The Python/C API

Release 3.13.2

Guido van Rossum and the Python development team

março 25, 2025

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

A Interface de Programação de Aplicações (API) para Python fornece aos programadores C e C++ acesso ao interpretador Python em uma variedade de níveis. A API pode ser usada igualmente em C++, mas, para abreviar, geralmente é chamada de API Python/C. Existem dois motivos fundamentalmente diferentes para usar a API Python/C. A primeira razão é escrever *módulos de extensão* para propósitos específicos; esses são módulos C que estendem o interpretador Python. Este é provavelmente o uso mais comum. O segundo motivo é usar Python como um componente em uma aplicação maior; esta técnica é geralmente referida como *incorporação* Python em uma aplicação.

Escrever um módulo de extensão é um processo relativamente bem compreendido, no qual uma abordagem de "livro de receitas" funciona bem. Existem várias ferramentas que automatizam o processo até certo ponto. Embora as pessoas tenham incorporado o Python em outras aplicações desde sua existência inicial, o processo de incorporação do Python é menos direto do que escrever uma extensão.

Muitas funções da API são úteis independentemente de você estar incorporando ou estendendo o Python; além disso, a maioria das aplicações que incorporam Python também precisará fornecer uma extensão customizada, portanto, é provavelmente uma boa ideia se familiarizar com a escrita de uma extensão antes de tentar incorporar Python em uma aplicação real.

1.1 Padrões de codificação

Se você estiver escrevendo código C para inclusão no CPython, **deve** seguir as diretrizes e padrões definidos na **PEP 7**. Essas diretrizes se aplicam independentemente da versão do Python com a qual você está contribuindo. Seguir essas convenções não é necessário para seus próprios módulos de extensão de terceiros, a menos que você eventualmente espere contribuí-los para o Python.

1.2 Arquivos de inclusão

Todas as definições de função, tipo e macro necessárias para usar a API Python/C estão incluídas em seu código pela seguinte linha:

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

Isso implica a inclusão dos seguintes cabeçalhos padrão: `<stdio.h>`, `<string.h>`, `<errno.h>`, `<limits.h>`, `<assert.h>` e `<stdlib.h>` (se disponível).

> **Nota**
>
> Uma vez que Python pode definir algumas definições de pré-processador que afetam os cabeçalhos padrão em alguns sistemas, você *deve* incluir `Python.h` antes de quaisquer cabeçalhos padrão serem incluídos.
>
> É recomendável sempre definir `PY_SSIZE_T_CLEAN` antes de incluir `Python.h`. Veja *Análise de argumentos e construção de valores* para uma descrição desta macro.

Todos os nomes visíveis ao usuário definidos por Python.h (exceto aqueles definidos pelos cabeçalhos padrão incluídos) têm um dos prefixos `Py` ou `_Py`. Nomes começando com `_Py` são para uso interno pela implementação Python e não devem ser usados por escritores de extensão. Os nomes dos membros da estrutura não têm um prefixo reservado.

> **Nota**
>
> O código do usuário nunca deve definir nomes que começam com `Py` ou `_Py`. Isso confunde o leitor e coloca em risco a portabilidade do código do usuário para versões futuras do Python, que podem definir nomes adicionais começando com um desses prefixos.

Os arquivos de cabeçalho são normalmente instalados com Python. No Unix, eles estão localizados nos diretórios *prefix*`/include/pythonversion/` e *exec_prefix*`/include/pythonversion/`, onde `prefix` e `exec_prefix` são definidos pelos parâmetros correspondentes ao script **`configure`** e *version* do Python é `'%d.%d' % sys.version_info[:2]`. No Windows, os cabeçalhos são instalados em *prefix*`/include`, onde `prefix` é o diretório de instalação especificado para o instalador.

Para incluir os cabeçalhos, coloque os dois diretórios (se diferentes) no caminho de pesquisa do compilador para as inclusões. *Não* coloque os diretórios pais no caminho de busca e então use `#include <pythonX.Y/Python.h>`; isto irá quebrar em compilações multiplataforma, uma vez que os cabeçalhos independentes da plataforma em `prefix` incluem os cabeçalhos específicos da plataforma de `exec_prefix`.

Os usuários de C++ devem notar que embora a API seja definida inteiramente usando C, os arquivos de cabeçalho declaram apropriadamente os pontos de entrada como `extern "C"`. Como resultado, não há necessidade de fazer nada especial para usar a API do C++.

1.3 Macros úteis

Diversas macros úteis são definidas nos arquivos de cabeçalho do Python. Muitas são definidas mais próximas de onde são úteis (por exemplo, *Py_RETURN_NONE*). Outras de utilidade mais geral são definidas aqui. Esta não é necessariamente uma lista completa.

PyMODINIT_FUNC

Declara uma função de inicialização do módulo de extensão `PyInit`. O tipo de retorno da função é *PyObject*`*`. A macro declara quaisquer declarações de ligação especial necessárias pela plataforma e, para C++, declara a função como `extern "C"`.

A função de inicialização deve ser nomeada `PyInit_`*name*, onde *name* é o nome do módulo, e deve ser o único item não-`static` definido no arquivo do módulo. Exemplo:

```
static struct PyModuleDef spam_module = {
    PyModuleDef_HEAD_INIT,
    .m_name = "spam",
    ...
};

PyMODINIT_FUNC
PyInit_spam(void)
{
```

(continua na próxima página)

(continuação da página anterior)

```
    return PyModule_Create(&spam_module);
}
```

Py_ABS(x)

Retorna o valor absoluto de `x`.

Adicionado na versão 3.3.

Py_ALWAYS_INLINE

Pede ao compilador para sempre embutir uma função em linha estática. O compilador pode ignorá-lo e decide não inserir a função.

Ele pode ser usado para inserir funções em linha estáticas críticas de desempenho ao compilar Python no modo de depuração com função de inserir em linha desabilitada. Por exemplo, o MSC desabilita a função de inserir em linha ao compilar no modo de depuração.

Marcar cegamente uma função em linha estática com Py_ALWAYS_INLINE pode resultar em desempenhos piores (devido ao aumento do tamanho do código, por exemplo). O compilador geralmente é mais inteligente que o desenvolvedor para a análise de custo/benefício.

Se o Python tiver sido compilado em modo de depuração (se a macro *Py_DEBUG* estiver definida), a macro *Py_ALWAYS_INLINE* não fará nada.

Deve ser especificado antes do tipo de retorno da função. Uso:

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

Adicionado na versão 3.11.

Py_CHARMASK(c)

O argumento deve ser um caractere ou um número inteiro no intervalo [-128, 127] ou [0, 255]. Esta macro retorna `c` convertido em um `unsigned char`.

Py_DEPRECATED(version)

Use isso para declarações descontinuadas. A macro deve ser colocada antes do nome do símbolo.

Exemplo:

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

Alterado na versão 3.8: Suporte a MSVC foi adicionado.

Py_GETENV(s)

Como `getenv(s)`, mas retorna `NULL` se a opção `-E` foi passada na linha de comando (veja *PyConfig.use_environment*).

Py_MAX(x, y)

Retorna o valor máximo entre `x` e `y`.

Adicionado na versão 3.3.

Py_MEMBER_SIZE(type, member)

Retorna o tamanho do `member` de uma estrutura (`type`) em bytes.

Adicionado na versão 3.6.

Py_MIN(x, y)

Retorna o valor mínimo entre `x` e `y`.

Adicionado na versão 3.3.

Py_NO_INLINE

Desabilita a inserção em linha em uma função. Por exemplo, isso reduz o consumo da pilha C: útil em compilações LTO+PGO que faz uso intenso de inserção em linha de código (veja bpo-33720).

Uso:

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

Adicionado na versão 3.11.

Py_STRINGIFY(x)

Converte `x` para uma string C. Por exemplo, `Py_STRINGIFY(123)` retorna `"123"`.

Adicionado na versão 3.4.

Py_UNREACHABLE()

Use isso quando você tiver um caminho de código que não pode ser alcançado por design. Por exemplo, na cláusula `default:` em uma instrução `switch` para a qual todos os valores possíveis são incluídos nas instruções `case`. Use isto em lugares onde você pode ficar tentado a colocar uma chamada `assert(0)` ou `abort()`.

No modo de lançamento, a macro ajuda o compilador a otimizar o código e evita um aviso sobre código inacessível. Por exemplo, a macro é implementada com `__builtin_unreachable()` no GCC em modo de lançamento.

Um uso para `Py_UNREACHABLE()` é seguir uma chamada de uma função que nunca retorna, mas que não é declarada com `_Py_NO_RETURN`.

Se um caminho de código for um código muito improvável, mas puder ser alcançado em casos excepcionais, esta macro não deve ser usada. Por exemplo, sob condição de pouca memória ou se uma chamada de sistema retornar um valor fora do intervalo esperado. Nesse caso, é melhor relatar o erro ao chamador. Se o erro não puder ser reportado ao chamador, *Py_FatalError()* pode ser usada.

Adicionado na versão 3.7.

Py_UNUSED(arg)

Use isso para argumentos não usados em uma definição de função para silenciar avisos do compilador. Exemplo: `int func(int a, int Py_UNUSED(b)) { return a; }`.

Adicionado na versão 3.4.

PyDoc_STRVAR(name, str)

Cria uma variável com o nome `name` que pode ser usada em docstrings. Se o Python for compilado sem docstrings, o valor estará vazio.

Use *PyDoc_STRVAR* para docstrings para ter suporte à compilação do Python sem docstrings, conforme especificado em **PEP 7**.

Exemplo:

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

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

PyDoc_STR(str)

Cria uma docstring para a string de entrada fornecida ou uma string vazia se docstrings estiverem desabilitadas.

Use *PyDoc_STR* ao especificar docstrings para ter suporte à compilação do Python sem docstrings, conforme especificado em **PEP 7**.

Exemplo:

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

1.4 Objetos, tipos e contagens de referências

A maioria das funções da API Python/C tem um ou mais argumentos, bem como um valor de retorno do tipo *PyObject**. Este tipo é um ponteiro para um tipo de dados opaco que representa um objeto Python arbitrário. Como todos os tipos de objeto Python são tratados da mesma maneira pela linguagem Python na maioria das situações (por exemplo, atribuições, regras de escopo e passagem de argumento), é adequado que eles sejam representados por um único tipo C. Quase todos os objetos Python vivem na pilha: você nunca declara uma variável automática ou estática do tipo *PyObject*, variáveis de apenas ponteiro do tipo *PyObject** podem ser declaradas. A única exceção são os objetos de tipo; uma vez que estes nunca devem ser desalocados, eles são normalmente objetos estáticos *PyTypeObject*.

Todos os objetos Python (mesmo inteiros Python) têm um *tipo* e uma *contagem de referências*. O tipo de um objeto determina que tipo de objeto ele é (por exemplo, um número inteiro, uma lista ou uma função definida pelo usuário; existem muitos mais, conforme explicado em types). Para cada um dos tipos conhecidos, há uma macro para verificar se um objeto é desse tipo; por exemplo, `PyList_Check(a)` é verdadeiro se (e somente se) o objeto apontado por *a* for uma lista Python.

1.4.1 Contagens de referências

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

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

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

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

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

Detalhes da contagem de referências

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

Por outro lado, quando uma função de chamada passa uma referência a um objeto, há duas possibilidades: a função *rouba* uma referência ao objeto, ou não. *Roubar uma referência* significa que quando você passa uma referência para uma função, essa função presume que agora ela possui essa referência e você não é mais responsável por ela.

Poucas funções roubam referências; as duas exceções notáveis são *PyList_SetItem()* e *PyTuple_SetItem()*, que roubam uma referência para o item (mas não para a tupla ou lista na qual o item é colocado!). Essas funções foram projetadas para roubar uma referência devido a um idioma comum para preencher uma tupla ou lista com objetos recém-criados; por exemplo, o código para criar a tupla `(1, 2, "three")` pode ser parecido com isto (esquecendo o tratamento de erros por enquanto; uma maneira melhor de codificar isso é mostrada abaixo):

```
PyObject *t;

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

Aqui, *PyLong_FromLong()* retorna uma nova referência que é imediatamente roubada por *PyTuple_SetItem()*. Quando você quiser continuar usando um objeto, embora a referência a ele seja roubada, use *Py_INCREF()* para obter outra referência antes de chamar a função de roubo de referência.

A propósito, *PyTuple_SetItem()* é a *única* maneira de definir itens de tupla; *PySequence_SetItem()* e *PyObject_SetItem()* se recusam a fazer isso, pois tuplas são um tipo de dados imutável. Você só deve usar *PyTuple_SetItem()* para tuplas que você mesmo está criando.

O código equivalente para preencher uma lista pode ser escrita usando *PyList_New()* e *PyList_SetItem()*.

No entanto, na prática, você raramente usará essas maneiras de criar e preencher uma tupla ou lista. Existe uma função genérica, *Py_BuildValue()*, que pode criar objetos mais comuns a partir de valores C, dirigidos por uma *string de formato*. Por exemplo, os dois blocos de código acima podem ser substituídos pelos seguintes (que também cuidam da verificação de erros):

```
PyObject *tuple, *list;

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

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

```
int
set_all(PyObject *target, PyObject *item)
{
```

(continua na próxima página)

(continuação da página anterior)

```
    Py_ssize_t i, n;

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

A situação é ligeiramente diferente para os valores de retorno da função. Embora passar uma referência para a maioria das funções não altere suas responsabilidades de propriedade para aquela referência, muitas funções que retornam uma referência a um objeto fornecem a propriedade da referência. O motivo é simples: em muitos casos, o objeto retornado é criado instantaneamente e a referência que você obtém é a única referência ao objeto. Portanto, as funções genéricas que retornam referências a objetos, como *PyObject_GetItem()* e *PySequence_GetItem()*, sempre retornam uma nova referência (o chamador torna-se o dono da referência).

É importante perceber que se você possui uma referência retornada por uma função depende de qual função você chama apenas — *a plumagem* (o tipo do objeto passado como um argumento para a função) *não entra nela!* Assim, se você extrair um item de uma lista usando *PyList_GetItem()*, você não possui a referência — mas se obtiver o mesmo item da mesma lista usando *PySequence_GetItem()* (que leva exatamente os mesmos argumentos), você possui uma referência ao objeto retornado.

Aqui está um exemplo de como você poderia escrever uma função que calcula a soma dos itens em uma lista de inteiros; uma vez usando *PyList_GetItem()*, e uma vez usando *PySequence_GetItem()*.

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

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

```
long
```

(continua na próxima página)

(continuação da página anterior)

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

1.4.2 Tipos

Existem alguns outros tipos de dados que desempenham um papel significativo na API Python/C; a maioria são tipos C simples, como `int`, `long`, `double` e `char*`. Alguns tipos de estrutura são usados para descrever tabelas estáticas usadas para listar as funções exportadas por um módulo ou os atributos de dados de um novo tipo de objeto, e outro é usado para descrever o valor de um número complexo. Eles serão discutidos junto com as funções que os utilizam.

type **`Py_ssize_t`**

Parte da ABI Estável. Um tipo integral assinado tal que `sizeof(Py_ssize_t) == sizeof(size_t)`. C99 não define tal coisa diretamente (size_t é um tipo integral não assinado). Veja **PEP 353** para mais detalhes. `PY_SSIZE_T_MAX` é o maior valor positivo do tipo *`Py_ssize_t`*.

1.5 Exceções

O programador Python só precisa lidar com exceções se o tratamento de erros específico for necessário; as exceções não tratadas são propagadas automaticamente para o chamador, depois para o chamador e assim por diante, até chegarem ao interpretador de nível superior, onde são relatadas ao usuário acompanhadas por um traceback (situação da pilha de execução).

Para programadores C, entretanto, a verificação de erros sempre deve ser explícita. Todas as funções na API Python/C podem levantar exceções, a menos que uma declaração explícita seja feita de outra forma na documentação de uma função. Em geral, quando uma função encontra um erro, ela define uma exceção, descarta todas as referências de objeto de sua propriedade e retorna um indicador de erro. Se não for documentado de outra forma, este indicador é `NULL` ou `-1`, dependendo do tipo de retorno da função. Algumas funções retornam um resultado booleano verdadeiro/falso, com falso indicando um erro. Muito poucas funções não retornam nenhum indicador de erro explícito ou têm um valor de retorno ambíguo e requerem teste explícito para erros com *`PyErr_Occurred()`*. Essas exceções são sempre documentadas explicitamente.

O estado de exceção é mantido no armazenamento por thread (isso é equivalente a usar o armazenamento global em

uma aplicação sem thread). Uma thread pode estar em um de dois estados: ocorreu uma exceção ou não. A função *PyErr_Occurred()* pode ser usada para verificar isso: ela retorna uma referência emprestada ao objeto do tipo de exceção quando uma exceção ocorreu, e `NULL` caso contrário. Existem várias funções para definir o estado de exceção: *PyErr_SetString()* é a função mais comum (embora não a mais geral) para definir o estado de exceção, e *PyErr_Clear()* limpa o estado da exceção.

O estado de exceção completo consiste em três objetos (todos os quais podem ser `NULL`): o tipo de exceção, o valor de exceção correspondente e o traceback. Eles têm os mesmos significados que o resultado do Python de `sys.exc_info()`; no entanto, eles não são os mesmos: os objetos Python representam a última exceção sendo tratada por uma instrução Python `try` … `except`, enquanto o estado de exceção de nível C só existe enquanto uma exceção está sendo transmitido entre funções C até atingir o loop principal do interpretador de bytecode Python, que se encarrega de transferi-lo para `sys.exc_info()` e amigos.

Observe que a partir do Python 1.5, a maneira preferida e segura para thread para acessar o estado de exceção do código Python é chamar a função `sys.exc_info()`, que retorna o estado de exceção por thread para o código Python. Além disso, a semântica de ambas as maneiras de acessar o estado de exceção mudou, de modo que uma função que captura uma exceção salvará e restaurará o estado de exceção de seu segmento de modo a preservar o estado de exceção de seu chamador. Isso evita bugs comuns no código de tratamento de exceções causados por uma função aparentemente inocente sobrescrevendo a exceção sendo tratada; também reduz a extensão da vida útil frequentemente indesejada para objetos que são referenciados pelos quadros de pilha no traceback.

Como princípio geral, uma função que chama outra função para realizar alguma tarefa deve verificar se a função chamada levantou uma exceção e, em caso afirmativo, passar o estado da exceção para seu chamador. Ele deve descartar todas as referências de objeto que possui e retornar um indicador de erro, mas *não* deve definir outra exceção — que sobrescreveria a exceção que acabou de ser gerada e perderia informações importantes sobre a causa exata do erro.

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
```

Aqui está o código C correspondente, em toda sua glória:

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

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

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

(continua na próxima página)

