is the same as `PyDict_GetItem()`, but *key* is specified as a `char*`, rather than a `PyObject*`.

*PyObject** **PyDict_Items**(*PyObject *p*)
Return value: New reference. Return a `PyListObject` containing all the items from the dictionary, as in the dictionary method `dict.items()`.

*PyObject** **PyDict_Keys**(*PyObject *p*)
Return value: New reference. Return a `PyListObject` containing all the keys from the dictionary, as in the dictionary method `dict.keys()`.

*PyObject** **PyDict_Values**(*PyObject *p*)
Return value: New reference. Return a `PyListObject` containing all the values from the dictionary *p*, as in the dictionary method `dict.values()`.

Py_ssize_t **PyDict_Size**(*PyObject *p*)
Retorna o número de itens no dicionário. Isso é equivalente à `len(p)` em um dicionário.

Alterado na versão 2.5: This function returned an `int` type. This might require changes in your code for properly supporting 64-bit systems.

int **PyDict_Next** (*PyObject* *p, Py_ssize_t *ppos, *PyObject* **pkey, *PyObject* **pvalue)

Iterate over all key-value pairs in the dictionary *p*. The `Py_ssize_t` referred to by *ppos* must be initialized to `0` prior to the first call to this function to start the iteration; the function returns true for each pair in the dictionary, and false once all pairs have been reported. The parameters *pkey* and *pvalue* should either point to `PyObject*` variables that will be filled in with each key and value, respectively, or may be *NULL*. Any references returned through them are borrowed. *ppos* should not be altered during iteration. Its value represents offsets within the internal dictionary structure, and since the structure is sparse, the offsets are not consecutive.

Por exemplo:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}
```

The dictionary *p* should not be mutated during iteration. It is safe (since Python 2.1) to modify the values of the keys as you iterate over the dictionary, but only so long as the set of keys does not change. For example:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    int i = PyInt_AS_LONG(value) + 1;
    PyObject *o = PyInt_FromLong(i);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
    Py_DECREF(o);
}
```

Alterado na versão 2.5: This function used an `int *` type for *ppos*. This might require changes in your code for properly supporting 64-bit systems.

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

Iterate over mapping object *b* adding key-value pairs to dictionary *a*. *b* may be a dictionary, or any object supporting `PyMapping_Keys()` and `PyObject_GetItem()`. If *override* is true, existing pairs in *a* will be replaced if a matching key is found in *b*, otherwise pairs will only be added if there is not a matching key in *a*. Return `0` on success or `-1` if an exception was raised.

Novo na versão 2.2.

int **PyDict_Update** (*PyObject* *a, *PyObject* *b)

This is the same as `PyDict_Merge(a, b, 1)` in C, and is similar to `a.update(b)` in Python except that `PyDict_Update()` doesn't fall back to the iterating over a sequence of key value pairs if the second argument has no "keys" attribute. Return `0` on success or `-1` if an exception was raised.

Novo na versão 2.2.

int **PyDict_MergeFromSeq2** (*PyObject* *a, *PyObject* *seq2, int *override*)

Update or merge into dictionary *a*, from the key-value pairs in *seq2*. *seq2* must be an iterable object producing iterable objects of length 2, viewed as key-value pairs. In case of duplicate keys, the last wins if *override* is true, else the first wins. Return `0` on success or `-1` if an exception was raised. Equivalent Python (except for the return value):

```
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value
```

Novo na versão 2.2.

7.5 Other Objects

7.5.1 Class and Instance Objects

Note that the class objects described here represent old-style classes, which will go away in Python 3. When creating new types for extension modules, you will want to work with type objects (section *Objetos de tipo*).

PyClassObject
The C structure of the objects used to describe built-in classes.

*PyObject** **PyClass_Type**
This is the type object for class objects; it is the same object as `types.ClassType` in the Python layer.

int **PyClass_Check**(*PyObject* *o)
Return true if the object *o* is a class object, including instances of types derived from the standard class object. Return false in all other cases.

int **PyClass_IsSubclass**(*PyObject* *klass, *PyObject* *base)
Return true if *klass* is a subclass of *base*. Return false in all other cases.

There are very few functions specific to instance objects.

PyTypeObject **PyInstance_Type**
Type object for class instances.

int **PyInstance_Check**(*PyObject* *obj)
Return true if *obj* is an instance.

*PyObject** **PyInstance_New**(*PyObject* *class, *PyObject* *arg, *PyObject* *kw)
Return value: New reference. Create a new instance of a specific class. The parameters *arg* and *kw* are used as the positional and keyword parameters to the object's constructor.

*PyObject** **PyInstance_NewRaw**(*PyObject* *class, *PyObject* *dict)
Return value: New reference. Create a new instance of a specific class without calling its constructor. *class* is the class of new object. The *dict* parameter will be used as the object's `__dict__`; if *NULL*, a new dictionary will be created for the instance.

7.5.2 Objetos de Função

Existem algumas funções específicas para as funções do Python.

PyFunctionObject
A estrutura C usada para funções.

PyTypeObject **PyFunction_Type**
Esta é uma instância de `PyTypeObject` e representa o tipo de função Python. Está exposto aos programadores Python como `types.FunctionType`.

int **PyFunction_Check**(*PyObject* *o)
Return true if *o* is a function object (has type `PyFunction_Type`). The parameter must not be *NULL*.

*PyObject** **PyFunction_New** (*PyObject *code*, *PyObject *globals*)
Return value: New reference. Retorna um novo objeto de função associado ao código objeto *code*. *globals* deve ser um dicionário com as variáveis globais acessíveis à função.

The function's docstring, name and *__module__* are retrieved from the code object, the argument defaults and closure are set to *NULL*.

*PyObject** **PyFunction_GetCode** (*PyObject *op*)
Return value: Borrowed reference. Retorna o objeto de código associado ao objeto de função *op*.

*PyObject** **PyFunction_GetGlobals** (*PyObject *op*)
Return value: Borrowed reference. Retorna o dicionário global associado ao objeto de função *op*.

*PyObject** **PyFunction_GetModule** (*PyObject *op*)
Return value: Borrowed reference. Retorna o atributo *__module__* do objeto de função *op*. Esta é normalmente uma string contendo o nome do módulo, mas pode ser configurada para qualquer outro objeto pelo código Python.

*PyObject** **PyFunction_GetDefaults** (*PyObject *op*)
Return value: Borrowed reference. Return the argument default values of the function object *op*. This can be a tuple of arguments or *NULL*.

int **PyFunction_SetDefaults** (*PyObject *op*, *PyObject *defaults*)
Set the argument default values for the function object *op*. *defaults* must be *Py_None* or a tuple.

Levanta `SystemError` e retorna `-1` em falha.

*PyObject** **PyFunction_GetClosure** (*PyObject *op*)
Return value: Borrowed reference. Return the closure associated with the function object *op*. This can be *NULL* or a tuple of cell objects.

int **PyFunction_SetClosure** (*PyObject *op*, *PyObject *closure*)
Set the closure associated with the function object *op*. *closure* must be *Py_None* or a tuple of cell objects.

Levanta `SystemError` e retorna `-1` em falha.

7.5.3 Objetos de Método

There are some useful functions that are useful for working with method objects.

PyTypeObject **PyMethod_Type**
This instance of `PyTypeObject` represents the Python method type. This is exposed to Python programs as `types.MethodType`.

int **PyMethod_Check** (*PyObject *o*)
Return true if *o* is a method object (has type `PyMethod_Type`). The parameter must not be *NULL*.

*PyObject** **PyMethod_New** (*PyObject *func*, *PyObject *self*, *PyObject *class*)
Return value: New reference. Return a new method object, with *func* being any callable object; this is the function that will be called when the method is called. If this method should be bound to an instance, *self* should be the instance and *class* should be the class of *self*, otherwise *self* should be *NULL* and *class* should be the class which provides the unbound method..

*PyObject** **PyMethod_Class** (*PyObject *meth*)
Return value: Borrowed reference. Return the class object from which the method *meth* was created; if this was created from an instance, it will be the class of the instance.

*PyObject** **PyMethod_GET_CLASS** (*PyObject *meth*)
Return value: Borrowed reference. Macro version of `PyMethod_Class()` which avoids error checking.

*PyObject** **PyMethod_Function** (*PyObject *meth*)
Return value: Borrowed reference. Return the function object associated with the method *meth*.

*PyObject** **PyMethod_GET_FUNCTION**(*PyObject *meth*)
Return value: Borrowed reference. Macro version of `PyMethod_Function()` which avoids error checking.

*PyObject** **PyMethod_Self**(*PyObject *meth*)
Return value: Borrowed reference. Return the instance associated with the method *meth* if it is bound, otherwise return *NULL*.

*PyObject** **PyMethod_GET_SELF**(*PyObject *meth*)
Return value: Borrowed reference. Macro version of `PyMethod_Self()` which avoids error checking.

int **PyMethod_ClearFreeList**()
Limpe a lista livre. Retorna o número total de itens liberados.

Novo na versão 2.6.

7.5.4 Objetos File

Python's built-in file objects are implemented entirely on the `FILE*` support from the C standard library. This is an implementation detail and may change in future releases of Python.

PyFileObject
This subtype of `PyObject` represents a Python file object.

PyTypeObject **PyFile_Type**
This instance of `PyTypeObject` represents the Python file type. This is exposed to Python programs as `file` and `types.FileType`.

int **PyFile_Check**(*PyObject *p*)
Return true if its argument is a `PyFileObject` or a subtype of `PyFileObject`.

Alterado na versão 2.2: Allowed subtypes to be accepted.

int **PyFile_CheckExact**(*PyObject *p*)
Return true if its argument is a `PyFileObject`, but not a subtype of `PyFileObject`.

Novo na versão 2.2.

*PyObject** **PyFile_FromString**(char **filename*, char **mode*)
Return value: New reference. On success, return a new file object that is opened on the file given by *filename*, with a file mode given by *mode*, where *mode* has the same semantics as the standard C routine `fopen()`. On failure, return *NULL*.

*PyObject** **PyFile_FromFile**(FILE **fp*, char **name*, char **mode*, int (**close*)(FILE*))
Return value: New reference. Create a new `PyFileObject` from the already-open standard C file pointer, *fp*. The function *close* will be called when the file should be closed. Return *NULL* and close the file using *close* on failure. *close* is optional and can be set to *NULL*.

FILE* **PyFile_AsFile**(*PyObject *p*)
Return the file object associated with *p* as a `FILE*`.

If the caller will ever use the returned `FILE*` object while the *GIL* is released it must also call the `PyFile_IncUseCount()` and `PyFile_DecUseCount()` functions described below as appropriate.

void **PyFile_IncUseCount**(*PyFileObject *p*)
Increments the PyFileObject's internal use count to indicate that the underlying `FILE*` is being used. This prevents Python from calling f_close() on it from another thread. Callers of this must call `PyFile_DecUseCount()` when they are finished with the `FILE*`. Otherwise the file object will never be closed by Python.

The *GIL* must be held while calling this function.

The suggested use is to call this after `PyFile_AsFile()` and before you release the GIL:

```
FILE *fp = PyFile_AsFile(p);
PyFile_IncUseCount(p);
/* ... */
Py_BEGIN_ALLOW_THREADS
do_something(fp);
Py_END_ALLOW_THREADS
/* ... */
PyFile_DecUseCount(p);
```

Novo na versão 2.6.

void **PyFile_DecUseCount** (*PyFileObject *p*)

Decrements the PyFileObject's internal unlocked_count member to indicate that the caller is done with its own use of the `FILE*`. This may only be called to undo a prior call to `PyFile_IncUseCount()`.

The *GIL* must be held while calling this function (see the example above).

Novo na versão 2.6.

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

Return value: New reference. Equivalent to `p.readline([n])`, this function reads one line from the object *p*. *p* may be a file object or any object with a `readline()` method. If *n* is `0`, exactly one line is read, regardless of the length of the line. If *n* is greater than `0`, no more than *n* bytes will be read from the file; a partial line can be returned. In both cases, an empty string is returned if the end of the file is reached immediately. If *n* is less than `0`, however, one line is read regardless of length, but `EOFError` is raised if the end of the file is reached immediately.

*PyObject** **PyFile_Name** (*PyObject *p*)

Return value: Borrowed reference. Return the name of the file specified by *p* as a string object.

void **PyFile_SetBufSize** (*PyFileObject *p*, int *n*)

Available on systems with `setvbuf()` only. This should only be called immediately after file object creation.

int **PyFile_SetEncoding** (*PyFileObject *p*, const char **enc*)

Set the file's encoding for Unicode output to *enc*. Return `1` on success and `0` on failure.

Novo na versão 2.3.

int **PyFile_SetEncodingAndErrors** (*PyFileObject *p*, const char **enc*, *errors)

Set the file's encoding for Unicode output to *enc*, and its error mode to *err*. Return `1` on success and `0` on failure.

Novo na versão 2.6.

int **PyFile_SoftSpace** (*PyObject *p*, int *newflag*)

This function exists for internal use by the interpreter. Set the `softspace` attribute of *p* to *newflag* and return the previous value. *p* does not have to be a file object for this function to work properly; any object is supported (thought its only interesting if the `softspace` attribute can be set). This function clears any errors, and will return `0` as the previous value if the attribute either does not exist or if there were errors in retrieving it. There is no way to detect errors from this function, but doing so should not be needed.

int **PyFile_WriteObject** (*PyObject *obj*, *PyObject *p*, int *flags*)

Write object *obj* to file object *p*. The only supported flag for *flags* is `Py_PRINT_RAW`; if given, the `str()` of the object is written instead of the `repr()`. Return `0` on success or `-1` on failure; the appropriate exception will be set.

int **PyFile_WriteString** (const char **s*, *PyObject *p*)

Write string *s* to file object *p*. Return `0` on success or `-1` on failure; the appropriate exception will be set.

7.5.5 Objeto Module

There are only a few functions special to module objects.

PyTypeObject **PyModule_Type**

This instance of *PyTypeObject* represents the Python module type. This is exposed to Python programs as `types.ModuleType`.

int **PyModule_Check** (*PyObject *p*)

Return true if *p* is a module object, or a subtype of a module object.

Alterado na versão 2.2: Allowed subtypes to be accepted.

int **PyModule_CheckExact** (*PyObject *p*)

Return true if *p* is a module object, but not a subtype of *PyModule_Type*.

Novo na versão 2.2.

*PyObject** **PyModule_New** (const char **name*)

Return value: New reference. Return a new module object with the `__name__` attribute set to *name*. Only the module's `__doc__` and `__name__` attributes are filled in; the caller is responsible for providing a `__file__` attribute.

*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. This function never fails. It is recommended extensions use other `PyModule_*()` and `PyObject_*()` functions rather than directly manipulate a module's `__dict__`.

char* **PyModule_GetName** (*PyObject *module*)

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.

char* **PyModule_GetFilename** (*PyObject *module*)

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 string, raise `SystemError` and return *NULL*.

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

Novo na versão 2.0.

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.

Novo na versão 2.0.

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.

Novo na versão 2.0.

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.

Novo na versão 2.6.

int **PyModule_AddStringMacro** (*PyObject *module*, macro)

Add a string constant to *module*.

Novo na versão 2.6.

7.5.6 Objetos Iteradores

O Python fornece dois objetos iteradores de uso geral. O primeiro, um iterador de sequência, trabalha com uma sequência arbitrária que suporta o método `__getitem__()`. O segundo funciona com um objeto que pode ser chamado e um valor de sentinela, chamando o que pode ser chamado para cada item na sequência e finalizando a iteração quando o valor de sentinela é retornado.

PyTypeObject **PySeqIter_Type**

Objeto de tipo para objetos iteradores retornados por `PySeqIter_New()` e a forma de um argumento da função embutida `iter()` para tipos de sequência embutidos.

Novo na versão 2.2.

int **PySeqIter_Check**(op)

Retorna true se o tipo de *op* for `PySeqIter_Type`.

Novo na versão 2.2.