(continuação da página anterior)

```
    const_one = PyLong_FromLong(1L);
    if (const_one == NULL)
        goto error;

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

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

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

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

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

Este exemplo representa um uso endossado da instrução `goto` em C! Ele ilustra o uso de *PyErr_ExceptionMatches()* e *PyErr_Clear()* para lidar com exceções específicas, e o uso de *Py_XDECREF()* para descartar referências de propriedade que podem ser `NULL` (observe o `'X'` no nome; *Py_DECREF()* travaria quando confrontado com uma referência `NULL`). É importante que as variáveis usadas para manter as referências de propriedade sejam inicializadas com `NULL` para que isso funcione; da mesma forma, o valor de retorno proposto é inicializado para `-1` (falha) e apenas definido para sucesso após a chamada final feita ser bem sucedida.

1.6 Incorporando Python

A única tarefa importante com a qual apenas os incorporadores (em oposição aos escritores de extensão) do interpretador Python precisam se preocupar é a inicialização e, possivelmente, a finalização do interpretador Python. A maior parte da funcionalidade do interpretador só pode ser usada após a inicialização do interpretador.

A função de inicialização básica é *Py_Initialize()*. Isso inicializa a tabela de módulos carregados e cria os módulos fundamentais `builtins`, `__main__` e `sys`. Ela também inicializa o caminho de pesquisa de módulos (`sys.path`).

Py_Initialize() não define a "lista de argumentos de script" (`sys.argv`). Se esta variável for necessária para o código Python que será executado posteriormente, *PyConfig.argv* e *PyConfig.parse_argv* devem estar definidas; veja *Configuração de inicialização do Python*.

Na maioria dos sistemas (em particular, no Unix e no Windows, embora os detalhes sejam ligeiramente diferentes), *Py_Initialize()* calcula o caminho de pesquisa de módulos com base em sua melhor estimativa para a localização do executável do interpretador Python padrão, presumindo que a biblioteca Python é encontrada em um local fixo em relação ao executável do interpretador Python. Em particular, ele procura por um diretório chamado `lib/pythonX.Y` relativo ao diretório pai onde o executável chamado `python` é encontrado no caminho de pesquisa de comandos do shell (a variável de ambiente `PATH`).

Por exemplo, se o executável Python for encontrado em `/usr/local/bin/python`, ele presumirá que as bibliotecas estão em `/usr/local/lib/pythonX.Y`. (Na verdade, este caminho particular também é o local reserva, usado quando nenhum arquivo executável chamado `python` é encontrado ao longo de `PATH`.) O usuário pode substituir este comportamento definindo a variável de ambiente `PYTHONHOME`, ou insira diretórios adicionais na frente do caminho padrão definindo `PYTHONPATH`.

The embedding application can steer the search by setting *PyConfig.program_name* *before* calling *Py_InitializeFromConfig()*. 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).

Às vezes, é desejável "desinicializar" o Python. Por exemplo, a aplicação pode querer iniciar novamente (fazer outra chamada para *Py_Initialize()*) ou a aplicação simplesmente termina com o uso de Python e deseja liberar memória alocada pelo Python. Isso pode ser feito chamando *Py_FinalizeEx()*. A função *Py_IsInitialized()* retorna verdadeiro se o Python está atualmente no estado inicializado. Mais informações sobre essas funções são fornecidas em um capítulo posterior. Observe que *Py_FinalizeEx()* *não* libera toda a memória alocada pelo interpretador Python, por exemplo, a memória alocada por módulos de extensão atualmente não pode ser liberada.

1.7 Compilações de depuração

Python pode ser compilado com várias macros para permitir verificações extras do interpretador e módulos de extensão. Essas verificações tendem a adicionar uma grande quantidade de sobrecarga ao tempo de execução, portanto, não são habilitadas por padrão.

Uma lista completa dos vários tipos de compilações de depuração está no arquivo Misc/SpecialBuilds.txt na distribuição do código-fonte do Python. Estão disponíveis compilações que oferecem suporte ao rastreamento de contagens de referências, depuração do alocador de memória ou criação de perfil de baixo nível do laço do interpretador principal. Apenas as compilações usadas com mais frequência serão descritas no restante desta seção.

Py_DEBUG

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

Além da depuração de contagem de referências descrita abaixo, verificações extras são realizadas, consulte Compilação de Depuração do Python.

Definir Py_TRACE_REFS habilita o rastreamento de referência (veja a opção opção --with-trace-refs de configure). Quando definida, uma lista circular duplamente vinculada de objetos ativos é mantida adicionando dois campos extras a cada *PyObject*. As alocações totais também são rastreadas. Ao sair, todas as referências existentes são impressas. (No modo interativo, isso acontece após cada instrução executada pelo interpretador.)

Consulte Misc/SpecialBuilds.txt na distribuição do código-fonte Python para informações mais detalhadas.

CAPÍTULO 2

Estabilidade da API C

A menos que documentado de outra forma, a API C do Python é coberta pela Política de Compatibilidade com versões anteriores, **PEP 387**. A maioria das alterações são compatíveis com a fonte (normalmente adicionando apenas uma nova API). A alteração ou remoção da API existente só é feita após um período de descontinuação ou para corrigir problemas sérios.

A Interface Binária de Aplicação (ABI) do CPython é compatível para frente e para trás através de uma versão menor (se elas forem compiladas da mesma maneira; veja *Considerações da plataforma* abaixo). Portanto, o código compilado para Python 3.10.0 funcionará em 3.10.8 e vice-versa, mas precisará ser compilado separadamente para 3.9.x e 3.11.x.

Existem dois níveis de API C com diferentes expectativas de estabilidade:

- *API Instável* ("Unstable API"), pode mudar em versões menores sem período de depreciação. É marcado pelo prefixo `PyUnstable` nos nomes.
- *API Limitada* ("Limited API"), é compatível em várias versões menores. Quando `Py_LIMITED_API` é definido, apenas este subconjunto é exposto de `Python.h`.

Elas são discutidas em mais detalhes abaixo.

Nomes prefixados por um sublinhado, como `_Py_InternalState`, são APIs privadas que podem ser alteradas sem aviso prévio, mesmo em lançamentos de correção. Se você precisa usar essa API, considere entrar em contato com os desenvolvedores do CPython para discutir a adição de uma API pública para o seu caso de uso.

2.1 API C Instável

Qualquer API nomeada com o prefixo "PyUnstable" expõe detalhes de implementação do CPython e pode mudar em cada versão menor (por exemplo, de 3.9 para 3.10) sem nenhum aviso de depreciação. No entanto, não mudará em uma versão de correção de bugs (por exemplo, de 3.10.0 para 3.10.1).

É geralmente destinado a ferramentas especializadas de baixo nível, como depuradores.

Projetos que utilizam esta API são esperados para seguir o desenvolvimento do CPython e dedicar esforço extra para se ajustar às mudanças.

2.2 Interface Binária de Aplicação Estável

Para simplificar, este documento fala sobre *extensões*, mas a API Limitada e a ABI Estável funcionam da mesma maneira para todos os usos da API – por exemplo, embutir o Python.

2.2.1 API C Limitada

Python 3.2 introduziu a *API Limitada*, um subconjunto da API C do Python. Extensões que apenas usam o Limited API podem ser compiladas uma vez e ser carregadas em várias versões do Python. Os conteúdos da API Limitada estão *listados abaixo*.

Py_LIMITED_API

Defina essa macro antes de incluir `Python.h` para optar por usar apenas a API Limitada e selecionar a versão da API Limitada.

Defina `Py_LIMITED_API` com o valor de *PY_VERSION_HEX* correspondente à versão mais baixa do Python que sua extensão suporta. A extensão terá compatibilidade com a ABI de todas as versões do Python 3 a partir daquela especificada, e poderá usar a API Limitada introduzida até aquela versão.

Em vez de usar diretamente a macro `PY_VERSION_HEX`, codifique uma versão menor mínima (por exemplo, `0x030A0000` para o Python 3.10) para garantir estabilidade ao compilar com versões futuras do Python.

Você também pode definir `Py_LIMITED_API` como `3`. Isso funciona da mesma forma que `0x03020000` (Python 3.2, a versão que introduziu a API Limitada).

2.2.2 ABI Estável

Para habilitar isso, o Python fornece uma *ABI Estável*: um conjunto de símbolos que permanecerão compatíveis com ABI em todas as versões do Python 3.x.

Nota

O ABI Estável previne problemas de ABI, como erros de ligador devido a símbolos ausentes ou corrupção de dados devido a alterações em layouts de estrutura ou assinaturas de função. No entanto, outras alterações no Python podem alterar o *comportamento* das extensões. Veja a Política de Retrocompatibilidade do Python (**PEP 387**) para detalhes.

A ABI Estável contém símbolos expostos na *API Limitada*, mas também outros – por exemplo, funções necessárias para suportar versões mais antigas da API Limitada.

No Windows, as extensões que usam a ABI Estável devem ser vinculadas a `python3.dll` em vez de uma biblioteca específica de versão, como `python39.dll`.

Em algumas plataformas, o Python procurará e carregará arquivos de biblioteca compartilhada com o nome marcado como `abi3` (por exemplo, `meumódulo.abi3.so`). Ele não verifica se essas extensões estão em conformidade com uma ABI Estável. O usuário (ou suas ferramentas de empacotamento) precisa garantir que, por exemplo, as extensões construídas com a API Limitada 3.10+ não sejam instaladas em versões mais baixas do Python.

Todas as funções na ABI estável estão presentes como funções na biblioteca compartilhada do Python, não apenas como macros. Isso as torna utilizáveis em linguagens que não utilizam o pré-processador C.

2.2.3 Escopo e Desempenho da API Limitada

O objetivo da API Limitada é permitir tudo o que é possível com a API C completa, mas possivelmente com uma penalidade de desempenho.

Por exemplo, enquanto *PyList_GetItem()* está disponível, sua variante de macro "insegura" *PyList_GET_ITEM()* não está. A macro pode ser mais rápida porque pode depender de detalhes de implementação específicos da versão do objeto da lista.

Sem a definição de `Py_LIMITED_API`, algumas funções da API C são colocadas "inline" ou substituídas por macros. Definir `Py_LIMITED_API` desativa esse inline, permitindo estabilidade à medida que as estruturas de dados do Python são aprimoradas, mas possivelmente reduzindo o desempenho.

Ao deixar de fora a definição `Py_LIMITED_API`, é possível compilar uma extensão da API Limitada com uma ABI específica da versão. Isso pode melhorar o desempenho para essa versão do Python, mas limitará a compatibilidade. Compilar com `Py_LIMITED_API` vai produzir uma extensão que pode ser distribuída quando uma específica da versão não estiver disponível – por exemplo, para pré-lançamentos de uma próxima versão do Python.

2.2.4 Limitações da API Limitada

Observe que compilar com `Py_LIMITED_API` *não* é uma garantia completa de que o código esteja em conformidade com a *API Limitada* ou com a *ABI Estável*. `Py_LIMITED_API` abrange apenas definições, mas uma API também inclui outras questões, como semântica esperada.

Uma questão que `Py_LIMITED_API` não protege é a chamada de uma função com argumentos inválidos em uma versão inferior do Python. Por exemplo, considere uma função que começa a aceitar `NULL` como argumento. No Python 3.9, `NULL` agora seleciona um comportamento padrão, mas no Python 3.8, o argumento será usado diretamente, causando uma referência `NULL` e uma falha. Um argumento similar funciona para campos de estruturas.

Outra questão é que alguns campos de estrutura não estão atualmente ocultos quando `Py_LIMITED_API` é definido, mesmo que eles façam parte da API Limitada.

Por esses motivos, recomendamos testar uma extensão com *todas* as versões menores do Python que ela oferece suporte e, preferencialmente, construir com a versão *mais baixa* dessas.

Também recomendamos revisar a documentação de todas as APIs utilizadas para verificar se ela faz parte explicitamente da API Limitada. Mesmo com a definição de `Py_LIMITED_API`, algumas declarações privadas são expostas por razões técnicas (ou até mesmo acidentalmente, como bugs).

Também observe que a API Limitada não é necessariamente estável: compilar com `Py_LIMITED_API` com Python 3.8 significa que a extensão será executada com Python 3.12, mas não necessariamente será *compilada* com Python 3.12. Em particular, partes da API Limitada podem ser descontinuadas e removidas, desde que a ABI Estável permaneça estável.

2.3 Considerações da plataforma

A estabilidade da ABI depende não apenas do Python, mas também do compilador utilizado, das bibliotecas de nível inferior e das opções do compilador. Para os fins da *ABI Estável*, esses detalhes definem uma "plataforma". Geralmente, eles dependem do tipo de sistema operacional e da arquitetura do processador.

É responsabilidade de cada distribuidor particular do Python garantir que todas as versões do Python em uma plataforma específica sejam construídas de forma a não quebrar a ABI estável. Isso é válido para as versões do Windows e macOS disponibilizadas pela `python.org` e por muitos distribuidores terceiros.

2.4 Conteúdo da API Limitada

Atualmente, a *API Limitada* inclui os seguintes itens:

- *PY_VECTORCALL_ARGUMENTS_OFFSET*
- *PyAIter_Check()*
- *PyArg_Parse()*
- *PyArg_ParseTuple()*
- *PyArg_ParseTupleAndKeywords()*
- *PyArg_UnpackTuple()*
- *PyArg_VaParse()*

- *PyArg_VaParseTupleAndKeywords()*
- *PyArg_ValidateKeywordArguments()*
- *PyBaseObject_Type*
- *PyBool_FromLong()*
- *PyBool_Type*
- *PyBuffer_FillContiguousStrides()*
- *PyBuffer_FillInfo()*
- *PyBuffer_FromContiguous()*
- *PyBuffer_GetPointer()*
- *PyBuffer_IsContiguous()*
- *PyBuffer_Release()*
- *PyBuffer_SizeFromFormat()*
- *PyBuffer_ToContiguous()*
- PyByteArrayIter_Type
- *PyByteArray_AsString()*
- *PyByteArray_Concat()*
- *PyByteArray_FromObject()*
- *PyByteArray_FromStringAndSize()*
- *PyByteArray_Resize()*
- *PyByteArray_Size()*
- *PyByteArray_Type*
- PyBytesIter_Type
- *PyBytes_AsString()*
- *PyBytes_AsStringAndSize()*
- *PyBytes_Concat()*
- *PyBytes_ConcatAndDel()*
- PyBytes_DecodeEscape()
- *PyBytes_FromFormat()*
- *PyBytes_FromFormatV()*
- *PyBytes_FromObject()*
- *PyBytes_FromString()*
- *PyBytes_FromStringAndSize()*
- PyBytes_Repr()
- *PyBytes_Size()*
- *PyBytes_Type*
- *PyCFunction*
- *PyCFunctionFast*
- *PyCFunctionFastWithKeywords*
- *PyCFunctionWithKeywords*

- PyCFunction_GetFlags()
- PyCFunction_GetFunction()
- PyCFunction_GetSelf()
- *PyCFunction_New()*
- *PyCFunction_NewEx()*
- PyCFunction_Type
- *PyCMethod_New()*
- *PyCallIter_New()*
- *PyCallIter_Type*
- *PyCallable_Check()*
- *PyCapsule_Destructor*
- *PyCapsule_GetContext()*
- *PyCapsule_GetDestructor()*
- *PyCapsule_GetName()*
- *PyCapsule_GetPointer()*
- *PyCapsule_Import()*
- *PyCapsule_IsValid()*
- *PyCapsule_New()*
- *PyCapsule_SetContext()*
- *PyCapsule_SetDestructor()*
- *PyCapsule_SetName()*
- *PyCapsule_SetPointer()*
- PyCapsule_Type
- PyClassMethodDescr_Type
- *PyCodec_BackslashReplaceErrors()*
- *PyCodec_Decode()*
- *PyCodec_Decoder()*
- *PyCodec_Encode()*
- *PyCodec_Encoder()*
- *PyCodec_IgnoreErrors()*
- *PyCodec_IncrementalDecoder()*
- *PyCodec_IncrementalEncoder()*
- *PyCodec_KnownEncoding()*
- *PyCodec_LookupError()*
- *PyCodec_NameReplaceErrors()*
- *PyCodec_Register()*
- *PyCodec_RegisterError()*
- *PyCodec_ReplaceErrors()*
- *PyCodec_StreamReader()*

- *PyCodec_StreamWriter()*
- *PyCodec_StrictErrors()*
- *PyCodec_Unregister()*
- *PyCodec_XMLCharRefReplaceErrors()*
- *PyComplex_FromDoubles()*
- *PyComplex_ImagAsDouble()*
- *PyComplex_RealAsDouble()*
- *PyComplex_Type*
- *PyDescr_NewClassMethod()*
- *PyDescr_NewGetSet()*
- *PyDescr_NewMember()*
- *PyDescr_NewMethod()*
- PyDictItems_Type
- PyDictIterItem_Type
- PyDictIterKey_Type
- PyDictIterValue_Type
- PyDictKeys_Type
- *PyDictProxy_New()*
- PyDictProxy_Type
- PyDictRevIterItem_Type
- PyDictRevIterKey_Type
- PyDictRevIterValue_Type
- PyDictValues_Type
- *PyDict_Clear()*
- *PyDict_Contains()*
- *PyDict_Copy()*
- *PyDict_DelItem()*
- *PyDict_DelItemString()*
- *PyDict_GetItem()*
- *PyDict_GetItemRef()*
- *PyDict_GetItemString()*
- *PyDict_GetItemStringRef()*
- *PyDict_GetItemWithError()*
- *PyDict_Items()*
- *PyDict_Keys()*
- *PyDict_Merge()*
- *PyDict_MergeFromSeq2()*
- *PyDict_New()*
- *PyDict_Next()*

- *PyDict_SetItem()*
- *PyDict_SetItemString()*
- *PyDict_Size()*
- *PyDict_Type*
- *PyDict_Update()*
- *PyDict_Values()*
- *PyEllipsis_Type*
- PyEnum_Type
- *PyErr_BadArgument()*
- *PyErr_BadInternalCall()*
- *PyErr_CheckSignals()*
- *PyErr_Clear()*
- PyErr_Display()
- *PyErr_DisplayException()*
- *PyErr_ExceptionMatches()*
- *PyErr_Fetch()*
- *PyErr_Format()*
- *PyErr_FormatV()*
- *PyErr_GetExcInfo()*
- *PyErr_GetHandledException()*
- *PyErr_GetRaisedException()*
- *PyErr_GivenExceptionMatches()*
- *PyErr_NewException()*
- *PyErr_NewExceptionWithDoc()*
- *PyErr_NoMemory()*
- *PyErr_NormalizeException()*
- *PyErr_Occurred()*
- *PyErr_Print()*
- *PyErr_PrintEx()*
- PyErr_ProgramText()
- *PyErr_ResourceWarning()*
- *PyErr_Restore()*
- *PyErr_SetExcFromWindowsErr()*
- *PyErr_SetExcFromWindowsErrWithFilename()*
- *PyErr_SetExcFromWindowsErrWithFilenameObject()*
- *PyErr_SetExcFromWindowsErrWithFilenameObjects()*
- *PyErr_SetExcInfo()*
- *PyErr_SetFromErrno()*
- *PyErr_SetFromErrnoWithFilename()*

- *PyErr_SetFromErrnoWithFilenameObject()*
- *PyErr_SetFromErrnoWithFilenameObjects()*
- *PyErr_SetFromWindowsErr()*
- *PyErr_SetFromWindowsErrWithFilename()*
- *PyErr_SetHandledException()*
- *PyErr_SetImportError()*
- *PyErr_SetImportErrorSubclass()*
- *PyErr_SetInterrupt()*
- *PyErr_SetInterruptEx()*
- *PyErr_SetNone()*
- *PyErr_SetObject()*
- *PyErr_SetRaisedException()*
- *PyErr_SetString()*
- *PyErr_SyntaxLocation()*
- *PyErr_SyntaxLocationEx()*
- *PyErr_WarnEx()*
- *PyErr_WarnExplicit()*
- *PyErr_WarnFormat()*
- *PyErr_WriteUnraisable()*
- *PyEval_AcquireThread()*
- *PyEval_EvalCode()*
- *PyEval_EvalCodeEx()*
- *PyEval_EvalFrame()*
- *PyEval_EvalFrameEx()*
- *PyEval_GetBuiltins()*
- *PyEval_GetFrame()*
- *PyEval_GetFrameBuiltins()*
- *PyEval_GetFrameGlobals()*
- *PyEval_GetFrameLocals()*
- *PyEval_GetFuncDesc()*
- *PyEval_GetFuncName()*
- *PyEval_GetGlobals()*
- *PyEval_GetLocals()*
- *PyEval_InitThreads()*
- *PyEval_ReleaseThread()*
- *PyEval_RestoreThread()*
- *PyEval_SaveThread()*
- PyExc_ArithmeticError
- PyExc_AssertionError

- PyExc_AttributeError
- PyExc_BaseException
- PyExc_BaseExceptionGroup
- PyExc_BlockingIOError
- PyExc_BrokenPipeError
- PyExc_BufferError
- PyExc_BytesWarning
- PyExc_ChildProcessError
- PyExc_ConnectionAbortedError
- PyExc_ConnectionError
- PyExc_ConnectionRefusedError
- PyExc_ConnectionResetError
- PyExc_DeprecationWarning
- PyExc_EOFError
- PyExc_EncodingWarning
- PyExc_EnvironmentError
- PyExc_Exception
- PyExc_FileExistsError
- PyExc_FileNotFoundError
- PyExc_FloatingPointError
- PyExc_FutureWarning
- PyExc_GeneratorExit
- PyExc_IOError
- PyExc_ImportError
- PyExc_ImportWarning
- PyExc_IndentationError
- PyExc_IndexError
- PyExc_InterruptedError
- PyExc_IsADirectoryError
- PyExc_KeyError
- PyExc_KeyboardInterrupt
- PyExc_LookupError
- PyExc_MemoryError
- PyExc_ModuleNotFoundError
- PyExc_NameError
- PyExc_NotADirectoryError
- PyExc_NotImplementedError
- PyExc_OSError
- PyExc_OverflowError

- PyExc_PendingDeprecationWarning
- PyExc_PermissionError
- PyExc_ProcessLookupError
- PyExc_RecursionError
- PyExc_ReferenceError
- PyExc_ResourceWarning
- PyExc_RuntimeError
- PyExc_RuntimeWarning
- PyExc_StopAsyncIteration
- PyExc_StopIteration
- PyExc_SyntaxError
- PyExc_SyntaxWarning
- PyExc_SystemError
- PyExc_SystemExit
- PyExc_TabError
- PyExc_TimeoutError
- PyExc_TypeError
- PyExc_UnboundLocalError
- PyExc_UnicodeDecodeError
- PyExc_UnicodeEncodeError
- PyExc_UnicodeError
- PyExc_UnicodeTranslateError
- PyExc_UnicodeWarning
- PyExc_UserWarning
- PyExc_ValueError
- PyExc_Warning
- PyExc_WindowsError
- PyExc_ZeroDivisionError
- PyExceptionClass_Name()
- *PyException_GetArgs()*
- *PyException_GetCause()*
- *PyException_GetContext()*
- *PyException_GetTraceback()*
- *PyException_SetArgs()*
- *PyException_SetCause()*
- *PyException_SetContext()*
- *PyException_SetTraceback()*
- *PyFile_FromFd()*
- *PyFile_GetLine()*

- *PyFile_WriteObject()*
- *PyFile_WriteString()*
- PyFilter_Type
- *PyFloat_AsDouble()*
- *PyFloat_FromDouble()*
- *PyFloat_FromString()*
- *PyFloat_GetInfo()*
- *PyFloat_GetMax()*
- *PyFloat_GetMin()*
- *PyFloat_Type*
- *PyFrameObject*
- *PyFrame_GetCode()*
- *PyFrame_GetLineNumber()*
- *PyFrozenSet_New()*
- *PyFrozenSet_Type*
- *PyGC_Collect()*
- *PyGC_Disable()*
- *PyGC_Enable()*
- *PyGC_IsEnabled()*
- *PyGILState_Ensure()*
- *PyGILState_GetThisThreadState()*
- *PyGILState_Release()*
- PyGILState_STATE
- *PyGetSetDef*
- PyGetSetDescr_Type
- *PyImport_AddModule()*
- *PyImport_AddModuleObject()*
- *PyImport_AddModuleRef()*
- *PyImport_AppendInittab()*
- *PyImport_ExecCodeModule()*
- *PyImport_ExecCodeModuleEx()*
- *PyImport_ExecCodeModuleObject()*
- *PyImport_ExecCodeModuleWithPathnames()*
- *PyImport_GetImporter()*
- *PyImport_GetMagicNumber()*
- *PyImport_GetMagicTag()*
- *PyImport_GetModule()*
- *PyImport_GetModuleDict()*
- *PyImport_Import()*

- *PyImport_ImportFrozenModule()*
- *PyImport_ImportFrozenModuleObject()*
- *PyImport_ImportModule()*
- *PyImport_ImportModuleLevel()*
- *PyImport_ImportModuleLevelObject()*
- *PyImport_ImportModuleNoBlock()*
- *PyImport_ReloadModule()*
- *PyIndex_Check()*
- *PyInterpreterState*
- *PyInterpreterState_Clear()*
- *PyInterpreterState_Delete()*
- *PyInterpreterState_Get()*
- *PyInterpreterState_GetDict()*
- *PyInterpreterState_GetID()*
- *PyInterpreterState_New()*
- *PyIter_Check()*
- *PyIter_Next()*
- *PyIter_Send()*
- PyListIter_Type
- PyListRevIter_Type
- *PyList_Append()*
- *PyList_AsTuple()*
- *PyList_GetItem()*
- *PyList_GetItemRef()*
- *PyList_GetSlice()*
- *PyList_Insert()*
- *PyList_New()*
- *PyList_Reverse()*
- *PyList_SetItem()*
- *PyList_SetSlice()*
- *PyList_Size()*
- *PyList_Sort()*
- *PyList_Type*
- *PyLongObject*
- PyLongRangeIter_Type
- *PyLong_AsDouble()*
- *PyLong_AsInt()*
- *PyLong_AsLong()*
- *PyLong_AsLongAndOverflow()*

- *PyLong_AsLongLong()*
- *PyLong_AsLongLongAndOverflow()*
- *PyLong_AsSize_t()*
- *PyLong_AsSsize_t()*
- *PyLong_AsUnsignedLong()*
- *PyLong_AsUnsignedLongLong()*
- *PyLong_AsUnsignedLongLongMask()*
- *PyLong_AsUnsignedLongMask()*
- *PyLong_AsVoidPtr()*
- *PyLong_FromDouble()*
- *PyLong_FromLong()*
- *PyLong_FromLongLong()*
- *PyLong_FromSize_t()*
- *PyLong_FromSsize_t()*
- *PyLong_FromString()*
- *PyLong_FromUnsignedLong()*
- *PyLong_FromUnsignedLongLong()*
- *PyLong_FromVoidPtr()*
- *PyLong_GetInfo()*
- *PyLong_Type*
- PyMap_Type
- *PyMapping_Check()*
- *PyMapping_GetItemString()*
- *PyMapping_GetOptionalItem()*
- *PyMapping_GetOptionalItemString()*
- *PyMapping_HasKey()*
- *PyMapping_HasKeyString()*
- *PyMapping_HasKeyStringWithError()*
- *PyMapping_HasKeyWithError()*
- *PyMapping_Items()*
- *PyMapping_Keys()*
- *PyMapping_Length()*
- *PyMapping_SetItemString()*
- *PyMapping_Size()*
- *PyMapping_Values()*
- *PyMem_Calloc()*
- *PyMem_Free()*
- *PyMem_Malloc()*
- *PyMem_RawCalloc()*

- *PyMem_RawFree()*
- *PyMem_RawMalloc()*
- *PyMem_RawRealloc()*
- *PyMem_Realloc()*
- *PyMemberDef*
- PyMemberDescr_Type
- *PyMember_GetOne()*
- *PyMember_SetOne()*
- *PyMemoryView_FromBuffer()*
- *PyMemoryView_FromMemory()*
- *PyMemoryView_FromObject()*
- *PyMemoryView_GetContiguous()*
- PyMemoryView_Type
- *PyMethodDef*
- PyMethodDescr_Type
- *PyModuleDef*
- PyModuleDef_Base
- *PyModuleDef_Init()*
- PyModuleDef_Type
- *PyModule_Add()*
- *PyModule_AddFunctions()*
- *PyModule_AddIntConstant()*
- *PyModule_AddObject()*
- *PyModule_AddObjectRef()*
- *PyModule_AddStringConstant()*
- *PyModule_AddType()*
- *PyModule_Create2()*
- *PyModule_ExecDef()*
- *PyModule_FromDefAndSpec2()*
- *PyModule_GetDef()*
- *PyModule_GetDict()*
- *PyModule_GetFilename()*
- *PyModule_GetFilenameObject()*
- *PyModule_GetName()*
- *PyModule_GetNameObject()*
- *PyModule_GetState()*
- *PyModule_New()*
- *PyModule_NewObject()*
- *PyModule_SetDocString()*

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

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

- *PyObject_Dir()*
- *PyObject_Format()*
- *PyObject_Free()*
- *PyObject_GC_Del()*
- *PyObject_GC_IsFinalized()*
- *PyObject_GC_IsTracked()*
- *PyObject_GC_Track()*
- *PyObject_GC_UnTrack()*
- *PyObject_GenericGetAttr()*
- *PyObject_GenericGetDict()*
- *PyObject_GenericSetAttr()*
- *PyObject_GenericSetDict()*
- *PyObject_GetAIter()*
- *PyObject_GetAttr()*
- *PyObject_GetAttrString()*
- *PyObject_GetBuffer()*
- *PyObject_GetItem()*
- *PyObject_GetIter()*
- *PyObject_GetOptionalAttr()*
- *PyObject_GetOptionalAttrString()*
- *PyObject_GetTypeData()*
- *PyObject_HasAttr()*
- *PyObject_HasAttrString()*
- *PyObject_HasAttrStringWithError()*
- *PyObject_HasAttrWithError()*
- *PyObject_Hash()*
- *PyObject_HashNotImplemented()*
- *PyObject_Init()*
- *PyObject_InitVar()*
- *PyObject_IsInstance()*
- *PyObject_IsSubclass()*
- *PyObject_IsTrue()*
- *PyObject_Length()*
- *PyObject_Malloc()*
- *PyObject_Not()*
- *PyObject_Realloc()*
- *PyObject_Repr()*
- *PyObject_RichCompare()*
- *PyObject_RichCompareBool()*

- *PyObject_SelfIter()*
- *PyObject_SetAttr()*
- *PyObject_SetAttrString()*
- *PyObject_SetItem()*
- *PyObject_Size()*
- *PyObject_Str()*
- *PyObject_Type()*
- *PyObject_Vectorcall()*
- *PyObject_VectorcallMethod()*
- *PyProperty_Type*
- PyRangeIter_Type
- PyRange_Type
- PyReversed_Type
- *PySeqIter_New()*
- *PySeqIter_Type*
- *PySequence_Check()*
- *PySequence_Concat()*
- *PySequence_Contains()*
- *PySequence_Count()*
- *PySequence_DelItem()*
- *PySequence_DelSlice()*
- *PySequence_Fast()*
- *PySequence_GetItem()*
- *PySequence_GetSlice()*
- PySequence_In()
- *PySequence_InPlaceConcat()*
- *PySequence_InPlaceRepeat()*
- *PySequence_Index()*
- *PySequence_Length()*
- *PySequence_List()*
- *PySequence_Repeat()*
- *PySequence_SetItem()*
- *PySequence_SetSlice()*
- *PySequence_Size()*
- *PySequence_Tuple()*
- PySetIter_Type
- *PySet_Add()*
- *PySet_Clear()*
- *PySet_Contains()*

- *PySet_Discard()*
- *PySet_New()*
- *PySet_Pop()*
- *PySet_Size()*
- *PySet_Type*
- *PySlice_AdjustIndices()*
- *PySlice_GetIndices()*
- *PySlice_GetIndicesEx()*
- *PySlice_New()*
- *PySlice_Type*
- *PySlice_Unpack()*
- *PyState_AddModule()*
- *PyState_FindModule()*
- *PyState_RemoveModule()*
- *PyStructSequence_Desc*
- *PyStructSequence_Field*
- *PyStructSequence_GetItem()*
- *PyStructSequence_New()*
- *PyStructSequence_NewType()*
- *PyStructSequence_SetItem()*
- *PyStructSequence_UnnamedField*
- PySuper_Type
- *PySys_Audit()*
- *PySys_AuditTuple()*
- *PySys_FormatStderr()*
- *PySys_FormatStdout()*
- *PySys_GetObject()*
- *PySys_GetXOptions()*
- *PySys_ResetWarnOptions()*
- *PySys_SetArgv()*
- *PySys_SetArgvEx()*
- *PySys_SetObject()*
- *PySys_WriteStderr()*
- *PySys_WriteStdout()*
- *PyThreadState*
- *PyThreadState_Clear()*
- *PyThreadState_Delete()*
- *PyThreadState_Get()*
- *PyThreadState_GetDict()*

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

- *PyTuple_Pack()*
- *PyTuple_SetItem()*
- *PyTuple_Size()*
- *PyTuple_Type*
- *PyTypeObject*
- *PyType_ClearCache()*
- *PyType_FromMetaclass()*
- *PyType_FromModuleAndSpec()*
- *PyType_FromSpec()*
- *PyType_FromSpecWithBases()*
- *PyType_GenericAlloc()*
- *PyType_GenericNew()*
- *PyType_GetFlags()*
- *PyType_GetFullyQualifiedName()*
- *PyType_GetModule()*
- *PyType_GetModuleByDef()*
- *PyType_GetModuleName()*
- *PyType_GetModuleState()*
- *PyType_GetName()*
- *PyType_GetQualName()*
- *PyType_GetSlot()*
- *PyType_GetTypeDataSize()*
- *PyType_IsSubtype()*
- *PyType_Modified()*
- *PyType_Ready()*
- *PyType_Slot*
- *PyType_Spec*
- *PyType_Type*
- *PyUnicodeDecodeError_Create()*
- *PyUnicodeDecodeError_GetEncoding()*
- *PyUnicodeDecodeError_GetEnd()*
- *PyUnicodeDecodeError_GetObject()*
- *PyUnicodeDecodeError_GetReason()*
- *PyUnicodeDecodeError_GetStart()*
- *PyUnicodeDecodeError_SetEnd()*
- *PyUnicodeDecodeError_SetReason()*
- *PyUnicodeDecodeError_SetStart()*
- *PyUnicodeEncodeError_GetEncoding()*
- *PyUnicodeEncodeError_GetEnd()*

- *PyUnicodeEncodeError_GetObject()*
- *PyUnicodeEncodeError_GetReason()*
- *PyUnicodeEncodeError_GetStart()*
- *PyUnicodeEncodeError_SetEnd()*
- *PyUnicodeEncodeError_SetReason()*
- *PyUnicodeEncodeError_SetStart()*
- PyUnicodeIter_Type
- *PyUnicodeTranslateError_GetEnd()*
- *PyUnicodeTranslateError_GetObject()*
- *PyUnicodeTranslateError_GetReason()*
- *PyUnicodeTranslateError_GetStart()*
- *PyUnicodeTranslateError_SetEnd()*
- *PyUnicodeTranslateError_SetReason()*
- *PyUnicodeTranslateError_SetStart()*
- PyUnicode_Append()
- PyUnicode_AppendAndDel()
- *PyUnicode_AsASCIIString()*
- *PyUnicode_AsCharmapString()*
- PyUnicode_AsDecodedObject()
- PyUnicode_AsDecodedUnicode()
- PyUnicode_AsEncodedObject()
- *PyUnicode_AsEncodedString()*
- PyUnicode_AsEncodedUnicode()
- *PyUnicode_AsLatin1String()*
- *PyUnicode_AsMBCSString()*
- *PyUnicode_AsRawUnicodeEscapeString()*
- *PyUnicode_AsUCS4()*
- *PyUnicode_AsUCS4Copy()*
- *PyUnicode_AsUTF16String()*
- *PyUnicode_AsUTF32String()*
- *PyUnicode_AsUTF8AndSize()*
- *PyUnicode_AsUTF8String()*
- *PyUnicode_AsUnicodeEscapeString()*
- *PyUnicode_AsWideChar()*
- *PyUnicode_AsWideCharString()*
- PyUnicode_BuildEncodingMap()
- *PyUnicode_Compare()*
- *PyUnicode_CompareWithASCIIString()*
- *PyUnicode_Concat()*

- *PyUnicode_Contains()*
- *PyUnicode_Count()*
- *PyUnicode_Decode()*
- *PyUnicode_DecodeASCII()*
- *PyUnicode_DecodeCharmap()*
- *PyUnicode_DecodeCodePageStateful()*
- *PyUnicode_DecodeFSDefault()*
- *PyUnicode_DecodeFSDefaultAndSize()*
- *PyUnicode_DecodeLatin1()*
- *PyUnicode_DecodeLocale()*
- *PyUnicode_DecodeLocaleAndSize()*
- *PyUnicode_DecodeMBCS()*
- *PyUnicode_DecodeMBCSStateful()*
- *PyUnicode_DecodeRawUnicodeEscape()*
- *PyUnicode_DecodeUTF16()*
- *PyUnicode_DecodeUTF16Stateful()*
- *PyUnicode_DecodeUTF32()*
- *PyUnicode_DecodeUTF32Stateful()*
- *PyUnicode_DecodeUTF7()*
- *PyUnicode_DecodeUTF7Stateful()*
- *PyUnicode_DecodeUTF8()*
- *PyUnicode_DecodeUTF8Stateful()*
- *PyUnicode_DecodeUnicodeEscape()*
- *PyUnicode_EncodeCodePage()*
- *PyUnicode_EncodeFSDefault()*
- *PyUnicode_EncodeLocale()*
- *PyUnicode_EqualToUTF8()*
- *PyUnicode_EqualToUTF8AndSize()*
- *PyUnicode_FSConverter()*
- *PyUnicode_FSDecoder()*
- *PyUnicode_Find()*
- *PyUnicode_FindChar()*
- *PyUnicode_Format()*
- *PyUnicode_FromEncodedObject()*
- *PyUnicode_FromFormat()*
- *PyUnicode_FromFormatV()*
- *PyUnicode_FromObject()*
- PyUnicode_FromOrdinal()
- *PyUnicode_FromString()*

- *PyUnicode_FromStringAndSize()*
- *PyUnicode_FromWideChar()*
- *PyUnicode_GetDefaultEncoding()*
- *PyUnicode_GetLength()*
- *PyUnicode_InternFromString()*
- *PyUnicode_InternInPlace()*
- *PyUnicode_IsIdentifier()*
- *PyUnicode_Join()*
- *PyUnicode_Partition()*
- *PyUnicode_RPartition()*
- *PyUnicode_RSplit()*
- *PyUnicode_ReadChar()*
- *PyUnicode_Replace()*
- PyUnicode_Resize()
- *PyUnicode_RichCompare()*
- *PyUnicode_Split()*
- *PyUnicode_Splitlines()*
- *PyUnicode_Substring()*
- *PyUnicode_Tailmatch()*
- *PyUnicode_Translate()*
- *PyUnicode_Type*
- *PyUnicode_WriteChar()*
- *PyVarObject*
- PyVarObject.ob_base
- *PyVarObject.ob_size*
- *PyVectorcall_Call()*
- *PyVectorcall_NARGS()*
- PyWeakReference
- *PyWeakref_GetObject()*
- *PyWeakref_GetRef()*
- *PyWeakref_NewProxy()*
- *PyWeakref_NewRef()*
- PyWrapperDescr_Type
- *PyWrapper_New()*
- PyZip_Type
- *Py_AddPendingCall()*
- *Py_AtExit()*
- *Py_BEGIN_ALLOW_THREADS*
- *Py_BLOCK_THREADS*

- *Py_BuildValue()*
- *Py_BytesMain()*
- *Py_CompileString()*
- *Py_DecRef()*
- *Py_DecodeLocale()*
- *Py_END_ALLOW_THREADS*
- *Py_EncodeLocale()*
- *Py_EndInterpreter()*
- *Py_EnterRecursiveCall()*
- *Py_Exit()*
- *Py_FatalError()*
- Py_FileSystemDefaultEncodeErrors
- Py_FileSystemDefaultEncoding
- *Py_Finalize()*
- *Py_FinalizeEx()*
- *Py_GenericAlias()*
- *Py_GenericAliasType*
- *Py_GetBuildInfo()*
- *Py_GetCompiler()*
- *Py_GetConstant()*
- *Py_GetConstantBorrowed()*
- *Py_GetCopyright()*
- *Py_GetExecPrefix()*
- *Py_GetPath()*
- *Py_GetPlatform()*
- *Py_GetPrefix()*
- *Py_GetProgramFullPath()*
- *Py_GetProgramName()*
- *Py_GetPythonHome()*
- Py_GetRecursionLimit()
- *Py_GetVersion()*
- Py_HasFileSystemDefaultEncoding
- *Py_IncRef()*
- *Py_Initialize()*
- *Py_InitializeEx()*
- *Py_Is()*
- *Py_IsFalse()*
- *Py_IsFinalizing()*
- *Py_IsInitialized()*

- *Py_IsNone()*
- *Py_IsTrue()*
- *Py_LeaveRecursiveCall()*
- *Py_Main()*
- Py_MakePendingCalls()
- *Py_NewInterpreter()*
- *Py_NewRef()*
- *Py_ReprEnter()*
- *Py_ReprLeave()*
- *Py_SetProgramName()*
- *Py_SetPythonHome()*
- Py_SetRecursionLimit()
- *Py_UCS4*
- *Py_UNBLOCK_THREADS*
- Py_UTF8Mode
- *Py_VaBuildValue()*
- *Py_Version*
- *Py_XNewRef()*
- *Py_buffer*
- Py_intptr_t
- *Py_ssize_t*
- Py_uintptr_t
- *allocfunc*
- *binaryfunc*
- *descrgetfunc*
- *descrsetfunc*
- *destructor*
- *getattrfunc*
- *getattrofunc*
- *getbufferproc*
- *getiterfunc*
- *getter*
- *hashfunc*
- *initproc*
- *inquiry*
- *iternextfunc*
- *lenfunc*
- *newfunc*
- *objobjargproc*

- *objobjproc*
- *releasebufferproc*
- *reprfunc*
- *richcmpfunc*
- *setattrfunc*
- *setattrofunc*
- *setter*
- *ssizeargfunc*
- *ssizeobjargproc*
- ssizessizeargfunc
- ssizessizeobjargproc
- symtable
- *ternaryfunc*
- *traverseproc*
- *unaryfunc*
- *vectorcallfunc*
- *visitproc*

CAPÍTULO 3

A camada de nível muito alto

As funções neste capítulo permitirão que você execute um código-fonte Python fornecido em um arquivo ou buffer, mas não permitirão que você interaja de forma mais detalhada com o interpretador.

Várias dessas funções aceitam um símbolo de início da gramática como parâmetro. Os símbolos de início disponíveis são `Py_eval_input`, `Py_file_input` e `Py_single_input`. Eles são descritos seguindo as funções que os aceitam como parâmetros.

Note também que várias dessas funções aceitam parâmetros `FILE*`. Um problema específico que precisa ser tratado com cuidado é que a estrutura `FILE` para diferentes bibliotecas C pode ser diferente e incompatível. No Windows (pelo menos), é possível que extensões vinculadas dinamicamente usem bibliotecas diferentes, então deve-se tomar cuidado para que os parâmetros `FILE*` sejam passados para essas funções somente se for certo que elas foram criadas pela mesma biblioteca que o tempo de execução do Python está usando.

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

Esta é uma interface simplificada para `PyRun_AnyFileExFlags()` abaixo, deixando *closeit* definido como `0` e *flags* definido como `NULL`.

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

Esta é uma interface simplificada para `PyRun_AnyFileExFlags()` abaixo, deixando o argumento *closeit* definido como `0`.

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

Esta é uma interface simplificada para `PyRun_AnyFileExFlags()` abaixo, deixando o argumento *flags* definido como `NULL`.

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

Se *fp* se referir a um arquivo associado a um dispositivo interativo (entrada de console ou terminal ou pseudo-terminal Unix), retorna o valor de `PyRun_InteractiveLoop()`, caso contrário, retorna o resultado de `PyRun_SimpleFile()`. *filename* é decodificado da codificação do sistema de arquivos (`sys.getfilesystemencoding()`). Se *filename* for `NULL`, esta função usa `"???"` como o nome do arquivo. Se *closeit* for true, o arquivo será fechado antes de `PyRun_SimpleFileExFlags()` retornar.

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

Esta é uma interface simplificada para `PyRun_SimpleStringFlags()` abaixo, deixando o argumento `PyCompilerFlags`* definido como `NULL`.

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

Executa o código-fonte Python de *command* no módulo `__main__` de acordo com o argumento *flags*. Se

__main__ ainda não existir, ele será criado. Retorna 0 em caso de sucesso ou -1 se uma exceção foi gerada. Se houve um erro, não há como obter as informações da exceção. Para o significado de *flags*, veja abaixo.

Observe que se uma exceção SystemExit não tratada for levantada, esta função não retornará -1, mas sairá do processo, desde que *PyConfig.inspect* seja zero.

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

Esta é uma interface simplificada para *PyRun_SimpleFileExFlags()* abaixo, deixando *closeit* definido como 0 e *flags* definido como NULL.

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

Esta é uma interface simplificada para *PyRun_SimpleFileExFlags()* abaixo, deixando o *flags* definido como NULL.

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

Semelhante a *PyRun_SimpleStringFlags()*, mas o código-fonte Python é lido de *fp* em vez de uma string na memória. *filename* deve ser o nome do arquivo, ele é decodificado a partir do *tratador de erros e codificação do sistema de arquivos*. Se *closeit* for true, o arquivo será fechado antes que PyRun_SimpleFileExFlags() retorne.

> **Nota**
>
> No Windows, *fp* deve ser aberto como modo binário (por exemplo, fopen(filename, "rb")). Caso contrário, o Python pode não manipular corretamente o arquivo de script com final de linha LF.

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

Esta é uma interface simplificada para *PyRun_InteractiveOneFlags()* abaixo, deixando *flags* definido como NULL.

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

Lê e executa uma única instrução de um arquivo associado a um dispositivo interativo de acordo com o argumento *flags*. O usuário será solicitado usando sys.ps1 e sys.ps2. *filename* é decodificado a partir do *tratador de erros e codificação do sistema de arquivos*.

Retorna 0 quando a entrada foi executada com sucesso, -1 se houve uma exceção ou um código de erro do arquivo de inclusão errcode.h distribuído como parte do Python se houve um erro de análise. (Observe que errcode.h não é incluído por Python.h, então deve ser incluído especificamente se necessário.)

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

Esta é uma interface simplificada para *PyRun_InteractiveLoopFlags()* abaixo, deixando *flags* definido como NULL.

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

Lê e executa instruções de um arquivo associado a um dispositivo interativo até que o EOF seja atingido. O usuário será consultado usando sys.ps1 e sys.ps2. *filename* é decodificado a partir do *tratador de erros e codificação do sistema de arquivos*. Retorna 0 no EOF ou um número negativo em caso de falha.

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

Parte da ABI Estável. Pode ser definido para apontar para uma função com o protótipo int func(void). A função será chamada quando o prompt do interpretador do Python estiver prestes a ficar ocioso e aguardar a entrada do usuário no terminal. O valor de retorno é ignorado. A substituição deste hook pode ser usada para integrar o prompt do interpretador com outros laços de eventos, como feito em Modules/_tkinter.c no código-fonte do Python.

Alterado na versão 3.12: Esta função só é chamada pelo *interpretador principal*.

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

Pode ser definido para apontar para uma função com o protótipo char *func(FILE *stdin, FILE *stdout, char *prompt), substituindo a função padrão usada para ler uma única linha de entrada no prompt do interpretador. Espera-se que a função produza a string *prompt* se não for NULL e, em seguida, leia

uma linha de entrada do arquivo de entrada padrão fornecido, retornando a string resultante. Por exemplo, o módulo `readline` define este gancho para fornecer recursos de edição de linha e preenchimento de tabulação.

O resultado deve ser uma string alocada por *PyMem_RawMalloc()* ou *PyMem_RawRealloc()*, ou `NULL` se ocorrer um erro.

Alterado na versão 3.4: O resultado deve ser alocado por *PyMem_RawMalloc()* ou *PyMem_RawRealloc()*, em vez de ser alocado por *PyMem_Malloc()* ou *PyMem_Realloc()*.

Alterado na versão 3.12: Esta função só é chamada pelo *interpretador principal*.

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

Retorna valor: Nova referência. Esta é uma interface simplificada para *PyRun_StringFlags()* abaixo, deixando *flags* definido como `NULL`.

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

Retorna valor: Nova referência. Execut o código-fonte Python de *str* no contexto especificado pelos objetos *globals* e *locals* com os sinalizadores do compilador especificados por *flags*. *globals* deve ser um dicionário; *locals* pode ser qualquer objeto que implemente o protocolo de mapeamento. O parâmetro *start* especifica o token de início que deve ser usado para analisar o código-fonte.

Retorna o resultado da execução do código como um objeto Python, ou `NULL` se uma exceção foi levantada.

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

Retorna valor: Nova referência. Esta é uma interface simplificada para *PyRun_FileExFlags()* abaixo, deixando *closeit* definido como `0` e *flags* definido como `NULL`.

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

Retorna valor: Nova referência. Esta é uma interface simplificada para *PyRun_FileExFlags()* abaixo, deixando *flags* definido como `NULL`.

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

Retorna valor: Nova referência. Esta é uma interface simplificada para *PyRun_FileExFlags()* abaixo, deixando *closeit* definido como `0`.

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

Retorna valor: Nova referência. Semelhante a *PyRun_StringFlags()*, mas o código-fonte Python é lido de *fp* em vez de uma string na memória. *filename* deve ser o nome do arquivo, ele é decodificado a partir do *tratador de erros e codificação do sistema de arquivos*. Se *closeit* for true, o arquivo será fechado antes que *PyRun_FileExFlags()* retorne.

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

Retorna valor: Nova referência. *Parte da* ABI Estável. Esta é uma interface simplificada para *Py_CompileStringFlags()* abaixo, deixando *flags* definido como `NULL`.

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

Retorna valor: Nova referência. Esta é uma interface simplificada para *Py_CompileStringExFlags()* abaixo, com *optimize* definido como `-1`.

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

Retorna valor: Nova referência. Analisa e compil o código-fonte Python em *str*, retornando o objeto de código resultante. O token inicial é fornecido por *start*; isso pode ser usado para restringir o código que pode ser compilado e deve ser *Py_eval_input*, *Py_file_input* ou *Py_single_input*. O nome do arquivo especificado por *filename* é usado para construir o objeto de código e pode aparecer em tracebacks ou mensagens de exceção `SyntaxError`. Isso retorna `NULL` se o código não puder ser analisado ou compilado.

O inteiro *optimize* especifica o nível de otimização do compilador; um valor de `-1` seleciona o nível de otimização do interpretador dado pela opção `-O`. Níveis explícitos são `0` (nenhuma otimização; `__debug__` é

verdadeiro), `1` (instruções `assert` são removidas, `__debug__` é falso) ou `2` (strings de documentação também são removidas).

Adicionado na versão 3.4.

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

Retorna valor: Nova referência. Como *`Py_CompileStringObject()`*, mas *filename* é uma string de bytes decodificada a partir do *tratador de erros e codificação do sistema de arquivos*.

Adicionado na versão 3.2.

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

Retorna valor: Nova referência. Parte da ABI Estável. Esta é uma interface simplificada para *`PyEval_EvalCodeEx()`*, com apenas o objeto código e variáveis locais e globais. Os outros argumentos são definidos como `NULL`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Avalia um objeto código pré-compilado, dado um ambiente particular para sua avaliação. Este ambiente consiste em um dicionário de variáveis globais, um objeto mapeamento de variáveis locais, vetores de argumentos, nomeados e padrões, um dicionário de valores padrões para argumentos *somente-nomeados* e uma tupla de clausura de células.

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

Retorna valor: Nova referência. Parte da ABI Estável. Avalia um quadro de execução. Esta é uma interface simplificada para *`PyEval_EvalFrameEx()`*, para retrocompatibilidade.

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

Retorna valor: Nova referência. Parte da ABI Estável. Esta é a função principal e sem retoques da interpretação Python. O objeto código associado ao quadro de execução *f* é executado, interpretando bytecode e executando chamadas conforme necessário. O parâmetro adicional *throwflag* pode ser ignorado na maioria das vezes - se verdadeiro, ele faz com que uma exceção seja levantada imediatamente; isso é usado para os métodos `throw()` de objetos geradores.

Alterado na versão 3.4: Essa função agora inclui uma asserção de depuração para ajudar a garantir que ela não descarte silenciosamente uma exceção ativa.

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

Esta função altera os sinalizadores do quadro de avaliação atual e retorna verdadeiro em caso de sucesso e falso em caso de falha.

int **Py_eval_input**

O símbolo inicial da gramática Python para expressões isoladas; para uso com *`Py_CompileString()`*.

int **Py_file_input**

O símbolo de início da gramática Python para sequências de instruções conforme lidas de um arquivo ou outra fonte; para uso com *`Py_CompileString()`*. Este é o símbolo a ser usado ao compilar código-fonte Python arbitrariamente longo.

int **Py_single_input**

O símbolo de início da gramática Python para uma única declaração; para uso com *`Py_CompileString()`*. Este é o símbolo usado para o laço do interpretador interativo.

struct **PyCompilerFlags**

Esta é a estrutura usada para manter os sinalizadores do compilador. Em casos onde o código está apenas sendo compilado, ele é passado como `int flags`, e em casos onde o código está sendo executado, ele é passado como `PyCompilerFlags *flags`. Neste caso, `from __future__ import` pode modificar *flags*.

Sempre que `PyCompilerFlags *flags` for `NULL`, *`cf_flags`* é tratado como igual a `0`, e qualquer modificação devido a `from __future__ import` é descartada.

int **cf_flags**

Sinalizadores do compilador.

int **cf_feature_version**

cf_feature_version é a versão secundária do Python. Deve ser inicializada como `PY_MINOR_VERSION`.

O campo é ignorado por padrão, ele é usado se e somente se o sinalizador `PyCF_ONLY_AST` estiver definido em *cf_flags*.

Alterado na versão 3.8: Adicionado campo *cf_feature_version*.

Os sinalizadores do compilador disponíveis são acessíveis como macros:

PyCF_ALLOW_TOP_LEVEL_AWAIT

PyCF_ONLY_AST

PyCF_OPTIMIZED_AST

PyCF_TYPE_COMMENTS

Veja sinalizadores do compilador na documentação do módulo Python `ast`, que exporta essas constantes com os mesmos nomes.

int **CO_FUTURE_DIVISION**

Este bit pode ser definido em *flags* para fazer com que o operador de divisão `/` seja interpretado como "divisão verdadeira" de acordo com a **PEP 238**.

CAPÍTULO 4

Contagem de Referências

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

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

Get the reference count of the Python object *o*.

Observe que o valor retornado pode não refletir realmente quantas referências ao objeto são realmente mantidas. Por exemplo, alguns objetos são *imortais* e têm uma refcount muito alta que não reflete o número real de referências. Consequentemente, não confie no valor retornado para ser preciso, exceto um valor de 0 ou 1.

Use the *Py_SET_REFCNT()* function to set an object reference count.

Alterado na versão 3.10: *Py_REFCNT()* is changed to the inline static function.

Alterado na versão 3.11: The parameter type is no longer `const` *PyObject*`*`.

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

Set the object *o* reference counter to *refcnt*.

Em uma construção do Python com threads threads livres, se *refcnt* for maior que `UINT32_MAX`, o objeto será transformado em *imortal*.

This function has no effect on *immortal* objects.

Adicionado na versão 3.9.

Alterado na versão 3.12: Immortal objects are not modified.

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

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

This function has no effect on *immortal* objects.

This function is usually used to convert a *borrowed reference* to a *strong reference* in-place. The *Py_NewRef()* function can be used to create a new *strong reference*.

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

The object must not be `NULL`; if you aren't sure that it isn't `NULL`, use *Py_XINCREF()*.

Do not expect this function to actually modify *o* in any way. For at least **some objects**, this function has no effect.

Alterado na versão 3.12: Immortal objects are not modified.

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

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

See also `Py_XNewRef()`.

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

Parte da ABI Estável *desde a versão 3.10.* Create a new *strong reference* to an object: call `Py_INCREF()` on *o* and return the object *o*.

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

The object *o* must not be `NULL`; use `Py_XNewRef()` if *o* can be `NULL`.

Por exemplo:

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

can be written as:

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

See also `Py_INCREF()`.

Adicionado na versão 3.10.

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

Parte da ABI Estável *desde a versão 3.10.* Similar to `Py_NewRef()`, but the object *o* can be NULL.

If the object *o* is `NULL`, the function just returns `NULL`.

Adicionado na versão 3.10.

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

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

This function has no effect on *immortal* objects.

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

This function is usually used to delete a *strong reference* before exiting its scope.

The object must not be `NULL`; if you aren't sure that it isn't `NULL`, use `Py_XDECREF()`.

Do not expect this function to actually modify *o* in any way. For at least **some objects**, this function has no effect.

> **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.

Alterado na versão 3.12: Immortal objects are not modified.

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

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

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

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

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

Alterado na versão 3.12: The macro argument is now only evaluated once. If the argument has side effects, these are no longer duplicated.

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

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

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

Parte da ABI Estável. Release a *strong reference* to object *o*. A function version of `Py_XDECREF()`. It can be used for runtime dynamic embedding of Python.

Py_SETREF(dst, src)

Macro safely releasing a *strong reference* to object *dst* and setting *dst* to *src*.

As in case of `Py_CLEAR()`, "the obvious" code can be deadly:

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

The safe way is:

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

That arranges to set *dst* to *src* _before_ releasing the reference to the old value of *dst*, so that any code triggered as a side-effect of *dst* getting torn down no longer believes *dst* points to a valid object.

Adicionado na versão 3.6.

Alterado na versão 3.12: The macro arguments are now only evaluated once. If an argument has side effects, these are no longer duplicated.

Py_XSETREF(dst, src)

Variant of `Py_SETREF` macro that uses `Py_XDECREF()` instead of `Py_DECREF()`.

Adicionado na versão 3.6.

Alterado na versão 3.12: The macro arguments are now only evaluated once. If an argument has side effects, these are no longer duplicated.

CAPÍTULO 5

Manipulando Exceções

As funções descritas nesse capítulo permitem você tratar e gerar exceções em Python. É importante entender alguns princípios básicos no tratamento de exceção no Python. Funciona de forma parecida com a variável POSIX `errno`: existe um indicador global (por thread) do último erro ocorrido. A maioria das funções da API C não limpa isso com êxito, mas indica a causa do erro na falha. A maioria das funções da API retorna um indicador de erro, geralmente, `NULL` se eles devem retornar um ponteiro, ou `-1` se retornarem um inteiro (exceção: as funções `PyArg_*` retornam `1` para sucesso e `0` para falha).

Concretamente, o indicador de erro consiste em três ponteiros de objeto: o tipo da exceção, o valor da exceção e o objeto de traceback. Qualquer um desses ponteiros pode ser `NULL` se não definido (embora algumas combinações sejam proibidas, por exemplo, você não pode ter um retorno não `NULL` se o tipo de exceção for `NULL`).

Quando uma função deve falhar porque devido à falha de alguma função que ela chamou, ela geralmente não define o indicador de erro; a função que ela chamou já o definiu. Ela é responsável por manipular o erro e limpar a exceção ou retornar após limpar todos os recursos que possui (como referências a objetos ou alocações de memória); ela *não* deve continuar normalmente se não estiver preparada para lidar com o erro. Se estiver retornando devido a um erro, é importante indicar ao chamador que um erro foi definido. Se o erro não for manipulado ou propagado com cuidado, chamadas adicionais para a API Python/C podem não se comportar conforme o esperado e podem falhar de maneiras misteriosas.

Nota

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

5.1 Impressão e limpeza

void **`PyErr_Clear`**()

Parte da ABI Estável. Limpe o indicador de erro. Se o indicador de erro não estiver definido, não haverá efeito.

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

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

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 variable `sys.last_exc` is set to the printed exception. For backwards compatibility, the deprecated variables `sys.last_type`, `sys.last_value` and `sys.last_traceback` are also set to the type, value and traceback of this exception, respectively.

Alterado na versão 3.12: The setting of `sys.last_exc` was added.

void **PyErr_Print**()

Parte da ABI Estável. Apelido para `PyErr_PrintEx(1)`.

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

Parte da ABI Estável. Chama `sys.unraisablehook()` usando a exceção atual e o argumento *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. If *obj* is `NULL`, only the traceback is printed.

Uma exceção deve ser definida ao chamar essa função.

Alterado na versão 3.4: Exibe o traceback (situação da pilha de execução). Exibe apenas o traceback se *obj* for `NULL`.

Alterado na versão 3.8: Utiliza `sys.unraisablehook()`.

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

Similar to *PyErr_WriteUnraisable()*, but the *format* and subsequent parameters help format the warning message; they have the same meaning and values as in *PyUnicode_FromFormat()*. `PyErr_WriteUnraisable(obj)` is roughly equivalent to `PyErr_FormatUnraisable("Exception ignored in: %R", obj)`. If *format* is `NULL`, only the traceback is printed.

Adicionado na versão 3.13.

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

Parte da ABI Estável *desde a versão 3.12*. Print the standard traceback display of `exc` to `sys.stderr`, including chained exceptions and notes.

Adicionado na versão 3.12.

5.2 Lançando exceções

Essas funções ajudam a definir o indicador de erro do thread. Por conveniência, algumas dessas funções sempre retornam um ponteiro `NULL` ao usar instrução com `return`.

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

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

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

Parte da ABI Estável. 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, ...)

Retorna valor: Sempre NULL. Parte da ABI Estável. This function sets the error indicator and returns `NULL`. *exception* should be a Python exception class. The *format* and subsequent parameters help format the error message; they have the same meaning and values as in *PyUnicode_FromFormat()*. *format* is an ASCII-encoded string.

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

Retorna valor: Sempre NULL. Parte da ABI Estável *desde a versão 3.5.* Igual a `PyErr_Format()`, mas usando o argumento `va_list` em vez de um número variável de argumentos.

Adicionado na versão 3.5.

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

Parte da ABI Estável. Isso é uma abreviação para `PyErr_SetObject(type, Py_None)`.

int **PyErr_BadArgument**()

Parte da ABI Estável. 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**()

Retorna valor: Sempre NULL. Parte da ABI Estável. Essa é uma abreviação para `PyErr_SetNone(PyExc_MemoryError)`; que retorna `NULL` para que uma função de alocação de objeto possa escrever `return PyErr_NoMemory();` quando ficar sem memória.

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

Retorna valor: Sempre NULL. Parte da ABI Estável. 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)

Retorna valor: Sempre NULL. Parte da ABI Estável. Similar to `PyErr_SetFromErrno()`, with the additional behavior that if *filenameObject* is not `NULL`, it is passed to the constructor of *type* as a third parameter. In the case of `OSError` exception, this is used to define the `filename` attribute of the exception instance.

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

Retorna valor: Sempre NULL. Parte da ABI Estável *desde a versão 3.7.* Similar to `PyErr_SetFromErrnoWithFilenameObject()`, but takes a second filename object, for raising errors when a function that takes two filenames fails.

Adicionado na versão 3.4.

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

Retorna valor: Sempre NULL. Parte da ABI Estável. Similar to `PyErr_SetFromErrnoWithFilenameObject()`, but the filename is given as a C string. *filename* is decoded from the *filesystem encoding and error handler*.

PyObject ***PyErr_SetFromWindowsErr**(int ierr)

Retorna valor: Sempre NULL. Parte da ABI Estável *on Windows desde a versão 3.7.* This is a convenience function to raise `OSError`. 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 `OSError` object with the `winerror` attribute set to the error code, the `strerror` attribute set to the corresponding error message (gotten from `FormatMessage()`), and then calls `PyErr_SetObject(PyExc_OSError, object)`. This function always returns `NULL`.

Disponibilidade: Windows.

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

Retorna valor: Sempre NULL. Parte da ABI Estável *on Windows desde a versão 3.7.* Similar to `PyErr_SetFromWindowsErr()`, with an additional parameter specifying the exception type to be raised.

Disponibilidade: Windows.

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

Retorna valor: Sempre NULL. Parte da ABI Estável *on Windows desde a versão 3.7.* Similar to `PyErr_SetFromWindowsErr()`, with the additional behavior that if *filename* is not `NULL`, it is decoded from the filesystem encoding (`os.fsdecode()`) and passed to the constructor of `OSError` as a third parameter to be used to define the `filename` attribute of the exception instance.

Disponibilidade: Windows.

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

Retorna valor: Sempre NULL. Parte da ABI Estável *on Windows desde a versão 3.7.* Similar to `PyErr_SetExcFromWindowsErr()`, with the additional behavior that if *filename* is not `NULL`, it is passed to the constructor of `OSError` as a third parameter to be used to define the `filename` attribute of the exception instance.

Disponibilidade: Windows.

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

Retorna valor: Sempre NULL. Parte da ABI Estável *on Windows desde a versão 3.7.* Similar à `PyErr_SetExcFromWindowsErrWithFilenameObject()`, mas aceita um segundo caminho do objeto.

Disponibilidade: Windows.

Adicionado na versão 3.4.

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

Retorna valor: Sempre NULL. Parte da ABI Estável *on Windows desde a versão 3.7.* Similar à `PyErr_SetFromWindowsErrWithFilename()`, com um parâmetro adicional especificando o tipo de exceção a ser gerado.

Disponibilidade: Windows.

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

Retorna valor: Sempre NULL. Parte da ABI Estável *desde a versão 3.7.* This is a convenience function to raise `ImportError`. *msg* will be set as the exception's message string. *name* and *path*, both of which can be `NULL`, will be set as the `ImportError`'s respective `name` and `path` attributes.

Adicionado na versão 3.3.

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

Retorna valor: Sempre NULL. Parte da ABI Estável *desde a versão 3.6.* Muito parecido com `PyErr_SetImportError()` mas a função permite especificar uma subclasse de `ImportError` para levantar uma exceção.

Adicionado na versão 3.6.

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

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

Adicionado na versão 3.4.

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

Parte da ABI Estável *desde a versão 3.7.* Like `PyErr_SyntaxLocationObject()`, but *filename* is a byte string decoded from the *filesystem encoding and error handler*.

Adicionado na versão 3.2.

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

Parte da ABI Estável. Like `PyErr_SyntaxLocationEx()`, but the *col_offset* parameter is omitted.

void **PyErr_BadInternalCall**()

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

5.3 Emitindo advertências

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

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

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

Warning categories must be subclasses of `PyExc_Warning`; `PyExc_Warning` is a subclass of `PyExc_Exception`; the default warning category is `PyExc_RuntimeWarning`. The standard Python warning categories are available as global variables whose names are enumerated at *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_WarnExplicitObject**(*PyObject* *category, *PyObject* *message, *PyObject* *filename, int lineno, *PyObject* *module, *PyObject* *registry)

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

Adicionado na versão 3.4.

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

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

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

Parte da ABI Estável. Function similar to *PyErr_WarnEx()*, but use *PyUnicode_FromFormat()* to format the warning message. *format* is an ASCII-encoded string.

Adicionado na versão 3.2.

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

Parte da ABI Estável *desde a versão 3.6.* Function similar to *PyErr_WarnFormat()*, but *category* is `ResourceWarning` and it passes *source* to `warnings.WarningMessage`.

Adicionado na versão 3.6.

5.4 Consultando o indicador de erro

PyObject ***PyErr_Occurred**()

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

The caller must hold the GIL.

> **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)

Parte da ABI Estável. 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)

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

PyObject ***PyErr_GetRaisedException**(void)

Retorna valor: Nova referência. *Parte da* ABI Estável *desde a versão 3.12.* Return the exception currently being raised, clearing the error indicator at the same time. Return `NULL` if the error indicator is not set.

This function is used by code that needs to catch exceptions, or code that needs to save and restore the error indicator temporarily.

Por exemplo:

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

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

   PyErr_SetRaisedException(exc);
}
```