*PyObject** **PySeqIter_New**(*PyObject *seq*)

Return value: New reference. Retorna um iterador que funcione com um objeto de sequência geral, *seq*. A iteração termina quando a sequência levanta `IndexError` para a operação de assinatura.

Novo na versão 2.2.

PyTypeObject **PyCallIter_Type**

Objeto de tipo para objetos iteradores retornados por `PyCallIter_New()` e a forma de dois argumentos da função embutida `iter()`.

Novo na versão 2.2.

int **PyCallIter_Check**(op)

Retorna true se o tipo de *op* é `PyCallIter_Type`.

Novo na versão 2.2.

*PyObject** **PyCallIter_New**(*PyObject *callable*, *PyObject *sentinel*)

Return value: New reference. Retorna um novo iterador. O primeiro parâmetro, *callable*, pode ser qualquer objeto chamável do Python que possa ser chamado sem parâmetros; cada chamada deve retornar o próximo item na iteração. Quando *callable* retorna um valor igual a *sentinel*, a iteração será encerrada.

Novo na versão 2.2.

7.5.7 Objetos Descritores

"Descritores" são objetos que descrevem algum atributo de um objeto. Eles são encontrados no dicionário de objetos de tipo.

PyTypeObject **PyProperty_Type**

O tipo de objeto para os tipos de descritores embutidos.

Novo na versão 2.2.

*PyObject** **PyDescr_NewGetSet**(*PyTypeObject *type*, struct *PyGetSetDef *getset*)

Return value: New reference. Novo na versão 2.2.

*PyObject** **PyDescr_NewMember**(*PyTypeObject *type*, struct *PyMemberDef *meth*)

Return value: New reference. Novo na versão 2.2.

*PyObject** **PyDescr_NewMethod**(*PyTypeObject *type*, struct *PyMethodDef *meth*)
Return value: New reference. Novo na versão 2.2.

*PyObject** **PyDescr_NewWrapper**(*PyTypeObject *type*, struct wrapperbase **wrapper*, void **wrapped*)
Return value: New reference. Novo na versão 2.2.

*PyObject** **PyDescr_NewClassMethod**(*PyTypeObject *type*, *PyMethodDef *method*)
Return value: New reference. Novo na versão 2.3.

int **PyDescr_IsData**(*PyObject *descr*)
Retorna True se os objetos descritores *descr* descrevem um atributo de dados, ou False se os mesmos descrevem um método. *descr* deve ser um objeto descritor; não há verificação de erros.

Novo na versão 2.2.

*PyObject** **PyWrapper_New**(*PyObject* *, *PyObject* *)
Return value: New reference. Novo na versão 2.2.

7.5.8 Objetos Slice

PyTypeObject **PySlice_Type**
The type object for slice objects. This is the same as `slice` and `types.SliceType`.

int **PySlice_Check**(*PyObject *ob*)
Return true if *ob* is a slice object; *ob* must not be *NULL*.

*PyObject** **PySlice_New**(*PyObject *start*, *PyObject *stop*, *PyObject *step*)
Return value: New reference. Return a new slice object with the given values. The *start*, *stop*, and *step* parameters are used as the values of the slice object attributes of the same names. Any of the values may be *NULL*, in which case the `None` will be used for the corresponding attribute. Return *NULL* if the new object could not be allocated.

int **PySlice_GetIndices**(PySliceObject **slice*, Py_ssize_t *length*, Py_ssize_t **start*, Py_ssize_t **stop*, Py_ssize_t **step*)
Retrieve the start, stop and step indices from the slice object *slice*, assuming a sequence of length *length*. Treats indices greater than *length* as errors.

Returns `0` on success and `-1` on error with no exception set (unless one of the indices was not `None` and failed to be converted to an integer, in which case `-1` is returned with an exception set).

You probably do not want to use this function. If you want to use slice objects in versions of Python prior to 2.3, you would probably do well to incorporate the source of `PySlice_GetIndicesEx()`, suitably renamed, in the source of your extension.

Alterado na versão 2.5: This function used an `int` type for *length* and an `int *` type for *start*, *stop*, and *step*. This might require changes in your code for properly supporting 64-bit systems.

int **PySlice_GetIndicesEx**(PySliceObject **slice*, Py_ssize_t *length*, Py_ssize_t **start*, Py_ssize_t **stop*, Py_ssize_t **step*, Py_ssize_t **slicelength*)
Usable replacement for `PySlice_GetIndices()`. Retrieve the start, stop, and step indices from the slice object *slice* assuming a sequence of length *length*, and store the length of the slice in *slicelength*. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Returns `0` on success and `-1` on error with exception set.

Novo na versão 2.3.

Alterado na versão 2.5: This function used an `int` type for *length* and an `int *` type for *start*, *stop*, *step*, and *slicelength*. This might require changes in your code for properly supporting 64-bit systems.

7.5.9 Ellipsis Object

PyObject ***Py_Ellipsis***

The Python `Ellipsis` object. This object has no methods. It needs to be treated just like any other object with respect to reference counts. Like `Py_None` it is a singleton object.

7.5.10 Objetos de Referência Fraca

O Python suporta *referências fracas* como objetos de primeira classe. Existem dois tipos de objetos específicos que implementam diretamente referências fracas. O primeiro é um objeto de referência simples, e o segundo atua como um proxy para o objeto original tanto quanto ele pode.

int **PyWeakref_Check**(ob)

Retorna True se *ob* for um objeto de referência ou proxy.

Novo na versão 2.2.

int **PyWeakref_CheckRef**(ob)

Retornar True se *ob* for um objeto de referência.

Novo na versão 2.2.

int **PyWeakref_CheckProxy**(ob)

Retornar True se *ob* for um objeto proxy.

Novo na versão 2.2.

*PyObject** **PyWeakref_NewRef**(*PyObject *ob*, *PyObject *callback*)

Return value: New reference. Return a weak reference object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object; an existing reference object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be `None` or *NULL*. If *ob* is not a weakly-referencable object, or if *callback* is not callable, `None`, or *NULL*, this will return *NULL* and raise `TypeError`.

Novo na versão 2.2.

*PyObject** **PyWeakref_NewProxy**(*PyObject *ob*, *PyObject *callback*)

Return value: New reference. Return a weak reference proxy object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object; an existing proxy object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be `None` or *NULL*. If *ob* is not a weakly-referencable object, or if *callback* is not callable, `None`, or *NULL*, this will return *NULL* and raise `TypeError`.

Novo na versão 2.2.

*PyObject** **PyWeakref_GetObject**(*PyObject *ref*)

Return value: Borrowed reference. Retorna o objeto referenciado de uma referência fraca, *ref*. Se o referente não estiver mais em tempo real, retorna `Py_None`.

Novo na versão 2.2.

> **Aviso:** Esta função retorna **referência emprestada** ao objeto referenciado. Isso significa que você deve sempre invocar `Py_INCREF()` no objeto, exceto se você souber que não pode ser destruído enquanto você ainda está usando.

*PyObject** **PyWeakref_GET_OBJECT**(*PyObject *ref*)
Return value: Borrowed reference. Semelhante a `PyWeakref_GetObject()`, mas implementado como uma macro que não verifica erros.

Novo na versão 2.2.

7.5.11 Capsules

Refer to using-capsules for more information on using these objects.

Novo na versão 2.7.

PyCapsule
This subtype of `PyObject` represents an opaque value, useful for C extension modules who need to pass an opaque value (as a `void*` pointer) through Python code to other C code. It is often used to make a C function pointer defined in one module available to other modules, so the regular import mechanism can be used to access C APIs defined in dynamically loaded modules.

PyCapsule_Destructor
The type of a destructor callback for a capsule. Defined as:

```
typedef void (*PyCapsule_Destructor)(PyObject *);
```

See `PyCapsule_New()` for the semantics of PyCapsule_Destructor callbacks.

int **PyCapsule_CheckExact**(*PyObject *p*)
Retorna true se seu argumento é um `PyCapsule`.

*PyObject** **PyCapsule_New**(void **pointer*, const char **name*, *PyCapsule_Destructor destructor*)
Return value: New reference. Create a `PyCapsule` encapsulating the *pointer*. The *pointer* argument may not be *NULL*.

On failure, set an exception and return *NULL*.

The *name* string may either be *NULL* or a pointer to a valid C string. If non-*NULL*, this string must outlive the capsule. (Though it is permitted to free it inside the *destructor*.)

If the *destructor* argument is not *NULL*, it will be called with the capsule as its argument when it is destroyed.

If this capsule will be stored as an attribute of a module, the *name* should be specified as `modulename.attributename`. This will enable other modules to import the capsule using `PyCapsule_Import()`.

void* **PyCapsule_GetPointer**(*PyObject *capsule*, const char **name*)
Retrieve the *pointer* stored in the capsule. On failure, set an exception and return *NULL*.

The *name* parameter must compare exactly to the name stored in the capsule. If the name stored in the capsule is *NULL*, the *name* passed in must also be *NULL*. Python uses the C function `strcmp()` to compare capsule names.

PyCapsule_Destructor **PyCapsule_GetDestructor**(*PyObject *capsule*)
Return the current destructor stored in the capsule. On failure, set an exception and return *NULL*.

It is legal for a capsule to have a *NULL* destructor. This makes a *NULL* return code somewhat ambiguous; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

void* **PyCapsule_GetContext**(*PyObject *capsule*)
Return the current context stored in the capsule. On failure, set an exception and return *NULL*.

It is legal for a capsule to have a *NULL* context. This makes a *NULL* return code somewhat ambiguous; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

const char* **PyCapsule_GetName** (*PyObject *capsule*)
Return the current name stored in the capsule. On failure, set an exception and return *NULL*.

It is legal for a capsule to have a *NULL* name. This makes a *NULL* return code somewhat ambiguous; use `PyCapsule_IsValid()` or `PyErr_Occurred()` to disambiguate.

void* **PyCapsule_Import** (const char **name*, int *no_block*)
Import a pointer to a C object from a capsule attribute in a module. The *name* parameter should specify the full name to the attribute, as in `module.attribute`. The *name* stored in the capsule must match this string exactly. If *no_block* is true, import the module without blocking (using `PyImport_ImportModuleNoBlock()`). If *no_block* is false, import the module conventionally (using `PyImport_ImportModule()`).

Return the capsule's internal *pointer* on success. On failure, set an exception and return *NULL*.

int **PyCapsule_IsValid** (*PyObject *capsule*, const char **name*)
Determines whether or not *capsule* is a valid capsule. A valid capsule is non-*NULL*, passes `PyCapsule_CheckExact()`, has a non-*NULL* pointer stored in it, and its internal name matches the *name* parameter. (See `PyCapsule_GetPointer()` for information on how capsule names are compared.)

In other words, if `PyCapsule_IsValid()` returns a true value, calls to any of the accessors (any function starting with `PyCapsule_Get()`) are guaranteed to succeed.

Return a nonzero value if the object is valid and matches the name passed in. Return `0` otherwise. This function will not fail.

int **PyCapsule_SetContext** (*PyObject *capsule*, void **context*)
Set the context pointer inside *capsule* to *context*.

Retorne `0` em caso de sucesso. Retorne diferente de zero e defina uma exceção em caso de falha.

int **PyCapsule_SetDestructor** (*PyObject *capsule*, *PyCapsule_Destructor destructor*)
Set the destructor inside *capsule* to *destructor*.

Retorne `0` em caso de sucesso. Retorne diferente de zero e defina uma exceção em caso de falha.

int **PyCapsule_SetName** (*PyObject *capsule*, const char **name*)
Set the name inside *capsule* to *name*. If non-*NULL*, the name must outlive the capsule. If the previous *name* stored in the capsule was not *NULL*, no attempt is made to free it.

Retorne `0` em caso de sucesso. Retorne diferente de zero e defina uma exceção em caso de falha.

int **PyCapsule_SetPointer** (*PyObject *capsule*, void **pointer*)
Set the void pointer inside *capsule* to *pointer*. The pointer may not be *NULL*.

Retorne `0` em caso de sucesso. Retorne diferente de zero e defina uma exceção em caso de falha.

7.5.12 CObjects

Aviso: The CObject API is deprecated as of Python 2.7. Please switch to the new *Capsules* API.

PyCObject
This subtype of `PyObject` represents an opaque value, useful for C extension modules who need to pass an opaque value (as a `void*` pointer) through Python code to other C code. It is often used to make a C function pointer defined in one module available to other modules, so the regular import mechanism can be used to access C APIs defined in dynamically loaded modules.

int **PyCObject_Check** (*PyObject *p*)
Return true if its argument is a `PyCObject`.

*PyObject** **PyCObject_FromVoidPtr**(void* *cobj*, void (**destr*)(void *))
: *Return value: New reference.* Create a `PyCObject` from the `void *` *cobj*. The *destr* function will be called when the object is reclaimed, unless it is *NULL*.

*PyObject** **PyCObject_FromVoidPtrAndDesc**(void* *cobj*, void* *desc*, void (**destr*)(void *, void *))
: *Return value: New reference.* Create a `PyCObject` from the `void *` *cobj*. The *destr* function will be called when the object is reclaimed. The *desc* argument can be used to pass extra callback data for the destructor function.

void* **PyCObject_AsVoidPtr**(*PyObject** *self*)
: Return the object `void *` that the `PyCObject` *self* was created with.

void* **PyCObject_GetDesc**(*PyObject** *self*)
: Return the description `void *` that the `PyCObject` *self* was created with.

int **PyCObject_SetVoidPtr**(*PyObject** *self*, void* *cobj*)
: Set the void pointer inside *self* to *cobj*. The `PyCObject` must not have an associated destructor. Return true on success, false on failure.

7.5.13 Objeto Célula

Objetos "Cell" são usados para implementar variáveis referenciadas por múltiplos escopos. Para cada variável, um objeto de célula é criado para armazenar o valor; as variáveis locais de cada quadro de pilha que referencia o valor contém uma referência para as células de escopos externos que também usam essa variável. Quando o valor é acessado, o valor contido na célula é usado em vez do próprio objeto da célula. Essa des-referência do objeto da célula requer suporte do código de bytes gerado; estes não são automaticamente desprezados quando acessados. Objetos de células provavelmente não serão úteis em outro lugar.

PyCellObject
: A estrutura C usada para objetos de célula.

PyTypeObject **PyCell_Type**
: O objeto de tipo correspondente aos objetos de célula.

int **PyCell_Check**(ob)
: Return true if *ob* is a cell object; *ob* must not be *NULL*.

*PyObject** **PyCell_New**(*PyObject* **ob*)
: *Return value: New reference.* Create and return a new cell object containing the value *ob*. The parameter may be *NULL*.

*PyObject** **PyCell_Get**(*PyObject* **cell*)
: *Return value: New reference.* Retorna o conteúdo da célula *cell*.

*PyObject** **PyCell_GET**(*PyObject* **cell*)
: *Return value: Borrowed reference.* Return the contents of the cell *cell*, but without checking that *cell* is non-*NULL* and a cell object.

int **PyCell_Set**(*PyObject* **cell*, *PyObject* **value*)
: Set the contents of the cell object *cell* to *value*. This releases the reference to any current content of the cell. *value* may be *NULL*. *cell* must be non-*NULL*; if it is not a cell object, `-1` will be returned. On success, `0` will be returned.

void **PyCell_SET**(*PyObject* **cell*, *PyObject* **value*)
: Sets the value of the cell object *cell* to *value*. No reference counts are adjusted, and no checks are made for safety; *cell* must be non-*NULL* and must be a cell object.

7.5.14 Objeto Geradores

Generator objects are what Python uses to implement generator iterators. They are normally created by iterating over a function that yields values, rather than explicitly calling `PyGen_New()`.

PyGenObject
A estrutura C usada para objetos geradores.

PyTypeObject **PyGen_Type**
O objeto de tipo correspondendo a objetos geradores.

int **PyGen_Check**(ob)
Return true if *ob* is a generator object; *ob* must not be *NULL*.

int **PyGen_CheckExact**(ob)
Return true if *ob*'s type is *PyGen_Type* is a generator object; *ob* must not be *NULL*.