> **Ver também**
>
> *PyErr_GetHandledException()*, to save the exception currently being handled.

Adicionado na versão 3.12.

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

Parte da ABI Estável *desde a versão 3.12.* Set *exc* as the exception currently being raised, clearing the existing exception if one is set.

> **Aviso**
>
> This call steals a reference to *exc*, which must be a valid exception.

Adicionado na versão 3.12.

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

Parte da ABI Estável. Descontinuado desde a versão 3.12: Use *PyErr_GetRaisedException()* instead.

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 legacy code that needs to catch exceptions or save and restore the error indicator temporarily.

Por exemplo:

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

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

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

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

Parte da ABI Estável. Descontinuado desde a versão 3.12: Use `PyErr_SetRaisedException()` instead.

Set the error indicator from the three objects, *type*, *value*, and *traceback*, clearing the existing exception if one is set. 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 legacy code that needs to save and restore the error indicator temporarily. Use `PyErr_Fetch()` to save the current error indicator.

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

Parte da ABI Estável. Descontinuado desde a versão 3.12: Use `PyErr_GetRaisedException()` instead, to avoid any possible de-normalization.

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.

Nota

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

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

PyObject ***PyErr_GetHandledException**(void)

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

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

Adicionado na versão 3.11.

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

Parte da ABI Estável *desde a versão 3.11.* Set the active exception, as known from `sys.exception()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. To clear the exception state, pass `NULL`.

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

Adicionado na versão 3.11.

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

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

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

Adicionado na versão 3.3.

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

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

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

Adicionado na versão 3.3.

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

5.5 Tratamento de sinal

int **PyErr_CheckSignals**()

Parte da ABI Estável. Essa função interage com o manipulador de sinais do Python.

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

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

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

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

> **Nota**
>
> The default Python signal handler for `SIGINT` raises the `KeyboardInterrupt` exception.

void **PyErr_SetInterrupt**()

Parte da ABI Estável. Simulate the effect of a `SIGINT` signal arriving. This is equivalent to `PyErr_SetInterruptEx(SIGINT)`.

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

int **PyErr_SetInterruptEx**(int signum)

Parte da ABI Estável *desde a versão 3.10.* Simulate the effect of a signal arriving. The next time *PyErr_CheckSignals()* is called, the Python signal handler for the given signal number will be called.

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

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

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

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

Adicionado na versão 3.10.

int **PySignal_SetWakeupFd**(int fd)

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

O valor `-1` desabilita o recurso; este é o estado inicial. Isso é equivalente à `signal.set_wakeup_fd()` em Python, mas sem nenhuma verificação de erro. *fd* deve ser um descritor de arquivo válido. A função só deve ser chamada a partir da thread principal.

Alterado na versão 3.5: No Windows, a função agora também suporta manipuladores de socket.

5.6 Classes de exceção

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

Retorna valor: Nova referência. Parte da ABI Estável. This utility function creates and returns a new exception class. The *name* argument must be the name of the new exception, a C string of the form `module.classname`. The *base* and *dict* arguments are normally `NULL`. This creates a class object derived from `Exception` (accessible in C as `PyExc_Exception`).

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

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

Retorna valor: Nova referência. Parte da ABI Estável. 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.

Adicionado na versão 3.2.

5.7 Objeto Exceção

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

Retorna valor: Nova referência. Parte da ABI Estável. Return the traceback associated with the exception as a new reference, as accessible from Python through the `__traceback__` attribute. If there is no traceback associated, this returns `NULL`.

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

Parte da ABI Estável. Defina o retorno traceback (situação da pilha de execução) associado à exceção como *tb*. Use `Py_None` para limpá-lo.

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

Retorna valor: Nova referência. Parte da ABI Estável. Return the context (another exception instance during whose handling *ex* was raised) associated with the exception as a new reference, as accessible from Python through the `__context__` attribute. If there is no context associated, this returns `NULL`.

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

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

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

Retorna valor: Nova referência. Parte da ABI Estável. Return the cause (either an exception instance, or `None`, set by `raise ... from ...`) associated with the exception as a new reference, as accessible from Python through the `__cause__` attribute.

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

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

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

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

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.12*. Return `args` of exception *ex*.

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

Parte da ABI Estável *desde a versão 3.12*. Set `args` of exception *ex* to *args*.

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

> Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Implement part of the interpreter's implementation of `except*`. *orig* is the original exception that was caught, and *excs* is the list of the exceptions that need to be raised. This list contains the unhandled part of *orig*, if any, as well as the exceptions that were raised from the `except*` clauses (so they have a different traceback from *orig*) and those that were reraised (and have the same traceback as *orig*). Return the `ExceptionGroup` that needs to be reraised in the end, or `None` if there is nothing to reraise.

Adicionado na versão 3.12.

5.8 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)

Retorna valor: Nova referência. Parte da ABI Estável. Create a `UnicodeDecodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.

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

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

Retorna valor: Nova referência. Parte da ABI Estável. 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 valor: Nova referência. Parte da ABI Estável. 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)

Parte da ABI Estável. Obtém o atributo *start* do objeto da exceção coloca-o em **start*. *start* não deve ser `NULL`. Retorna `0` se não der erro, `-1` caso dê erro.

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)

Parte da ABI Estável. 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)

Parte da ABI Estável. Obtenha o atributo *end* dado no objeto de exceção e coloque **end*. O *end* não deve ser `NULL`. Em caso de sucesso, retorna `0`, em caso de falha, retorna `-1`.

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

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

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

Parte da ABI Estável. 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 valor: Nova referência. Parte da ABI Estável. 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)

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

5.9 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). They are also not needed for *tp_call* implementations because the *call protocol* takes care of recursion handling.

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

Parte da ABI Estável *desde a versão 3.9.* 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()*. If this is the case, it sets a `MemoryError` and returns a nonzero value.

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

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

Alterado na versão 3.9: This function is now also available in the *limited API*.

void **Py_LeaveRecursiveCall**(void)

Parte da ABI Estável *desde a versão 3.9.* Ends a *Py_EnterRecursiveCall()*. Must be called once for each *successful* invocation of *Py_EnterRecursiveCall()*.

Alterado na versão 3.9: This function is now also available in the *limited API*.

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

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

Parte da ABI Estável. Chamado no início da implementação *tp_repr* para detectar ciclos.

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

A função retornará um inteiro negativo se o limite da recursão for atingido. Nesse caso a implementação *tp_repr* deverá, normalmente,. retornar `NULL`.

Caso contrário, a função retorna zero e a implementação *tp_repr* poderá continuar normalmente.

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

Parte da ABI Estável. Termina a *Py_ReprEnter()*. Deve ser chamado uma vez para cada chamada de *Py_ReprEnter()* que retorna zero.