*PyObject** **PyGen_New**(PyFrameObject **frame*)
Return value: New reference. Create and return a new generator object based on the *frame* object. A reference to *frame* is stolen by this function. The parameter must not be *NULL*.

7.5.15 Objetos DateTime

Vários objetos de data e hora são fornecidos pelo módulo `datetime`. Antes de usar qualquer uma dessas funções, o arquivo de cabeçalho `datetime.h` deve ser incluído na sua fonte (observe que isso não é incluído por `Python.h`) e a macro `PyDateTime_IMPORT` deve ser chamada, geralmente como parte da função de inicialização do módulo. A macro coloca um ponteiro para uma estrutura C em uma variável estática, `PyDateTimeAPI`, usada pelas macros a seguir.

Macros de verificação de tipo:

int **PyDate_Check**(*PyObject* **ob*)
Return true if *ob* is of type `PyDateTime_DateType` or a subtype of `PyDateTime_DateType`. *ob* must not be *NULL*.

Novo na versão 2.4.

int **PyDate_CheckExact**(*PyObject* **ob*)
Return true if *ob* is of type `PyDateTime_DateType`. *ob* must not be *NULL*.

Novo na versão 2.4.

int **PyDateTime_Check**(*PyObject* **ob*)
Return true if *ob* is of type `PyDateTime_DateTimeType` or a subtype of `PyDateTime_DateTimeType`. *ob* must not be *NULL*.

Novo na versão 2.4.

int **PyDateTime_CheckExact**(*PyObject* **ob*)
Return true if *ob* is of type `PyDateTime_DateTimeType`. *ob* must not be *NULL*.

Novo na versão 2.4.

int **PyTime_Check**(*PyObject* **ob*)
Return true if *ob* is of type `PyDateTime_TimeType` or a subtype of `PyDateTime_TimeType`. *ob* must not be *NULL*.

Novo na versão 2.4.

int **PyTime_CheckExact**(*PyObject* **ob*)
Return true if *ob* is of type `PyDateTime_TimeType`. *ob* must not be *NULL*.

Novo na versão 2.4.

int **PyDelta_Check** (*PyObject* *ob)
Return true if *ob* is of type `PyDateTime_DeltaType` or a subtype of `PyDateTime_DeltaType`. *ob* must not be *NULL*.

Novo na versão 2.4.

int **PyDelta_CheckExact** (*PyObject* *ob)
Return true if *ob* is of type `PyDateTime_DeltaType`. *ob* must not be *NULL*.

Novo na versão 2.4.

int **PyTZInfo_Check** (*PyObject* *ob)
Return true if *ob* is of type `PyDateTime_TZInfoType` or a subtype of `PyDateTime_TZInfoType`. *ob* must not be *NULL*.

Novo na versão 2.4.

int **PyTZInfo_CheckExact** (*PyObject* *ob)
Return true if *ob* is of type `PyDateTime_TZInfoType`. *ob* must not be *NULL*.

Novo na versão 2.4.

Macros para criar objetos:

*PyObject** **PyDate_FromDate** (int *year*, int *month*, int *day*)
Return value: New reference. Return a `datetime.date` object with the specified year, month and day.

Novo na versão 2.4.

*PyObject** **PyDateTime_FromDateAndTime** (int *year*, int *month*, int *day*, int *hour*, int *minute*, int *second*, int *usecond*)
Return value: New reference. Return a `datetime.datetime` object with the specified year, month, day, hour, minute, second and microsecond.

Novo na versão 2.4.

*PyObject** **PyTime_FromTime** (int *hour*, int *minute*, int *second*, int *usecond*)
Return value: New reference. Return a `datetime.time` object with the specified hour, minute, second and microsecond.

Novo na versão 2.4.

*PyObject** **PyDelta_FromDSU** (int *days*, int *seconds*, int *useconds*)
Return value: New reference. Return a `datetime.timedelta` object representing the given number of days, seconds and microseconds. Normalization is performed so that the resulting number of microseconds and seconds lie in the ranges documented for `datetime.timedelta` objects.

Novo na versão 2.4.

Macros to extract fields from date objects. The argument must be an instance of `PyDateTime_Date`, including subclasses (such as `PyDateTime_DateTime`). The argument must not be *NULL*, and the type is not checked:

int **PyDateTime_GET_YEAR** (PyDateTime_Date *o)
Retorna o ano, como um inteiro positivo.

Novo na versão 2.4.

int **PyDateTime_GET_MONTH** (PyDateTime_Date *o)
Retorna o mês, como um inteiro de 1 a 12.

Novo na versão 2.4.

int **PyDateTime_GET_DAY** (PyDateTime_Date *o*)
Retorna o dia, como um inteiro de 1 a 31.

Novo na versão 2.4.

Macros to extract fields from datetime objects. The argument must be an instance of `PyDateTime_DateTime`, including subclasses. The argument must not be *NULL*, and the type is not checked:

int **PyDateTime_DATE_GET_HOUR** (PyDateTime_DateTime *o*)
Retorna a hora, como um inteiro de 0 a 23.

Novo na versão 2.4.

int **PyDateTime_DATE_GET_MINUTE** (PyDateTime_DateTime *o*)
Retorna o minuto, como um inteiro de 0 a 59.

Novo na versão 2.4.

int **PyDateTime_DATE_GET_SECOND** (PyDateTime_DateTime *o*)
Retorna o segundo, como um inteiro de 0 a 59.

Novo na versão 2.4.

int **PyDateTime_DATE_GET_MICROSECOND** (PyDateTime_DateTime *o*)
Retorna o microssegundo, como um inteiro de 0 a 999999.

Novo na versão 2.4.

Macros to extract fields from time objects. The argument must be an instance of `PyDateTime_Time`, including subclasses. The argument must not be *NULL*, and the type is not checked:

int **PyDateTime_TIME_GET_HOUR** (PyDateTime_Time *o*)
Retorna a hora, como um inteiro de 0 a 23.

Novo na versão 2.4.

int **PyDateTime_TIME_GET_MINUTE** (PyDateTime_Time *o*)
Retorna o minuto, como um inteiro de 0 a 59.

Novo na versão 2.4.

int **PyDateTime_TIME_GET_SECOND** (PyDateTime_Time *o*)
Retorna o segundo, como um inteiro de 0 a 59.

Novo na versão 2.4.

int **PyDateTime_TIME_GET_MICROSECOND** (PyDateTime_Time *o*)
Retorna o microssegundo, como um inteiro de 0 a 999999.

Novo na versão 2.4.

Macros para a conveniência de módulos implementando a API de DB:

*PyObject** **PyDateTime_FromTimestamp** (*PyObject *args*)
Return value: New reference. Create and return a new `datetime.datetime` object given an argument tuple suitable for passing to `datetime.datetime.fromtimestamp()`.

Novo na versão 2.4.

*PyObject** **PyDate_FromTimestamp** (*PyObject *args*)
Return value: New reference. Create and return a new `datetime.date` object given an argument tuple suitable for passing to `datetime.date.fromtimestamp()`.

Novo na versão 2.4.

7.5.16 Objeto Set

Novo na versão 2.5.

This section details the public API for `set` and `frozenset` objects. Any functionality not listed below is best accessed using the 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 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**
Essa é uma instância de `PyTypeObject` representando o tipo Python `set`

PyTypeObject **PyFrozenSet_Type**
This is an instance of `PyTypeObject` representing the Python `frozenset` type.

The following type check macros work on pointers to any Python object. Likewise, the constructor functions work with any iterable Python object.

int **PySet_Check**(*PyObject* **p*)
Return true if *p* is a `set` object or an instance of a subtype.

Novo na versão 2.6.

int **PyFrozenSet_Check**(*PyObject* **p*)
Return true if *p* is a `frozenset` object or an instance of a subtype.

Novo na versão 2.6.

int **PyAnySet_Check**(*PyObject* **p*)
Return true if *p* is a `set` object, a `frozenset` object, or an instance of a subtype.

int **PyAnySet_CheckExact**(*PyObject* **p*)
Return true if *p* is a `set` object or a `frozenset` object but not an instance of a subtype.

int **PyFrozenSet_CheckExact**(*PyObject* **p*)
Return true if *p* is a `frozenset` object but not an instance of a subtype.

*PyObject** **PySet_New**(*PyObject* **iterable*)
Return value: New reference. Return a new `set` containing objects returned by the *iterable*. The *iterable* may be *NULL* to create a new empty set. Return the new set on success or *NULL* on failure. Raise `TypeError` if *iterable* is not actually iterable. The constructor is also useful for copying a set (`c=set(s)`).

*PyObject** **PyFrozenSet_New**(*PyObject* **iterable*)
Return value: New reference. Return a new `frozenset` containing objects returned by the *iterable*. The *iterable* may be *NULL* to create a new empty frozenset. Return the new set on success or *NULL* on failure. Raise `TypeError` if *iterable* is not actually iterable.

Alterado na versão 2.6: Now guaranteed to return a brand-new `frozenset`. Formerly, frozensets of zero-length were a singleton. This got in the way of building-up new frozensets with `PySet_Add()`.

The following functions and macros are available for instances of `set` or `frozenset` or instances of their subtypes.

Py_ssize_t **PySet_Size**(*PyObject *anyset*)
Return the length of a `set` or `frozenset` object. Equivalent to `len(anyset)`. Raises a `PyExc_SystemError` if *anyset* is not a `set`, `frozenset`, or an instance of a subtype.

Alterado na versão 2.5: This function returned an `int`. This might require changes in your code for properly supporting 64-bit systems.

Py_ssize_t **PySet_GET_SIZE**(*PyObject *anyset*)
Macro form of `PySet_Size()` without error checking.

int **PySet_Contains**(*PyObject *anyset*, *PyObject *key*)
Return `1` if found, `0` if not found, and `-1` if an error is encountered. Unlike the Python `__contains__()` method, this function does not automatically convert unhashable sets into temporary frozensets. Raise a `TypeError` if the *key* is unhashable. Raise `PyExc_SystemError` if *anyset* is not a `set`, `frozenset`, or an instance of a subtype.

int **PySet_Add**(*PyObject *set*, *PyObject *key*)
Add *key* to a `set` instance. Does not apply to `frozenset` instances. 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.

Alterado na versão 2.6: Now works with instances of `frozenset` or its subtypes. Like `PyTuple_SetItem()` in that it can be used to fill-in the values of brand new frozensets before they are exposed to other code.

The following functions are available for instances of `set` or its subtypes but not for instances of `frozenset` or its subtypes.

int **PySet_Discard**(*PyObject *set*, *PyObject *key*)
Return `1` if found and removed, `0` if not found (no action taken), and `-1` if an error is encountered. Does not raise `KeyError` for missing keys. Raise a `TypeError` if the *key* is unhashable. Unlike the Python `discard()` method, this function does not automatically convert unhashable sets into temporary frozensets. Raise `PyExc_SystemError` if *set* is not an instance of `set` or its subtype.

*PyObject** **PySet_Pop**(*PyObject *set*)
Return value: New reference. Return a new reference to an arbitrary object in the *set*, and removes the object from the *set*. Return *NULL* on failure. Raise `KeyError` if the set is empty. Raise a `SystemError` if *set* is not an instance of `set` or its subtype.

int **PySet_Clear**(*PyObject *set*)
Limpa todos os elementos de um set existente

7.5.17 Objetos Código

Os objetos código são um detalhe de baixo nível da implementação do CPython. Cada um representa um pedaço de código executável que ainda não foi vinculado a uma função.

PyCodeObject
A estrutura C dos objetos usados para descrever objetos de código. Os campos deste tipo estão sujeitos a alterações a qualquer momento.

PyTypeObject **PyCode_Type**
Esta é uma instância de `PyTypeObject` representando o tipo Python `code`.

int **PyCode_Check**(*PyObject *co*)
Retorna *True* se *co* for um objeto `code`.

int **PyCode_GetNumFree**(*PyObject *co*)
Retorna o número de variáveis livres em *co*.

PyCodeObject ***PyCode_New**(int *argcount*, 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*)

Retorna um novo objeto de código. Se você precisa de um objeto código fictício para criar um quadro, use `PyCode_NewEmpty()` no caso. Chamar `PyCode_New()` diretamente pode vinculá-lo a uma versão precisa do Python, uma vez que a definição do bytecode muda frequentemente.

int **PyCode_NewEmpty**(const char **filename*, const char **funcname*, int *firstlineno*)

Return a new empty code object with the specified filename, function name, and first line number. It is illegal to `exec` or `eval()` the resulting code object.

CAPÍTULO 8

Inicialização, Finalização e Threads

8.1 Inicializando e encerrando o interpretador

void **Py_Initialize**()

Initialize the Python interpreter. In an application embedding Python, this should be called before using any other Python/C API functions; with the exception of *Py_SetProgramName()*, *Py_SetPythonHome()*, *PyEval_InitThreads()*, *PyEval_ReleaseLock()*, and *PyEval_AcquireLock()*. This initializes the table of loaded modules (`sys.modules`), and creates the fundamental modules `__builtin__`, `__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_Finalize()* first). There is no return value; it is a fatal error if the initialization fails.

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.

Novo na versão 2.4.

int **Py_IsInitialized**()

Return true (nonzero) when the Python interpreter has been initialized, false (zero) if not. After *Py_Finalize()* is called, this returns false until *Py_Initialize()* is called again.

void **Py_Finalize**()

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). There is no return value; errors during finalization are ignored.

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_Finalize()* more than once.

8.2 Process-wide parameters

void **Py_SetProgramName**(char **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. 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 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.

char* **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.

char* **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.

char* **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.

char* **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`.

char* **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 Mac OS X, `';'` 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.

const char* **Py_GetVersion**()
Return the version of this Python interpreter. This is a string that looks something like

```
"1.5 (#67, Dec 31 1997, 22:34:28) [GCC 2.7.2.2]"
```

The first word (up to the first space character) is the current Python version; the first three 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 Mac OS X, 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*, char ***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 (`"."`).