5.10 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
PyExc_Exception	Exception	Página 66, 1
PyExc_ArithmeticError	ArithmeticError	Página 66, 1
PyExc_AssertionError	AssertionError	
PyExc_AttributeError	AttributeError	
PyExc_BlockingIOError	BlockingIOError	
PyExc_BrokenPipeError	BrokenPipeError	
PyExc_BufferError	BufferError	
PyExc_ChildProcessError	ChildProcessError	
PyExc_ConnectionAbortedErro	ConnectionAbortedError	
PyExc_ConnectionError	ConnectionError	
PyExc_ConnectionRefusedErro	ConnectionRefusedError	
PyExc_ConnectionResetError	ConnectionResetError	
PyExc_EOFError	EOFError	
PyExc_FileExistsError	FileExistsError	
PyExc_FileNotFoundError	FileNotFoundError	
PyExc_FloatingPointError	FloatingPointError	
PyExc_GeneratorExit	GeneratorExit	
PyExc_ImportError	ImportError	
PyExc_IndentationError	IndentationError	
PyExc_IndexError	IndexError	
PyExc_InterruptedError	InterruptedError	
PyExc_IsADirectoryError	IsADirectoryError	
PyExc_KeyError	KeyError	
PyExc_KeyboardInterrupt	KeyboardInterrupt	
PyExc_LookupError	LookupError	Página 66, 1
PyExc_MemoryError	MemoryError	
PyExc_ModuleNotFoundError	ModuleNotFoundError	
PyExc_NameError	NameError	
PyExc_NotADirectoryError	NotADirectoryError	
PyExc_NotImplementedError	NotImplementedError	
PyExc_OSError	OSError	Página 66, 1
PyExc_OverflowError	OverflowError	
PyExc_PermissionError	PermissionError	
PyExc_ProcessLookupError	ProcessLookupError	
PyExc_PythonFinalizationErr	PythonFinalizationError	
PyExc_RecursionError	RecursionError	
PyExc_ReferenceError	ReferenceError	
PyExc_RuntimeError	RuntimeError	
PyExc_StopAsyncIteration	StopAsyncIteration	
PyExc_StopIteration	StopIteration	
PyExc_SyntaxError	SyntaxError	
PyExc_SystemError	SystemError	
PyExc_SystemExit	SystemExit	
PyExc_TabError	TabError	
PyExc_TimeoutError	TimeoutError	
PyExc_TypeError	TypeError	
PyExc_UnboundLocalError	UnboundLocalError	
PyExc_UnicodeDecodeError	UnicodeDecodeError	

continua na próxima página

Tabela 1 – continuação da página anterior

Nome C	Nome Python	Notas
PyExc_UnicodeEncodeError	UnicodeEncodeError	
PyExc_UnicodeError	UnicodeError	
PyExc_UnicodeTranslateError	UnicodeTranslateError	
PyExc_ValueError	ValueError	
PyExc_ZeroDivisionError	ZeroDivisionError	

Adicionado na versão 3.3: PyExc_BlockingIOError, PyExc_BrokenPipeError, PyExc_ChildProcessError, PyExc_ConnectionError, PyExc_ConnectionAbortedError, PyExc_ConnectionRefusedError, PyExc_ConnectionResetError, PyExc_FileExistsError, PyExc_FileNotFoundError, PyExc_InterruptedError, PyExc_IsADirectoryError, PyExc_NotADirectoryError, PyExc_PermissionError, PyExc_ProcessLookupError e PyExc_TimeoutError foram introduzidos seguindo a **PEP 3151**.

Adicionado na versão 3.5: PyExc_StopAsyncIteration e PyExc_RecursionError.

Adicionado na versão 3.6: PyExc_ModuleNotFoundError.

Esses são os aliases de compatibilidade para PyExc_OSError:

Nome C	Notas
PyExc_EnvironmentError	
PyExc_IOError	
PyExc_WindowsError	[2]

Alterado na versão 3.3: Esses aliases costumavam ser tipos de exceção separados.

Notas:

5.11 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	[3]
PyExc_BytesWarning	BytesWarning	
PyExc_DeprecationWarning	DeprecationWarning	
PyExc_FutureWarning	FutureWarning	
PyExc_ImportWarning	ImportWarning	
PyExc_PendingDeprecationWarning	PendingDeprecationWarning	
PyExc_ResourceWarning	ResourceWarning	
PyExc_RuntimeWarning	RuntimeWarning	
PyExc_SyntaxWarning	SyntaxWarning	
PyExc_UnicodeWarning	UnicodeWarning	
PyExc_UserWarning	UserWarning	

Adicionado na versão 3.2: PyExc_ResourceWarning.

Notas:

[1] Esta é uma classe base para outras exceções padrão.
[2] Defina apenas no Windows; proteja o código que usa isso testando se a macro do pré-processador MS_WINDOWS está definida.
[3] Esta é uma classe base para outras categorias de aviso padrão.

CAPÍTULO 6

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.

6.1 Utilitários do sistema operacional

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

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.6.* Retorna a representação do sistema de arquivos para *path*. Se o objeto for um objeto `str` ou `bytes`, então uma nova *referência forte* é retornada. Se o objeto implementa a interface `os.PathLike`, então `__fspath__()` é retornado desde que seja um objeto `str` ou `bytes`. Caso contrário, `TypeError` é levantada e `NULL` é retornado.

Adicionado na versão 3.6.

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

Retorna verdadeiro (não zero) se o arquivo padrão de E/S *fp* com o nome *filename* for considerado interativo. Este é o caso dos arquivos para os quais `isatty(fileno(fp))` é verdade. Se *`PyConfig.interactive`* for não zero, esta função também retorna true se o ponteiro *filename* for `NULL` ou se o nome for igual a uma das strings `'<stdin>'` ou `'???'`.

Esta função não deve ser chamada antes da inicialização do Python.

void **PyOS_BeforeFork**()

Parte da ABI Estável *on platforms with fork() desde a versão 3.7.* Função para preparar algum estado interno antes de ser feito um fork do processo. Isso deve ser chamado antes de chamar `fork()` ou qualquer função semelhante que clone o processo atual. Disponível apenas em sistemas onde `fork()` é definido.

> **Aviso**
>
> A chamada C `fork()` só deve ser feita a partir da *thread "main"* (do *interpretador "main"*). O mesmo vale para `PyOS_BeforeFork()`.

Adicionado na versão 3.7.

void **PyOS_AfterFork_Parent**()

Parte da ABI Estável *on platforms with fork() desde a versão 3.7.* Função para atualizar algum estado interno depois de ser feito um fork do processo. Isso deve ser chamado a partir do processo pai depois de chamar `fork()` ou qualquer função semelhante que clone o processo atual, independentemente da clonagem do processo ter sido bem-sucedida ou não. Disponível apenas em sistemas onde `fork()` é definido.

Aviso

A chamada C `fork()` só deve ser feita a partir da *thread "main"* (do *interpretador "main"*). O mesmo vale para `PyOS_AfterFork_Parent()`.

Adicionado na versão 3.7.

void **PyOS_AfterFork_Child**()

Parte da ABI Estável *on platforms with fork() desde a versão 3.7.* Função para atualizar o estado interno do interpretador depois de ser feito um fork do processo. Isso deve ser chamado a partir do processo filho depois de chamar `fork()` ou qualquer função semelhante que clone o processo atual, se houver alguma chance do processo ter uma chamada de retorno para o interpretador Python. Disponível apenas em sistemas onde `fork()` é definido.

Aviso

A chamada C `fork()` só deve ser feita a partir da *thread "main"* (do *interpretador "main"*). O mesmo vale para `PyOS_AfterFork_Child()`.

Adicionado na versão 3.7.

Ver também

`os.register_at_fork()` permite registrar funções personalizadas do Python para serem chamadas por *`PyOS_BeforeFork()`*, *`PyOS_AfterFork_Parent()`* e *`PyOS_AfterFork_Child()`*.

void **PyOS_AfterFork**()

Parte da ABI Estável *on platforms with fork().* Função para atualizar algum estado interno após ser feito um fork de processo; isso deve ser chamado no novo processo se o interpretador do Python continuar a ser usado. Se um novo executável é carregado no novo processo, esta função não precisa ser chamada.

Descontinuado desde a versão 3.7: Esta função foi sucedida por *`PyOS_AfterFork_Child()`*.

int **PyOS_CheckStack**()

Parte da ABI Estável *on platforms with USE_STACKCHECK desde a versão 3.7.* Retorna verdadeiro quando o interpretador fica sem espaço na pilha. Esta é uma verificação confiável, mas só está disponível quando `USE_STACKCHECK` é definido (atualmente em certas versões do Windows usando o compilador Microsoft Visual C++). `USE_STACKCHECK` será definido automaticamente; você nunca deve alterar a definição em seu próprio código.

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

Parte da ABI Estável.

PyOS_sighandler_t **PyOS_getsig**(int i)

Parte da ABI Estável. Retorna o manipulador de sinal atual para o sinal *i*. Este é um invólucro fino em torno de `sigaction()` ou `signal()`. Não chame essas funções diretamente!

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

Parte da ABI Estável. Define o manipulador de sinal para o sinal *i* como *h*; retornar o manipulador de sinal antigo. Este é um invólucro fino em torno de `sigaction()` ou `signal()`. Não chame essas funções diretamente!

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

Parte da ABI Estável *desde a versão 3.7.*

> **Aviso**
>
> Esta função não deve ser chamada diretamente: use a API `PyConfig` com a função `PyConfig_SetBytesString()` que garante que *Python esteja pré-inicializado*.
>
> Esta função não deve ser chamada antes de *Python estar pré-inicializado* e para que a localidade LC_CTYPE seja configurada corretamente: consulte a função `Py_PreInitialize()`.

Decodifica uma string de bytes do *tratador de erros e codificação do sistema de arquivos*. Se o tratador de erros for o tratador de errors surrogateescape, bytes não decodificáveis são decodificados como caracteres no intervalo U+DC80..U+DCFF; e se uma string de bytes puder ser decodificada como um caractere substituto, os bytes são escapados usando o tratador de erros surrogateescape em vez de decodificá-los.

Retorna um ponteiro para uma string de caracteres largos recém-alocada, usa `PyMem_RawFree()` para liberar a memória. Se o tamanho não for `NULL`, escreve o número de caracteres largos excluindo o caractere nulo em `*size`

Retorna `NULL` em erro de decodificação ou erro de alocação de memória. Se *size* não for `NULL`, `*size` é definido como `(size_t)-1` em erro de memória ou definido como `(size_t)-2` em erro de decodificação.

tratador de erros e codificação do sistema de arquivos são selecionados por `PyConfig_Read()`: veja os membros `filesystem_encoding` e `filesystem_errors` de `PyConfig`.

Erros de decodificação nunca devem acontecer, a menos que haja um bug na biblioteca C.

Use a função `Py_EncodeLocale()` para codificar a string de caracteres de volta para uma string de bytes.

> **Ver também**
>
> As funções `PyUnicode_DecodeFSDefaultAndSize()` e `PyUnicode_DecodeLocaleAndSize()`.

Adicionado na versão 3.5.

Alterado na versão 3.7: A função agora usa a codificação UTF-8 no Modo UTF-8 do Python.

Alterado na versão 3.8: A função agora usa a codificação UTF-8 no Windows se `PyPreConfig.legacy_windows_fs_encoding` for zero;

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

Parte da ABI Estável *desde a versão 3.7.* Encode a wide character string to the *filesystem encoding and error handler*. If the error handler is surrogateescape error handler, surrogate characters in the range U+DC80..U+DCFF are converted to bytes 0x80..0xFF.

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

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

tratador de erros e codificação do sistema de arquivos são selecionados por `PyConfig_Read()`: veja os membros `filesystem_encoding` e `filesystem_errors` de `PyConfig`.

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

> **Aviso**
>
> Esta função não deve ser chamada antes de *Python estar pré-inicializado* e para que a localidade LC_CTYPE seja configurada corretamente: consulte a função `Py_PreInitialize()`.

> **Ver também**
>
> The *PyUnicode_EncodeFSDefault()* and *PyUnicode_EncodeLocale()* functions.

Adicionado na versão 3.5.

Alterado na versão 3.7: A função agora usa a codificação UTF-8 no Modo UTF-8 do Python.

Alterado na versão 3.8: The function now uses the UTF-8 encoding on Windows if *PyPreConfig.legacy_windows_fs_encoding* is zero.

6.2 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**(const char *name)

Retorna valor: Referência emprestada. Parte da ABI Estável. Return the object *name* from the `sys` module or `NULL` if it does not exist, without setting an exception.

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

Parte da ABI Estável. 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**()

Parte da ABI Estável. Reset `sys.warnoptions` to an empty list. This function may be called prior to *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15: Clear `sys.warnoptions` and `warnings.filters` instead.

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

Parte da ABI Estável. 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, ...)

Parte da ABI Estável. As *PySys_WriteStdout()*, but write to `sys.stderr` or *stderr* instead.

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

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

Adicionado na versão 3.2.

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

Parte da ABI Estável. As *PySys_FormatStdout()*, but write to `sys.stderr` or *stderr* instead.

Adicionado na versão 3.2.

PyObject ***PySys_GetXOptions**()

Retorna valor: Referência emprestada. Parte da ABI Estável *desde a versão 3.7.* Return the current dictionary of `-X` options, similarly to `sys._xoptions`. On error, `NULL` is returned and an exception is set.

Adicionado na versão 3.2.

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

Parte da ABI Estável *desde a versão 3.13.* Raise an auditing event with any active hooks. Return zero for success and non-zero with an exception set on failure.

The *event* string argument must not be *NULL*.

If any hooks have been added, *format* and other arguments will be used to construct a tuple to pass. Apart from `N`, the same format characters as used in *Py_BuildValue()* are available. If the built value is not a tuple, it will be added into a single-element tuple.

The `N` format option must not be used. It consumes a reference, but since there is no way to know whether arguments to this function will be consumed, using it may cause reference leaks.

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

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

See also *PySys_AuditTuple()*.

Adicionado na versão 3.8.

Alterado na versão 3.8.2: Require *Py_ssize_t* for `#` format characters. Previously, an unavoidable deprecation warning was raised.

int **PySys_AuditTuple**(const char *event, *PyObject* *args)

Parte da ABI Estável *desde a versão 3.13.* Similar to *PySys_Audit()*, but pass arguments as a Python object. *args* must be a `tuple`. To pass no arguments, *args* can be *NULL*.

Adicionado na versão 3.13.

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

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

O ponteiro *userData* é passado para a função de gancho. Como as funções de gancho podem ser chamadas de diferentes tempos de execução, esse ponteiro não deve se referir diretamente ao estado do Python.

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

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

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

If the interpreter is initialized, this function raises an auditing event `sys.addaudithook` with no arguments. If any existing hooks raise an exception derived from `Exception`, the new hook will not be added and the exception is cleared. As a result, callers cannot assume that their hook has been added unless they control all existing hooks.

typedef int (***Py_AuditHookFunction**)(const char *event, *PyObject* *args, void *userData)

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

Adicionado na versão 3.8.

6.3 Process Control

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

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

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

Alterado na versão 3.9: Log the function name automatically.

void **Py_Exit**(int status)

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

Alterado na versão 3.6: Errors from finalization no longer ignored.

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

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

Ver também

PyUnstable_AtExit() for passing a `void *data` argument.

6.4 Importando módulos

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

Retorna valor: Nova referência. Parte da ABI Estável. This is a wrapper around *PyImport_Import()* which takes a `const char*` as an argument instead of a *PyObject**.

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

Retorna valor: Nova referência. Parte da ABI Estável. Esta função é um alias descontinuado de *PyImport_ImportModule()*.

Alterado na versão 3.3: Essa função falhava em alguns casos, quando a trava de importação era mantida por outra thread. No Python 3.3, no entanto, o esquema de trava mudou passando a ser de travas por módulo na maior parte, dessa forma, o comportamento especial dessa função não é mais necessário.

Deprecated since version 3.13, will be removed in version 3.15: Use *PyImport_ImportModule()* instead.

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

Retorna valor: Nova referência. Importa um módulo. Isso é melhor descrito referindo-se à função embutida do Python `__import__()`.

O valor de retorno é uma nova referência ao módulo importado ou pacote de nível superior, ou `NULL` com uma exceção definida em caso de falha. Como para `__import__()`, o valor de retorno quando um submódulo de um pacote é solicitado é normalmente o pacote de nível superior, a menos que um *fromlist* não vazio seja fornecido.

As importações com falhas removem objetos incompletos do módulo, como em *PyImport_ImportModule()*.

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

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.7.* Importa um módulo. Isso é melhor descrito referindo-se à função embutida do Python `__import__()`, já que a função padrão `__import__()` chama essa função diretamente.

O valor de retorno é uma nova referência ao módulo importado ou pacote de nível superior, ou `NULL` com uma exceção definida em caso de falha. Como para `__import__()`, o valor de retorno quando um submódulo de um pacote é solicitado é normalmente o pacote de nível superior, a menos que um *fromlist* não vazio seja fornecido.

Adicionado na versão 3.3.

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

Retorna valor: Nova referência. Parte da ABI Estável. Semelhante para *PyImport_ImportModuleLevelObject()*, mas o nome é uma string codificada em UTF-8 de um objeto Unicode.

Alterado na versão 3.3: Valores negativos para *level* não são mais aceitos.

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

Retorna valor: Nova referência. Parte da ABI Estável. Essa é uma interface de alto nível que chama a atual "função auxiliar de importação" (com um *level* explícito de 0, significando importação absoluta). Invoca a função `__import__()` a partir de `__builtins__` da global atual. Isso significa que a importação é feita usando quaisquer extras de importação instalados no ambiente atual.

Esta função sempre usa importações absolutas.

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

Retorna valor: Nova referência. Parte da ABI Estável. Recarrega um módulo. Retorna uma nova referência para o módulo recarregado, ou `NULL` com uma exceção definida em caso de falha (o módulo ainda existe neste caso).

PyObject ***PyImport_AddModuleRef**(const char *name)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.13.* 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 a *strong reference* to the module on success. Return `NULL` with an exception set on failure.

The module name *name* is decoded from UTF-8.

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

Adicionado na versão 3.13.

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

Retorna valor: Referência emprestada. Parte da ABI Estável *desde a versão 3.7.* Similar to *PyImport_AddModuleRef()*, but return a *borrowed reference* and *name* is a Python `str` object.

Adicionado na versão 3.3.

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

Retorna valor: Referência emprestada. Parte da ABI Estável. Similar to *PyImport_AddModuleRef()*, but return a *borrowed reference*.

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

Retorna valor: Nova referência. Parte da ABI Estável. Given a module name (possibly of the form `package.module`) and a code object read from a Python bytecode file or obtained from the built-in function `compile()`, load the module. Return a new reference to the module object, or `NULL` with an exception set if an error

occurred. *name* is removed from `sys.modules` in error cases, even if *name* was already in `sys.modules` on entry to *PyImport_ExecCodeModule()*. Leaving incompletely initialized modules in `sys.modules` is dangerous, as imports of such modules have no way to know that the module object is an unknown (and probably damaged with respect to the module author's intents) state.

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

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

Esta função recarregará o módulo se este já tiver sido importado. Veja *PyImport_ReloadModule()* para forma desejada de recarregar um módulo.

Se *name* apontar para um nome pontilhado no formato de `package.module`, quaisquer estruturas de pacote ainda não criadas ainda não serão criadas.

Veja também *PyImport_ExecCodeModuleEx()* e *PyImport_ExecCodeModuleWithPathnames()*.

Alterado na versão 3.12: The setting of `__cached__` and `__loader__` is deprecated. See `ModuleSpec` for alternatives.

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

Retorna valor: Nova referência. *Parte da* ABI Estável. Like *PyImport_ExecCodeModule()*, but the `__file__` attribute of the module object is set to *pathname* if it is non-`NULL`.

Veja também *PyImport_ExecCodeModuleWithPathnames()*.

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

Retorna valor: Nova referência. *Parte da* ABI Estável *desde a versão 3.7.* Like *PyImport_ExecCodeModuleEx()*, but the `__cached__` attribute of the module object is set to *cpathname* if it is non-`NULL`. Of the three functions, this is the preferred one to use.

Adicionado na versão 3.3.

Alterado na versão 3.12: Setting `__cached__` is deprecated. See `ModuleSpec` for alternatives.

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

Retorna valor: Nova referência. *Parte da* ABI Estável. Como *PyImport_ExecCodeModuleObject()*, mas *name*, *pathname* e *cpathname* são strings codificadas em UTF-8. Também são feitas tentativas para descobrir qual valor para *pathname* deve ser de *cpathname* se o primeiro estiver definido como `NULL`.

Adicionado na versão 3.2.

Alterado na versão 3.3: Uses `imp.source_from_cache()` in calculating the source path if only the bytecode path is provided.

Alterado na versão 3.12: No longer uses the removed `imp` module.

long **PyImport_GetMagicNumber**()

Parte da ABI Estável. Retorna o número mágico para arquivos de bytecode Python (também conhecido como arquivo `.pyc`). O número mágico deve estar presente nos primeiros quatro bytes do arquivo bytecode, na ordem de bytes little-endian. Retorna `-1` em caso de erro.

Alterado na versão 3.3: Retorna o valor de `-1` no caso de falha.

const char ***PyImport_GetMagicTag**()

Parte da ABI Estável. Retorna a string de tag mágica para nomes de arquivo de bytecode Python no formato de **PEP 3147**. Tenha em mente que o valor em `sys.implementation.cache_tag` é autoritativo e deve ser usado no lugar desta função.

Adicionado na versão 3.2.

PyObject ***PyImport_GetModuleDict**()

Retorna valor: Referência emprestada. Parte da ABI Estável. Retorna o dicionário usado para a administração do módulo (também conhecido como `sys.modules`). Observe que esta é uma variável por interpretador.

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

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.8.* Retorna o módulo já importado com o nome fornecido. Se o módulo ainda não foi importado, retorna `NULL`, mas não define um erro. Retorna `NULL` e define um erro se a pesquisa falhar.

Adicionado na versão 3.7.

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

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

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

Parte da ABI Estável *desde a versão 3.7.* Carrega um módulo congelado chamado *name*. Retorna `1` para sucesso, `0` se o módulo não for encontrado e `-1` com uma exceção definida se a inicialização falhar. Para acessar o módulo importado em um carregamento bem-sucedido, use *`PyImport_ImportModule()`*. (Observe o nome incorreto — esta função recarregaria o módulo se ele já tivesse sido importado.)

Adicionado na versão 3.3.

Alterado na versão 3.4: O atributo `__file__` não está mais definido no módulo.

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

Parte da ABI Estável. Semelhante a *`PyImport_ImportFrozenModuleObject()`*, mas o nome é uma string codificada em UTF-8 em vez de um objeto Unicode.

struct **_frozen**

Esta é a definição do tipo de estrutura para descritores de módulo congelados, conforme gerado pelo utilitário **freeze** (veja `Tools/freeze/` na distribuição fonte do Python). Sua definição, encontrada em `Include/import.h`, é:

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

Alterado na versão 3.11: O novo campo `is_package` indica se o módulo é um pacote ou não. Isso substitui a configuração do campo `size` para um valor negativo.

const struct *_frozen* ***PyImport_FrozenModules**

Este ponteiro é inicializado para apontar para um vetor de registros de *`_frozen`*, terminado por um cujos membros são todos `NULL` ou zero. Quando um módulo congelado é importado, ele é pesquisado nesta tabela. O código de terceiros pode fazer truques com isso para fornecer uma coleção criada dinamicamente de módulos congelados.

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

Parte da ABI Estável. Adiciona um único módulo à tabela existente de módulos embutidos. Este é um invólucro prático em torno de *`PyImport_ExtendInittab()`*, retornando `-1` se a tabela não puder ser estendida. O novo módulo pode ser importado pelo nome *name* e usa a função *initfunc* como a função de inicialização chamada na primeira tentativa de importação. Deve ser chamado antes de *`Py_Initialize()`*.

struct **_inittab**

Structure describing a single entry in the list of built-in modules. Programs which embed Python may use an

array of these structures in conjunction with *PyImport_ExtendInittab()* to provide additional built-in modules. The structure consists of two members:

const char ***name**

The module name, as an ASCII encoded string.

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

Initialization function for a module built into the interpreter.

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

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

Se Python é inicializado várias vezes, *PyImport_AppendInittab()* ou *PyImport_ExtendInittab()* devem ser chamados antes de cada inicialização do Python.

6.5 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.

O módulo oferece suporte a duas versões do formato de dados: a versão 0 é a versão histórica, a versão 1 compartilha strings internas no arquivo e após a desserialização. A versão 2 usa um formato binário para números de ponto flutuante. `Py_MARSHAL_VERSION` indica o formato do arquivo atual (atualmente 2).

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

Aplica *marshalling* em um inteiro `long`, *value*, para *file*. Isso escreverá apenas os 32 bits menos significativos de *value*; independentemente do tamanho do tipo nativo `long`. *version* indica o formato do arquivo.

Esta função pode falhar, caso em que define o indicador de erro. Use *PyErr_Occurred()* para verificar isso.

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

Aplica *marshalling* em um objeto Python, *value*, para *file*. *version* indica o formato do arquivo.

Esta função pode falhar, caso em que define o indicador de erro. Use *PyErr_Occurred()* para verificar isso.

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

Retorna valor: Nova referência. Retorna um objeto de bytes que contém a representação pós-*marshalling* de *value*. *version* indica o formato do arquivo.

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

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`.

Em caso de erro, define a exceção apropriada (`EOFError`) e retorna `-1`.

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`.

Em caso de erro, define a exceção apropriada (`EOFError`) e retorna `-1`.

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

Retorna valor: Nova referência. Retorna um objeto Python do fluxo de dados em um `FILE*` aberto para leitura.

Em caso de erro, define a exceção apropriada (`EOFError`, `ValueError` ou `TypeError`) e retorna `NULL`.

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

Retorna valor: Nova referência. Retorna um objeto Python do fluxo de dados em um `FILE*` aberto para leitura. Diferentemente de *PyMarshal_ReadObjectFromFile()*, essa função presume que nenhum objeto adicional será lido do arquivo, permitindo que ela carregue agressivamente os dados do arquivo na memória, para que a desserialização possa operar a partir de dados na memória em vez de ler um byte por vez do arquivo. Use essas variantes apenas se tiver certeza de que não estará lendo mais nada do arquivo.

Em caso de erro, define a exceção apropriada (`EOFError`, `ValueError` ou `TypeError`) e retorna `NULL`.

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

Retorna valor: Nova referência. Retorna um objeto Python do fluxo de dados em um buffer de bytes contendo *len* bytes apontados por *data*.

Em caso de erro, define a exceção apropriada (`EOFError`, `ValueError` ou `TypeError`) e retorna `NULL`.

6.6 Análise de argumentos e construção de valores

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

As três primeiras funções descritas, *PyArg_ParseTuple()*, *PyArg_ParseTupleAndKeywords()*, e *PyArg_Parse()*, todas usam a *string de formatação* que informam à função sobre os argumentos esperados. As strings de formato usam a mesma sintaxe para cada uma dessas funções.

6.6.1 Análise de argumentos

Uma string de formato 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 poucas 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, a forma citada 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.

Strings e buffers

> **Nota**
>
> No Python 3.12 e anteriores, a macro `PY_SSIZE_T_CLEAN` deve ser definida antes da inclusão de `Python.h` para usar todas as variantes no formato `#` (`s#`, `y#`, etc.) explicadas abaixo. Isso não é necessário no Python 3.13 e posteriores.

Esses formatos permitem acessar um objeto como um pedaço contíguo de memória. Você não precisa fornecer armazenamento bruto para a área de unicode ou bytes retornada.

Salvo indicação em contrário, os buffers não são terminados em NUL.

Existem três maneiras pelas quais strings e buffers podem ser convertidos em C:

- Formatos como `y*` e `s*` estão dentro de uma estrutura *Py_buffer*. Isso bloqueia o buffer subjacente para que o chamador possa posteriormente usar o buffer, mesmo dentro de um bloco *Py_BEGIN_ALLOW_THREADS* sem que haja o risco de que dados mutáveis sejam redimensionados ou destruídos. Dessa forma, **você precisa chamar** *PyBuffer_Release()* depois de ter concluído o processamento de dados (ou em qualquer caso de interrupção precoce).
- Os formatos `es`, `es#`, `et` e `et#` alocam o buffer resultante. **Você precisa chamar** *PyMem_Free()* depois de ter concluído o processamento de dados (ou em qualquer caso de interrupção precoce).
- Outros formatos usam um `str` ou um *objeto byte ou similar* somente leitura, como `bytes`, e fornecem um ponteiro `const char *` para seu buffer. Nesse caso, o buffer é "emprestado": ele é gerenciado pelo ob-

jeto Python correspondente e compartilha o tempo de vida desse objeto. Você mesmo não precisará liberar nenhuma memória.

Para garantir que o buffer subjacente possa ser emprestado com segurança, o campo *PyBufferProcs.bf_releasebuffer* do objeto deve ser `NULL`. Isso não permite objetos mutáveis comuns, como `bytearray`, mas também alguns objetos somente leitura, como `memoryview` ou `bytes`.

Além desse requisito `bf_releasebuffer`, não há nenhuma verificação para saber se o objeto de entrada é imutável (por exemplo, se ele atenderia a uma solicitação de um buffer gravável ou se outro thread pode alterar os dados).

`s` (`str`) [const char *]
Converte um objeto Unicode para um ponteiro em C para uma string. Um ponteiro para uma string existente é armazenado na variável do ponteiro do caractere cujo o endereço que você está passando. A string em C é terminada em NULO. A string no Python não deve conter pontos de código nulo embutidos; se isso acontecer, uma exceção `ValueError` é levantada. Objetos Unicode são convertidos para strings em C usando a codificação `'utf-8'`. Se essa conversão falhar, uma exceção `UnicodeError` é levantada.

> **Nota**
>
> Esse formato não aceita *objetos byte ou similar*. Se você quer aceitar caminhos de arquivos do sistema e convertê-los para strings em C, é preferível que use o formato `O&` com *PyUnicode_FSConverter()* como *conversor*.

Alterado na versão 3.5: Anteriormente, a exceção `TypeError` era levantada quando pontos de código nulo embutidos em string Python eram encontrados.

`s*` (`str` ou *objeto byte ou similar*) [Py_buffer]
Esse formato aceita tanto objetos Unicode quanto objetos byte ou similar. Preenche uma estrutura *Py_buffer* fornecida pelo chamador. Nesse caso, a string em C resultante pode conter bytes NUL embutidos. Objetos Unicode são convertidos para strings em C usando codificação `'utf-8'`.

`s#` (`str`, *objeto byte ou similar* somente leitura) [const char *, *Py_ssize_t*]
Como `s*`, exceto que não fornece um *buffer emprestado*. O resultado é armazenado em duas variáveis em C, a primeira é um ponteiro para uma string em C, a segunda é o tamanho. A string pode conter bytes nulos embutidos. Objetos Unicode são convertidos para strings em C usando codificação `'utf-8'`.

`z` (`str` ou `None`) [const char *]
Como `s`, mas o objeto Python também pode ser `None`, nesse caso o ponteiro C é definido como `NULL`.

`z*` (`str`, *objeto byte ou similar* ou `None`) [Py_buffer]
Como `s*`, mas o objeto Python também pode ser `None`, nesse caso o membro `buf` da estrutura *Py_buffer* é definido como `NULL`.

`z#` (`str`, *objeto byte ou similar* somente leitura ou `None`) [const char *, *Py_ssize_t*]
Como `s#`, mas o objeto Python também pode ser `None`, nesse caso o ponteiro C é definido como `NULL`.

`y` (*objeto byte ou similar* somente leitura) [const char *]
Este formato converte um objeto byte ou similar para um ponteiro C para uma string de caracteres *emprestada*; não aceita objetos Unicode. O buffer de bytes não pode conter bytes nulos embutidos; se isso ocorrer uma exceção `ValueError` será levantada.

Alterado na versão 3.5: Anteriormente, a exceção `TypeError` era levantada quando pontos de código nulo embutidos em string Python eram encontrados no buffer de bytes.

`y*` (*objeto byte ou similar*) [Py_buffer]
Esta variante em `s*` não aceita objetos unicode, apenas objetos byte ou similar. **Esta é a maneira recomendada para aceitar dados binários.**

`y#` (*objeto byte ou similar* somente leitura) [const char *, *Py_ssize_t*]
Esta variação de `s#` não aceita objetos Unicode, apenas objetos byte ou similar.

S (bytes) [PyBytesObject *]
Exige que o objeto Python seja um objeto `bytes`, sem tentar nenhuma conversão. Levanta `TypeError` se o objeto não for um objeto byte. A variável C pode ser declarada como *PyObject**.

Y (bytearray) [PyByteArrayObject *]
Exige que o objeto Python seja um objeto `bytearray`, sem aceitar qualquer conversão. Levanta `TypeError` se o objeto não é um objeto `bytearray`. A variável C apenas pode ser declarada como *PyObject**.