Nota: 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 2.6.6, 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");
```

Novo na versão 2.6.6.

void **PySys_SetArgv**(int *argc*, char ***argv*)

This function works like `PySys_SetArgvEx()` with *updatepath* set to `1`.

void **Py_SetPythonHome**(char **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.

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.

8.3 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.setcheckinterval()`). 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()`.

8.3.1 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. These two macros are still available when Python is compiled without thread support (they simply have an empty expansion).

When thread support is enabled, 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.

Nota: 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.

8.3.2 Non-Python created threads

When threads are created using the dedicated Python APIs (such as the `threading` module), a thread state is automatically associated to them and the code showed above is therefore correct. However, when threads are created from C (for example by a third-party library with its own thread management), they don't hold the GIL, nor is there a thread state structure for them.

If you need to call Python code from these threads (often this will be part of a callback API provided by the aforementioned third-party library), you must first register these threads with the interpreter by creating a thread state data structure, then acquiring the GIL, and finally storing their thread state pointer, before you can start using the Python/C API. When you are done, you should reset the thread state pointer, release the GIL, and finally free the thread state data structure.

The `PyGILState_Ensure()` and `PyGILState_Release()` functions do all of the above automatically. The typical idiom for calling into Python from a C thread is:

```
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Release the thread. No Python API allowed beyond this point. */
PyGILState_Release(gstate);
```

Note that the `PyGILState_*()` functions assume there is only one global interpreter (created automatically by `Py_Initialize()`). Python supports the creation of additional interpreters (using `Py_NewInterpreter()`), but mixing multiple interpreters and the `PyGILState_*()` API is unsupported.

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. That also 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()` tries to reset the necessary locks, but is not always able to.

8.3.3 High-level 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 `PyInterpreterState *interp`, which points to this thread's interpreter state.

void **PyEval_InitThreads**()

Initialize and acquire the global interpreter lock. It should be called in the main thread before creating a second thread or engaging in any other thread operations such as `PyEval_ReleaseLock()` or `PyEval_ReleaseThread(tstate)`. It is not needed before calling `PyEval_SaveThread()` or `PyEval_RestoreThread()`.

This is a no-op when called for a second time. It is safe to call this function before calling `Py_Initialize()`.

Nota: When only the main thread exists, no GIL operations are needed. This is a common situation (most Python programs do not use threads), and the lock operations slow the interpreter down a bit. Therefore, the lock is not created initially. This situation is equivalent to having acquired the lock: when there is only a single thread, all object accesses are safe. Therefore, when this function initializes the global interpreter lock, it also acquires it. Before the Python `_thread` module creates a new thread, knowing that either it has the lock or the lock hasn't been created yet, it calls *PyEval_InitThreads()*. When this call returns, it is guaranteed that the lock has been created and that the calling thread has acquired it.

It is **not** safe to call this function when it is unknown which thread (if any) currently has the global interpreter lock.

This function is not available when thread support is disabled at compile time.

---

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. This function is not available when thread support is disabled at compile time.

Novo na versão 2.4.

*PyThreadState** **PyEval_SaveThread**()

Release the global interpreter lock (if it has been created and thread support is enabled) 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. (This function is available even when thread support is disabled at compile time.)

void **PyEval_RestoreThread**(*PyThreadState *tstate*)

Acquire the global interpreter lock (if it has been created and thread support is enabled) 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. (This function is available even when thread support is disabled at compile time.)

*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.

void **PyEval_ReInitThreads**()

This function is called from *PyOS_AfterFork()* to ensure that newly created child processes don't hold locks referring to threads which are not running in the child process.

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.

Novo na versão 2.3.

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.

Novo na versão 2.3.

*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.

Novo na versão 2.3.

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. It is a no-op when thread support is disabled at compile time.

Py_END_ALLOW_THREADS

This macro expands to `PyEval_RestoreThread(_save); }`. Note that it contains a closing brace; it must be matched with an earlier `Py_BEGIN_ALLOW_THREADS` macro. See above for further discussion of this macro. It is a no-op when thread support is disabled at compile time.

Py_BLOCK_THREADS

This macro expands to `PyEval_RestoreThread(_save);`: it is equivalent to `Py_END_ALLOW_THREADS` without the closing brace. It is a no-op when thread support is disabled at compile time.

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. It is a no-op when thread support is disabled at compile time.

8.3.4 Low-level API

All of the following functions are only available when thread support is enabled at compile time, and must be called only when the global interpreter lock has been created.

*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.

void **PyInterpreterState_Clear**(*PyInterpreterState *interp*)

Reset all information in an interpreter state object. The global interpreter lock must be held.

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.

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()`.

*PyObject** **PyThreadState_GetDict** ()
Return value: Borrowed reference. Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no current thread state is available. If this function returns *NULL*, no exception has been raised and the caller should assume no current thread state is available.

Alterado na versão 2.3: Previously this could only be called when a current thread is active, and *NULL* meant that an exception was raised.

int **PyThreadState_SetAsyncExc** (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.

Novo na versão 2.3.

void **PyEval_AcquireThread** (*PyThreadState *tstate*)
Acquire the global interpreter lock and set the current thread state to *tstate*, which should not be *NULL*. The lock must have been created earlier. If this thread already has the lock, deadlock ensues.

`PyEval_RestoreThread()` is a higher-level function which is always available (even when thread support isn't enabled or 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 thread support isn't enabled or 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.

Aviso: This function does not change the current thread state. Please use `PyEval_RestoreThread()` or `PyEval_AcquireThread()` instead.

void **PyEval_ReleaseLock** ()
Release the global interpreter lock. The lock must have been created earlier.

Aviso: This function does not change the current thread state. Please use `PyEval_SaveThread()` or `PyEval_ReleaseThread()` instead.

8.4 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. 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: 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. Note that this is different from what happens when an extension is imported after the interpreter has been completely re-initialized by calling *Py_Finalize()* and *Py_Initialize()*; in that case, the extension's `initmodule` function *is* called again.

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_Finalize()* will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

8.4.1 Bugs and caveats

Because sub-interpreters (and the main interpreter) are part of the same process, the insulation between them isn't perfect — for example, using low-level file operations like `os.close()` they can (accidentally or maliciously) affect each other's open files. Because of the way extensions are shared between (sub-)interpreters, some extensions may not work properly; this is especially likely when the extension makes use of (static) global variables, or when the extension manipulates its module's dictionary after its initialization. It is possible to insert objects created in one sub-interpreter into a namespace of another sub-interpreter; this should be done with great care 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.

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.

8.5 Asynchronous Notifications

A mechanism is provided to make asynchronous notifications to the main interpreter thread. These notifications take the form of a function pointer and a void pointer argument.

int **Py_AddPendingCall**(int (**func*)(void *), void **arg*)

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.

> **Aviso:** 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*.

Novo na versão 2.7.

8.6 Profiling and Tracing

The Python interpreter provides some low-level support for attaching profiling and execution tracing facilities. These are used for profiling, debugging, and coverage analysis tools.

Starting with Python 2.2, the implementation of this facility was substantially revised, and an interface from C was added. 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`, or `PyTrace_C_RETURN`, and *arg* depends on the value of *what*:

Value of *what*	Meaning of *arg*
`PyTrace_CALL`	Always `Py_None`.
`PyTrace_EXCEPTION`	Exception information as returned by `sys.exc_info()`.
`PyTrace_LINE`	Always `Py_None`.
`PyTrace_RETURN`	Value being returned to the caller, or *NULL* if caused by an exception.
`PyTrace_C_CALL`	Function object being called.
`PyTrace_C_EXCEPTION`	Function object being called.
`PyTrace_C_RETURN`	Function object being called.

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 trace function (but not a profiling function) when a line-number event is being reported.

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.

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` and `PyTrace_EXCEPTION`.

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 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.

*PyObject** **PyEval_GetCallStats**(*PyObject *self*)
Return a tuple of function call counts. There are constants defined for the positions within the tuple:

Name	Value
PCALL_ALL	0
PCALL_FUNCTION	1
PCALL_FAST_FUNCTION	2
PCALL_FASTER_FUNCTION	3
PCALL_METHOD	4
PCALL_BOUND_METHOD	5
PCALL_CFUNCTION	6
PCALL_TYPE	7
PCALL_GENERATOR	8
PCALL_OTHER	9
PCALL_POP	10

`PCALL_FAST_FUNCTION` means no argument tuple needs to be created. `PCALL_FASTER_FUNCTION` means that the fast-path frame setup code is used.

If there is a method call where the call can be optimized by changing the argument tuple and calling the function directly, it gets recorded twice.

This function is only present if Python is compiled with `CALL_PROFILE` defined.

8.7 Advanced Debugger Support

These functions are only intended to be used by advanced debugging tools.

*PyInterpreterState** **PyInterpreterState_Head**()

Return the interpreter state object at the head of the list of all such objects.

Novo na versão 2.2.

*PyInterpreterState** **PyInterpreterState_Next**(*PyInterpreterState *interp*)

Return the next interpreter state object after *interp* from the list of all such objects.

Novo na versão 2.2.

PyThreadState * **PyInterpreterState_ThreadHead**(*PyInterpreterState *interp*)

Return the pointer to the first `PyThreadState` object in the list of threads associated with the interpreter *interp*.

Novo na versão 2.2.

*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.

Novo na versão 2.2.

CAPÍTULO 9

Gerenciamento de Memória

9.1 Visão Geral

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the *Python memory manager*. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

To avoid memory corruption, extension writers should never try to operate on Python objects with the functions exported by the C library: `malloc()`, `calloc()`, `realloc()` and `free()`. This will result in mixed calls between the C allocator and the Python memory manager with fatal consequences, because they implement different algorithms and operate on different heaps. However, one may safely allocate and release memory blocks with the C library allocator for individual purposes, as shown in the following example:

```
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyString_FromString(buf);
```

(continua na próxima página)

(continuação da página anterior)

```
free(buf); /* malloc'ed */
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the string object returned as a result.

In most situations, however, it is recommended to allocate memory from the Python heap specifically because the latter is under control of the Python memory manager. For example, this is required when the interpreter is extended with new object types written in C. Another reason for using the Python heap is the desire to *inform* the Python memory manager about the memory needs of the extension module. Even when the requested memory is used exclusively for internal, highly-specific purposes, delegating all memory requests to the Python memory manager causes the interpreter to have a more accurate image of its memory footprint as a whole. Consequently, under certain circumstances, the Python memory manager may or may not trigger appropriate actions, like garbage collection, memory compaction or other preventive procedures. Note that by using the C library allocator as shown in the previous example, the allocated memory for the I/O buffer escapes completely the Python memory manager.

9.2 Interface da Memória

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap:

void* **PyMem_Malloc**(size_t *n*)
: Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or *NULL* if the request fails. Requesting zero bytes returns a distinct non-*NULL* pointer if possible, as if `PyMem_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void* **PyMem_Realloc**(void **p*, size_t *n*)
: Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes. If *p* is *NULL*, the call is equivalent to `PyMem_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-*NULL*. Unless *p* is *NULL*, it must have been returned by a previous call to `PyMem_Malloc()` or `PyMem_Realloc()`. If the request fails, `PyMem_Realloc()` returns *NULL* and *p* remains a valid pointer to the previous memory area.

void **PyMem_Free**(void **p*)
: Frees the memory block pointed to by *p*, which must have been returned by a previous call to `PyMem_Malloc()` or `PyMem_Realloc()`. Otherwise, or if `PyMem_Free(p)` has been called before, undefined behavior occurs. If *p* is *NULL*, no operation is performed.

The following type-oriented macros are provided for convenience. Note that *TYPE* refers to any C type.

TYPE* **PyMem_New**(TYPE, size_t *n*)
: Same as `PyMem_Malloc()`, but allocates `(n * sizeof(TYPE))` bytes of memory. Returns a pointer cast to `TYPE*`. The memory will not have been initialized in any way.

TYPE* **PyMem_Resize**(void **p*, TYPE, size_t *n*)
: Same as `PyMem_Realloc()`, but the memory block is resized to `(n * sizeof(TYPE))` bytes. Returns a pointer cast to `TYPE*`. On return, *p* will be a pointer to the new memory area, or *NULL* in the event of failure. This is a C preprocessor macro; p is always reassigned. Save the original value of p to avoid losing memory when handling errors.

void **PyMem_Del**(void **p*)
: Same as `PyMem_Free()`.

In addition, the following macro sets are provided for calling the Python memory allocator directly, without involving the C API functions listed above. However, note that their use does not preserve binary compatibility across Python versions and is therefore deprecated in extension modules.

PyMem_MALLOC(), PyMem_REALLOC(), PyMem_FREE().

PyMem_NEW(), PyMem_RESIZE(), PyMem_DEL().

9.3 Object allocators

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

By default, these functions use *pymalloc memory allocator*.

> **Aviso:** The *GIL* must be held when using these functions.

void* **PyObject_Malloc**(size_t *n*)
: Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or *NULL* if the request fails.

  Requesting zero bytes returns a distinct non-*NULL* pointer if possible, as if `PyObject_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void* **PyObject_Realloc**(void **p*, size_t *n*)
: Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

  If *p* is *NULL*, the call is equivalent to `PyObject_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-*NULL*.

  Unless *p* is *NULL*, it must have been returned by a previous call to *`PyObject_Malloc()`*, *`PyObject_Realloc()`* or `PyObject_Calloc()`.

  If the request fails, *`PyObject_Realloc()`* returns *NULL* and *p* remains a valid pointer to the previous memory area.

void **PyObject_Free**(void **p*)
: Frees the memory block pointed to by *p*, which must have been returned by a previous call to *`PyObject_Malloc()`*, *`PyObject_Realloc()`* or `PyObject_Calloc()`. Otherwise, or if `PyObject_Free(p)` has been called before, undefined behavior occurs.

  If *p* is *NULL*, no operation is performed.

In addition, the following macro sets are provided:

- `PyObject_MALLOC()`: alias to *`PyObject_Malloc()`*
- `PyObject_REALLOC()`: alias to *`PyObject_Realloc()`*
- `PyObject_FREE()`: alias to *`PyObject_Free()`*
- *`PyObject_Del()`*: alias to *`PyObject_Free()`*
- `PyObject_DEL()`: alias to `PyObject_FREE()` (so finally an alias to *`PyObject_Free()`*)

9.4 The pymalloc allocator

Python has a *pymalloc* allocator optimized for small objects (smaller or equal to 512 bytes) with a short lifetime. It uses memory mappings called "arenas" with a fixed size of 256 KiB. It falls back to `malloc()` and `realloc()` for allocations larger than 512 bytes.

pymalloc is the default allocator of `PyObject_Malloc()`.

The arena allocator uses the following functions:

- `mmap()` e `munmap()` se disponível,
- `malloc()` e `free()` do contrário.

Alterado na versão 2.7.7: The threshold changed from 256 to 512 bytes. The arena allocator now uses `mmap()` if available.

9.5 Exemplos

Here is the example from section *Visão Geral*, rewritten so that the I/O buffer is allocated from the Python heap by using the first function set:

```
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyString_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

The same code using the type-oriented function set:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyString_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

Note that in the two examples above, the buffer is always manipulated via functions belonging to the same set. Indeed, it is required to use the same memory API family for a given memory block, so that the risk of mixing different allocators is reduced to a minimum. The following code sequence contains two errors, one of which is labeled as *fatal* because it mixes two different allocators operating on different heaps.

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);  /* Wrong -- should be PyMem_Free() */
free(buf2);       /* Right -- allocated via malloc() */
free(buf1);       /* Fatal -- should be PyMem_Del()  */
```

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with `PyObject_New()`, `PyObject_NewVar()` and `PyObject_Del()`.

These will be explained in the next chapter on defining and implementing new object types in C.

CAPÍTULO 10

Suporte a implementação de Objetos

Este capítulo descreve as funções, tipos e macros usados ao definir novos tipos de objeto.

10.1 Alocar objetos na pilha

*PyObject** **_PyObject_New** (*PyTypeObject *type*)
  Return value: New reference.

*PyVarObject** **_PyObject_NewVar** (*PyTypeObject *type*, Py_ssize_t *size*)
  Return value: New reference. Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

void **_PyObject_Del** (*PyObject *op*)

*PyObject** **PyObject_Init** (*PyObject *op*, *PyTypeObject *type*)
  Return value: Borrowed reference. Inicializa um objeto *op* recém-alocado com seu tipo e sua referência inicial. Retorna o objeto inicializado. Se *type* indicar no detector de lixo cíclico que o objeto participa, ele é adicionado ao grupo do detector de objetos observados. Outros campos do objeto não são afetados.

*PyVarObject** **PyObject_InitVar** (*PyVarObject *op*, *PyTypeObject *type*, Py_ssize_t *size*)
  Return value: Borrowed reference. Isso faz tudo que `PyObject_Init()` faz, e também inicializa a informação de comprimento para um objeto tamanho de variável.

  Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

TYPE* **PyObject_New** (TYPE, *PyTypeObject *type*)
  Return value: New reference. Aloca um novo objeto Python usando o tipo de estrutura C *TYPE* e o tipo de objeto Python *type*. Campos não definidos no cabeçalho do objeto Python não são inicializados; a contagem de referências do objeto será um deles. O tamanho da alocação de memória é determinado do campo `tp_basicsize` do objeto tipo.

TYPE* **PyObject_NewVar** (TYPE, *PyTypeObject *type*, Py_ssize_t *size*)
  Return value: New reference. Aloca um novo objeto Python usando o tipo de estrutura C *TYPE* e o tipo de objeto Python *type*. Campos não definidos pelo cabeçalho do objeto Python não são inicializados. A memória alocada

permite a estrutura *TYPE* e os campos *size* do tamanho dado pelo campo `tp_itemsize` do tipo *type*. Isto é útil para implementar objetos como tuplas, que são capazes de determinar seu tamanho em tempo de construção. Incorporando o vetor de campos dentro da mesma alocação diminuindo o numero de alocações, melhorando a eficiência de gerenciamento de memória.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

void **`PyObject_Del`**(*PyObject* *op*)
Libera memória alocada a um objeto usando `PyObject_New()` ou `PyObject_NewVar()`. Isso é normalmente chamado por `tp_dealloc` manipulador especificado no tipo do objeto. Os campos do objeto não devem ser acessados após esta chamada, já que a memória não é mais um objeto Python válido.

*PyObject** **`Py_InitModule`**(char **name*, *PyMethodDef* **methods*)
Return value: Borrowed reference. Create a new module object based on a name and table of functions, returning the new module object.

Alterado na versão 2.3: Older versions of Python did not support *NULL* as the value for the *methods* argument.

*PyObject** **`Py_InitModule3`**(char **name*, *PyMethodDef* **methods*, char **doc*)
Return value: Borrowed reference. Create a new module object based on a name and table of functions, returning the new module object. If *doc* is non-*NULL*, it will be used to define the docstring for the module.

Alterado na versão 2.3: Older versions of Python did not support *NULL* as the value for the *methods* argument.

*PyObject** **`Py_InitModule4`**(char **name*, *PyMethodDef* **methods*, char **doc*, *PyObject* **self*, int *apiver*)
Return value: Borrowed reference. Create a new module object based on a name and table of functions, returning the new module object. If *doc* is non-*NULL*, it will be used to define the docstring for the module. If *self* is non-*NULL*, it will be passed to the functions of the module as their (otherwise *NULL*) first parameter. (This was added as an experimental feature, and there are no known uses in the current version of Python.) For *apiver*, the only value which should be passed is defined by the constant `PYTHON_API_VERSION`.

Nota: Most uses of this function should probably be using the `Py_InitModule3()` instead; only use this if you are sure you need it.

Alterado na versão 2.3: Older versions of Python did not support *NULL* as the value for the *methods* argument.

PyObject **`_Py_NoneStruct`**
Object which is visible in Python as `None`. This should only be accessed using the `Py_None` macro, which evaluates to a pointer to this object.

10.2 Estruturas Comuns de Objetos

There are a large number of structures which are used in the definition of object types for Python. This section describes these structures and how they are used.

All Python objects ultimately share a small number of fields at the beginning of the object's representation in memory. These are represented by the `PyObject` and `PyVarObject` types, which are defined, in turn, by the expansions of some macros also used, whether directly or indirectly, in the definition of all other Python objects.

`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. It corresponds to the fields defined by the expansion of the `PyObject_HEAD` macro.

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. It corresponds to the fields defined by the expansion of the `PyObject_VAR_HEAD` macro.

These macros are used in the definition of *PyObject* and *PyVarObject*:

PyObject_HEAD
This is a macro which expands to the declarations of the fields of the *PyObject* type; it is used when declaring new types which represent objects without a varying length. The specific fields it expands to depend on the definition of `Py_TRACE_REFS`. By default, that macro is not defined, and *PyObject_HEAD* expands to:

```
Py_ssize_t ob_refcnt;
PyTypeObject *ob_type;
```

When `Py_TRACE_REFS` is defined, it expands to:

```
PyObject *_ob_next, *_ob_prev;
Py_ssize_t ob_refcnt;
PyTypeObject *ob_type;
```

PyObject_VAR_HEAD
This is a macro which expands to the declarations of the fields of the *PyVarObject* type; it is used when declaring new types which represent objects with a length that varies from instance to instance. This macro always expands to:

```
PyObject_HEAD
Py_ssize_t ob_size;
```

Note that *PyObject_HEAD* is part of the expansion, and that its own expansion varies depending on the definition of `Py_TRACE_REFS`.

Py_TYPE(o)
This macro is used to access the `ob_type` member of a Python object. It expands to:

```
(((PyObject*)(o))->ob_type)
```

Novo na versão 2.6.

Py_REFCNT(o)
This macro is used to access the `ob_refcnt` member of a Python object. It expands to:

```
(((PyObject*)(o))->ob_refcnt)
```

Novo na versão 2.6.

Py_SIZE(o)
This macro is used to access the `ob_size` member of a Python object. It expands to:

```
(((PyVarObject*)(o))->ob_size)
```

Novo na versão 2.6.

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,
```

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.

PyMethodDef
Structure used to describe a method of an extension type. This structure has four fields:

Campo	Tipo em C	Significado
`ml_name`	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`	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. Of the calling convention flags, only *METH_VARARGS* and *METH_KEYWORDS* can be combined. Any of the calling convention flags can be combined with a binding flag.

METH_VARARGS
This is the typical calling convention, where the methods have the type *PyCFunction*. The function expects two *PyObject** values. The first one is the *self* object for methods; for module functions, it is the module object. The second parameter (often called *args*) is a tuple object representing all arguments. This parameter is typically processed using *PyArg_ParseTuple()* or *PyArg_UnpackTuple()*.

METH_KEYWORDS
Methods with these flags must be of type `PyCFunctionWithKeywords`. The function expects three parameters: *self*, *args*, and a dictionary of all the keyword arguments. The flag is typically combined with *METH_VARARGS*, and the parameters are typically processed using *PyArg_ParseTupleAndKeywords()*.

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.

METH_OLDARGS
This calling convention is deprecated. The method must be of type *PyCFunction*. The second argument is *NULL* if no arguments are given, a single object if exactly one argument is given, and a tuple of objects if more than one argument is given. There is no way for a function using this convention to distinguish between a call with multiple arguments and a call with a tuple as the only 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.

Novo na versão 2.3.

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.

Novo na versão 2.3.

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.

Novo na versão 2.4.

PyMemberDef
Structure which describes an attribute of a type which corresponds to a C struct member. Its fields are:

Campo	Tipo em C	Significado
`name`	char *	name of the member
`type`	int	the type of the member in the C struct
`offset`	Py_stamanho_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`	char *	points to the contents of the docstring

`type` can be one of many `T_` macros corresponding to various C types. When the member is accessed in Python, it will be converted to the equivalent Python type.

Macro name	C type
T_SHORT	short
T_INT	int
T_LONG	extenso, comprido
T_FLOAT	float
T_DOUBLE	double
T_STRING	char *
T_OBJECT	PyObject *
T_OBJECT_EX	PyObject *
T_CHAR	char
T_BYTE	char
T_UBYTE	unsigned char
T_UINT	unsigned int
T_USHORT	unsigned short
T_ULONG	unsigned long
T_BOOL	char
T_LONGLONG	long long
T_ULONGLONG	unsigned long long
T_PYSSIZET	Py_stamanho_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`. Only `T_OBJECT` and `T_OBJECT_EX` members can be deleted. (They are set to *NULL*).

PyGetSetDef

Structure to define property-like access for a type. See also description of the *PyTypeObject.tp_getset* slot.

Campo	Tipo em C	Significado
name	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	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.

*PyObject** **Py_FindMethod**(*PyMethodDef table[]*, *PyObject *ob*, char **name*)

Return value: New reference. Return a bound method object for an extension type implemented in C. This can be useful in the implementation of a `tp_getattro` or `tp_getattr` handler that does not use the `PyObject_GenericGetAttr()` function.

10.3 Objetos de tipo

Talvez uma das estruturas mais importantes do sistema de objetos Python seja a estrutura que define um novo tipo: a estrutura `PyTypeObject`. Objetos de tipo podem ser manipulados usando qualquer uma das funções `PyObject_*()` ou `PyType_*()`, mas não oferecem muita coisa interessante para a maioria dos aplicativos Python. Esses objetos são fundamentais para o comportamento dos objetos, portanto, são muito importantes para o próprio interpretador e para qualquer módulo de extensão que implemente novos tipos.

Os objetos de tipo são bastante grandes em comparação com a maioria dos tipos padrão. A razão para o tamanho é que cada objeto de tipo armazena um grande número de valores, principalmente indicadores de função C, cada um dos quais implementa uma pequena parte da funcionalidade do tipo. Os campos do objeto de tipo são examinados em detalhes nesta seção. Os campos serão descritos na ordem em que ocorrem na estrutura.

Typedefs: unaryfunc, binaryfunc, ternaryfunc, inquiry, coercion, intargfunc, intintargfunc, intobjargproc, intintobjargproc, objobjargproc, destructor, freefunc, printfunc, getattrfunc, getattrofunc, setattrfunc, setattrofunc, cmpfunc, reprfunc, hashfunc

The structure definition for `PyTypeObject` can be found in `Include/object.h`. For convenience of reference, this repeats the definition found there:

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    char *tp_name; /* For printing, in format "<module>.<name>" */
    int tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    printfunc tp_print;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    cmpfunc tp_compare;
    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;
```

(continua na próxima página)

(continuação da página anterior)

```
    /* Flags to define presence of optional/expanded features */
    long tp_flags;

    char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    long tp_weaklistoffset;

    /* Added in release 2.2 */
    /* 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;
    long 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;

} PyTypeObject;
```

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.

These fields are not inherited by subtypes.

Py_ssize_t **PyObject.ob_refcnt**

This is the type object's reference count, initialized to `1` by the `PyObject_HEAD_INIT` macro. Note that for statically allocated type objects, the type's instances (objects whose `ob_type` points back to the type) do *not* count as references. But for dynamically allocated type objects, the instances *do* count as references.

This field is not inherited by subtypes.

Alterado na versão 2.5: This field used to be an `int` type. This might require changes in your code for properly supporting 64-bit systems.

*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: in Python 2.2, it is set to `&PyType_Type`; in Python 2.2.1 and later it is initialized to the `ob_type` field of the base class. `PyType_Ready()` will not change this field if it is non-zero.

In Python 2.2, this field is not inherited by subtypes. In 2.2.1, and in 2.3 and beyond, it is inherited by subtypes.

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.

This field is not inherited by subtypes.

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 is not inherited by subtypes.

Py_ssize_t **PyTypeObject.tp_basicsize**
Py_ssize_t **PyTypeObject.tp_itemsize**

These fields allow calculating the size in bytes of instances of the type.

There are two kinds of types: types with fixed-length instances have a zero `tp_itemsize` field, types with variable-length instances have a non-zero `tp_itemsize` field. For a type with fixed-length instances, all instances have the same size, given in `tp_basicsize`.

For a type with variable-length instances, the instances must have an `ob_size` field, and the instance size is `tp_basicsize` plus N times `tp_itemsize`, where N is the "length" of the object. The value of N is typically stored in the instance's `ob_size` field. There are exceptions: for example, long 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 (this is new in Python 2.2; in 2.1 and 2.0, the GC header size was included in `tp_basicsize`).

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).

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`).

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 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 finally (as its last action) 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()`.

This field is inherited by subtypes.

printfunc **PyTypeObject.tp_print**

An optional pointer to the instance print function.

The print function is only called when the instance is printed to a *real* file; when it is printed to a pseudo-file (like a `StringIO` instance), the instance's `tp_repr` or `tp_str` function is called to convert it to a string. These are also called when the type's `tp_print` field is *NULL*. A type should never implement `tp_print` in a way that produces different output than `tp_repr` or `tp_str` would.

The print function is called with the same signature as `PyObject_Print()`: `int tp_print(PyObject *self, FILE *file, int flags)`. The *self* argument is the instance to be printed. The *file* argument is the stdio file to which it is to be printed. The *flags* argument is composed of flag bits. The only flag bit currently defined is `Py_PRINT_RAW`. When the `Py_PRINT_RAW` flag bit is set, the instance should be printed the same way as `tp_str` would format it; when the `Py_PRINT_RAW` flag bit is clear, the instance should be printed the same was as `tp_repr` would format it. It should return `-1` and set an exception condition when an error occurred during the comparison.

It is possible that the `tp_print` field will be deprecated. In any case, it is recommended not to define `tp_print`, but instead to rely on `tp_repr` and `tp_str` for printing.

This field is inherited by subtypes.

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. The signature is

```
PyObject * tp_getattr(PyObject *o, char *attr_name);
```

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. The signature is

```
PyObject * tp_setattr(PyObject *o, char *attr_name, PyObject *v);
```

The *v* argument is set to *NULL* to delete the attribute. 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*.

cmpfunc **PyTypeObject.tp_compare**
An optional pointer to the three-way comparison function.

The signature is the same as for *PyObject_Compare()*. The function should return `1` if *self* greater than *other*, `0` if *self* is equal to *other*, and `-1` if *self* less than *other*. It should return `-1` and set an exception condition when an error occurred during the comparison.

This field is inherited by subtypes together with *tp_richcompare* and *tp_hash*: a subtypes inherits all three of *tp_compare*, *tp_richcompare*, and *tp_hash* when the subtype's *tp_compare*, *tp_richcompare*, and *tp_hash* are all *NULL*.

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()*; it 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.

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.

This field is inherited by subtypes.

*PyNumberMethods** **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*.

The `tp_as_number` field is not inherited, but the contained fields are inherited individually.

*PySequenceMethods** **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*.

The `tp_as_sequence` field is not inherited, but the contained fields are inherited individually.

*PyMappingMethods** **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*.

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()`; it must return a C long. 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`.

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()`.

When this field is not set, two possibilities exist: if the `tp_compare` and `tp_richcompare` fields are both *NULL*, a default hash value based on the object's address is returned; otherwise, a `TypeError` is raised.

This field is inherited by subtypes together with `tp_richcompare` and `tp_compare`: a subtypes inherits all three of `tp_compare`, `tp_richcompare`, and `tp_hash`, when the subtype's `tp_compare`, `tp_richcompare` and `tp_hash` are all *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()`.

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()`; it must return a string or a Unicode object. This function should return a "friendly" string representation of the object, as this is the representation that will be used by the print statement.

When this field is not set, `PyObject_Repr()` is called to return a string representation.

This field is inherited by subtypes.

getattrofunc **PyTypeObject.tp_getattro**

An optional pointer to the get-attribute function.

The signature is the same as for `PyObject_GetAttr()`. It is usually convenient to set this field to `PyObject_GenericGetAttr()`, which implements the normal way of looking for object attributes.

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*.

setattrofunc **PyTypeObject.tp_setattro**

An optional pointer to the function for setting and deleting attributes.

The signature is the same as for `PyObject_SetAttr()`, but setting *v* 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.

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*.

*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*.

The `tp_as_buffer` field is not inherited, but the contained fields are inherited individually.

long **PyTypeObject.tp_flags**

This field is a bit mask of various flags. Some flags indicate variant semantics for certain situations; others are used to indicate that certain fields in the type object (or in the extension structures referenced via `tp_as_number`, `tp_as_sequence`, `tp_as_mapping`, and `tp_as_buffer`) that were historically not always present are valid; if such a flag bit is clear, the type fields it guards must not be accessed and must be considered to have a zero or *NULL* value instead.

Inheritance 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 (as indicated by the `Py_TPFLAGS_HAVE_RICHCOMPARE` flag bit) and have *NULL* values.

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_HAVE_GETCHARBUFFER

If this bit is set, the `PyBufferProcs` struct referenced by `tp_as_buffer` has the `bf_getcharbuffer` field.

Py_TPFLAGS_HAVE_SEQUENCE_IN

If this bit is set, the `PySequenceMethods` struct referenced by `tp_as_sequence` has the `sq_contains` field.

Py_TPFLAGS_GC

This bit is obsolete. The bit it used to name is no longer in use. The symbol is now defined as zero.

Py_TPFLAGS_HAVE_INPLACEOPS

If this bit is set, the `PySequenceMethods` struct referenced by `tp_as_sequence` and the `PyNumberMethods` structure referenced by `tp_as_number` contain the fields for in-place operators. In particular, this means that the `PyNumberMethods` structure has the fields `nb_inplace_add`, `nb_inplace_subtract`, `nb_inplace_multiply`, `nb_inplace_divide`, `nb_inplace_remainder`, `nb_inplace_power`, `nb_inplace_lshift`, `nb_inplace_rshift`, `nb_inplace_and`, `nb_inplace_xor`, and `nb_inplace_or`; and the `PySequenceMethods` struct has the fields `sq_inplace_concat` and `sq_inplace_repeat`.