U (str) [PyObject *]
Exige que o objeto python seja um objeto Unicode, sem tentar alguma conversão. Levanta `TypeError` se o objeto não for um objeto Unicode. A variável C deve ser declarada como *PyObject**.

w* (*objeto byte ou similar* de leitura e escrita) [Py_buffer]
Este formato aceita qualquer objeto que implemente a interface do buffer de leitura e escrita. Ele preenche uma estrutura *Py_buffer* fornecida pelo chamador. O buffer pode conter bytes nulos incorporados. O chamador deve chamar *PyBuffer_Release()* quando isso for feito com o buffer.

es (str) [const char *encoding, char **buffer]
Esta variante em `s` é utilizada para codificação do Unicode em um buffer de caracteres. Ele só funciona para dados codificados sem NUL bytes incorporados.

Este formato exige dois argumentos. O primeiro é usado apenas como entrada e deve ser a `const char*` que aponta para o nome de uma codificação como uma string terminada em NUL ou `NULL`, nesse caso a codificação `'utf-8'` é usada. Uma exceção é levantada se a codificação nomeada não for conhecida pelo Python. O segundo argumento deve ser um `char**`; o valor do ponteiro a que ele faz referência será definido como um buffer com o conteúdo do texto do argumento. O texto será codificado na codificação especificada pelo primeiro argumento.

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 (str, bytes ou bytearray) [const char *encoding, char **buffer]
O mesmo que `es`, exceto que os objetos strings de bytes são passados sem os recodificar. Em vez disso, a implementação presume que o objeto string de bytes usa a codificação passada como parâmetro.

es# (str) [const char *encoding, char **buffer, *Py_ssize_t* *buffer_length]
Essa variante em `s#` é usada para codificar Unicode em um buffer de caracteres. Diferente do formato `es`, essa variante permite a entrada de dados que contêm caracteres NUL.

Exige três argumentos. O primeiro é usado apenas como entrada e deve ser a `const char*` que aponta para o nome de uma codificação como uma string terminada em NUL ou `NULL`, nesse caso a codificação `'utf-8'` é usada. Uma exceção será gerada se a codificação nomeada não for conhecida pelo Python. O segundo argumento deve ser um `char**`; o valor do ponteiro a que ele faz referência será definido como um buffer com o conteúdo do texto do argumento. O texto será codificado na codificação especificada pelo primeiro argumento. O terceiro argumento deve ser um ponteiro para um número inteiro; o número inteiro referenciado será definido como o número de bytes no buffer de saída.

Há dois modos de operação:

Se **buffer* apontar um ponteiro `NULL`, a função irá alocar um buffer do tamanho necessário, copiar os dados codificados para dentro desse buffer e configurar **buffer* para referenciar o novo armazenamento alocado. O chamador é responsável por chamar *PyMem_Free()* para liberar o buffer alocado após o uso.

Se **buffer* apontar para um ponteiro que não seja `NULL` (um buffer já alocado), *PyArg_ParseTuple()* irá usar essa localização como buffer e interpretar o valor inicial de **buffer_length* como sendo o tamanho do buffer. Depois ela vai copiar os dados codificados para dentro do buffer e terminá-lo com NUL. Se o buffer não for suficientemente grande, um `ValueError` será definido.

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

et# (str, bytes ou bytearray) [const char *encoding, char **buffer, *Py_ssize_t* *buffer_length]
O mesmo que `es#`, exceto que os objetos strings de bytes são passados sem que sejam recodificados. Em vez disso, a implementação presume que o objeto string de bytes usa a codificação passada como parâmetro.

Alterado na versão 3.12: `u`, `u#`, `Z` e `Z#` foram removidos porque usavam uma representação herdada de `Py_UNICODE*`.

Números

Esses formatos permitem representar números Python ou caracteres únicos como números C. Formatos que exigem `int`, `float` ou `complex` também podem usar os métodos especiais correspondentes `__index__()`, `__float__()` ou `__complex__()` para converter o objeto Python para o tipo necessário.

Para formatos inteiros com sinal, `OverflowError` é levantada se o valor estiver fora do intervalo para o tipo C. Para formatos inteiros não assinados, nenhuma verificação de intervalo é feita — os bits mais significativos são silenciosamente truncados quando o campo de recebimento é muito pequeno para receber o valor.

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

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

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

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

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

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

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

`k` (`int`) [unsigned long]
Converte um inteiro Python para um `unsigned long` do C sem verificação de estouro.

`L` (`int`) [longo longo]
Converte um inteiro Python para um `long long` do C.

`K` (`int`) [unsigned long long]
Converte um inteiro Python para um `unsigned long long` do C sem verificação de estouro.

`n` (`int`) [*`Py_ssize_t`*]
Converte um inteiro Python para um *`Py_ssize_t`* do C.

`c` (`bytes` ou `bytearray` de comprimento 1) [char]
Converte um byte Python, representado com um objeto `byte` ou `bytearray` de comprimento 1, para um `char` do C.

Alterado na versão 3.3: Permite objetos `bytearray`.

`C` (`str` de comprimento 1) [int]
Converte um caractere Python, representado como uma `str` objeto de comprimento 1, para um `int` do C

`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`*

Outros objetos

`O` (objeto) [PyObject*]
Armazena um objeto Python (sem qualquer conversão) em um ponteiro de objeto C. O programa C então recebe o objeto real que foi passado. Uma nova *referência forte* ao objeto não é criado (isto é sua contagem de referências não é aumentada). O ponteiro armazenado não é `NULL`.

`O!` (objeto) [*typeobject*, PyObject *]
Armazena um objeto Python em um ponteiro de objeto C. Isso é similar a `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 levantada.

`O&` (objeto) [*converter*, *address*]
Converte um objeto Python em uma variável C através de uma função *converter*. Isso leva dois argumentos: o primeiro é a função, o segundo é o endereço da variável C (de tipo arbitrário), convertendo para `void*`. A função *converter* por sua vez, é chamada da seguinte maneira:

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

onde *object* é o objeto Python a ser convertido e *address* é o argumento `void*` que foi passado para a função `PyArg_Parse*`. O *status* retornado deve ser `1` para uma conversão bem-sucedida e `0` se a conversão falhar. Quando a conversão falha, a função *converter* deve levantar uma exceção e deixar o conteúdo de *address* inalterado. Se o *converter* retornar `Py_CLEANUP_SUPPORTED`, ele poderá ser chamado uma segunda vez se a análise do argumento eventualmente falhar, dando ao conversor a chance de liberar qualquer memória que já havia alocado. Nesta segunda chamada, o parâmetro *object* será `NULL`; *address* terá o mesmo valor que na chamada original.

Exemplos de conversores: *`PyUnicode_FSConverter()`* e *`PyUnicode_FSDecoder()`*.

Alterado na versão 3.1: `Py_CLEANUP_SUPPORTED` foi adicionado.

`p` (`bool`) [int]
Testa o valor transmitido para a verdade (um booleano **p**redicado) e converte o resultado em seu valor inteiro C verdadeiro/falso equivalente. Define o int como `1` se a expressão for verdadeira e `0` se for falsa. Isso aceita qualquer valor válido do Python. Veja truth para obter mais informações sobre como o Python testa valores para a verdade.

Adicionado na versão 3.3.

`(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.

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

`|`
Indica que os argumentos restantes na lista de argumentos do Python são opcionais. As variáveis C correspondentes a argumentos opcionais devem ser inicializadas para seus valores padrão — quando um argumento opcional não é especificado, *`PyArg_ParseTuple()`* não toca no conteúdo da(s) variável(eis) C correspondente(s).

`$`
`PyArg_ParseTupleAndKeywords()` apenas: Indica que os argumentos restantes na lista de argumentos do Python são somente-nomeados. Atualmente, todos os argumentos somente-nomeados devem ser também argumentos opcionais, então `|` deve sempre ser especificado antes de `$` na string de formatação.

Adicionado na versão 3.3.

`:`
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 libera-as (isto é, 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.

Funções da API

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

Parte da ABI Estável. 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)

Parte da ABI Estável. 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 *const *keywords, ...)

Parte da ABI Estável. Analisa os parâmetros de uma função que aceita tanto parâmetros posicionais quando parâmetros nomeados e os converte em variáveis locais. O argumento *keywords* é um array terminado em `NULL` de nomes de parâmetros nomeados especificados como strings C codificadas em ASCII ou UTF-8 e terminadas com nulo. Nomes vazios representam *parâmetros somente-posicional*. Retorna verdadeiro em caso de sucesso; em caso de falha, retorna falso e levanta a exceção apropriada.

> **Nota**
>
> A declaração do parâmetro *keywords* é `char *const*` em C e `const char *const*` em C++. Isso pode ser substituído com a macro *PY_CXX_CONST*.

Alterado na versão 3.6: Adicionado suporte para *positional-only parameters*.

Alterado na versão 3.13: O parâmetro *keywords* é agora do tipo `char *const*` em C e `const char *const*` em C++, no lugar de `char**`. Adicionado suporte para nomes de parâmetros nomeados não-ASCII.

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

Parte da ABI Estável. Idêntico a *PyArg_ParseTupleAndKeywords()*, exceto que aceita uma va_list ao invés de um número variável de argumentos.

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

Parte da ABI Estável. Garante que as chaves no dicionário de argumento de palavras reservadas são strings. Isso só é necessário se *PyArg_ParseTupleAndKeywords()* não é usado, já que o último já faz essa checagem.

Adicionado na versão 3.2.

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

Parte da ABI Estável. Analisa o parâmetro de uma função que recebe um único parâmetro posicional e o converte em uma variável local. Retorna verdadeiro em caso de sucesso; em caso de falha, retorna falso e levanta a exceção apropriada.

Exemplo:

```
// Função usando a convenção de chamada ao método METH_O
static PyObject*
my_function(PyObject *module, PyObject *arg)
{
    int value;
    if (!PyArg_Parse(arg, "i:my_function", &value)) {
        return NULL;
    }
    // ... usar valor ...
}
```

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

Parte da ABI Estável. Uma forma mais simples de recuperação de parâmetro que não usa uma string de formato para especificar os tipos de argumentos. Funções que usam este 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*; deve realmente ser uma tupla. O comprimento da tupla deve ser de pelo menos *min* e não mais do que *max*; *min* e *max* podem ser iguais. Argumentos adicionais devem ser passados para a função, cada um dos quais deve ser um ponteiro para uma variável *PyObject**; eles serão preenchidos com os valores de *args*; eles conterão *referências emprestadas*. As variáveis que correspondem a parâmetros opcionais não fornecidos por *args* não serão preenchidas; estes devem ser inicializados pelo chamador. Esta função retorna verdadeiro em caso de sucesso e falso se *args* não for uma tupla ou contiver o número incorreto de elementos; uma exceção será definida se houver 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)
```

PY_CXX_CONST

O valor para ser inserido, se houver, antes de `char *const*` na declaração do parâmetro *keywords* das funções *PyArg_ParseTupleAndKeywords()* e *PyArg_VaParseTupleAndKeywords()*. Por padrão, é vazio para C e `const` for C++ (`const char *const*`). Para substituir, defina o valor desejado antes de incluir `Python.h`.

Adicionado na versão 3.13.

6.6.2 Construindo valores

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

Retorna valor: Nova referência. Parte da ABI Estável. Cria um novo valor baseado em uma string de formatação similar àquelas aceitas pela família de funções `PyArg_Parse*` e uma sequência de valores. Retorna o valor ou `NULL` em caso de erro; uma exceção será levantada se `NULL` for retornado.

`Py_BuildValue()` não constrói sempre uma tupla. Ela constrói uma tupla apenas se a sua string de formatação contém duas ou mais unidades de formatação. Se a string de formatação estiver vazia, ela retorna `None`; se ela contém exatamente uma unidade de formatação, ela retorna qualquer que seja o objeto que for descrito pela unidade de formatação. Para forçar ela a retornar uma tupla de tamanho 0 ou um, use parênteses na string de formatação.

Quando buffers de memória são passados como parâmetros para fornecer dados para construir objetos, como nos formatos `s` e `s#`, os dados necessários são copiados. Buffers fornecidos pelo chamador nunca são referenciados pelos objetos criados por *`Py_BuildValue()`*. Em outras palavras, se o seu código invoca `malloc()` e passa a memória alocada para *`Py_BuildValue()`*, seu código é responsável por chamar `free()` para aquela memória uma vez que *`Py_BuildValue()`* tiver retornado.

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` (`str` ou `None`) [const char *]
Converte uma string C terminada em NULL em um objeto Python `str` usando codificação `'utf-8'`. Se o ponteiro da string C é `NULL`, `None` é usado.

`s#` (`str` ou `None`) [const char *, *`Py_ssize_t`*]
Converte uma string C e seu comprimento em um objeto Python `str` usando a codificação `'utf-8'`. Se o ponteiro da string C é `NULL`, o comprimento é ignorado e `None` é retornado.

`y` (`bytes`) [const char *]
Isso converte uma string C para um objeto Python `bytes`. Se o ponteiro da string C é `NULL`, `None` é retornado.

`y#` (`bytes`) [const char *, *`Py_ssize_t`*]
Isso converte uma string C e seu comprimento para um objeto Python. Se o ponteiro da string C é `NULL`, `None` é retornado.

`z` (`str` ou `None`) [const char *]
O mesmo de `s`.

`z#` (`str` ou `None`) [const char *, *`Py_ssize_t`*]
O mesmo de `s#`.

`u` (`str`) [const wchar_t *]
Converte um buffer terminado por null `wchar_t` de dados Unicode (UTF-16 ou UCS-4) para um objeto Python Unicode. Se o ponteiro do buffer Unicode é `NULL`, `None` é retornado.

`u#` (`str`) [const wchar_t *, *`Py_ssize_t`*]
Converte um buffer de dados Unicode (UTF-17 ou UCS-4) e seu comprimento em um objeto Python Unicode. Se o ponteiro do buffer Unicode é `NULL`, o comprimento é ignorado e `None` é retornado.

`U` (`str` ou `None`) [const char *]
O mesmo de `s`.

`U#` (`str` ou `None`) [const char *, *`Py_ssize_t`*]
O mesmo de `s#`.

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

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

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

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

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

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

I (int) [unsigned int]
Converte um `unsigned int` do C em um objeto inteiro do Python.

k (int) [unsigned long]
Converte um `unsigned long` do C em um objeto inteiro do Python.

L (int) [longo longo]
Converte um `long long` do C em um objeto inteiro do Python.

K (int) [unsigned long long]
Converte um `unsigned long long` do C em um objeto inteiro do Python.

n (int) [*Py_ssize_t*]
Converte um *Py_ssize_t* do C em um objeto inteiro do Python.

c (bytes de comprimento 1) [char]
Converte um `int` representando um byte do C em um objeto `bytes` de comprimento 1 do Python.

C (str de comprimento 1) [int]
Converte um `int` representando um caractere do C em um objeto `str` de comprimento 1 do Python.

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

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

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

O (objeto) [PyObject*]
Passa um objeto Python intocado, mas cria uma nova *referência forte* a ele (isto é, sua contagem de referências é incrementada por um). Se o objeto passado é um ponteiro `NULL`, presume-se que isso foi causado porque a chamada que produziu o argumento encontrou um erro e definiu uma exceção. Portanto, *Py_BuildValue()* irá retornar `NULL` mas não irá levantar uma exceção. Se nenhuma exceção foi levantada ainda, `SystemError` é definida.

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

N (objeto) [PyObject *]
O mesmo que `O`, exceto que não cria uma nova *referência forte*. Útil quando o objeto é criado por uma chamada a um construtor de objeto na lista de argumento.

O& (objeto) [*converter*, *anything*]
Converte *anything* para um objeto Python através de uma função *converter*. A função é chamada com *anything* (que deve ser compatível com o `void*`) como argumento e deve retornar um "novo" objeto Python, ou `NULL` se um erro ocorreu.

(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} (dict) [*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.

Se existir um erro na string de formatação, a exceção `SystemError` é definida e `NULL` é retornado.

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

Retorna valor: Nova referência. Parte da ABI Estável. Idêntico a *`Py_BuildValue()`*, exceto que aceita uma va_list ao invés de um número variável de argumentos.

6.7 Conversão e formação de strings

Funções para conversão de números e saída formatada de strings.

int **PyOS_snprintf**(char *str, size_t size, const char *format, ...)

Parte da ABI Estável. Saída não superior a *size* bytes para *str* de acordo com a string de formato *format* e os argumentos extras. Veja a página man do Unix *`snprintf(3)`*.

int **PyOS_vsnprintf**(char *str, size_t size, const char *format, va_list va)

Parte da ABI Estável. Saída não superior a *size* bytes para *str* de acordo com o string de formato *format* e a variável argumento de lista *va*. Página man do Unix *`vsnprintf(3)`*.

`PyOS_snprintf()` e *`PyOS_vsnprintf()`* envolvem as funções `snprintf()` e `vsnprintf()` da biblioteca Standard C. Seu objetivo é garantir um comportamento consistente em casos extremos, o que as funções do Standard C não garantem.

Os invólucros garantem que `str[size-1]` seja sempre `'\0'` no retorno. Eles nunca escrevem mais do que *size* bytes (incluindo o `'\0'` ao final) em str. Ambas as funções exigem que `str != NULL`, `size > 0`, `format != NULL` e `size < INT_MAX`. Note que isso significa que não há equivalente ao `n = snprintf(NULL, 0, ...)` do C99 que determinaria o tamanho de buffer necessário.

O valor de retorno (*rv*) para essas funções deve ser interpretado da seguinte forma:

- Quando `0 <= rv < size`, a conversão de saída foi bem-sucedida e os caracteres de *rv* foram escritos em *str* (excluindo o `'\0'` byte em `str[rv]`).
- Quando `rv >= size`, a conversão de saída foi truncada e um buffer com `rv + 1` bytes teria sido necessário para ter sucesso. `str[size-1]` é `'\0'` neste caso.
- Quando `rv < 0`, "aconteceu algo de errado." `str[size-1]` é `'\0'` neste caso também, mas o resto de *str* é indefinido. A causa exata do erro depende da plataforma subjacente.

As funções a seguir fornecem strings independentes de localidade para conversões de números.

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

Parte da ABI Estável. Converte a parte inicial da string em `str` para um valor `unsigned long` de acordo com a `base` fornecida, que deve estar entre `2` e `36` inclusive, ou ser o valor especial `0`.

Espaços em branco iniciais e diferenciação entre caracteres maiúsculos e minúsculos são ignorados. Se `base` for zero, procura um `0b`, `0o` ou `0x` inicial para informar qual base. Se estes estiverem ausentes, o padrão é `10`. Base deve ser 0 ou entre 2 e 36 (inclusive). Se `ptr` não for `NULL`, conterá um ponteiro para o fim da varredura.

Se o valor convertido ficar fora do intervalo do tipo de retorno correspondente, ocorrerá um erro de intervalo (`errno` é definido como `ERANGE`) e `ULONG_MAX` será retornado. Se nenhuma conversão puder ser realizada, `0` será retornado.

Veja também a página man do Unix *`strtoul(3)`*.

Adicionado na versão 3.2.

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

Parte da ABI Estável. Converte a parte inicial da string em `str` para um valor `long` de acordo com a `base` fornecida, que deve estar entre `2` e `36` inclusive, ou ser o valor especial `0`.

O mesmo que *PyOS_strtoul()*, mas retorna um valor `long` e `LONG_MAX` em caso de estouro.

Veja também a página man do Unix *strtol(3)*.

Adicionado na versão 3.2.

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

Parte da ABI Estável. Converte uma string `s` em `double`, levantando uma exceção Python em caso de falha. O conjunto de strings aceitas corresponde ao conjunto de strings aceito pelo construtor `float()` do Python, exceto que `s` não deve ter espaços em branco à esquerda ou à direita. A conversão é independente da localidade atual.

Se `endptr` for `NULL`, converte a string inteira. Levanta `ValueError` e retorna `-1.0` se a string não for uma representação válida de um número de ponto flutuante.

Se endptr não for `NULL`, converte o máximo possível da string e define `*endptr` para apontar para o primeiro caractere não convertido. Se nenhum segmento inicial da string for a representação válida de um número de ponto flutuante, define `*endptr` para apontar para o início da string, levanta ValueError e retorna `-1.0`.

Se `s` representa um valor que é muito grande para armazenar em um ponto flutuante (por exemplo, `"1e500"` é uma string assim em muitas plataformas), então se `overflow_exception` for `NULL` retorna `Py_HUGE_VAL` (com um sinal apropriado) e não define nenhuma exceção. Caso contrário, `overflow_exception` deve apontar para um objeto de exceção Python; levanta essa exceção e retorna `-1.0`. Em ambos os casos, define `*endptr` para apontar para o primeiro caractere após o valor convertido.

Se qualquer outro erro ocorrer durante a conversão (por exemplo, um erro de falta de memória), define a exceção Python apropriada e retorna `-1.0`.

Adicionado na versão 3.1.

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

Parte da ABI Estável. Converte um *val* `double` para uma string usando *format_code*, *precision* e *flags* fornecidos.

format_code deve ser um entre `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'` ou `'r'`. Para `'r'`, a precisão *precision* fornecida deve ser 0 e é ignorada. O código de formato `'r'` especifica o formato padrão de `repr()`.

flags pode ser zero ou mais de valores `Py_DTSF_SIGN`, `Py_DTSF_ADD_DOT_0` ou `Py_DTSF_ALT`, alternados por operador lógico OU:

- `Py_DTSF_SIGN` significa sempre preceder a string retornada com um caractere de sinal, mesmo se *val* não for negativo.
- `Py_DTSF_ADD_DOT_0` significa garantir que a string retornada não se pareça com um inteiro.
- `Py_DTSF_ALT` significa aplicar regras de formatação "alternativas". Veja a documentação para o especificador `'#'` de *PyOS_snprintf()* para detalhes.

Se *type* não for `NULL`, então o valor para o qual ele aponta será definido como um dos `Py_DTST_FINITE`, `Py_DTST_INFINITE` ou `Py_DTST_NAN`, significando que *val* é um número finito, um número infinito ou não um número, respectivamente.

O valor de retorno é um ponteiro para *buffer* com a string convertida ou `NULL` se a conversão falhou. O chamador é responsável por liberar a string retornada chamando *PyMem_Free()*.

Adicionado na versão 3.1.

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

Comparação de strings sem diferença entre maiúsculas e minúsculas. A função funciona quase de forma idêntica a `strcmp()` exceto que ignora a diferença entre maiúsculas e minúsculas.

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

Comparação de strings sem diferença entre maiúsculas e minúsculas. A função funciona quase de forma idêntica a `strncmp()` exceto que ignora a diferença entre maiúsculas e minúsculas.

6.8 API do PyHash

Veja também o membro *PyTypeObject.tp_hash* e numeric-hash.

type **Py_hash_t**

Tipo de valor do hash: inteiro com sinal.

Adicionado na versão 3.2.

type **Py_uhash_t**

Tipo de valor do hash: inteiro sem sinal.

Adicionado na versão 3.2.

PyHASH_MODULUS

O primo de Mersenne `P = 2**n -1`, usado para esquema de hash numérico.

Adicionado na versão 3.13.

PyHASH_BITS

O expoente `n` de `P` em *PyHASH_MODULUS*.

Adicionado na versão 3.13.

PyHASH_MULTIPLIER

Multiplicador de primo usado em strings e vários outros hashes.

Adicionado na versão 3.13.

PyHASH_INF

O valor de hash retornado para um infinito positivo.

Adicionado na versão 3.13.

PyHASH_IMAG

O multiplicador usado para a parte imaginária de um número complexo.

Adicionado na versão 3.13.

type **PyHash_FuncDef**

Definição de função de hash usada por *PyHash_GetFuncDef()*.

const char ***name**

Nome de função hash (string codificada em UTF-8).

const int **hash_bits**

Tamanho interno do valor do hash em bits.

const int **seed_bits**

Tamanho da entrada de seed em bits.

Adicionado na versão 3.4.

PyHash_FuncDef ***PyHash_GetFuncDef**(void)

Obtém a definição de função de hash.

> **Ver também**
>
> **PEP 456** "Algoritmo de hash seguro e intercambiável".

Adicionado na versão 3.4.

Py_hash_t **Py_HashPointer**(const void *ptr)

Hash de um valor de ponteiro: processa o valor do ponteiro como um inteiro (converte-o para `uintptr_t` internamente). O ponteiro não é desreferenciado.

A função não pode falhar: ela não pode retornar `-1`.

Adicionado na versão 3.13.

Py_hash_t **PyObject_GenericHash**(*PyObject* *obj)

Função de hash genérica que deve ser colocada no slot `tp_hash` de um objeto de tipo. Seu resultado depende apenas da identidade do objeto.

No CPython, é equivalente a *Py_HashPointer()*.

Adicionado na versão 3.13.

6.9 Reflexão

PyObject ***PyEval_GetBuiltins**(void)

Retorna valor: Referência emprestada. *Parte da* ABI Estável. Descontinuado desde a versão 3.13: Use *PyEval_GetFrameBuiltins()*.

Retorna um dicionário dos componentes embutidos no quadro de execução atual ou o interpretador do estado da thread, se nenhum quadro estiver em execução no momento.

PyObject ***PyEval_GetLocals**(void)

Retorna valor: Referência emprestada. *Parte da* ABI Estável. Descontinuado desde a versão 3.13: Use *PyEval_GetFrameLocals()* para obter o mesmo comportamento que chamar `locals()` no código Python, ou então chame *PyFrame_GetLocals()* no resultado de *PyEval_GetFrame()* para acessar o atributo `f_locals` do quadro atualmente em execução.

Retorna um mapeamento fornecendo acesso às variáveis locais no quadro de execução atual ou `NULL` se nenhum quadro estiver sendo executado no momento.

Consulte `locals()` para detalhes do mapeamento retornado em diferentes escopos.

Como esta função retorna uma *referência emprestada*, o dicionário retornado para *escopos otimizados* é armazenado em cache no objeto frame e permanecerá ativo enquanto o objeto frame o fizer. Ao contrário de *PyEval_GetFrameLocals()* e `locals()`, chamadas subsequentes para esta função no mesmo quadro atualizarão o conteúdo do dicionário em cache para refletir as mudanças no estado das variáveis locais em vez de retornar um novo snapshot.

Alterado na versão 3.13: Como parte da **PEP 667**, *PyFrame_GetLocals()*, `locals()` e `FrameType.f_locals` não fazem mais uso do dicionário de cache compartilhado. Consulte a entrada de O Que Há de Novo para detalhes adicionais.

PyObject ***PyEval_GetGlobals**(void)

Retorna valor: Referência emprestada. *Parte da* ABI Estável. Descontinuado desde a versão 3.13: Use *PyEval_GetFrameGlobals()*.

Retorna um dicionário das variáveis globais no quadro de execução atual ou `NULL` se nenhum quadro estiver sendo executado no momento.

PyFrameObject ***PyEval_GetFrame**(void)

Retorna valor: Referência emprestada. *Parte da* ABI Estável. Retorna o quadro do estado atual da thread, que é `NULL` se nenhum quadro estiver em execução no momento.

Veja também *PyThreadState_GetFrame()*.

PyObject ***PyEval_GetFrameBuiltins**(void)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.13.* Retorna um dicionário dos componentes embutidos no quadro de execução atual ou o interpretador do estado da thread, se nenhum quadro estiver em execução no momento.

Adicionado na versão 3.13.

PyObject ***PyEval_GetFrameLocals**(void)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.13.* Retorna um dicionário das variáveis locais no quadro de execução atual ou `NULL` se nenhum quadro estiver sendo executado no momento. Equivalente a chamar `locals()` em código Python.

Para acessar `f_locals` no quadro atual sem fazer um snapshot independente em *escopos otimizados*, chame *`PyFrame_GetLocals()`* no resultado de *`PyEval_GetFrame()`*.

Adicionado na versão 3.13.

PyObject ***PyEval_GetFrameGlobals**(void)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.13.* Retorna um dicionário das variáveis globais no quadro de execução atual ou `NULL` se nenhum quadro estiver sendo executado no momento. Equivalente a chamar `globals()` em código Python.

Adicionado na versão 3.13.

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

Parte da ABI Estável. 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)

Parte da ABI Estável. Retorna uma sequência de caracteres de descrição, dependendo do tipo de *func*. Os valores de retorno incluem "()" para funções e métodos, " constructor", " instance" e " object".. Concatenado com o resultado de *`PyEval_GetFuncName()`*, o resultado será uma descrição de *func*.

6.10 Registro de codec e funções de suporte

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

Parte da ABI Estável. 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_Unregister**(*PyObject* *search_function)

Parte da ABI Estável *desde a versão 3.10.* Cancela o registro de uma função de busca de codec e limpa o cache de registro. Se a função de busca não está registrada, não faz nada. Retorna 0 no sucesso. Levanta uma exceção e retorna -1 em caso de erro.

Adicionado na versão 3.10.

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

Parte da ABI Estável. Retorna `1` ou `0` dependendo se há um codec registrado para a dada codificação *encoding*. Essa função sempre é bem-sucedida.

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

Retorna valor: Nova referência. Parte da ABI Estável. API de codificação baseada em codec genérico.

object é passado através da função de codificação encontrada para a codificação fornecida por meio de *encoding*, usando o método de tratamento de erros definido por *errors*. *errors* pode ser `NULL` para usar o método padrão definido para o codec. Levanta um `LookupError` se nenhum codificador puder ser encontrado.

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

Retorna valor: Nova referência. Parte da ABI Estável. API de decodificação baseada em decodificador genérico.

object é passado através da função de decodificação encontrada para a codificação fornecida por meio de *encoding*, usando o método de tratamento de erros definido por *errors*. *errors* pode ser `NULL` para usar o método padrão definido para o codec. Levanta um `LookupError` se nenhum codificador puder ser encontrado.

6.10.1 API de pesquisa de codec

Nas funções a seguir, a string *encoding* é pesquisada com todos os caracteres sendo convertidos para minúsculo, o que faz com que as codificações pesquisadas por esse mecanismo não façam distinção entre maiúsculas e minúsculas. Se nenhum codec for encontrado, um `KeyError` é definido e `NULL` é retornado.

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

Retorna valor: Nova referência. Parte da ABI Estável. Obtém uma função de codificação para o *encoding* dado.

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

Retorna valor: Nova referência. Parte da ABI Estável. Obtém uma função de decodificação para o *encoding* dado.

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

Retorna valor: Nova referência. Parte da ABI Estável. Obtém um objeto `IncrementalEncoder` para o *encoding* dado.

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

Retorna valor: Nova referência. Parte da ABI Estável. Obtém um objeto `IncrementalDecoder` para o *encoding* dado.

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

Retorna valor: Nova referência. Parte da ABI Estável. Obtém uma função de fábrica `StreamReader` para o *encoding* dado.

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

Retorna valor: Nova referência. Parte da ABI Estável. Obtém uma função de fábrica `StreamWriter` para o *encoding* dado.

6.10.2 API de registro de tratamentos de erros de decodificação Unicode

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

Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. Pesquisa a função de retorno de chamada de tratamento de erros registrada em *name*. Como um caso especial, `NULL` pode ser passado; nesse caso, o erro no tratamento de retorno de chamada para "strict" será retornado.

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

Retorna valor: Sempre NULL. Parte da ABI Estável. Levanta *exc* como uma exceção.

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

Retorna valor: Nova referência. Parte da ABI Estável. Ignora o erro de unicode, ignorando a entrada que causou o erro.

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

Retorna valor: Nova referência. Parte da ABI Estável. Substitui o erro de unicode por `?` ou `U+FFFD`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Substitui o erro de unicode por caracteres da referência XML.

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

Retorna valor: Nova referência. Parte da ABI Estável. Substitui o erro de unicode com escapes de barra invertida (`\x`, `\u` e `\U`).

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

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.7.* Substitui os erros de codificação unicode com escapes `\N{...}`.

Adicionado na versão 3.5.

6.11 API C PyTime

Adicionado na versão 3.13.

A API C de relógios provê acesso a relógios do sistema. Ela é similar ao módulo Python `time`.

Para uma API C relacionada ao módulo `datetime`, veja *Objetos DateTime*.

6.11.1 Tipos

type **PyTime_t**

Um registro de data e hora ou uma duração em nanossegundos, representados como um inteiro de 64 bits com sinal.

O ponto de referência para registros de data e hora depende do relógio usado. Por exemplo, *PyTime_Time()* retorna registros relativos à época UNIX.

É suportada uma amplitude em torno de [-292.3 anos; +292.3 anos]. Usando o início da época UNIX (1º de Janeiro de 1970) como referência, a gama de datas suportadas é em torno de [1677-09-21; 2262-04-11]. Os limites exatos são expostos como constantes:

PyTime_t **PyTime_MIN**

Valor mínimo de *PyTime_t*.

PyTime_t **PyTime_MAX**

Valor máximo de *PyTime_t*.

6.11.2 Funções de relógio

As funções a seguir aceitam um ponteiro para *PyTime_t* ao qual elas atribuem o valor de um determinado relógio. Detalhes de cada relógio estão disponíveis nas documentações das funções Python correspondentes.

As funções retornam `0` em caso de sucesso, ou `-1` (com uma exceção definida) em caso de falha.

Em caso de estouro de inteiros, elas definem a exceção `PyExc_OverflowError` e definem `*result` como o valor limitado ao intervalo `[PyTime_MIN; PyTime_MAX]`. (Em sistemas atuais, estouros de inteiros são provavelmente causados por uma má configuração do tempo do sistema).

Como qualquer outra API C (se não especificado o contrário), as funções devem ser chamadas sob posse da *GIL*.

int **PyTime_Monotonic**(*PyTime_t* *result)

Lê o relógio monótono. Veja `time.monotonic()` para detalhes importantes sobre este relógio.

int **PyTime_PerfCounter**(*PyTime_t* *result)

Lê o contador de desempenho. Veja `time.perf_counter()` para detalhes importantes sobre este relógio.

int **PyTime_Time**(*PyTime_t* *result)

Lê o "relógio de parede". Veja `time.time()` para detalhes importantes sobre este relógio.

6.11.3 Funções de relógio brutas

Similares às funções de relógio, mas não definem uma exceção em condições de erro, e não requerem que o chamador possua a trava *GIL*.

Em caso de sucesso, as funções retornam `0`.

Em caso de falha, elas definem `*result` como `0` e retornam `-1`, *sem* definir uma exceção. Para acessar a causa do erro, obtenha a GIL e chame a função regular (sem o sufixo `Raw`). Observrve que a função regular pode ser bem-sucedida mesmo após a bruta falhar.

int **PyTime_MonotonicRaw**(*PyTime_t* *result)

Similar a *PyTime_Monotonic()*, mas não define uma exceção em caso de erro, e não exige a posse da GIL.

int **PyTime_PerfCounterRaw**(*PyTime_t* *result)

Similar a *PyTime_PerfCounter()*, mas não define uma exceção em caso de erro e não requer a posse da GIL.

int **PyTime_TimeRaw**(*PyTime_t* *result)

Similar a *PyTime_Time()*, mas não define uma exceção em caso de erro e não requer a posse da GIL.

6.11.4 Funções de conversão

double **PyTime_AsSecondsDouble**(*PyTime_t* t)

Converte um registro de data e hora para uma quantidade de segundos como um `double` C.

Esta função nunca falha, mas note que `double` tem acurácia limitada para valores grandes.

6.12 Suporte a Mapas do Perf

Em plataformas suportadas (no momento em que este livro foi escrito, apenas Linux), o tempo de execução pode tirar vantagem dos *arquivos de mapa perf* para tornar as funções Python visíveis para uma ferramenta de perfilação externa (como perf). Um processo em execução pode criar um arquivo no diretório `/tmp`, que contém entradas que podem mapear uma seção de código executável para um nome. Esta interface é descrita na documentação da ferramenta Linux Perf.

Em Python, essas APIs auxiliares podem ser usadas por bibliotecas e recursos que dependem da geração de código de máquina dinamicamente.

Observe que manter a trava global do interpretador (GIL) não é necessário para essas APIs.

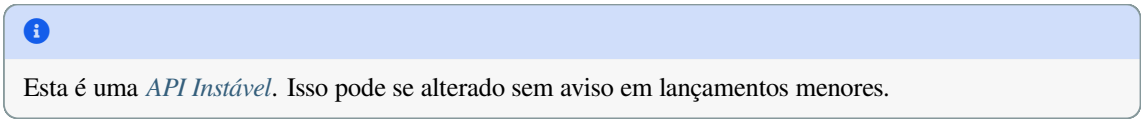
int **PyUnstable_PerfMapState_Init**(void)

> Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Abre o arquivo `/tmp/perf-$pid.map`, a menos que já esteja aberto, e cria uma trava para garantir escritas seguras para thread no arquivo (desde que as escritas sejam feitas através de *PyUnstable_WritePerfMapEntry()*). Normalmente, não há necessidade de chamar isso explicitamente; basta usar *PyUnstable_WritePerfMapEntry()* e ele inicializará o estado na primeira chamada.

Retorna `0` em caso de sucesso, `-1` em caso de falha ao criar/abrir o arquivo de mapa de desempenho ou `-2` em caso de falha na criação de uma trava. Verifique `errno` para mais informações sobre a causa de uma falha.

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



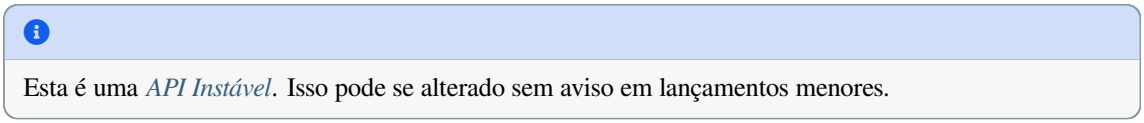
Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Escreve uma única entrada no arquivo `/tmp/perf-$pid.map`. Esta função é segura para thread. Aqui está a aparência de um exemplo de entrada:

```
# endereço  tamanho  nome
7f3529fcf759 b     py::bar:/run/t.py
```

Chamará *PyUnstable_PerfMapState_Init()* antes de escrever a entrada, se o arquivo de mapa de desempenho ainda não estiver aberto. Retorna `0` em caso de sucesso ou os mesmos códigos de erro que *PyUnstable_PerfMapState_Init()* em caso de falha.

void **PyUnstable_PerfMapState_Fini**(void)



Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Fecha o arquivo de mapa do perf aberto por *PyUnstable_PerfMapState_Init()*. Isso é chamado pelo próprio tempo de execução durante o desligamento do interpretador. Em geral, não deve haver motivo para chamar isso explicitamente, exceto para lidar com cenários específicos, como bifurcação.

CAPÍTULO 7

Camada de Objetos Abstratos

As funções neste capítulo interagem com objetos Python independentemente do seu tipo, ou com classes amplas de tipos de objetos (por exemplo, todos os tipos numéricos ou todos os tipos de sequência). Quando usadas em tipos de objetos para os quais não se aplicam, elas irão levantar uma exceção Python.

Não é possível usar essas funções em objetos que não foram inicializados corretamente, como um objeto de lista criado por `PyList_New()`, mas cujos itens ainda não foram definidos para algum valor diferente de `NULL`.

7.1 Protocolo de objeto

PyObject ***Py_GetConstant**(unsigned int constant_id)

Parte da ABI Estável *desde a versão 3.13.* Obtém uma *referência forte* para uma constante.

Define uma exceção e retorna `NULL` se *constant_id* for inválido.

constant_id deve ser um destes identificadores constantes:

Identificador da constante	Valor	Objeto retornado
Py_CONSTANT_NONE	0	None
Py_CONSTANT_FALSE	1	False
Py_CONSTANT_TRUE	2	True
Py_CONSTANT_ELLIPSIS	3	Ellipsis
Py_CONSTANT_NOT_IMPLEMENTI	4	NotImplemented
Py_CONSTANT_ZERO	5	0
Py_CONSTANT_ONE	6	1
Py_CONSTANT_EMPTY_STR	7	''
Py_CONSTANT_EMPTY_BYTES	8	b''
Py_CONSTANT_EMPTY_TUPLE	9	()

Valores numéricos são fornecidos apenas para projetos que não podem usar identificadores constantes.

Adicionado na versão 3.13.

No CPython, todas essas constantes são *imortais*.

PyObject ***Py_GetConstantBorrowed**(unsigned int constant_id)

Parte da ABI Estável *desde a versão 3.13.* Semelhante a *Py_GetConstant()*, mas retorna uma *referência emprestada*.

Esta função destina-se principalmente à compatibilidade com versões anteriores: usar *Py_GetConstant()* é recomendado para novo código.

A referência é emprestada do interpretador e é válida até a finalização do interpretador.

Adicionado na versão 3.13.

PyObject ***Py_NotImplemented**

O singleton `NotImplemented`, usado para sinalizar que uma operação não foi implementada para a combinação de tipo fornecida.

Py_RETURN_NOTIMPLEMENTED

Manipula adequadamente o retorno de *Py_NotImplemented* de dentro de uma função C (ou seja, cria uma nova *referência forte* para `NotImplemented` e retorna-a).

Py_PRINT_RAW

Sinaliza a ser usado com múltiplas funções que imprimem o objeto (como *PyObject_Print()* e *PyFile_WriteObject()*). Se passada, esta função usaria o `str()` do objeto em vez do `repr()`.

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

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

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

Parte da ABI Estável *desde a versão 3.13.* Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This is equivalent to the Python expression `hasattr(o, attr_name)`. On failure, return `-1`.

Adicionado na versão 3.13.

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

Parte da ABI Estável *desde a versão 3.13.* This is the same as *PyObject_HasAttrWithError()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

Adicionado na versão 3.13.

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

Parte da ABI Estável. Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This function always succeeds.

> **Nota**
>
> Exceptions that occur when this calls `__getattr__()` and `__getattribute__()` methods aren't propagated, but instead given to `sys.unraisablehook()`. For proper error handling, use *PyObject_HasAttrWithError()*, *PyObject_GetOptionalAttr()* or *PyObject_GetAttr()* instead.

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

Parte da ABI Estável. This is the same as *PyObject_HasAttr()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

> **Nota**
>
> Exceptions that occur when this calls `__getattr__()` and `__getattribute__()` methods or while creating the temporary `str` object are silently ignored. For proper error handling, use *PyObject_HasAttrStringWithError()*, *PyObject_GetOptionalAttrString()* or *PyObject_GetAttrString()* instead.

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

Retorna valor: Nova referência. Parte da ABI Estável. 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`.

If the missing attribute should not be treated as a failure, you can use *PyObject_GetOptionalAttr()* instead.

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

Retorna valor: Nova referência. Parte da ABI Estável. This is the same as *PyObject_GetAttr()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

If the missing attribute should not be treated as a failure, you can use *PyObject_GetOptionalAttrString()* instead.

int **PyObject_GetOptionalAttr**(*PyObject* *obj, *PyObject* *attr_name, *PyObject* **result);

Parte da ABI Estável *desde a versão 3.13.* Variant of *PyObject_GetAttr()* which doesn't raise `AttributeError` if the attribute is not found.

If the attribute is found, return `1` and set **result* to a new *strong reference* to the attribute. If the attribute is not found, return `0` and set **result* to `NULL`; the `AttributeError` is silenced. If an error other than `AttributeError` is raised, return `-1` and set **result* to `NULL`.

Adicionado na versão 3.13.

int **PyObject_GetOptionalAttrString**(*PyObject* *obj, const char *attr_name, *PyObject* **result);

Parte da ABI Estável *desde a versão 3.13.* This is the same as `PyObject_GetOptionalAttr()`, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Adicionado na versão 3.13.

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

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Parte da ABI Estável. Set the value of the attribute named *attr_name*, for object *o*, to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o.attr_name = v`.

If *v* is `NULL`, the attribute is deleted. This behaviour is deprecated in favour of using `PyObject_DelAttr()`, but there are currently no plans to remove it.

int **PyObject_SetAttrString**(*PyObject* *o, const char *attr_name, *PyObject* *v)

Parte da ABI Estável. This is the same as `PyObject_SetAttr()`, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

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

The number of different attribute names passed to this function should be kept small, usually by using a statically allocated string as *attr_name*. For attribute names that aren't known at compile time, prefer calling `PyUnicode_FromString()` and `PyObject_SetAttr()` directly. For more details, see `PyUnicode_InternFromString()`, which may be used internally to create a key object.

int **PyObject_GenericSetAttr**(*PyObject* *o, *PyObject* *name, *PyObject* *value)

Parte da ABI Estável. 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)

Parte da ABI Estável *desde a versão 3.13.* 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)

Parte da ABI Estável *desde a versão 3.13.* This is the same as `PyObject_DelAttr()`, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

The number of different attribute names passed to this function should be kept small, usually by using a statically allocated string as *attr_name*. For attribute names that aren't known at compile time, prefer calling `PyUnicode_FromString()` and `PyObject_DelAttr()` directly. For more details, see `PyUnicode_InternFromString()`, which may be used internally to create a key object for lookup.

PyObject ***PyObject_GenericGetDict**(*PyObject* *o, void *context)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.10.* A generic implementation for the getter of a `__dict__` descriptor. It creates the dictionary if necessary.

This function may also be called to get the `__dict__` of the object *o*. Pass `NULL` for *context* when calling it. Since this function may need to allocate memory for the dictionary, it may be more efficient to call `PyObject_GetAttr()` when accessing an attribute on the object.

On failure, returns `NULL` with an exception set.

Adicionado na versão 3.3.

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

Parte da ABI Estável *desde a versão 3.7.* A generic implementation for the setter of a `__dict__` descriptor. This implementation does not allow the dictionary to be deleted.

Adicionado na versão 3.3.

PyObject ****_PyObject_GetDictPtr**(*PyObject* *obj)

Return a pointer to `__dict__` of the object *obj*. If there is no `__dict__`, return `NULL` without setting an exception.

This function may need to allocate memory for the dictionary, so it may be more efficient to call `PyObject_GetAttr()` when accessing an attribute on the object.

PyObject ***PyObject_RichCompare**(*PyObject* *o1, *PyObject* *o2, int opid)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Parte da ABI Estável. Compare the values of *o1* and *o2* using the operation specified by *opid*, like `PyObject_RichCompare()`, but returns `-1` on error, `0` if the result is false, `1` otherwise.

Nota

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

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

Parte da ABI Estável. Format *obj* using *format_spec*. This is equivalent to the Python expression `format(obj, format_spec)`.

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

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

Retorna valor: Nova referência. Parte da ABI Estável. 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.

Alterado na versão 3.4: Essa função agora inclui uma asserção de depuração para ajudar a garantir que ela não descarte silenciosamente uma exceção ativa.

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

Retorna valor: Nova referência. Parte da ABI Estável. As `PyObject_Repr()`, compute a string representation of object *o*, but escape the non-ASCII characters in the string returned by `PyObject_Repr()` with `\x`, `\u` or `\U` escapes. This generates a string similar to that returned by `PyObject_Repr()` in Python 2. Called by the `ascii()` built-in function.

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

Retorna valor: Nova referência. Parte da ABI Estável. Compute a string representation of object *o*. Returns the string representation on success, `NULL` on failure. This is the equivalent of the Python expression `str(o)`. Called by the `str()` built-in function and, therefore, by the `print()` function.

Alterado na versão 3.4: Essa função agora inclui uma asserção de depuração para ajudar a garantir que ela não descarte silenciosamente uma exceção ativa.

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

Retorna valor: Nova referência. Parte da ABI Estável. Compute a bytes representation of object *o*. `NULL` is returned on failure and a bytes object on success. This is equivalent to the Python expression `bytes(o)`, when *o* is not an integer. Unlike `bytes(o)`, a TypeError is raised when *o* is an integer instead of a zero-initialized bytes object.

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

Parte da ABI Estável. Return `1` if the class *derived* is identical to or derived from the class *cls*, otherwise return `0`. In case of an error, return `-1`.

If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `0`.

If *cls* has a `__subclasscheck__()` method, it will be called to determine the subclass status as described in **PEP 3119**. Otherwise, *derived* is a subclass of *cls* if it is a direct or indirect subclass, i.e. contained in `cls.__mro__`.

Normally only class objects, i.e. instances of `type` or a derived class, are considered classes. However, objects can override this by having a `__bases__` attribute (which must be a tuple of base classes).

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

Parte da ABI Estável. Return `1` if *inst* is an instance of the class *cls* or a subclass of *cls*, or `0` if not. On error, returns `-1` and sets an exception.

If *cls* is a tuple, the check will be done against every entry in *cls*. The result will be `1` when at least one of the checks returns `1`, otherwise it will be `0`.

If *cls* has a `__instancecheck__()` method, it will be called to determine the subclass status as described in **PEP 3119**. Otherwise, *inst* is an instance of *cls* if its class is a subclass of *cls*.

An instance *inst* can override what is considered its class by having a `__class__` attribute.

An object *cls* can override if it is considered a class, and what its base classes are, by having a `__bases__` attribute (which must be a tuple of base classes).

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

Parte da ABI Estável. Compute and return the hash value of an object *o*. On failure, return `-1`. This is the equivalent of the Python expression `hash(o)`.

Alterado na versão 3.2: The return type is now Py_hash_t. This is a signed integer the same size as *Py_ssize_t*.

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

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

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

Parte da ABI Estável. 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)

Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. When *o* is non-`NULL`, returns a type object corresponding to the object type of object *o*. On failure, raises `SystemError` and returns `NULL`. This is equivalent to the Python expression `type(o)`. This function creates a new *strong reference* to the return value. There's really no reason to use this function instead of the *Py_TYPE()* function, which returns a pointer of type *PyTypeObject**, except when a new *strong reference* is needed.

int **PyObject_TypeCheck**(*PyObject* *o, *PyTypeObject* *type)

Return non-zero if the object *o* is of type *type* or a subtype of *type*, and `0` otherwise. Both parameters must be non-`NULL`.

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

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

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

Py_ssize_t **PyObject_LengthHint**(*PyObject* *o, *Py_ssize_t* defaultvalue)

Return an estimated length for the object *o*. First try to return its actual length, then an estimate using `__length_hint__()`, and finally return the default value. On error return `-1`. This is the equivalent to the Python expression `operator.length_hint(o, defaultvalue)`.

Adicionado na versão 3.4.

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

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Parte da ABI Estável. Map the object *key* to the value *v*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o[key] = v`. This function *does not* steal a reference to *v*.

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

Parte da ABI Estável. Remove the mapping for the object *key* from the object *o*. Return `-1` on failure. This is equivalent to the Python statement `del o[key]`.

int **PyObject_DelItemString**(*PyObject* *o, const char *key)

Parte da ABI Estável. É o mesmo que *PyObject_DelItem()*, mas *key* é especificada como uma string de bytes `const char*` codificada em UTF-8, em vez de um *PyObject**.

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

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. This is equivalent to the Python expression `iter(o)`. It returns a new iterator for the object argument, or the object itself if the object is already an iterator. Raises `TypeError` and returns `NULL` if the object cannot be iterated.

PyObject ***PyObject_SelfIter**(*PyObject* *obj)

Retorna valor: Nova referência. Parte da ABI Estável. This is equivalent to the Python `__iter__(self): return self` method. It is intended for *iterator* types, to be used in the *PyTypeObject.tp_iter* slot.

PyObject ***PyObject_GetAIter**(*PyObject* *o)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.10.* This is the equivalent to the Python expression `aiter(o)`. Takes an `AsyncIterable` object and returns an `AsyncIterator` for it. This is typically a new iterator but if the argument is an `AsyncIterator`, this returns itself. Raises `TypeError` and returns `NULL` if the object cannot be iterated.

Adicionado na versão 3.10.

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

Parte da ABI Estável *desde a versão 3.12.* Get a pointer to subclass-specific data reserved for *cls*.

The object *o* must be an instance of *cls*, and *cls* must have been created using negative *PyType_Spec.basicsize*. Python does not check this.

On error, set an exception and return `NULL`.

Adicionado na versão 3.12.

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

Parte da ABI Estável *desde a versão 3.12.* Return the size of the instance memory space reserved for *cls*, i.e. the size of the memory `PyObject_GetTypeData()` returns.

This may be larger than requested using `-PyType_Spec.basicsize`; it is safe to use this larger size (e.g. with `memset()`).

The type *cls* **must** have been created using negative `PyType_Spec.basicsize`. Python does not check this.

On error, set an exception and return a negative value.

Adicionado na versão 3.12.

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

Get a pointer to per-item data for a class with `Py_TPFLAGS_ITEMS_AT_END`.

On error, set an exception and return `NULL`. `TypeError` is raised if *o* does not have `Py_TPFLAGS_ITEMS_AT_END` set.

Adicionado na versão 3.12.

int **PyObject_VisitManagedDict**(*PyObject* *obj, *visitproc* visit, void *arg)

Visit the managed dictionary of *obj*.

This function must only be called in a traverse function of the type which has the `Py_TPFLAGS_MANAGED_DICT` flag set.

Adicionado na versão 3.13.

void **PyObject_ClearManagedDict**(*PyObject* *obj)

Clear the managed dictionary of *obj*.

This function must only be called in a traverse function of the type which has the `Py_TPFLAGS_MANAGED_DICT` flag set.

Adicionado na versão 3.13.

7.2 Protocolo de chamada

O CPython permite dois protocolos de chamada: *tp_call* e vectorcall.

7.2.1 O protocolo *tp_call*

Instâncias de classe que definem `tp_call` são chamáveis. A assinatura do slot é:

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

Uma chamada é feita usando uma tupla para os argumentos posicionais e um dicionário para os argumentos nomeados, similar a `callable(*args, **kwargs)` no Python. *args* não pode ser nulo (utilize uma tupla vazia se não houver argumentos), mas *kwargs* pode ser *NULL* se não houver argumentos nomeados.

Esta convenção não é somente usada por *tp_call*: `tp_new` e `tp_init` também passam argumento dessa forma.

Para chamar um objeto, use `PyObject_Call()` ou outra *call API*.

7.2.2 O protocolo vectorcall

Adicionado na versão 3.9.

O protocolo vectorcall foi introduzido pela **PEP 590** como um protocolo adicional para tornar invocações mais eficientes.

Como regra de bolso. CPython vai preferir o vectorcall para invocações internas se o chamável suportar. Entretanto, isso não é uma regra rígida. Ademais, alguma extensões de terceiros usam diretamente *tp_call* (em vez de utilizar `PyObject_Call()`). Portanto, uma classe que suporta vectorcall precisa também implementar `tp_call`. Além disso, o chamável precisa se comportar da mesma forma independe de qual protocolo é utilizado. A forma recomendada de alcançar isso é definindo `tp_call` para `PyVectorcall_Call()`. Vale a pena repetir:

Aviso

Uma classe que suporte vectorcall também **precisa** implementar `tp_call` com a mesma semântica.

Alterado na versão 3.12: O sinalizador `Py_TPFLAGS_HAVE_VECTORCALL` agora é removida da classe quando o método `__call__()` está reatribuído. (Internamente, isso apenas define `tp_call`, e portanto, pode fazê-lo comportar-se de forma diferente da função vetorcall. Em versões anteriores do Python, vectorcall só deve ser usado com tipos `imutáveis` ou estáticos.

Uma classe não deve implementar vectorcall se for mais lento que *tp_call*. Por exemplo, se o chamador precisa converter os argumentos para uma tupla args e um dicionário kwargs de qualquer forma, então não é necessário implementar vectorcall.

Classes podem implementar o protocolo vectorcall ativando o sinalizador `Py_TPFLAGS_HAVE_VECTORCALL` e configurando `tp_vectorcall_offset` para o offset dentro da estrutura do objeto onde uma *vectorcallfunc* aparece. Este é um ponteiro para uma função com a seguinte assinatura:

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

Parte da ABI Estável *desde a versão 3.12.*

- *callable* é o objeto sendo chamado.
- ***args* é um array C formado pelos argumentos posicionais seguidos de**
  valores dos argumentos nomeados. Este pode ser *NULL* se não existirem argumentos.
- ***nargsf* é o número de argumentos posicionais somado á possível**
  Sinalizador `PY_VECTORCALL_ARGUMENTS_OFFSET`. Para obter o número real de argumentos posicionais de *nargsf*, use `PyVectorcall_NARGS()`.
- ***kwnames* é uma tupla contendo os nomes dos argumentos nomeados;**
  em outras palavras, as chaves do dicionário kwargs. Estes nomes devem ser strings (instâncias de `str` ou uma subclasse) e eles devem ser únicos. Se não existem argumentos nomeados, então *kwnames* deve então ser *NULL*.

PY_VECTORCALL_ARGUMENTS_OFFSET

Parte da ABI Estável *desde a versão 3.12.* Se esse sinalizador é definido em um argumento *nargsf* do vectorcall, deve ser permitido ao chamado temporariamente mudar `args[-1]`. Em outras palavras, *args* aponta para o argumento 1 (não 0) no vetor alocado. O chamado deve restaurar o valor de `args[-1]` antes de retornar.

Para `PyObject_VectorcallMethod()`, este sinalizador significa que `args[0]` pode ser alterado.

Sempre que podem realizar a um custo tão baixo (sem alocações adicionais), invocadores são encorajados a usar `PY_VECTORCALL_ARGUMENTS_OFFSET`. Isso permitirá invocados como métodos vinculados a instâncias fazerem suas próprias invocações (o que inclui um argumento *self*) muito eficientemente.

Adicionado na versão 3.8.

Para invocar um objeto que implementa vectorcall, utilize a função *call API* como qualquer outra invocável. `PyObject_Vectorcall()` será normalmente mais eficiente.

Controle de recursão

Quando utilizando *tp_call*, invocadores não precisam se preocupar sobre *recursão*: CPython usa `Py_EnterRecursiveCall()` e `Py_LeaveRecursiveCall()` para chamadas utilizando *tp_call*.

Por questão de eficiência, este não é o caso de chamadas utilizando o vectorcall: o que chama deve utilizar *Py_EnterRecursiveCall* e *Py_LeaveRecursiveCall* se necessário.

API de suporte à chamada de vetores

Py_ssize_t **PyVectorcall_NARGS**(size_t nargsf)

Parte da ABI Estável *desde a versão 3.12.* Dado um argumento de chamada de vetor *nargsf*, retorna o número real de argumentos. Atualmente equivalente a:

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

Entretanto, a função `PyVectorcall_NARGS` deve ser usada para permitir para futuras extensões.

Adicionado na versão 3.8.

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

Se *op* não suporta o protocolo de chamada de vetor (seja porque o tipo ou a instância específica não suportam), retorne *NULL*. Se não, retorne o ponteiro da função chamada de vetor armazenado em *op*. Esta função nunca levanta uma exceção.

É mais útil checar se *op* suporta ou não chamada de vetor, o que pode ser feito checando `PyVectorcall_Function(op) != NULL`.

Adicionado na versão 3.9.

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

Parte da ABI Estável *desde a versão 3.12.* Chama o *vectorcallfunc* de *callable* com argumentos posicionais e nomeados dados em uma tupla e dicionário, respectivamente.

Esta é uma função especializada, feita para ser colocada no slot *tp_call* ou usada em uma implementação de `tp_call`. Ela não verifica o sinalizador *Py_TPFLAGS_HAVE_VECTORCALL* e não retorna para `tp_call`.

Adicionado na versão 3.8.

7.2.3 API de chamada de objetos

Várias funções estão disponíveis para chamar um objeto Python. Cada uma converte seus argumentos para uma convenção suportada pelo objeto chamado – seja *tp_call* ou chamada de vetor. Para fazer o mínimo possível de conversões, escolha um que melhor se adapte ao formato de dados que você tem disponível.

A tabela a seguir resume as funções disponíveis; por favor, veja a documentação individual para detalhes.

Função	chamável	args	kwargs
PyObject_Call()	`PyObject *`	tupla	dict/`NULL`
PyObject_CallNoArgs()	`PyObject *`	—	—
PyObject_CallOneArg()	`PyObject *`	1 objeto	—
PyObject_CallObject()	`PyObject *`	tupla/`NULL`	—
PyObject_CallFunction()	`PyObject *`	formato	—
PyObject_CallMethod()	obj + `char*`	formato	—
PyObject_CallFunctionObjArgs()	`PyObject *`	variádica	—
PyObject_CallMethodObjArgs()	obj + nome	variádica	—
PyObject_CallMethodNoArgs()	obj + nome	—	—
PyObject_CallMethodOneArg()	obj + nome	1 objeto	—
PyObject_Vectorcall()	`PyObject *`	vectorcall	vectorcall
PyObject_VectorcallDict()	`PyObject *`	vectorcall	dict/`NULL`
PyObject_VectorcallMethod()	arg + nome	vectorcall	vectorcall

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

Retorna valor: Nova referência. Parte da ABI Estável. Chama um objeto Python chamável de *callable*, com argumentos dados pela tupla *args*, e argumentos nomeados dados pelo dicionário *kwargs*.

args não deve ser *NULL*; use uma tupla vazia se não precisar de argumentos. Se nenhum argumento nomeado é necessário, *kwargs* pode ser *NULL*.

Retorna o resultado da chamada em sucesso, ou levanta uma exceção e retorna *NULL* em caso de falha.

Esse é o equivalente da expressão Python: `callable(*args, **kwargs)`.

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

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.10.* Chama um objeto Python chamável de *callable* sem nenhum argumento. É o jeito mais eficiente de chamar um objeto Python sem nenhum argumento.

Retorna o resultado da chamada em sucesso, ou levanta uma exceção e retorna *NULL* em caso de falha.

Adicionado na versão 3.9.

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

Retorna valor: Nova referência. Chama um objeto Python chamável de *callable* com exatamente 1 argumento posicional *arg* e nenhum argumento nomeado.

Retorna o resultado da chamada em sucesso, ou levanta uma exceção e retorna *NULL* em caso de falha.

Adicionado na versão 3.9.

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

Retorna valor: Nova referência. Parte da ABI Estável. Chama um objeto Python chamável de *callable* com argumentos dados pela tupla *args*. Se nenhum argumento é necessário, *args* pode ser *NULL*.

Retorna o resultado da chamada em sucesso, ou levanta uma exceção e retorna *NULL* em caso de falha.

Este é o equivalente da expressão Python: `callable(*args)`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Chama um objeto Python chamável de *callable*, com um número variável de argumentos C. Os argumentos C são descritos usando uma string de estilo no formato *Py_BuildValue()*. O formato pode ser *NULL*, indicando que nenhum argumento foi provido.

Retorna o resultado da chamada em sucesso, ou levanta uma exceção e retorna *NULL* em caso de falha.

Este é o equivalente da expressão Python: `callable(*args)`.

Note que se você apenas passa argumentos *PyObject**, *PyObject_CallFunctionObjArgs()* é uma alternativa mais rápida.

Alterado na versão 3.4: O tipo de *format* foi mudado de `char *`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Chama o método chamado *name* do objeto *obj* com um número variável de argumentos C. Os argumentos C são descritos com uma string de formato *Py_BuildValue()* que deve produzir uma tupla.

O formato pode ser *NULL*, indicado que nenhum argumento foi provido.

Retorna o resultado da chamada em sucesso, ou levanta uma exceção e retorna *NULL* em caso de falha.

Este é o equivalente da expressão Python: `obj.name(arg1, arg2, ...)`.

Note que se você apenas passa argumentos *PyObject**, *PyObject_CallMethodObjArgs()* é uma alternativa mais rápida.

Alterado na versão 3.4: Os tipos de *name* e *format* foram mudados de `char *`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Chama um objeto Python chamável de *callable*, com um número variável de argumentos *PyObject**. Os argumentos são providos como um número variável de parâmetros seguidos por um *NULL*.

Retorna o resultado da chamada em sucesso, ou levanta uma exceção e retorna *NULL* em caso de falha.

Este é o equivalente da expressão Python: `callable(arg1, arg2, ...)`.

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

Retorna valor: Nova referência. *Parte da* ABI Estável. Chama um método do objeto Python *obj*, onde o nome do método é dado como um objeto string Python em *name*. É chamado com um número variável de argumentos `PyObject*`. Os argumentos são providos como um número variável de parâmetros seguidos por um *NULL*.

Retorna o resultado da chamada em sucesso, ou levanta uma exceção e retorna *NULL* em caso de falha.

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

Chama um método do objeto Python *obj* sem argumentos, onde o nome do método é fornecido como um objeto string do Python em *name*.

Retorna o resultado da chamada em sucesso, ou levanta uma exceção e retorna *NULL* em caso de falha.

Adicionado na versão 3.9.

PyObject ***PyObject_CallMethodOneArg**(*PyObject* *obj, *PyObject* *name, *PyObject* *arg)

Chama um método do objeto Python *obj* com um argumento posicional *arg*, onde o nome do método é fornecido como um objeto string do Python em *name*.

Retorna o resultado da chamada em sucesso, ou levanta uma exceção e retorna *NULL* em caso de falha.

Adicionado na versão 3.9.

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

Parte da ABI Estável *desde a versão 3.12.* Chama um objeto Python chamável *callable*. Os argumentos são os mesmos de `vectorcallfunc`. Se *callable* tiver suporte a *vectorcall*, isso chamará diretamente a função vectorcall armazenada em *callable*.

Retorna o resultado da chamada em sucesso, ou levanta uma exceção e retorna *NULL* em caso de falha.

Adicionado na versão 3.9.

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

Chama *callable* com argumentos posicionais passados exatamente como no protocolo *vectorcall*, mas com argumentos nomeados passados como um dicionário *kwdict*. O array *args* contém apenas os argumentos posicionais.

Independentemente de qual protocolo é usado internamente, uma conversão de argumentos precisa ser feita. Portanto, esta função só deve ser usada se o chamador já tiver um dicionário pronto para usar para os argumentos nomeados, mas não uma tupla para os argumentos posicionais.

Adicionado na versão 3.9.

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

Parte da ABI Estável *desde a versão 3.12.* Chama um método usando a convenção de chamada vectorcall. O nome do método é dado como uma string Python *name*. O objeto cujo método é chamado é *args[0]*, e o array *args* começando em *args[1]* representa os argumentos da chamada. Deve haver pelo menos um argumento posicional. *nargsf* é o número de argumentos posicionais incluindo *args[0]*, mais `PY_VECTORCALL_ARGUMENTS_OFFSET` se o valor de `args[0]` puder ser alterado temporariamente. Argumentos nomeados podem ser passados como em `PyObject_Vectorcall()`.

Se o objeto tem o recurso `Py_TPFLAGS_METHOD_DESCRIPTOR`, isso irá chamar o objeto de método não vinculado com o vetor *args* inteiro como argumentos.

Retorna o resultado da chamada em sucesso, ou levanta uma exceção e retorna *NULL* em caso de falha.

Adicionado na versão 3.9.

7.2.4 API de suporte a chamadas

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

Parte da ABI Estável. Determine se o objeto *o* é chamável. Devolva `1` se o objeto é chamável e `0` caso contrário. Esta função sempre tem êxito.

7.3 Protocolo de número

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

Parte da ABI Estável. Retorna `1` se o objeto *o* fornece protocolos numéricos; caso contrário, retorna falso. Esta função sempre tem sucesso.

Alterado na versão 3.8: Retorna `1` se *o* for um número inteiro de índice.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna o resultado da adição de *o1* e *o2*, ou `NULL` em caso de falha. Este é o equivalente da expressão Python `o1 + o2`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna o resultado da subtração de *o2* por *o1*, ou `NULL` em caso de falha. Este é o equivalente da expressão Python `o1 - o2`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna o resultado da multiplicação de *o1* e *o2*, ou `NULL` em caso de falha. Este é o equivalente da expressão Python `o1 * o2`.

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

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.7.* Retorna o resultado da multiplicação da matriz em *o1* e *o2*, ou `NULL` em caso de falha. Este é o equivalente da expressão Python `o1 @ o2`.

Adicionado na versão 3.5.

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

Retorna valor: Nova referência. Parte da ABI Estável. Return the floor of *o1* divided by *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 // o2`.

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

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

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

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. Returns *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `+o`.

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

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. Returns the result of multiplying *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 *= o2`.

PyObject ***PyNumber_InPlaceMatrixMultiply**(*PyObject* *o1, *PyObject* *o2)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.7.* Returns the result of matrix multiplication on *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 @= o2`.

Adicionado na versão 3.5.

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

Retorna valor: Nova referência. Parte da ABI Estável. Returns the mathematical floor of dividing *o1* by *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 //= o2`.

PyObject ***PyNumber_InPlaceTrueDivide**(*PyObject* *o1, *PyObject* *o2)

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

PyObject ***PyNumber_InPlaceRemainder**(*PyObject* *o1, *PyObject* *o2)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. Returns the "bitwise or" of *o1* and *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 |= o2`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Returns the *o* converted to an integer object on success, or `NULL` on failure. This is the equivalent of the Python expression `int(o)`.

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

Retorna valor: Nova referência. Parte da ABI Estável. 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)

Retorna valor: Nova referência. Parte da ABI Estável. Returns the *o* converted to a Python int on success or `NULL` with a `TypeError` exception raised on failure.

Alterado na versão 3.10: O resultado sempre tem o tipo exato `int`. Anteriormente, o resultado poderia ter sido uma instância de uma subclasse de `int`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Returns the integer *n* converted to base *base* as a string. The *base* argument must be one of 2, 8, 10, or 16. For base 2, 8, or 16, the returned string is prefixed with a base

marker of `'0b'`, `'0o'`, or `'0x'`, respectively. If *n* is not a Python int, it is converted with *PyNumber_Index()* first.

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

Parte da ABI Estável. Returns *o* converted to a *Py_ssize_t* value if *o* can be interpreted as an integer. If the call fails, an exception is raised and `-1` is returned.

If *o* can be converted to a Python int but the attempt to convert to a *Py_ssize_t* value would raise an `OverflowError`, then the *exc* argument is the type of exception that will be raised (usually `IndexError` or `OverflowError`). If *exc* is `NULL`, then the exception is cleared and the value is clipped to `PY_SSIZE_T_MIN` for a negative integer or `PY_SSIZE_T_MAX` for a positive integer.

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

Parte da ABI Estável *desde a versão 3.8.* Returns `1` if *o* is an index integer (has the `nb_index` slot of the `tp_as_number` structure filled in), and `0` otherwise. This function always succeeds.

7.4 Protocolo de sequência

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

Parte da ABI Estável. Retorna `1` se o objeto fornecer o protocolo de sequência e `0` caso contrário. Note que ele retorna `1` para classes Python com um método `__getitem__()` a menos que sejam subclasses de `dict` já que no caso geral é impossível determinar que tipo de chaves a classe provê. Esta função sempre obtém sucesso.

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

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

Parte da ABI Estável. Retorna o número de objetos na sequência *o* em caso de sucesso, ou `-1` em caso de falha. Equivale à expressão Python `len(o)`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna a concatenação de *o1* e *o2* em caso de sucesso, e `NULL` em caso de falha. Equivale à expressão Python `o1 + o2`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna o resultado da repetição do objeto sequência *o* *count* vezes ou `NULL` em caso de falha. Equivale à expressão Python `o * count`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna a concatenação de *o1* e *o2* em caso de sucesso, e `NULL` em caso de falha. A operação é feita *localmente* se *o1* permitir. Equivale à expressão Python `o1 += o2`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna o resultado da repetição do objeto sequência *o* *count* vezes ou `NULL` em caso de falha. A operação é feita *localmente* se *o* permitir. Equivale à expressão Python `o *= count`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna o elemento *i* de *o* ou `NULL` em caso de falha. Equivale à expressão Python `o[i]`.

PyObject ***PySequence_GetSlice**(*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna a fatia do objeto sequência *o* entre *i1* e *i2*, ou `NULL` em caso de falha. Equivale à expressão Python `o[i1:i2]`.

int **PySequence_SetItem**(*PyObject* *o, *Py_ssize_t* i, *PyObject* *v)

Parte da ABI Estável. Atribui o objeto *v* ao elemento *i* de *o*. Levanta uma exceção e retorna `-1` em caso de falha; retorna `0` em caso de sucesso. Esta função *não* rouba uma referência a *v*. Equivale à instrução Python `o[i]=v`.

Se *v* for `NULL`, o elemento será removido, mas este recurso foi descontinuado em favor do uso de *PySequence_DelItem()*.

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

Parte da ABI Estável. Exclui o elemento *i* do objeto *o*. Retorna `-1` em caso de falha. Equivale à instrução Python `del o[i]`.

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

Parte da ABI Estável. Atribui o objeto sequência *v* à fatia no objeto sequência *o* de *i1* a *i2*. Equivale à instrução Python `o[i1:i2] = v`.

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

Parte da ABI Estável. Exclui a fatia no objeto sequência *o* de *i1* a *i2*. Retorna `-1` em caso de falha. Equivale à instrução Python `del o[i1:i2]`.

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

Parte da ABI Estável. Retorna a quantidade de ocorrências de *value* em *o*, isto é, retorna a quantidade de chaves onde `o[key] == value`. Em caso de falha, retorna `-1`. Equivale à expressão Python `o.count(value)`.

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

Parte da ABI Estável. Determina se *o* contém *value*. Se um item em *o* for igual a *value*, retorna `1`, senão, retorna `0`. Em caso de erro, retorna `-1`. Equivale à expressão Python `value in o`.

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

Parte da ABI Estável. Retorna o primeiro índice *i* tal que `o[i] == value`. Em caso de erro, retorna `-1`. Equivale à expressão Python `o.index(value)`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um objeto lista com o mesmo conteúdo da sequência ou iterável *o*, ou `NULL` em caso de falha. Garante-se que a lista retornada será nova. Equivale à expressão Python `list(o)`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna o objeto tupla com o mesmo conteúdo da sequência ou iterável *o*, ou `NULL` em caso de falha. Se *o* for uma tupla, retorna uma nova referência. Senão, uma tupla será construída com o conteúdo apropriado. Equivale à expressão Python `tuple(o)`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna a sequência ou iterável *o* como um objeto usável por outras funções da família `PySequence_Fast*`. Se o objeto não for uma sequência ou iterável, levanta `TypeError` com *m* sendo o texto da mensagem. Retorna `NULL` em caso de falha.

As funções `PySequence_Fast*` têm esse nome porque presumem que *o* é um *PyTupleObject* ou um *PyListObject* e porque acessam os campos de dados de *o* diretamente.

Como detalhe de implementação de CPython, se *o* já for uma sequência ou lista, ele será retornado.

Py_ssize_t **PySequence_Fast_GET_SIZE**(*PyObject* *o)

Retorna o comprimento de *o*, presumindo que *o* foi retornado por *PySequence_Fast()* e que *o* não seja `NULL`. O tamanho também pode ser obtido ao chamar *PySequence_Size()* em *o*, mas *PySequence_Fast_GET_SIZE()* é mais rápida, ao supor que *o* é uma lista ou tupla.

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

Retorna valor: Referência emprestada. Retorna o elemento *i* de *o*, presumindo que *o* foi retornado por *PySequence_Fast()*, que *o* seja `NULL`, e que *i* esteja nos limites.

PyObject ****PySequence_Fast_ITEMS**(*PyObject* *o)

Retorna o vetor subjacente de ponteiros PyObject. Presume que *o* foi retornado por *PySequence_Fast()* e que *o* não seja `NULL`.

Note que, se uma lista for redimensionada, a realocação poderá reposicionar o vetor de itens. Portanto, só use o ponteiro de vetor subjacente em contextos onde a sequência não mudará.

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

Retorna valor: Nova referência. Retorna o elemento *i* de *o*, ou `NULL` em caso de falha. É uma forma mais rápida de `PySequence_GetItem()`, mas sem verificar se `PySequence_Check()` em *o* é verdadeiro e sem ajustar índices negativos.

7.5 Protocolo de mapeamento

Veja também `PyObject_GetItem()`, `PyObject_SetItem()` e `PyObject_DelItem()`.

int **PyMapping_Check**(*PyObject* *o)

Parte da ABI Estável. Retorna `1` se o objeto fornece protocolo de mapeamento ou suporta fatiamento e `0` caso contrário. Note que ele retorna `1` para classes Python com um método `__getitem__()` visto que geralmente é impossível determinar a que tipo de chaves a classe tem suporte. Esta função sempre tem sucesso.

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

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

Parte da ABI Estável. Retorna o número de chaves no objeto *o* em caso de sucesso e `-1` em caso de falha. Isso é equivalente à expressão Python `len(o)`.

PyObject ***PyMapping_GetItemString**(*PyObject* *o, const char *key)

Retorna valor: Nova referência. Parte da ABI Estável. É o mesmo que `PyObject_GetItem()`, mas *key* é especificada como uma string de bytes `const char*` codificada em UTF-8, em vez de um `PyObject*`.

int **PyMapping_GetOptionalItem**(*PyObject* *obj, *PyObject* *key, *PyObject* **result)

Parte da ABI Estável *desde a versão 3.13.* Variante de `PyObject_GetItem()` que não levanta `KeyError` se a chave não for encontrada.

Se a chave for encontrada, retorna `1` e define **result* como uma nova *referência forte* para o valor correspondente. Se a chave não for encontrada, retorna `0` e define **result* como `NULL`; o `KeyError` é silenciado. Se um erro diferente de `KeyError` for levanta, retorna `-1` e define **result* como `NULL`.

Adicionado na versão 3.13.

int **PyMapping_GetOptionalItemString**(*PyObject* *obj, const char *key, *PyObject* **result)

Parte da ABI Estável *desde a versão 3.13.* É o mesmo que `PyMapping_GetOptionalItem()`, mas *key* é especificado como uma string de bytes codificada em UTF-8 `const char*`, em vez de um `PyObject*`.

Adicionado na versão 3.13.

int **PyMapping_SetItemString**(*PyObject* *o, const char *key, *PyObject* *v)

Parte da ABI Estável. É o mesmo que `PyObject_SetItem()`, mas *key* é especificada como uma string de bytes `const char*` codificada em UTF-8, em vez de um `PyObject*`.

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

Este é um apelido de `PyObject_DelItem()`.

int **PyMapping_DelItemString**(*PyObject* *o, const char *key)

É o mesmo que `PyObject_DelItem()`, mas *key* é especificada como uma string de bytes `const char*` codificada em UTF-8, em vez de um `PyObject*`.

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

Parte da ABI Estável *desde a versão 3.13.* Retorna `1` se o objeto de mapeamento tiver a chave *key* e `0` caso contrário. Isso é equivalente à expressão Python `key in o`. Em caso de falha, retorna `-1`.

Adicionado na versão 3.13.

int **PyMapping_HasKeyStringWithError**(*PyObject* *o, const char *key)

Parte da ABI Estável *desde a versão 3.13.* É o mesmo que `PyMapping_HasKeyWithError()`, mas *key* é especificada como uma string de bytes `const char*` codificada em UTF-8, em vez de um `PyObject*`.

Adicionado na versão 3.13.

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

Parte da ABI Estável. Retorna `1` se o objeto de mapeamento tiver a chave *key* e `0` caso contrário. Isso é equivalente à expressão Python `key in o`. Esta função sempre tem sucesso.

> **Nota**
>
> As exceções que ocorrem quando esse método chama `__getitem__()` são silenciosamente ignoradas. Para o tratamento adequado de erros, use *PyMapping_HasKeyWithError()*, *PyMapping_GetOptionalItem()* ou *PyObject_GetItem()* em vez disso.

int **PyMapping_HasKeyString**(*PyObject* *o, const char *key)

Parte da ABI Estável. É o mesmo que *PyMapping_HasKey()*, mas *key* é especificada como uma string de bytes `const char*` codificada em UTF-8, em vez de um *PyObject**.

> **Nota**
>
> As exceções que ocorrem quando esse método chama `__getitem__()` ou ao criar o objeto temporário `str` são silenciosamente ignoradas. Para o tratamento adequado de erros, use *PyMapping_HasKeyStringWithError()*, *PyMapping_GetOptionalItemString()* ou *PyMapping_GetItemString()* em vez disso.

PyObject ***PyMapping_Keys**(*PyObject* *o)

Retorna valor: Nova referência. Parte da ABI Estável. Em caso de sucesso, retorna uma lista das chaves no objeto *o*. Em caso de falha, retorna `NULL`.

Alterado na versão 3.7: Anteriormente, a função retornava uma lista ou tupla.

PyObject ***PyMapping_Values**(*PyObject* *o)

Retorna valor: Nova referência. Parte da ABI Estável. Em caso de sucesso, retorna uma lista dos valores no objeto *o*. Em caso de falha, retorna `NULL`.

Alterado na versão 3.7: Anteriormente, a função retornava uma lista ou tupla.

PyObject ***PyMapping_Items**(*PyObject* *o)

Retorna valor: Nova referência. Parte da ABI Estável. Em caso de sucesso, retorna uma lista dos itens no objeto *o*, onde cada item é uma tupla contendo um par de valores-chave. Em caso de falha, retorna `NULL`.

Alterado na versão 3.7: Anteriormente, a função retornava uma lista ou tupla.

7.6 Protocolo Iterador

Existem duas funções especificas para trabalhar com iteradores.

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

Parte da ABI Estável *desde a versão 3.8.* Retorna valor diferente de zero se o objeto *o* puder ser passado com segurança para *PyIter_Next()*, e `0` caso contrário. Esta função sempre é bem-sucedida.

int **PyAIter_Check**(*PyObject* *o)

Parte da ABI Estável *desde a versão 3.10.* Retorna valor diferente de zero se o objeto *o* fornecer o protocolo `AsyncIterator` e `0` caso contrário. Esta função sempre é bem-sucedida.

Adicionado na versão 3.10.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna o próximo valor do iterador *o*. O objeto deve ser um iterador de acordo com *PyIter_Check()* (cabe ao chamador verificar isso). Se não houver valores restantes, retorna `NULL` sem nenhuma exceção definida. Se ocorrer um erro ao recuperar o item, retorna `NULL` e passa a exceção.

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