Py_TPFLAGS_CHECKTYPES

If this bit is set, the binary and ternary operations in the `PyNumberMethods` structure referenced by `tp_as_number` accept arguments of arbitrary object types, and do their own type conversions if needed. If this bit is clear, those operations require that all arguments have the current type as their type, and the caller is supposed to perform a coercion operation first. This applies to `nb_add`, `nb_subtract`, `nb_multiply`, `nb_divide`, `nb_remainder`, `nb_divmod`, `nb_power`, `nb_lshift`, `nb_rshift`, `nb_and`, `nb_xor`, and `nb_or`.

Py_TPFLAGS_HAVE_RICHCOMPARE

If this bit is set, the type object has the `tp_richcompare` field, as well as the `tp_traverse` and the `tp_clear` fields.

Py_TPFLAGS_HAVE_WEAKREFS

If this bit is set, the *tp_weaklistoffset* field is defined. Instances of a type are weakly referenceable if the type's *tp_weaklistoffset* field has a value greater than zero.

Py_TPFLAGS_HAVE_ITER
If this bit is set, the type object has the *tp_iter* and *tp_iternext* fields.

Py_TPFLAGS_HAVE_CLASS
If this bit is set, the type object has several new fields defined starting in Python 2.2: *tp_methods*, *tp_members*, *tp_getset*, *tp_base*, *tp_dict*, *tp_descr_get*, *tp_descr_set*, *tp_dictoffset*, *tp_init*, *tp_alloc*, *tp_new*, *tp_free*, *tp_is_gc*, *tp_bases*, *tp_mro*, *tp_cache*, *tp_subclasses*, and *tp_weaklist*.

Py_TPFLAGS_HEAPTYPE
This bit is set when the type object itself is allocated on the heap. 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).

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).

Py_TPFLAGS_READY
This bit is set when the type object has been fully initialized by *PyType_Ready()*.

Py_TPFLAGS_READYING
This bit is set while *PyType_Ready()* is in the process of initializing the type object.

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 *Suporte a Coleta de Lixo Cíclica*. This bit also implies that the GC-related fields *tp_traverse* and *tp_clear* are present in the type object; but those fields also exist when *Py_TPFLAGS_HAVE_GC* is clear but *Py_TPFLAGS_HAVE_RICHCOMPARE* is set.

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_GETCHARBUFFER*, *Py_TPFLAGS_HAVE_SEQUENCE_IN*, *Py_TPFLAGS_HAVE_INPLACEOPS*, *Py_TPFLAGS_HAVE_RICHCOMPARE*, *Py_TPFLAGS_HAVE_WEAKREFS*, *Py_TPFLAGS_HAVE_ITER*, and *Py_TPFLAGS_HAVE_CLASS*.

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.

This field is *not* inherited by subtypes.

The following three fields only exist if the *Py_TPFLAGS_HAVE_RICHCOMPARE* flag bit is set.

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. More information about Python's garbage collection scheme can be found in section *Suporte a Coleta de Lixo Cíclica*.

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. 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.

Note that `Py_VISIT()` requires the *visit* and *arg* parameters to `local_traverse()` to have these specific names; don't name them just anything.

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 *and* the subtype has the `Py_TPFLAGS_HAVE_RICHCOMPARE` flag bit set.

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 `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.

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 *Suporte a Coleta de Lixo Cíclica*.

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 *and* the subtype has the *Py_TPFLAGS_HAVE_RICHCOMPARE* flag bit set.

richcmpfunc **PyTypeObject.tp_richcompare**

An optional pointer to the rich comparison function, whose signature is `PyObject *tp_richcompare(PyObject *a, PyObject *b, int op)`.

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.

Nota: If you want to implement a type for which only a limited set of comparisons makes sense (e.g. == and !=, but not < and friends), directly raise `TypeError` in the rich comparison function.

This field is inherited by subtypes together with *tp_compare* and *tp_hash*: a subtype inherits all three of *tp_compare*, *tp_richcompare*, and *tp_hash*, when the subtype's *tp_compare*, *tp_richcompare*, and *tp_hash* are all *NULL*.

The following constants are defined to be used as the third argument for *tp_richcompare* and for *PyObject_RichCompare()*:

Constante	Comparação
`Py_LT`	<
`Py_LE`	<=
`Py_EQ`	==
`Py_NE`	!=
`Py_GT`	>
`Py_GE`	>=

The next field only exists if the *Py_TPFLAGS_HAVE_WEAKREFS* flag bit is set.

long **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.

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.

The next two fields only exist if the *Py_TPFLAGS_HAVE_ITER* flag bit is set.

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, and classic instances always have this function, even if they don't define an `__iter__()` method).

This function has the same signature as *PyObject_GetIter()*.

This field is inherited by subtypes.

iternextfunc **PyTypeObject.tp_iternext**

An optional pointer to a function that returns the next item in an iterator. 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 normally signals that the instances of this type are iterators (although classic instances always have this function, even if they don't define a `next()` method).

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()*.

This field is inherited by subtypes.

The next fields, up to and including *tp_weaklist*, only exist if the *Py_TPFLAGS_HAVE_CLASS* flag bit is set.

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.

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.

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.

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.

This field is not inherited by subtypes (obviously), but it 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__()`).

This field is not inherited by subtypes (though the attributes defined in here are inherited through a different mechanism).

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);
```

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. This field is inherited by subtypes.

long **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 long 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()`.)

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.)

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. (VERSION NOTE: described here is what is implemented in Python 2.2.1 and later. In Python 2.2, the *tp_init* of the type of the object returned by *tp_new* was always called, if not *NULL*.)

This field is inherited by subtypes.

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)
```

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*.

Do not use this function to do any other instance initialization, not even to allocate additional memory; that should be done by *tp_new*.

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement); in the latter, this field is always set to *PyType_GenericAlloc()*, to force a standard heap allocation strategy. That is also the recommended value for statically defined types.

newfunc **PyTypeObject.tp_new**

An optional pointer to an instance creation function.

If this function is *NULL* for a particular type, that type cannot be called to create new instances; presumably there is some other way to create instances, like a factory 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*.

This field is inherited by subtypes, except it is not inherited by static types whose *tp_base* is *NULL* or `&PyBaseObject_Type`. The latter exception is a precaution so that old extension types don't become callable simply by being linked with Python 2.2.

destructor **PyTypeObject.tp_free**

An optional pointer to an instance deallocation function.

The signature of this function has changed slightly: in Python 2.2 and 2.2.1, its signature is `destructor`:

```
void tp_free(PyObject *)
```

In Python 2.3 and beyond, its signature is `freefunc`:

```
void tp_free(void *)
```

The only initializer that is compatible with both versions is `_PyObject_Del`, whose definition has suitably adapted in Python 2.3.

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement); in the latter, this field is set to a deallocator suitable to match *PyType_GenericAlloc()* and the value of the *Py_TPFLAGS_HAVE_GC* flag bit.

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.)

This field is inherited by subtypes. (VERSION NOTE: in Python 2.2, it was not inherited. It is inherited in 2.2.1 and later versions.)

*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.

This field is not inherited.

*PyObject** **PyTypeObject.tp_mro**

Tuple containing the expanded set of base types, starting with the type itself and ending with `object`, in Method Resolution Order.

This field is not inherited; it is calculated fresh by *PyType_Ready()*.

*PyObject** **PyTypeObject.tp_cache**
Unused. Not inherited. Internal use only.

*PyObject** **PyTypeObject.tp_subclasses**
List of weak references to subclasses. Not inherited. Internal use only.

*PyObject** **PyTypeObject.tp_weaklist**
Weak reference list head, for weak references to this type object. Not inherited. Internal use only.

The remaining fields are only defined if the feature test macro `COUNT_ALLOCS` is defined, and are for internal use only. They are documented here for completeness. None of these fields are inherited by subtypes. See the `PYTHONSHOWALLOCCOUNT` environment variable.

Py_ssize_t **PyTypeObject.tp_allocs**
Number of allocations.

Py_ssize_t **PyTypeObject.tp_frees**
Number of frees.

Py_ssize_t **PyTypeObject.tp_maxalloc**
Maximum simultaneously allocated objects.

*PyTypeObject** **PyTypeObject.tp_next**
Pointer to the next type object with a non-zero `tp_allocs` field.

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.

10.4 Number Object Structures

PyNumberMethods
This structure holds pointers to the functions which an object uses to implement the number protocol. Almost every function below is used by the function of similar name documented in the *Número de Protocolo* section.

Here is the structure definition:

```
typedef struct {
     binaryfunc nb_add;
     binaryfunc nb_subtract;
     binaryfunc nb_multiply;
     binaryfunc nb_divide;
     binaryfunc nb_remainder;
     binaryfunc nb_divmod;
     ternaryfunc nb_power;
     unaryfunc nb_negative;
     unaryfunc nb_positive;
     unaryfunc nb_absolute;
     inquiry nb_nonzero;       /* Used by PyObject_IsTrue */
     unaryfunc nb_invert;
     binaryfunc nb_lshift;
     binaryfunc nb_rshift;
     binaryfunc nb_and;
     binaryfunc nb_xor;
     binaryfunc nb_or;
```

(continua na próxima página)

(continuação da página anterior)

```
     coercion nb_coerce;       /* Used by the coerce() function */
     unaryfunc nb_int;
     unaryfunc nb_long;
     unaryfunc nb_float;
     unaryfunc nb_oct;
     unaryfunc nb_hex;

     /* Added in release 2.0 */
     binaryfunc nb_inplace_add;
     binaryfunc nb_inplace_subtract;
     binaryfunc nb_inplace_multiply;
     binaryfunc nb_inplace_divide;
     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;

     /* Added in release 2.2 */
     binaryfunc nb_floor_divide;
     binaryfunc nb_true_divide;
     binaryfunc nb_inplace_floor_divide;
     binaryfunc nb_inplace_true_divide;

     /* Added in release 2.5 */
     unaryfunc nb_index;
} PyNumberMethods;
```

Binary and ternary functions may receive different kinds of arguments, depending on the flag bit *Py_TPFLAGS_CHECKTYPES*:

- If *Py_TPFLAGS_CHECKTYPES* is not set, the function arguments are guaranteed to be of the object's type; the caller is responsible for calling the coercion method specified by the `nb_coerce` member to convert the arguments:

  coercion **PyNumberMethods.nb_coerce**

  This function is used by *PyNumber_CoerceEx()* and has the same signature. The first argument is always a pointer to an object of the defined type. If the conversion to a common "larger" type is possible, the function replaces the pointers with new references to the converted objects and returns `0`. If the conversion is not possible, the function returns `1`. If an error condition is set, it will return `-1`.

- If the *Py_TPFLAGS_CHECKTYPES* flag is set, 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). This is the recommended way; with Python 3 coercion will disappear completely.

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.

10.5 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_Length()* 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 has the same signature. 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()* and *PyObject_DelItem()*. 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.

10.6 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.

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. 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. 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.

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.

10.7 Buffer Object Structures

The buffer interface exports a model where an object can expose its internal data as a set of chunks of data, where each chunk is specified as a pointer/length pair. These chunks are called *segments* and are presumed to be non-contiguous in memory.

If an object does not export the buffer interface, then its `tp_as_buffer` member in the `PyTypeObject` structure should be *NULL*. Otherwise, the `tp_as_buffer` will point to a `PyBufferProcs` structure.

Nota: It is very important that your `PyTypeObject` structure uses `Py_TPFLAGS_DEFAULT` for the value of the `tp_flags` member rather than `0`. This tells the Python runtime that your `PyBufferProcs` structure contains the `bf_getcharbuffer` slot. Older versions of Python did not have this member, so a new Python interpreter using an old extension needs to be able to test for its presence before using it.

PyBufferProcs

Structure used to hold the function pointers which define an implementation of the buffer protocol.

The first slot is `bf_getreadbuffer`, of type `readbufferproc`. If this slot is *NULL*, then the object does not support reading from the internal data. This is non-sensical, so implementors should fill this in, but callers should test that the slot contains a non-*NULL* value.

The next slot is `bf_getwritebuffer` having type `writebufferproc`. This slot may be *NULL* if the object does not allow writing into its returned buffers.

The third slot is `bf_getsegcount`, with type `segcountproc`. This slot must not be *NULL* and is used to inform the caller how many segments the object contains. Simple objects such as `PyString_Type` and `PyBuffer_Type` objects contain a single segment.

The last slot is `bf_getcharbuffer`, of type `charbufferproc`. This slot will only be present if the `Py_TPFLAGS_HAVE_GETCHARBUFFER` flag is present in the `tp_flags` field of the object's `PyTypeObject`. Before using this slot, the caller should test whether it is present by using the `PyType_HasFeature()` function. If the flag is present, `bf_getcharbuffer` may be *NULL*, indicating that the object's contents cannot be used as *8-bit characters*. The slot function may also raise an error if the object's contents cannot be interpreted as 8-bit characters. For example, if the object is an array which is configured to hold floating point values, an exception may be raised if a caller attempts to use `bf_getcharbuffer` to fetch a sequence of 8-bit characters. This notion of exporting the internal buffers as "text" is used to distinguish between objects that are binary in nature, and those which have character-based content.

Nota: The current policy seems to state that these characters may be multi-byte characters. This implies that a buffer size of *N* does not mean there are *N* characters present.

Py_TPFLAGS_HAVE_GETCHARBUFFER

Flag bit set in the type structure to indicate that the `bf_getcharbuffer` slot is known. This being set does not indicate that the object supports the buffer interface or that the `bf_getcharbuffer` slot is non-*NULL*.

Py_ssize_t **(*readbufferproc)** (*PyObject *self*, Py_ssize_t *segment*, void ***ptrptr*)

Return a pointer to a readable segment of the buffer in `*ptrptr`. This function is allowed to raise an exception, in which case it must return `-1`. The *segment* which is specified must be zero or positive, and strictly less than the number of segments returned by the `bf_getsegcount` slot function. On success, it returns the length of the segment, and sets `*ptrptr` to a pointer to that memory.

Py_ssize_t **(*writebufferproc)** (*PyObject *self*, Py_ssize_t *segment*, void ***ptrptr*)

Return a pointer to a writable memory buffer in `*ptrptr`, and the length of that segment as the function return value. The memory buffer must correspond to buffer segment *segment*. Must return `-1` and set an exception on error. `TypeError` should be raised if the object only supports read-only buffers, and `SystemError` should be raised when *segment* specifies a segment that doesn't exist.

Py_ssize_t **(*segcountproc)** (*PyObject* **self*, Py_ssize_t **lenp*)
Return the number of memory segments which comprise the buffer. If *lenp* is not *NULL*, the implementation must report the sum of the sizes (in bytes) of all segments in `*lenp`. The function cannot fail.

Py_ssize_t **(*charbufferproc)** (*PyObject* **self*, Py_ssize_t *segment*, char ***ptrptr*)
Return the size of the segment *segment* that *ptrptr* is set to. `*ptrptr` is set to the memory buffer. Returns `-1` on error.

10.8 Suporte a Coleta de Lixo Cíclica

Python's support for detecting and collecting garbage which involves circular references requires support from object types which are "containers" for other objects which may also be containers. Types which do not store references to other objects, or which only store references to atomic types (such as numbers or strings), do not need to provide any explicit support for garbage collection.

To create a container type, the `tp_flags` field of the type object must include the `Py_TPFLAGS_HAVE_GC` and provide an implementation of the `tp_traverse` handler. If instances of the type are mutable, a `tp_clear` implementation must also be provided.

Py_TPFLAGS_HAVE_GC
Objects with a type with this flag set must conform with the rules documented here. For convenience these objects will be referred to as container objects.

Constructors for container types must conform to two rules:

1. The memory for the object must be allocated using `PyObject_GC_New()` or `PyObject_GC_NewVar()`.
2. Once all the fields which may contain references to other containers are initialized, it must call `PyObject_GC_Track()`.

TYPE* **PyObject_GC_New** (TYPE, *PyTypeObject* **type*)
Analogous to `PyObject_New()` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

TYPE* **PyObject_GC_NewVar** (TYPE, *PyTypeObject* **type*, Py_ssize_t *size*)
Analogous to `PyObject_NewVar()` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

Alterado na versão 2.5: This function used an `int` type for *size*. This might require changes in your code for properly supporting 64-bit systems.

TYPE* **PyObject_GC_Resize** (TYPE, *PyVarObject* **op*, Py_ssize_t *newsize*)
Resize an object allocated by `PyObject_NewVar()`. Returns the resized object or *NULL* on failure. *op* must not be tracked by the collector yet.

Alterado na versão 2.5: This function used an `int` type for *newsize*. This might require changes in your code for properly supporting 64-bit systems.

void **PyObject_GC_Track** (*PyObject* **op*)
Adds the object *op* to the set of container objects tracked by the collector. The collector can run at unexpected times so objects must be valid while being tracked. This should be called once all the fields followed by the `tp_traverse` handler become valid, usually near the end of the constructor.

void **_PyObject_GC_TRACK** (*PyObject* **op*)
A macro version of `PyObject_GC_Track()`. It should not be used for extension modules.

Similarly, the deallocator for the object must conform to a similar pair of rules:

1. Before fields which refer to other containers are invalidated, `PyObject_GC_UnTrack()` must be called.

2. The object's memory must be deallocated using `PyObject_GC_Del()`.

void **PyObject_GC_Del**(void **op*)
Releases memory allocated to an object using `PyObject_GC_New()` or `PyObject_GC_NewVar()`.

void **PyObject_GC_UnTrack**(void **op*)
Remove the object *op* from the set of container objects tracked by the collector. Note that `PyObject_GC_Track()` can be called again on this object to add it back to the set of tracked objects. The deallocator (`tp_dealloc` handler) should call this for the object before any of the fields used by the `tp_traverse` handler become invalid.

void **_PyObject_GC_UNTRACK**(PyObject **op*)
A macro version of `PyObject_GC_UnTrack()`. It should not be used for extension modules.

The `tp_traverse` handler accepts a function parameter of this type:

int **(*visitproc)**(PyObject **object*, void **arg*)
Type of the visitor function passed to the `tp_traverse` handler. The function should be called with an object to traverse as *object* and the third parameter to the `tp_traverse` handler as *arg*. The Python core uses several visitor functions to implement cyclic garbage detection; it's not expected that users will need to write their own visitor functions.

The `tp_traverse` handler must have the following type:

int **(*traverseproc)**(PyObject **self*, visitproc *visit*, void **arg*)
Traversal function for a container object. Implementations must call the *visit* function for each object directly contained by *self*, with the parameters to *visit* being the contained object and the *arg* value passed to the handler. The *visit* function must not be called with a *NULL* object argument. If *visit* returns a non-zero value that value should be returned immediately.

To simplify writing `tp_traverse` handlers, a `Py_VISIT()` macro is provided. In order to use this macro, the `tp_traverse` implementation must name its arguments exactly *visit* and *arg*:

void **Py_VISIT**(PyObject **o*)
If *o* is not *NULL*, call the *visit* callback, with arguments *o* and *arg*. If *visit* returns a non-zero value, then return it. Using this macro, `tp_traverse` handlers look like:

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

Novo na versão 2.4.

The `tp_clear` handler must be of the `inquiry` type, or *NULL* if the object is immutable.

int **(*inquiry)**(PyObject **self*)
Drop references that may have created reference cycles. Immutable objects do not have to define this method since they can never directly create reference cycles. Note that the object must still be valid after calling this method (don't just call `Py_DECREF()` on a reference). The collector will call this method if it detects that this object is involved in a reference cycle.

APÊNDICE A

Glossário

>>> O prompt padrão do shell interativo do Python. Normalmente visto em exemplos de código que podem ser executados interativamente no interpretador.

... The default Python prompt of the interactive shell when entering code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

2to3 Uma ferramenta que tenta converter código Python 2.x em código Python 3.x tratando a maioria das incompatibilidades que podem se detectadas com análise do código-fonte e navegação na árvore sintática.

O 2to3 está disponível na biblioteca padrão como `lib2to3`; um ponto de entrada é disponibilizado como `Tools/scripts/2to3`. Veja 2to3-reference.

classe base abstrata Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections` module), numbers (in the `numbers` module), and streams (in the `io` module). You can create your own ABCs with the `abc` module.

argumento A value passed to a *function* (or *method*) when calling the function. There are two types of arguments:

- *argumento nomeado*: um argumento precedido por um identificador (por exemplo, `nome=`) na chamada de uma função ou passada como um valor em um dicionário precedido por `**`. Por exemplo, `3` e `5` são ambos argumentos nomeados na chamada da função `complex()` a seguir:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *argumento posicional*: um argumento que não é um argumento nomeado. Argumentos posicionais podem aparecer no início da lista de argumentos e/ou podem ser passados com elementos de um *iterável* precedido por `*`. Por exemplo, `3` e `5` são ambos argumentos posicionais nas chamadas a seguir:

```
complex(3, 5)
complex(*(3, 5))
```

Argumentos são atribuídos às variáveis locais nomeadas no corpo da função. Veja a seção calls para as regras de atribuição. Sintaticamente, qualquer expressão pode ser usada para representar um argumento; avaliada a expressão, o valor é atribuído à variável local.

See also the *parameter* glossary entry and the FAQ question on the difference between arguments and parameters.

atributo Um valor associado a um objeto que é referenciado pelo nome separado por um ponto. Por exemplo, se um objeto *o* tem um atributo *a* esse seria referenciado como *o.a*.

BDFL Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.

objeto byte ou similar An object that supports the *buffer protocol*, like `str`, `bytearray` or `memoryview`. Bytes-like objects can be used for various operations that expect binary data, such as compression, saving to a binary file or sending over a socket. Some operations need the binary data to be mutable, in which case not all bytes-like objects can apply.

bytecode Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This "intermediate language" is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

Uma lista de instruções bytecode pode ser encontrada na documentação para o módulo dis.

Classe Um modelo para criação de objetos definidos pelo usuário. Definições de classe normalmente contém definições de métodos que operam sobre instâncias da classe.

classic class Any class which does not inherit from `object`. See *new-style class*. Classic classes have been removed in Python 3.

coerção The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer `3`, but in `3+4.5`, each argument is of a different type (one int, one float), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` built-in function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

número complexo Uma extensão ao familiar sistema de números reais em que todos os números são expressos como uma soma de uma parte real e uma parte imaginária. Números imaginários são múltiplos reais da unidade imaginária (a raiz quadrada de `-1`), normalmente escrita como `i` em matemática ou `j` em engenharia. O Python tem suporte nativo para números complexos, que são escritos com esta última notação; a parte imaginária escrita com um sufixo `j`, p.ex., `3+1j`. Para ter acesso aos equivalentes para números complexos do módulo `math`, utilize `cmath`. O uso de números complexos é uma funcionalidade matemática bastante avançada. Se você não sabe se irá precisar deles, é quase certo que você pode ignorá-los sem problemas.

gerenciador de contexto Um objeto que controla o ambiente visto numa instrução `with` por meio da definição dos métodos `__enter__()` e `__exit__()`. Veja **PEP 343**.

CPython A implementação canônica da linguagem de programação Python, como disponibilizada pelo python.org. O termo "CPython" é quando for necessário distinguir esta implementação de outras como Jython ou IronPython.

decorador Uma função que retorna outra função, geralmente aplicada como uma transformação de função usando a sintaxe `@wrapper`. Exemplos comuns para decoradores são `classmethod()` e `staticmethod()`.

A sintaxe do decorador é meramente um açúcar-sintático, as duas definições de funções a seguir são semanticamente equivalentes:

```
def f(...):
    ...
f = staticmethod(f)
```

(continua na próxima página)

(continuação da página anterior)

```
@staticmethod
def f(...):
    ...
```

O mesmo conceito existe para as classes, mas não é comumente utilizado. Veja a documentação de function definitions e class definitions para obter mais informações sobre decoradores.

descritor Any *new-style* object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

Para obter mais informações sobre os métodos dos descritores, veja: descriptors.

dicionário An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

visualização de dicionário The objects returned from `dict.viewkeys()`, `dict.viewvalues()`, and `dict.viewitems()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See dict-views.

docstring Uma string literal que aparece como primeira expressão numa classe, função ou módulo. Ainda que sejam ignoradas quando a suíte é executada, é reconhecida pelo compilador que a coloca no atributo `__doc__` da classe, função ou módulo que a encapsula. Como ficam disponíveis por meio de introspecção, docstrings são o lugar canônico para documentação do objeto.

duck-typing (tipagem pato) Um estilo de programação que não verifica o tipo do objeto para determinar se ele possui a interface correta; em vez disso, o método ou atributo é simplesmente chamado ou utilizado ("Se se parece com um pato e grasna como um pato, então deve ser um pato.") Enfatizando interfaces ao invés de tipos específicos, o código bem desenvolvido aprimora sua flexibilidade por permitir substituição polimórfica. Tipagem pato evita necessidade de testes que usem `type()` ou `isinstance()`. (Note, porém, que a a tipagem pato pode ser complementada com o uso de *classes base abstratas*.) Ao invés disso, são normalmente empregados testes `hasattr()` ou programação *EAFP*.

EAFP Iniciais da expressão em inglês "easier to ask for forgiveness than permission" que significa "é mais fácil pedir perdão que permissão". Este estilo de codificação comum em Python assume a existência de chaves ou atributos válidos e captura exceções caso essa premissa se prove falsa. Este estilo limpo e rápido se caracteriza pela presença de várias instruções `try` e `except`. A técnica diverge do estilo *LBYL*, comum em outras linguagens como C, por exemplo.

expressão A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as `print` or `if`. Assignments are also statements, not expressions.

extension module (módulo de extensão) Um módulo escrito em C ou C++, usando a API C de Python para interagir tanto com código de usuário quanto do núcleo.

objeto arquivo Um objeto que expõe uma API orientada a arquivos (com métodos tais como `read()` ou `write()`) para um recurso subjacente. Dependendo da maneira como foi criado, um objeto arquivo pode mediar o acesso a um arquivo real no disco ou outro tipo de dispositivo de armazenamento ou de comunicação (por exemplo a entrada/saída padrão, buffers em memória, soquetes, pipes, etc.). Objetos arquivo também são chamados de *file-like objects* ou *streams*.

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the io module. The canonical way to create a file object is by using the open() function.

objeto como-arquivo Um sinônimo do termo *file object*.

localizador An object that tries to find the *loader* for a module. It must implement a method named find_module(). See **PEP 302** for details.

divisão pelo piso Divisão matemática que arredonda para baixo para o inteiro mais próximo. O operador de divisão pelo piso é //. Por exemplo, a expressão 11 // 4 retorna o valor 2 ao invés de 2.75, que seria retornado pela divisão de ponto flutuante. Note que (-11) // 4 é -3 porque é -2.75 arredondado *para baixo*. Consulte a **PEP 238**.

função Uma série de instruções que retorna algum valor para um chamador. Também pode ser passado zero ou mais *argumentos* que podem ser usados na execução do corpo. Veja também *parâmetro*, *métodos* e a seção function.

__future__ A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression 11/4 currently evaluates to 2. If the module in which it is executed had enabled *true division* by executing:

```
from __future__ import division
```

the expression 11/4 would evaluate to 2.75. By importing the __future__ module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection (coletor de lixo) The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

gerador A function which returns an iterator. It looks like a normal function except that it contains yield statements for producing a series of values usable in a for-loop or that can be retrieved one at a time with the next() function. Each yield temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression An expression that returns an iterator. It looks like a normal expression followed by a for expression defining a loop variable, range, and an optional if expression. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

GIL Veja *global interpreter lock*.

global interpreter lock (bloqueio global do interpretador) O mecanismo utilizado pelo interpretador *CPython* para garantir que apenas uma thread execute o *bytecode* Python por vez. Isto simplifica a implementação do CPython ao fazer com que o modelo de objetos (incluindo tipos internos críticos como o dict) ganhem segurança implícita contra acesso concorrente. Travar todo o interpretador facilita que o interpretador em si seja multitarefa, às custas de muito do paralelismo já provido por máquinas multiprocessador.

No entanto, alguns módulos de extensão, tanto da biblioteca padrão quanto de terceiros, são desenvolvidos de forma a liberar o GIL ao realizar tarefas computacionalmente muito intensas, como compactação ou cálculos de hash. Além disso, o GIL é sempre liberado nas operações de E/S.

No passado, esforços para criar um interpretador que lidasse plenamente com threads (travando dados compartilhados numa granularidade bem mais fina) não foram bem sucedidos devido a queda no desempenho ao serem executados em processadores de apenas um núcleo. Acredita-se que superar essa questão de desempenho acabaria tornando a implementação muito mais complicada e bem mais difícil de manter.

hasheável An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` or `__cmp__()` method). Hashable objects which compare equal must have the same hash value.

A hashabilidade faz com que um objeto possa ser usado como chave de um dicionário e como membro de um conjunto, pois estas estruturas de dados utilizam os valores de hash internamente.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal (except with themselves), and their hash value is derived from their `id()`.

IDLE Um ambiente de desenvolvimento integrado para Python. IDLE é um editor básico e um ambiente interpretador que vem junto com a distribuição padrão do Python.

imutável Um objeto que possui um valor fixo. Objetos imutáveis incluem números, strings e tuplas. Estes objetos não podem ser alterados. Um novo objeto deve ser criado se um valor diferente tiver de ser armazenado. Objetos imutáveis têm um papel importante em lugares onde um valor constante de hash seja necessário, como por exemplo uma chave em um dicionário.

integer division Mathematical division discarding any remainder. For example, the expression `11/4` currently evaluates to `2` in contrast to the `2.75` returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a `float`), the result will be coerced (see *coercion*) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also *__future__*.

importing (importando) O processo pelo qual o código Python em um módulo é disponibilizado para o código Python em outro módulo.

importer Um objeto que localiza e carrega um módulo; Tanto um *finder* e o objeto *loader*.

interactive Python tem um interpretador interativo, o que significa que você pode digitar comandos e expressões no prompt do interpretador, executá-los imediatamente e ver seus resultados. Apenas execute `python` sem argumentos (possivelmente selecionando-o a partir do menu de aplicações de seu sistema operacional). O interpretador interativo é uma maneira poderosa de testar novas ideias ou aprender mais sobre módulos e pacotes (lembre-se do comando `help(x)`).

interpreted Python é uma linguagem interpretada, em oposição àquelas que são compiladas, embora esta distinção possa ser nebulosa devido à presença do compilador de bytecode. Isto significa que os arquivos-fontes podem ser executados diretamente sem necessidade explícita de se criar um arquivo executável. Linguagens interpretadas normalmente têm um ciclo de desenvolvimento/depuração mais curto que as linguagens compiladas, apesar de seus programas geralmente serem executados mais lentamente. Veja também *interativo*.

iterável An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, …). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterador An object representing a stream of data. Repeated calls to the iterator's `next()` method return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

Mais informações podem ser encontradas em typeiter.

key function (função chave) Uma função chave ou função colação é algo que retorna um valor utilizado para ordenação ou classificação. Por exemplo, `locale.strxfrm()` é usada para produzir uma chave de ordenação que leva o locale em consideração para fins de ordenação.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example. the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, the `operator` module provides three key function constructors: `attrgetter()`, `itemgetter()`, and `methodcaller()`. See the Sorting HOW TO for examples of how to create and use key functions.

argumento nomeado Veja o *argument*.

lambda Uma função de linha anônima consistindo de uma única *expression*, que é avaliada quando a função é chamada. A sintaxe para criar uma função lambda é `lambda [parameters]: expression`

LBYL Iniciais da expressão em inglês "look before you leap", que significa algo como "olhe antes de pisar". Este estilo de codificação testa as pré-condições explicitamente antes de fazer chamadas ou buscas. Este estilo contrasta com a abordagem *EAFP* e é caracterizada pela presença de muitos comandos `if`.