```
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;

if (iterator == NULL) {
    /* propaga o erro */
}

while ((item = PyIter_Next(iterator))) {
    /* faz algo com o item */
    ...
    /* liberar a referência quando concluído */
    Py_DECREF(item);
}

Py_DECREF(iterator);

if (PyErr_Occurred()) {
    /* propaga o erro */
}
else {
    /* continua fazendo algum trabalho útil */
}
```

type **PySendResult**

O valor de enum usado para representar diferentes resultados de *PyIter_Send()*.

Adicionado na versão 3.10.

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

Parte da ABI Estável *desde a versão 3.10.* Envia o valor *arg* para o iterador *iter*. Retorna:

- `PYGEN_RETURN` se o iterador retornar. O valor de retorno é retornado via *presult*.
- `PYGEN_NEXT` se o iterador render. O valor preduzido é retornado via *presult*.
- `PYGEN_ERROR` se o iterador tiver levantado uma exceção. *presult* é definido como `NULL`.

Adicionado na versão 3.10.

7.7 Protocolo de Buffer

Certos objetos disponíveis no Python envolvem o acesso a um vetor ou *buffer* de memória subjacente. Esses objetos incluem as `bytes` e `bytearray` embutidas, e alguns tipos de extensão como `array.array`. As bibliotecas de terceiros podem definir seus próprios tipos para fins especiais, como processamento de imagem ou análise numérica.

Embora cada um desses tipos tenha sua própria semântica, eles compartilham a característica comum de serem suportados por um buffer de memória possivelmente grande. É desejável, em algumas situações, acessar esse buffer diretamente e sem cópia intermediária.

Python fornece essa facilidade no nível C sob a forma de *protocolo de buffer*. Este protocolo tem dois lados:

- do lado do produtor, um tipo pode exportar uma "interface de buffer" que permite que objetos desse tipo exponham informações sobre o buffer subjacente. Esta interface é descrita na seção *Buffer Object Structures*;
- do lado do consumidor, vários meios estão disponíveis para obter o ponteiro para os dados subjacentes de um objeto (por exemplo, um parâmetro de método).

Objetos simples como `bytes` e `bytearray` expõem seu buffer subjacente em uma forma orientada a byte. Outras formas são possíveis; por exemplo, os elementos expostos por uma `array.array` podem ser valores de vários bytes.

Um exemplo de interface de um consumidor de buffer é o método `write()` de objetos arquivo: qualquer objeto que possa exportar uma série de bytes por meio da interface de buffer pode ser gravado em um arquivo. Enquanto o `write()` precisa apenas de acesso de somente leitura ao conteúdo interno do objeto passado, outros métodos, como `readinto()`, precisam de acesso de somente escrita ao conteúdo interno. A interface de buffer permite que o objetos possam permitir ou rejeitar a exportação para buffers de leitura e escrita ou somente leitura.

Existem duas maneiras para um consumidor da interface de buffer adquirir um buffer em um objeto alvo:

- chamada de *`PyObject_GetBuffer()`* com os parâmetros certos;
- chamada de *`PyArg_ParseTuple()`* (ou um dos seus irmãos) com um dos *códigos de formatação* `y*`, `w*` ou `s*`.

Em ambos os casos, *`PyBuffer_Release()`* deve ser chamado quando o buffer não é mais necessário. A falta de tal pode levar a várias questões, tais como vazamentos de recursos.

7.7.1 Estrutura de Buffer

As estruturas de buffer (ou simplesmente "buffers") são úteis como uma maneira de expor os dados binários de outro objeto para o programador Python. Eles também podem ser usados como um mecanismo de cópia silenciosa. Usando sua capacidade de fazer referência a um bloco de memória, é possível expor facilmente qualquer dado ao programador Python. A memória pode ser uma matriz grande e constante em uma extensão C, pode ser um bloco bruto de memória para manipulação antes de passar para uma biblioteca do sistema operacional, ou pode ser usado para transmitir dados estruturados no formato nativo e formato de memória.

Ao contrário da maioria dos tipos de dados expostos pelo interpretador Python, os buffers não são ponteiros *`PyObject`* mas sim estruturas C simples. Isso permite que eles sejam criados e copiados de forma muito simples. Quando um invólucro genérico em torno de um buffer é necessário, um objeto *memoryview* pode ser criado.

Para obter instruções curtas sobre como escrever um objeto exportador, consulte *Buffer Object Structures*. Para obter um buffer, veja *`PyObject_GetBuffer()`*.

type **`Py_buffer`**

Parte da ABI Estável *(incluindo todos os membros) desde a versão 3.11.*

void ***`buf`**

Um ponteiro para o início da estrutura lógica descrita pelos campos do buffer. Este pode ser qualquer local dentro do bloco de memória física subjacente do exportador. Por exemplo, com negativo *`strides`* o valor pode apontar para o final do bloco de memória.

Para vetores *contíguos*, o valor aponta para o início do bloco de memória.

PyObject ***`obj`**

Uma nova referência ao objeto sendo exporta. A referência pertence ao consumidor e é automaticamente liberada (por exemplo, a contagem de referências é decrementada) e é atribuída para `NULL` por *`PyBuffer_Release()`*. O campo é equivalmente ao valor de retorno de qualquer função do padrão C-API.

Como um caso especial, para buffers *temporários* que são encapsulados por *`PyMemoryView_FromBuffer()`* ou *`PyBuffer_FillInfo()`* esse campo é `NULL`. Em geral, objetos exportadores NÃO DEVEM usar esse esquema.

Py_ssize_t **`len`**

`product(shape) * itemsize`. Para matrizes contíguas, este é o comprimento do bloco de memória subjacente. Para matrizes não contíguas, é o comprimento que a estrutura lógica teria se fosse copiado para uma representação contígua.

Acessando `((char *)buf)[0] up to ((char *)buf)[len-1]` só é válido se o buffer tiver sido obtido por uma solicitação que garanta a contiguidade. Na maioria dos casos, esse pedido será *`PyBUF_SIMPLE`* ou *`PyBUF_WRITABLE`*.

int **`readonly`**

Um indicador de se o buffer é somente leitura. Este campo é controlado pelo sinalizador *`PyBUF_WRITABLE`*.

Py_ssize_t **itemsize**

O tamanho do item em bytes de um único elemento. O mesmo que o valor de `struct.calcsize()` chamado em valores não `NULL` de *format*.

Exceção importante: Se um consumidor requisita um buffer sem sinalizador *PyBUF_FORMAT*, *format* será definido como `NULL`, mas *itemsize* ainda terá seu valor para o formato original.

Se *shape* está presente, a igualdade `product(shape) * itemsize == len` ainda é válida e o usuário pode usar *itemsize* para navegar o buffer.

Se *shape* é `NULL` como resultado de uma *PyBUF_SIMPLE* ou uma requisição *PyBUF_WRITABLE*, o consumidor deve ignorar *itemsize* e presumir `itemsize == 1`.

char ***format**

Uma string terminada por *NULL* no estilo de sintaxe de módulo `struct` descrevendo os conteúdos de um único item. Se isso é `NULL`, `"B"` (unsigned bytes) é presumido.

Este campo é controlado pelo sinalizador *PyBUF_FORMAT*.

int **ndim**

O número de dimensões de memória representado como um array n-dimensional. Se for `0`, *buf* aponta para um único elemento representando um escalar. Neste caso, *shape*, *strides* e *suboffsets* DEVEM ser `NULL`. O número máximo de dimensões é dado por *PyBUF_MAX_NDIM*.

Py_ssize_t ***shape**

Uma matriz de *Py_ssize_t* do comprimento *ndim* indicando a forma da memória como uma matriz n-dimensional. Observe que a forma `shape[0] * ... * shape[ndim-1] * itemsize` DEVE ser igual a *len*.

Os valores da forma são restritos a `shape[n] >= 0`. The case `shape[n] == 0` requer atenção especial. Veja *complex arrays* para mais informações.

A forma de acesso a matriz é de somente leitura para o usuário.

Py_ssize_t ***strides**

Um vetor de *Py_ssize_t* de comprimento *ndim* dando o número de bytes para saltar para obter um novo elemento em cada dimensão.

Os valores de Stride podem ser qualquer número inteiro. Para arrays regulares, os passos são geralmente positivos, mas um consumidor DEVE ser capaz de lidar com o caso `strides[n] <= 0`. Veja *complex arrays* para mais informações.

A matriz de passos é somente leitura para o consumidor.

Py_ssize_t ***suboffsets**

Uma matriz de *Py_ssize_t* de comprimento *ndim*. Se `suboffsets[n] >= 0`, os valores armazenados ao longo da n-ésima dimensão são ponteiros e o valor suboffset determina quantos bytes para adicionar a cada ponteiro após desreferenciar. Um valor de suboffset que é negativo indica que não deve ocorrer desreferenciação (caminhando em um bloco de memória contíguo).

Se todos os subconjuntos forem negativos (ou seja, não é necessário fazer referência), então este campo deve ser `NULL` (o valor padrão).

Esse tipo de representação de matriz é usado pela Python Imaging Library (PIL). Veja *complex arrays* para obter mais informações sobre como acessar elementos dessa matriz.a matriz.

A matriz de subconjuntos é somente leitura para o consumidor.

void ***internal**

Isso é para uso interno pelo objeto exportador. Por exemplo, isso pode ser re-moldado como um número inteiro pelo exportador e usado para armazenar bandeiras sobre se os conjuntos de forma, passos e suboffsets devem ou não ser liberados quando o buffer é liberado. O consumidor NÃO DEVE alterar esse valor.

Constantes:

PyBUF_MAX_NDIM

O número máximo de dimensões que a memória representa. Exportadores DEVEM respeitar esse limite, consumidores de buffers multi-dimensionais DEVEM ser capazes de liader com até `PyBUF_MAX_NDIM` dimensões. Atualmente definido como 64.

7.7.2 Tipos de solicitação do buffer

Os buffers geralmente são obtidos enviando uma solicitação de buffer para um objeto exportador via *PyObject_GetBuffer()*. Uma vez que a complexidade da estrutura lógica da memória pode variar drasticamente, o consumidor usa o argumento *flags* para especificar o tipo de buffer exato que pode manipular.

Todos os campos *Py_buffer* são definidos de forma não-ambígua pelo tipo de requisição.

campos independentes do pedido

Os seguintes campos não são influenciados por *flags* e devem sempre ser preenchidos com os valores corretos: *obj*, *buf*, *len*, *itemsize*, *ndim*.

apenas em formato

PyBUF_WRITABLE

Controla o campo *readonly* . Se configurado, o exportador DEVE fornecer um buffer gravável ou então relatar falha. Do contrário, o exportador PODE fornecer um buffer somente leitura ou um buffer gravável , mas a escolha DEVE ser consistente para todos os consumidores. Por exemplo, *PyBUF_SIMPLE* | *PyBUF_WRITABLE* pode ser usado para requisitar um buffer simples gravável.

PyBUF_FORMAT

Controla o campo *format*. Se configurado, este campo DEVE ser preenchido corretamente. Caso contrário, este campo DEVE ser `NULL`.

:*PyBUF_WRITABLE* pode ser |'d para qualquer um dos sinalizadores na próxima seção. Uma vez que *PyBUF_WRITABLE* é definido como 0, *PyBUF_WRITABLE* pode ser usado como uma bandeira autônoma para solicitar um buffer simples gravável.

PyBUF_FORMAT deve ser |'d para qualquer um dos sinalizadores exceto *PyBUF_SIMPLE*, uma vez que este já indica que o formato `B` (unsigned bytes). `PyBUF_FORMAT` não pode ser utilizado sozinho.

forma, avanços, suboffsets

As bandeiras que controlam a estrutura lógica da memória estão listadas em ordem decrescente de complexidade. Observe que cada bandeira contém todos os bits das bandeiras abaixo.

Solicitação	Forma	Avanços	subconjuntos
PyBUF_INDIRECT	sim	sim	se necessário
PyBUF_STRIDES	sim	sim	NULL
PyBUF_ND	sim	NULL	NULL
PyBUF_SIMPLE	NULL	NULL	NULL

requisições contíguas

contiguity do C ou Fortran podem ser explicitamente solicitadas, com ou sem informação de avanço. Sem informação de avanço, o buffer deve ser C-contíguo.

Solicitação	Forma	Avanços	subconjuntos	contig
PyBUF_C_CONTIGUOUS	sim	sim	NULL	C
PyBUF_F_CONTIGUOUS	sim	sim	NULL	F
PyBUF_ANY_CONTIGUOUS	sim	sim	NULL	C ou F
PyBUF_ND	sim	NULL	NULL	C

requisições compostas

Todas as requisições possíveis foram completamente definidas por alguma combinação dos sinalizadores na seção anterior. Por conveniência, o protocolo do buffer fornece combinações frequentemente utilizadas como sinalizadores únicos.

Na seguinte tabela *U* significa contiguidade indefinida. O consumidor deve chamar *PyBuffer_IsContiguous()* para determinar a contiguidade.

Solicitação	Forma	Avanços	subconjuntos	contig	readonly	formato
PyBUF_FULL	sim	sim	se necessário	U	0	sim
PyBUF_FULL_RO	sim	sim	se necessário	U	1 ou 0	sim
PyBUF_RECORDS	sim	sim	NULL	U	0	sim
PyBUF_RECORDS_RO	sim	sim	NULL	U	1 ou 0	sim
PyBUF_STRIDED	sim	sim	NULL	U	0	NULL
PyBUF_STRIDED_RO	sim	sim	NULL	U	1 ou 0	NULL
PyBUF_CONTIG	sim	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	sim	NULL	NULL	C	1 ou 0	NULL

7.7.3 Vetores Complexos

Estilo NumPy: forma e avanços

A estrutura lógica de vetores do estilo NumPy é definida por *itemsize*, *ndim*, *shape* e *strides*.

Se `ndim == 0`, a localização da memória apontada para *buf* é interpretada como um escalar de tamanho *itemsize*. Nesse caso, ambos *shape* e *strides* são `NULL`.

Se *strides* é `NULL`, o vetor é interpretado como um vetor C n-dimensional padrão. Caso contrário, o consumidor deve acessar um vetor n-dimensional como a seguir:

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

Como notado acima, *buf* pode apontar para qualquer localização dentro do bloco de memória em si. Um exportador pode verificar a validade de um buffer com essa função:

```
def verify_structure(memlen, itemsize, ndim, shape, strides, offset):
    """Verifica se os parâmetros representa um vetor válido dentro
       dos limites da memória alocada:
           char *mem: início do bloco de memória física
           memlen: comprimento do bloco de memória física
           offset: (char *)buf - mem
    """
    if offset % itemsize:
        return False
    if offset < 0 or offset+itemsize > memlen:
        return False
    if any(v % itemsize for v in strides):
        return False

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

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

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

Estilo-PIL: forma, avanços e suboffsets

Além dos itens normais, uma matriz em estilo PIL pode conter ponteiros que devem ser seguidos para se obter o próximo elemento em uma dimensão. Por exemplo, a matriz tridimensional em C `char v[2][2][3]` também pode ser vista como um vetor de 2 ponteiros para duas matrizes bidimensionais: `char (*v[2])[2][3]`. Na representação por suboffsets, esses dois ponteiros podem ser embutidos no início de *buf*, apontando para duas matrizes `char x[2][3]` que podem estar localizadas em qualquer lugar na memória.