Em um ambiente multithread, a abordagem LBYL pode arriscar a introdução de uma condição de corrida entre "o olhar" e "o pisar". Por exemplo, o código `if key in mapping: return mapping[key]` pode falhar se outra thread remover *key* do *mapping* após o teste, mas antes da olhada. Esse problema pode ser resolvido com bloqueios ou usando a abordagem EAFP.

lista Uma *sequence* embutida no Python. Apesar do seu nome, é mais próximo de um vetor em outras linguagens do que uma lista encadeada, como o acesso aos elementos é da ordem O(1).

list comprehension A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

loader An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See **PEP 302** for details.

método mágico Um sinônimo informal para um *special method*.

mapeamento A container object that supports arbitrary key lookups and implements the methods specified in the `Mapping` or `MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

metaclass A classe de uma classe. Definições de classe criam um nome de classe, um dicionário de classe e uma lista de classes base. A metaclasse é responsável por receber estes três argumentos e criar a classe. A maioria das linguagens de programação orientadas a objetos provê uma implementação default. O que torna o Python especial é o fato de ser possível criar metaclasses personalizadas. A maioria dos usuários nunca vai precisar deste recurso, mas quando houver necessidade, metaclasses possibilitam soluções poderosas e elegantes. Metaclasses têm sido utilizadas para gerar registros de acesso a atributos, para incluir proteção contra acesso concorrente, rastrear a criação de objetos, implementar singletons, dentre muitas outras tarefas.

More information can be found in metaclasses.

method (método) Uma função que é definida dentro do corpo de uma classe. Se chamada como um atributo de uma instância daquela classe, o método receberá a instância do objeto como seu primeiro *argumento* (que comumente é chamado de `self`). Veja *função* e *nested scope*.

method resolution order (ordem de resolução de método) Ordem de resolução de métodos é a ordem em que os membros de uma classe base são buscados durante a pesquisa. Veja A ordem de resolução de métodos do Python 2.3.

módulo Um objeto que serve como uma unidade organizacional de código Python. Os módulos têm um namespace contendo objetos Python arbitrários. Os módulos são carregados pelo Python através do processo de *importação*.

Veja também *pacote*.

MRO Veja *method resolution order*.

mutável Objeto mutável é aquele que pode modificar seus valor mas manter seu `id()`. Veja também *immutable*.

tupla nomeada Any tuple-like class whose indexable elements are also accessible using named attributes (for example, `time.localtime()` returns a tuple-like object where the *year* is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as `time.struct_time`, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function `collections.namedtuple()`. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

namespace The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

nested scope (escopo aninhado) The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

new-style class (novo estilo de classes) Any class which inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, and `__getattribute__()`.

More information can be found in newstyle.

object (objeto) Qualquer dado que tenha estado (atributos ou valores) e comportamento definidos (métodos). Também a última classe base de qualquer *new-style class*.

pacote Um *module* Python é capaz de conter submódulos ou recursivamente, sub-pacotes. Tecnicamente, um pacote é um módulo Python com um atributo `__path__`.

parâmetro A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are four types of parameters:

- *posicional-ou-nomeado*: especifica um argumento que pode ser tanto *posicional* quanto *nomeado*. Esse é o tipo padrão de parâmetro, por exemplo *foo* e *bar* a seguir:

```
def func(foo, bar=None): ...
```

- *positional-only*: specifies an argument that can be supplied only by position. Python has no syntax for defining positional-only parameters. However, some built-in functions have positional-only parameters (e.g. `abs()`).

- *var-posicional*: especifica quem uma sequência arbitrária de argumentos posicionais pode ser fornecida (em adição a qualquer argumento posicional já aceito por outros parâmetros). Tal parâmetro pode ser definido colocando um `*` antes do nome, por exemplo *args* a seguir:

```
def func(*args, **kwargs): ...
```

- *var-nomeado*: especifica que, arbitrariamente, muitos argumentos nomeados podem ser fornecidos (em adição a qualquer argumento nomeado já aceito por outros parâmetros). Tal parâmetro pode definido colocando-se `**` antes do nome, por exemplo *kwargs* no exemplo acima.

Parâmetros podem especificar tanto argumentos opcionais quanto obrigatórios, assim como valores padrões para alguns argumentos opcionais.

See also the *argument* glossary entry, the FAQ question on the difference between arguments and parameters, and the function section.

PEP Proposta de melhoria do Python. Uma PEP é um documento de design que fornece informação para a comunidade Python, ou descreve uma nova funcionalidade para o Python ou seus predecessores ou ambientes. PEPs devem prover uma especificação técnica concisa e um racional para funcionalidades propostas.

PEPs tem a intenção de ser os mecanismos primários para propor novas funcionalidades significativas, para coletar opiniões da comunidade sobre um problema, e para documentar as decisões de design que foram adicionadas ao Python. O autor da PEP é responsável por construir um consenso dentro da comunidade e documentar opiniões dissidentes.

Veja **PEP 1**.

positional argument (argumento posicional) Veja o *argument*.

Python 3000 Apelido para a versão do Python 3.x linha de lançamento (cunhado há muito tempo, quando o lançamento da versão 3 era algo em um futuro muito distante.) Esse termo possui a seguinte abreviação: "Py3k".

Pythonic Uma ideia ou um pedaço de código que segue de perto os idiomas mais comuns da linguagem Python, ao invés de implementar códigos usando conceitos comuns a outros idiomas. Por exemplo, um idioma comum em Python é fazer um loop sobre todos os elementos de uma iterável usando a instrução `for`. Muitas outras linguagens não têm esse tipo de construção, então as pessoas que não estão familiarizadas com o Python usam um contador numérico:

```
for i in range(len(food)):
    print food[i]
```

Ao contrário do método limpo, ou então, Pythônico:

```
for piece in food:
    print piece
```

reference count O número de referências para um objeto. Quando a contagem de referências de um objeto atinge zero, ele é desalocado. Contagem de referências geralmente não é visível no código Python, mas é um elemento chave da implementação *CPython*. O módulo `sys` define a função `getrefcount()` que programadores podem chamar para retornar a contagem de referências para um objeto em particular.

__slots__ A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequência An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `len()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

fatia An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally (or in older versions, `__getslice__()` and `__setslice__()`).

método especial Um método que é chamado implicitamente pelo Python para executar uma certa operação em um tipo, como uma adição por exemplo. Tais métodos tem nomes iniciando e terminando com dois underscores. Métodos especiais estão documentados em specialnames.

instrução Uma instrução é parte de uma suíte (um "bloco" de código). Uma instrução é ou uma *expression* ou uma de várias construções com uma palavra-chave, tal como `if`, `while` ou `for`.

struct sequence A tuple with named elements. Struct sequences expose an interface similiar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

aspas triplas Uma string que está definida com três ocorrências de aspas duplas (") ou apóstrofes ('). Enquanto elas não fornecem nenhuma funcionalidade não disponível com strings de aspas simples, elas são úteis para inúmeras razões. Elas permitem que você inclua aspas simples e duplas não encerradas dentro de uma string, e elas podem utilizar múltiplas linhas sem o uso de caractere de continuação, fazendo-as especialmente úteis quando escrevemos documentação em docstrings.

type O tipo de um objeto Python determina qual tipo de objeto ele é; cada objeto tem um tipo. Um tipo de objeto é acessível pelo atributo `__class__` ou pode ser recuperado com `type(obj)`.

Novas linhas universais A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See **PEP 278** and **PEP 3116**, as well as `str.splitlines()` for an additional use.

ambiente virtual Um ambiente de execução isolado que permite usuários Python e aplicações instalarem e atualizarem pacotes Python sem interferir no comportamento de outras aplicações Python em execução no mesmo sistema.

máquina virtual Um computador definido inteiramente em software. A máquina virtual de Python executa o *bytecode* emitido pelo compilador de bytecode.

Zen of Python Lista de princípios de projeto e filosofias do Python que são úteis para a compreensão e uso da linguagem. A lista é exibida quando se digita "`import this`" no console interativo.

APÊNDICE B

Sobre esses documentos

Esses documentos são gerados a partir de reStructuredText pelo Sphinx, um processador de documentos especificamente escrito para documentação do Python.

O desenvolvimento da documentação e de suas ferramentas é um esforço totalmente voluntário, como o Python em si. Se você quer contribuir, por favor dê uma olhada na página reporting-bugs para informações sobre como fazer. Novos voluntários são sempre bem vindos!

Agradecimentos especiais para:

- Fred L. Drake, Jr., o criador do primeiro conjunto de ferramentas para documentar o Python e escritor de boa parte do conteúdo;
- O projeto Docutils para criar reStructuredText e o pacote Docutils;
- Fredrik Lundh por seu projeto Referência Alternativa para Python do qual Sphinx teve muitas ideias boas.

B.1 Contribuidores da Documentação do Python

Muitas pessoas tem contribuído para a linguagem Python, sua biblioteca padrão e sua documentação. Veja Misc/ACKS na distribuição do código do Python para ver uma lista parcial de contribuidores.

Tudo isso só foi possível com o esforço e a contribuição da comunidade Python, por isso temos essa maravilhosa documentação – Obrigado a todos!

APÊNDICE C

História e Licença

C.1 História do software

O Python foi criado no início dos anos 1990 por Guido van Rossum na Stichting Mathematisch Centrum (CWI, veja https://www.cwi.nl/) na Holanda como um sucessor de uma linguagem chamada ABC. Guido continua a ser o principal autor de Python, embora inclua muitas contribuições de outros.

Em 1995, Guido continuou seu trabalho em Python na Corporação para Iniciativas Nacionais de Pesquisa (CNRI, veja https://www.cnri.reston.va.us/) em Reston, Virgínia, onde lançou várias versões do software.

Em maio de 2000, Guido e a equipe principal de desenvolvimento do Python mudaram-se para o BeOpen.com para formar a equipe BeOpen PythonLabs. Em outubro do mesmo ano, a equipe da PythonLabs mudou para a Digital Creations (agora Zope Corporation; veja https://www.zope.org/). Em 2001, formou-se a Python Software Foundation (PSF, ver https://www.python.org/psf/), uma organização sem fins lucrativos criada especificamente para possuir propriedade intelectual relacionada a Python. A Zope Corporation é um membro patrocinador do PSF.

Todas as versões do Python são de código aberto (consulte https://opensource.org/ para a definição de código aberto). Historicamente, a maioria, mas não todas, versões do Python também são compatíveis com GPL; a tabela abaixo resume os vários lançamentos.

Release	Derivado de	Ano	Proprietário	GPL compatível?
0.9.0 a 1.2	n/a	1991-1995	CWI	sim
1.3 a 1.5.2	1.2	1995-1999	CNRI	sim
1.6	1.5.2	2000	CNRI	não
2.0	1.6	2000	BeOpen.com	não
1.6.1	1.6	2001	CNRI	não
2.1	2.0+1.6.1	2001	PSF	não
2.0.1	2.0+1.6.1	2001	PSF	sim
2.1.1	2.1+2.0.1	2001	PSF	sim
2.1.2	2.1.1	2002	PSF	sim
2.1.3	2.1.2	2002	PSF	sim
2.2 e acima	2.1.1	2001-agora	PSF	sim

Nota: Compatível com GPL não significa que estamos distribuindo Python sob a GPL. Todas as licenças do Python, ao contrário da GPL, permitem distribuir uma versão modificada sem fazer alterações em código aberto. As licenças compatíveis com GPL possibilitam combinar o Python com outro software lançado sob a GPL; os outros não.

Graças aos muitos voluntários externos que trabalharam sob a direção de Guido para tornar esses lançamentos possíveis.

C.2 Termos e condições para acessar ou usar Python

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 2.7.18

```
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 2.7.18, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 ACORDO DE LICENÇA DA BEOPEN.COM PARA PYTHON 2.0

CONTRATO DE LICENÇA DE FONTE ABERTA DO BEOPEN PYTHON VERSÃO 1

```
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
```

(continua na próxima página)

(continuação da página anterior)

```
   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 CONTRATO DE LICENÇA DA CNRI PARA O PYTHON 1.6.1

```
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
```

(continua na próxima página)

(continuação da página anterior)

```
   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 ACORDO DE LICENÇA DA CWI PARA PYTHON 0.9.0 A 1.2

```
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.3 Licenças e Reconhecimentos para Software Incorporado

Esta seção é uma lista incompleta, mas crescente, de licenças e confirmações para softwares de terceiros incorporados na distribuição do Python.

C.3.1 Mersenne Twister

O módulo `_random` inclui código baseado em um download de http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. A seguir estão os comentários literais do código original:

```
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).

```

(continua na próxima página)

(continuação da página anterior)

C.3.2 Sockets

O módulo `socket` usa as funções `getaddrinfo()` e `getnameinfo()`, que são codificadas em arquivos de origem separados do Projeto WIDE, http://www.wide.ad.jp/.

(continua na próxima página)

(continuação da página anterior)

C.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

C.3.4 MD5 message digest algorithm

The source code for the `md5` module contains the following notice:

```
Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

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.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
      http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A.  It does not include
any code or documentation that is identified in the RFC as being
copyrighted.

The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>.  Other authors are noted in the change history
that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed
      references to Ghostscript; clarified derivation from RFC 1321;
      now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
      added conditionalization for C++ compilation from Martin
      Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
```

C.3.5 Serviços de soquete assíncrono

Os módulos `asynchat` e `asyncore` contêm o seguinte aviso:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.6 Gerenciamento de cookies

The `Cookie` module contains the following notice:

```
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.3.7 Rastreamento de execução

O módulo `trace` contém o seguinte aviso:

```
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.3.8 Funções UUencode e UUdecode

O módulo `uu` contém o seguinte aviso:

```
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
```

(continua na próxima página)

(continuação da página anterior)

```
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.9 Chamadas de Procedimento Remoto XML

The `xmlrpclib` module contains the following notice:

```
    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.3.10 test_epoll

The `test_epoll` contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(continua na próxima página)

(continuação da página anterior)

```
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.3.11 Selecione o kqueue

The `select` and contains the following notice for the kqueue interface:

```
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.3.12 strtod e dtoa

O arquivo `Python/dtoa.c`, que fornece as funções C dtoa e strtod para conversão de duplas de C para e de strings, é derivado do arquivo com o mesmo nome de David M. Gay, atualmente disponível em http://www.netlib.org/fp/. O arquivo original, conforme recuperado em 16 de março de 2009, contém os seguintes avisos de direitos autorais e de licenciamento:

```
/****************************************************************
 *
 * 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.
 *
```

(continua na próxima página)

(continuação da página anterior)

```
 * 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.3.13 OpenSSL

Os módulos `hashlib`, `posix`, `ssl`, `crypt` usam a biblioteca OpenSSL para desempenho adicional se forem disponibilizados pelo sistema operacional. Além disso, os instaladores do Windows e do Mac OS X para Python podem incluir uma cópia das bibliotecas do OpenSSL, portanto incluímos uma cópia da licença do OpenSSL aqui:

```
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
```

(continua na próxima página)

(continuação da página anterior)

```
 *    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
```

(continua na próxima página)

(continuação da página anterior)

```
 *    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.3.14 expat

A extensão `pyexpat` é construída usando uma cópia incluída das fontes de expatriadas, a menos que a compilação esteja configurada `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.15 libffi

A extensão `_ctypes` é construída usando uma cópia incluída das fontes libffi, a menos que a compilação esteja configurada `--with-system-libffi`:

```
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.3.16 zlib

A extensão `zlib` é construída usando uma cópia incluída das fontes zlib se a versão do zlib encontrada no sistema for muito antiga para ser usada na construção:

```
Copyright (C) 1995-2010 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
```

APÊNDICE D

Copyright

Python e essa documentação é:

Veja: *História e Licença* para informações completas de licença e permissões.

Índice

Não alfabético

A

B

C

P

R

S