Esta é uma função que retorna um ponteiro para o elemento em uma matriz N-D apontada por um índice N-dimensional onde existem ambos passos e subconjuntos não-`NULL`:

```
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++) {
```

(continua na próxima página)

(continuação da página anterior)

```
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >=0 ) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

7.7.4 Funções relacionadas ao Buffer

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

Parte da ABI Estável *desde a versão 3.11.* Retorna `1` se *obj* oferece suporte à interface de buffer, se não, `0`. Quando `1` é retornado, isso não garante que *PyObject_GetBuffer()* será bem sucedida. Esta função é sempre bem sucedida.

int **PyObject_GetBuffer**(*PyObject* *exporter, *Py_buffer* *view, int flags)

Parte da ABI Estável *desde a versão 3.11.* Envia uma requisição ao *exporter* para preencher a *view* conforme especificado por *flags*. Se o exporter não conseguir prover um buffer do tipo especificado, ele DEVE levantar `BufferError`, definir `view->obj` para `NULL` e retornar `-1`.

Em caso de sucesso, preenche *view*, define `view->obj` para uma nova referência para *exporter* e retorna 0. No caso de provedores de buffer encadeados que redirecionam requisições para um único objeto, `view->obj` DEVE se referir a este objeto em vez de *exporter* (Veja *Buffer Object Structures*).

Chamadas bem sucedidas para *PyObject_GetBuffer()* devem ser emparelhadas a chamadas para *PyBuffer_Release()*, similar para `malloc()` e `free()`. Assim, após o consumidor terminar com o buffer, *PyBuffer_Release()* deve ser chamado exatamente uma vez.

void **PyBuffer_Release**(*Py_buffer* *view)

Parte da ABI Estável *desde a versão 3.11.* Libera o buffer de *view* e libera o *strong reference* (por exemplo, decrementa o contador de referências) para o objeto de suporte da view, `view->obj`. Esta função DEVE ser chamada quando o buffer não estiver mais sendo usado, ou o vazamento de referências pode acontecer.

É um erro chamar essa função em um buffer que não foi obtido via *PyObject_GetBuffer()*.

Py_ssize_t **PyBuffer_SizeFromFormat**(const char *format)

Parte da ABI Estável *desde a versão 3.11.* Retorna o *itemsize* implícito de *format*. Em erro, levantar e exceção e retornar -1.

Adicionado na versão 3.9.

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

Parte da ABI Estável *desde a versão 3.11.* Retorna `1` se a memória definida pela *view* é *contígua* no estilo C (*order* é `'C'`) ou no estilo Fortran (*order* é `'F'`) ou qualquer outra (*order* é `'A'`). Retorna `0` caso contrário. Essa função é sempre bem sucedida.

void ***PyBuffer_GetPointer**(const *Py_buffer* *view, const *Py_ssize_t* *indices)

Parte da ABI Estável *desde a versão 3.11.* Recebe a área de memória apontada pelos *indices* dentro da *view* dada. *indices* deve apontar para um array de `view->ndim` índices.

int **PyBuffer_FromContiguous**(const *Py_buffer* *view, const void *buf, *Py_ssize_t* len, char fort)

Parte da ABI Estável *desde a versão 3.11.* Copia *len* bytes contíguos de *buf* para *view*. *fort* pode ser `'C'` ou `'F'` (para ordenação estilo C ou estilo Fortran). Retorna `0` em caso de sucesso e `-1` em caso de erro.

int **PyBuffer_ToContiguous**(void *buf, const *Py_buffer* *src, *Py_ssize_t* len, char order)

Parte da ABI Estável *desde a versão 3.11.* Copia *len* bytes de *src* para sua representação contígua em *buf*. *order* pode ser `'C'` ou `'F'` ou `'A'` (para ordenação estilo C, Fortran ou qualquer uma). O retorno é `0` em caso de sucesso e `-1` em caso de falha.

Esta função falha se *len* != *src->len*.

int **PyObject_CopyData**(*PyObject* *dest, *PyObject* *src)

Parte da ABI Estável *desde a versão 3.11.* Copia os dados do buffer *src* para o buffer *dest*. Pode converter entre buffers de estilo C e/ou estilo Fortran.

`0` é retornado em caso de sucesso, `-1` em caso de erro.

void **PyBuffer_FillContiguousStrides**(int ndims, *Py_ssize_t* *shape, *Py_ssize_t* *strides, int itemsize, char order)

Parte da ABI Estável *desde a versão 3.11.* Preenche o array *strides* com byte-strides de um array *contíguo* (estilo C se *order* é `'C'` ou estilo Fortran se *order* for `'F'`) da forma dada com o número dado de bytes por elemento.

int **PyBuffer_FillInfo**(*Py_buffer* *view, *PyObject* *exporter, void *buf, *Py_ssize_t* len, int readonly, int flags)

Parte da ABI Estável *desde a versão 3.11.* Manipula requisições de buffer para um exportador que quer expor *buf* de tamanho *len* com capacidade de escrita definida de acordo com *readonly*. *buf* é interpretada como uma sequência de bytes sem sinal.

O argumento *flags* indica o tipo de requisição. Esta função sempre preenche *view* como especificado por *flags*, a não ser que *buf* seja designado como somente leitura e *PyBUF_WRITABLE* esteja definido em *flags*.

Em caso de sucesso, defina `view->obj` como um novo referência para *exporter* e retorna 0. Caso contrário, levante `BufferError` , defina `view->obj` para `NULL` e retorne `-1` ;

Se esta função é usada como parte de um *getbufferproc*, *exporter* DEVE ser definida para o objeto de exportação e *flags* deve ser passado sem modificações. Caso contrário, *exporter* DEVE ser `NULL`.

CAPÍTULO 8

Camada de Objetos Concretos

As funções neste capítulo são específicas para certos tipos de objetos Python. Passar para eles um objeto do tipo errado não é uma boa ideia; se você receber um objeto de um programa Python e não tiver certeza de que ele tem o tipo certo, primeiro execute uma verificação de tipo; por exemplo, para verificar se um objeto é um dicionário, use `PyDict_Check()`. O capítulo está estruturado como a "árvore genealógica" dos tipos de objetos Python.

Aviso

Enquanto as funções descritas neste capítulo verificam cuidadosamente o tipo de objetos passados, muitos deles não verificam a passagem de `NULL` em vez de um objeto válido. Permitir a passagem de `NULL` pode causar violações ao acesso à memória e encerramento imediato do interpretador.

8.1 Objetos Fundamentais

Esta seção descreve os objetos de tipo Python e o objeto singleton `None`.

8.1.1 Objetos tipo

type **PyTypeObject**

Parte da API Limitada *(como uma estrutura opaca).* A estrutura C dos objetos usados para descrever tipos embutidos.

PyTypeObject **PyType_Type**

Parte da ABI Estável. Este é o objeto de tipo para objetos tipo; é o mesmo objeto que `type` na camada Python.

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

Retorna valor diferente de zero se o objeto *o* for um objeto tipo, incluindo instâncias de tipos derivados do objeto tipo padrão. Retorna 0 em todos os outros casos. Esta função sempre tem sucesso.

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

Retorna valor diferente de zero se o objeto *o* for um objeto tipo, mas não um subtipo do objeto tipo padrão. Retorna 0 em todos os outros casos. Esta função sempre tem sucesso.

unsigned int **PyType_ClearCache**()

Parte da ABI Estável. Limpa o cache de pesquisa interno. Retorna a marcação de versão atual.

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

Parte da ABI Estável. Return the `tp_flags` member of *type*. This function is primarily meant for use with `Py_LIMITED_API`; the individual flag bits are guaranteed to be stable across Python releases, but access to `tp_flags` itself is not part of the *limited API*.

Adicionado na versão 3.2.

Alterado na versão 3.4: O tipo de retorno é agora um `unsigned long` em vez de um `long`.

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

Return the type object's internal namespace, which is otherwise only exposed via a read-only proxy (`cls.__dict__`). This is a replacement for accessing `tp_dict` directly. The returned dictionary must be treated as read-only.

This function is meant for specific embedding and language-binding cases, where direct access to the dict is necessary and indirect access (e.g. via the proxy or `PyObject_GetAttr()`) isn't adequate.

Extension modules should continue to use `tp_dict`, directly or indirectly, when setting up their own types.

Adicionado na versão 3.12.

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

Parte da ABI Estável. Invalida o cache de pesquisa interna para o tipo e todos os seus subtipos. Esta função deve ser chamada após qualquer modificação manual dos atributos ou classes bases do tipo.

int **PyType_AddWatcher**(*PyType_WatchCallback* callback)

Register *callback* as a type watcher. Return a non-negative integer ID which must be passed to future calls to `PyType_Watch()`. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Adicionado na versão 3.12.

int **PyType_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* (previously returned from `PyType_AddWatcher()`). Return `0` on success, `-1` on error (e.g. if *watcher_id* was never registered.)

An extension should never call `PyType_ClearWatcher` with a *watcher_id* that was not returned to it by a previous call to `PyType_AddWatcher()`.

Adicionado na versão 3.12.

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

Mark *type* as watched. The callback granted *watcher_id* by `PyType_AddWatcher()` will be called whenever `PyType_Modified()` reports a change to *type*. (The callback may be called only once for a series of consecutive modifications to *type*, if `_PyType_Lookup()` is not called on *type* between the modifications; this is an implementation detail and subject to change.)

An extension should never call `PyType_Watch` with a *watcher_id* that was not returned to it by a previous call to `PyType_AddWatcher()`.

Adicionado na versão 3.12.

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

Type of a type-watcher callback function.

The callback must not modify *type* or cause `PyType_Modified()` to be called on *type* or any type in its MRO; violating this rule could cause infinite recursion.

Adicionado na versão 3.12.

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

Retorna valor diferente de zero se o objeto tipo *o* define o recurso *feature*. Os recursos de tipo são denotados por sinalizadores de bit único.

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

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

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

Parte da ABI Estável. Retorna verdadeiro se *a* for um subtipo de *b*.

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

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

Retorna valor: Nova referência. *Parte da* ABI Estável. Manipulador genérico para o slot `tp_alloc` de um objeto tipo. Use o mecanismo de alocação de memória padrão do Python para alocar uma nova instância e inicializar todo o seu conteúdo para `NULL`.

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

Retorna valor: Nova referência. *Parte da* ABI Estável. Manipulador genérico para o slot `tp_new` de um objeto tipo. Cria uma nova instância usando o slot `tp_alloc` do tipo.

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

Parte da ABI Estável. Finaliza um objeto tipo. Isso deve ser chamado em todos os objetos tipo para finalizar sua inicialização. Esta função é responsável por adicionar slots herdados da classe base de um tipo. Retorna `0` em caso de sucesso, ou retorna `-1` e define uma exceção em caso de erro.

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

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

Retorna valor: Nova referência. *Parte da* ABI Estável *desde a versão 3.11.* Return the type's name. Equivalent to getting the type's `__name__` attribute.

Adicionado na versão 3.11.

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

Retorna valor: Nova referência. *Parte da* ABI Estável *desde a versão 3.11.* Return the type's qualified name. Equivalent to getting the type's `__qualname__` attribute.

Adicionado na versão 3.11.

PyObject ***PyType_GetFullyQualifiedName**(*PyTypeObject* *type)

Parte da ABI Estável *desde a versão 3.13.* Return the type's fully qualified name. Equivalent to `f"{type.__module__}.{type.__qualname__}"`, or `type.__qualname__` if `type.__module__` is not a string or is equal to `"builtins"`.

Adicionado na versão 3.13.

PyObject ***PyType_GetModuleName**(*PyTypeObject* *type)

Parte da ABI Estável *desde a versão 3.13.* Return the type's module name. Equivalent to getting the `type.__module__` attribute.

Adicionado na versão 3.13.

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

Parte da ABI Estável *desde a versão 3.4.* Retorna o ponteiro de função armazenado no slot fornecido. Se o resultado for `NULL`, isso indica que o slot é `NULL` ou que a função foi chamada com parâmetros inválidos. Os chamadores normalmente lançarão o ponteiro do resultado no tipo de função apropriado.

Veja `PyType_Slot.slot` por possíveis valores do argumento *slot*.

Adicionado na versão 3.4.

Alterado na versão 3.10: `PyType_GetSlot()` can now accept all types. Previously, it was limited to *heap types*.

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

Parte da ABI Estável *desde a versão 3.10.* Retorna o objeto de módulo associado ao tipo fornecido quando o tipo foi criado usando `PyType_FromModuleAndSpec()`.

Se nenhum módulo estiver associado com o tipo fornecido, define `TypeError` e retorna `NULL`.

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

Adicionado na versão 3.9.

void ***PyType_GetModuleState**(*PyTypeObject* *type)

Parte da ABI Estável *desde a versão 3.10.* Return the state of the module object associated with the given type. This is a shortcut for calling `PyModule_GetState()` on the result of `PyType_GetModule()`.

Se nenhum módulo estiver associado com o tipo fornecido, define `TypeError` e retorna `NULL`.

If the *type* has an associated module but its state is `NULL`, returns `NULL` without setting an exception.

Adicionado na versão 3.9.

PyObject ***PyType_GetModuleByDef**(*PyTypeObject* *type, struct *PyModuleDef* *def)

Parte da ABI Estável *desde a versão 3.13.* Find the first superclass whose module was created from the given `PyModuleDef` *def*, and return that module.

If no module is found, raises a `TypeError` and returns `NULL`.

This function is intended to be used together with `PyModule_GetState()` to get module state from slot methods (such as `tp_init` or `nb_add`) and other places where a method's defining class cannot be passed using the `PyCMethod` calling convention.

Adicionado na versão 3.11.

int **PyUnstable_Type_AssignVersionTag**(*PyTypeObject* *type)

Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Attempt to assign a version tag to the given type.

Returns 1 if the type already had a valid version tag or a new one was assigned, or 0 if a new tag could not be assigned.

Adicionado na versão 3.12.

Creating Heap-Allocated Types

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

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

Parte da ABI Estável *desde a versão 3.12.* Create and return a *heap type* from the *spec* (see `Py_TPFLAGS_HEAPTYPE`).

The metaclass *metaclass* is used to construct the resulting type object. When *metaclass* is `NULL`, the metaclass is derived from *bases* (or *Py_tp_base[s]* slots if *bases* is `NULL`, see below).

Metaclasses that override `tp_new` are not supported, except if `tp_new` is `NULL`. (For backwards compatibility, other `PyType_From*` functions allow such metaclasses. They ignore `tp_new`, which may result in incomplete initialization. This is deprecated and in Python 3.14+ such metaclasses will not be supported.)

The *bases* argument can be used to specify base classes; it can either be only one class or a tuple of classes. If *bases* is `NULL`, the *Py_tp_bases* slot is used instead. If that also is `NULL`, the *Py_tp_base* slot is used instead. If that also is `NULL`, the new type derives from `object`.

The *module* argument can be used to record the module in which the new class is defined. It must be a module object or `NULL`. If not `NULL`, the module is associated with the new type and can later be retrieved with *PyType_GetModule()*. The associated module is not inherited by subclasses; it must be specified for each class individually.

This function calls *PyType_Ready()* on the new type.

Note that this function does *not* fully match the behavior of calling `type()` or using the `class` statement. With user-provided base types or metaclasses, prefer *calling* `type` (or the metaclass) over `PyType_From*` functions. Specifically:

- `__new__()` is not called on the new class (and it must be set to `type.__new__`).
- `__init__()` is not called on the new class.
- `__init_subclass__()` is not called on any bases.
- `__set_name__()` is not called on new descriptors.

Adicionado na versão 3.12.

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

Retorna valor: Nova referência. *Parte da ABI Estável desde a versão 3.10.* Equivalent to `PyType_FromMetaclass(NULL, module, spec, bases)`.

Adicionado na versão 3.9.

Alterado na versão 3.10: The function now accepts a single class as the *bases* argument and `NULL` as the `tp_doc` slot.

Alterado na versão 3.12: The function now finds and uses a metaclass corresponding to the provided base classes. Previously, only `type` instances were returned.

The *tp_new* of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides *tp_new* is deprecated and in Python 3.14+ it will be no longer allowed.

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

Retorna valor: Nova referência. *Parte da ABI Estável desde a versão 3.3.* Equivalent to `PyType_FromMetaclass(NULL, NULL, spec, bases)`.

Adicionado na versão 3.3.

Alterado na versão 3.12: The function now finds and uses a metaclass corresponding to the provided base classes. Previously, only `type` instances were returned.

The *tp_new* of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides *tp_new* is deprecated and in Python 3.14+ it will be no longer allowed.

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

Retorna valor: Nova referência. *Parte da ABI Estável.* Equivalent to `PyType_FromMetaclass(NULL, NULL, spec, NULL)`.

Alterado na versão 3.12: The function now finds and uses a metaclass corresponding to the base classes provided in *Py_tp_base[s]* slots. Previously, only `type` instances were returned.

The *tp_new* of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides *tp_new* is deprecated and in Python 3.14+ it will be no longer allowed.

type **PyType_Spec**

Parte da ABI Estável (incluindo todos os membros). Structure defining a type's behavior.

const char ***name**

Name of the type, used to set *PyTypeObject.tp_name*.

int **basicsize**

If positive, specifies the size of the instance in bytes. It is used to set *PyTypeObject.tp_basicsize*.

If zero, specifies that *tp_basicsize* should be inherited.

If negative, the absolute value specifies how much space instances of the class need *in addition* to the superclass. Use *PyObject_GetTypeData()* to get a pointer to subclass-specific memory reserved this way. For negative `basicsize`, Python will insert padding when needed to meet *tp_basicsize*'s alignment requirements.

Alterado na versão 3.12: Previously, this field could not be negative.

int **itemsize**

Size of one element of a variable-size type, in bytes. Used to set *PyTypeObject.tp_itemsize*. See `tp_itemsize` documentation for caveats.

If zero, *tp_itemsize* is inherited. Extending arbitrary variable-sized classes is dangerous, since some types use a fixed offset for variable-sized memory, which can then overlap fixed-sized memory used by a subclass. To help prevent mistakes, inheriting `itemsize` is only possible in the following situations:

- The base is not variable-sized (its *tp_itemsize*).
- The requested *PyType_Spec.basicsize* is positive, suggesting that the memory layout of the base class is known.
- The requested *PyType_Spec.basicsize* is zero, suggesting that the subclass does not access the instance's memory directly.
- With the *Py_TPFLAGS_ITEMS_AT_END* flag.

unsigned int **flags**

Type flags, used to set *PyTypeObject.tp_flags*.

If the `Py_TPFLAGS_HEAPTYPE` flag is not set, *PyType_FromSpecWithBases()* sets it automatically.

PyType_Slot ***slots**

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

Each slot ID should be specified at most once.

type **PyType_Slot**

Parte da ABI Estável *(incluindo todos os membros).* Structure defining optional functionality of a type, containing a slot ID and a value pointer.

int **slot**

A slot ID.

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

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

The following "offset" fields cannot be set using *PyType_Slot*:

- *tp_weaklistoffset* (use *Py_TPFLAGS_MANAGED_WEAKREF* instead if possible)
- *tp_dictoffset* (use *Py_TPFLAGS_MANAGED_DICT* instead if possible)
- *tp_vectorcall_offset* (use `"__vectorcalloffset__"` in *PyMemberDef*)

If it is not possible to switch to a `MANAGED` flag (for example, for vectorcall or to support Python older than 3.12), specify the offset in *Py_tp_members*. See *PyMemberDef documentation* for details.

The following fields cannot be set at all when creating a heap type:

- *tp_vectorcall* (use *tp_new* and/or *tp_init*)
- Internal fields: *tp_dict*, *tp_mro*, *tp_cache*, *tp_subclasses*, and *tp_weaklist*.

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

Alterado na versão 3.9: Slots in *PyBufferProcs* may be set in the unlimited API.

Alterado na versão 3.11: *bf_getbuffer* and *bf_releasebuffer* are now available under the *limited API*.

void ***pfunc**

The desired value of the slot. In most cases, this is a pointer to a function.

Slots other than `Py_tp_doc` may not be `NULL`.

8.1.2 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`, denotando falta de valor. Este objeto não tem métodos e é *imortal*.

Alterado na versão 3.12: *Py_None* é *imortal*.

Py_RETURN_NONE

Retorna *Py_None* de uma função.

8.2 Objetos Numéricos

8.2.1 Objetos Inteiros

Todos os inteiros são implementados como objetos inteiros "longos" de tamanho arbitrário.

Em caso de erro, a maioria das APIs `PyLong_As*` retorna `(tipo de retorno)-1` que não pode ser distinguido de um número. Use *PyErr_Occurred()* para desambiguar.

type **PyLongObject**

Parte da API Limitada *(como uma estrutura opaca).* Este subtipo de *PyObject* representa um objeto inteiro Python.

PyTypeObject **PyLong_Type**

Parte da ABI Estável. Esta instância de *PyTypeObject* representa o tipo inteiro Python. Este é o mesmo objeto que `int` na camada Python.

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

Retorna true se seu argumento é um *PyLongObject* ou um subtipo de *PyLongObject*. Esta função sempre tem sucesso.

int **PyLong_CheckExact**(*PyObject* *p)

Retorna true se seu argumento é um *PyLongObject*, mas não um subtipo de *PyLongObject*. Esta função sempre tem sucesso.

PyObject ***PyLong_FromLong**(long v)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um novo objeto *PyLongObject* de *v* ou `NULL` em caso de falha.

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

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

Retorna valor: Nova referência. *Parte da* ABI Estável. Return a new *PyLongObject* object from a C `unsigned long`, or `NULL` on failure.

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

Retorna valor: Nova referência. *Parte da* ABI Estável. Retorna um novo objeto *PyLongObject* de um *Py_ssize_t* C ou `NULL` em caso de falha.

PyObject ***PyLong_FromSize_t**(size_t v)

Retorna valor: Nova referência. *Parte da* ABI Estável. Retorna um novo objeto *PyLongObject* de um `size_t` C ou `NULL` em caso de falha.

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

Retorna valor: Nova referência. *Parte da* ABI Estável. Return a new *PyLongObject* object from a C `long long`, or `NULL` on failure.

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

Retorna valor: Nova referência. *Parte da* ABI Estável. Return a new *PyLongObject* object from a C `unsigned long long`, or `NULL` on failure.

PyObject ***PyLong_FromDouble**(double v)

Retorna valor: Nova referência. *Parte da* ABI Estável. Retorna um novo objeto *PyLongObject* da parte inteira de *v* ou `NULL` em caso de falha.

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

Retorna valor: Nova referência. *Parte da* ABI Estável. Return a new *PyLongObject* based on the string value in *str*, which is interpreted according to the radix in *base*, or `NULL` on failure. If *pend* is non-`NULL`, **pend* will point to the end of *str* on success or to the first character that could not be processed on error. If *base* is `0`, *str* is interpreted using the integers definition; in this case, leading zeros in a non-zero decimal number raises a `ValueError`. If *base* is not `0`, it must be between `2` and `36`, inclusive. Leading and trailing whitespace and single underscores after a base specifier and between digits are ignored. If there are no digits or *str* is not NULL-terminated following the digits and trailing whitespace, `ValueError` will be raised.

> **Ver também**
>
> Python methods `int.to_bytes()` and `int.from_bytes()` to convert a *PyLongObject* to/from an array of bytes in base `256`. You can call those from C using *PyObject_CallMethod()*.

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

Retorna valor: Nova referência. Converte uma sequência de dígitos Unicode na string *u* para um valor inteiro Python.

Adicionado na versão 3.3.

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

Retorna valor: Nova referência. *Parte da* ABI Estável. Cria um inteiro Python a partir do ponteiro *p*. O valor do ponteiro pode ser recuperado do valor resultante usando *PyLong_AsVoidPtr()*.

PyObject ***PyLong_FromNativeBytes**(const void *buffer, size_t n_bytes, int flags)

Create a Python integer from the value contained in the first *n_bytes* of *buffer*, interpreted as a two's-complement signed number.

flags are as for *PyLong_AsNativeBytes()*. Passing `-1` will select the native endian that CPython was compiled with and assume that the most-significant bit is a sign bit. Passing `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` will produce the same result as calling *PyLong_FromUnsignedNativeBytes()*. Other flags are ignored.

Adicionado na versão 3.13.

PyObject ***PyLong_FromUnsignedNativeBytes**(const void *buffer, size_t n_bytes, int flags)

Create a Python integer from the value contained in the first *n_bytes* of *buffer*, interpreted as an unsigned number.

flags are as for `PyLong_AsNativeBytes()`. Passing `-1` will select the native endian that CPython was compiled with and assume that the most-significant bit is not a sign bit. Flags other than endian are ignored.

Adicionado na versão 3.13.

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

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

Raise `OverflowError` if the value of *obj* is out of range for a `long`.

Retorna `-1` no caso de erro. Use `PyErr_Occurred()` para desambiguar.

Alterado na versão 3.8: Usa `__index__()`, se disponível.

Alterado na versão 3.10: This function will no longer use `__int__()`.

long **PyLong_AS_LONG**(*PyObject* *obj)

A *soft deprecated* alias. Exactly equivalent to the preferred `PyLong_AsLong`. In particular, it can fail with `OverflowError` or another exception.

Descontinuado desde a versão 3.14: The function is soft deprecated.

int **PyLong_AsInt**(*PyObject* *obj)

Parte da ABI Estável *desde a versão 3.13*. Similar to `PyLong_AsLong()`, but store the result in a C `int` instead of a C `long`.

Adicionado na versão 3.13.

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

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

If the value of *obj* is greater than `LONG_MAX` or less than `LONG_MIN`, set **overflow* to `1` or `-1`, respectively, and return `-1`; otherwise, set **overflow* to `0`. If any other exception occurs set **overflow* to `0` and return `-1` as usual.

Retorna `-1` no caso de erro. Use `PyErr_Occurred()` para desambiguar.

Alterado na versão 3.8: Usa `__index__()`, se disponível.

Alterado na versão 3.10: This function will no longer use `__int__()`.

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

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

Raise `OverflowError` if the value of *obj* is out of range for a `long long`.

Retorna `-1` no caso de erro. Use `PyErr_Occurred()` para desambiguar.

Alterado na versão 3.8: Usa `__index__()`, se disponível.

Alterado na versão 3.10: This function will no longer use `__int__()`.

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

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

If the value of *obj* is greater than `LLONG_MAX` or less than `LLONG_MIN`, set **overflow* to `1` or `-1`, respectively, and return `-1`; otherwise, set **overflow* to `0`. If any other exception occurs set **overflow* to `0` and return `-1` as usual.

Retorna `-1` no caso de erro. Use `PyErr_Occurred()` para desambiguar.

Adicionado na versão 3.2.

Alterado na versão 3.8: Usa `__index__()`, se disponível.

Alterado na versão 3.10: This function will no longer use `__int__()`.

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

Parte da ABI Estável. Retorna uma representação de *Py_ssize_t* C de *pylong*. *pylong* deve ser uma instância de *PyLongObject*.

Levanta `OverflowError` se o valor de *pylong* estiver fora do intervalo de um *Py_ssize_t*.

Retorna `-1` no caso de erro. Use *PyErr_Occurred()* para desambiguar.

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

Parte da ABI Estável. Return a C `unsigned long` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a `unsigned long`.

Retorna `(unsigned long)-1` no caso de erro. Use *PyErr_Occurred()* para desambiguar.

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

Parte da ABI Estável. Retorna uma representação de `size_t` C de *pylong*. *pylong* deve ser uma instância de *PyLongObject*.

Levanta `OverflowError` se o valor de *pylong* estiver fora do intervalo de um `size_t`.

Retorna `(size)-1` no caso de erro. Use *PyErr_Occurred()* para desambiguar.

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

Parte da ABI Estável. Return a C `unsigned long long` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for an `unsigned long long`.

Retorna `(unsigned long long)-1` no caso de erro. Use *PyErr_Occurred()* para desambiguar.

Alterado na versão 3.1: Um *pylong* negativo agora levanta `OverflowError`, não `TypeError`.

unsigned long **PyLong_AsUnsignedLongMask**(*PyObject* *obj)

Parte da ABI Estável. Return a C `unsigned long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

If the value of *obj* is out of range for an `unsigned long`, return the reduction of that value modulo `ULONG_MAX + 1`.

Retorna `(unsigned long)-1` no caso de erro. Use *PyErr_Occurred()* para desambiguar.

Alterado na versão 3.8: Usa `__index__()`, se disponível.

Alterado na versão 3.10: This function will no longer use `__int__()`.

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

Parte da ABI Estável. Return a C `unsigned long long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

If the value of *obj* is out of range for an `unsigned long long`, return the reduction of that value modulo `ULLONG_MAX + 1`.

Retorna `(unsigned long long)-1` no caso de erro. Use *PyErr_Occurred()* para desambiguar.

Alterado na versão 3.8: Usa `__index__()`, se disponível.

Alterado na versão 3.10: This function will no longer use `__int__()`.

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

Parte da ABI Estável. Return a C `double` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a `double`.

Retorna `-1.0` no caso de erro. Use *PyErr_Occurred()* para desambiguar.

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

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

Retorna `NULL` no caso de erro. Use *PyErr_Occurred()* para desambiguar.

Py_ssize_t **PyLong_AsNativeBytes**(*PyObject* *pylong, void *buffer, *Py_ssize_t* n_bytes, int flags)

Copy the Python integer value *pylong* to a native *buffer* of size *n_bytes*. The *flags* can be set to `-1` to behave similarly to a C cast, or to values documented below to control the behavior.

Returns `-1` with an exception raised on error. This may happen if *pylong* cannot be interpreted as an integer, or if *pylong* was negative and the `Py_ASNATIVEBYTES_REJECT_NEGATIVE` flag was set.

Otherwise, returns the number of bytes required to store the value. If this is equal to or less than *n_bytes*, the entire value was copied. All *n_bytes* of the buffer are written: large buffers are padded with zeroes.

If the returned value is greater than than *n_bytes*, the value was truncated: as many of the lowest bits of the value as could fit are written, and the higher bits are ignored. This matches the typical behavior of a C-style downcast.

> **Nota**
>
> Overflow is not considered an error. If the returned value is larger than *n_bytes*, most significant bits were discarded.

`0` will never be returned.

Values are always copied as two's-complement.

Usage example:

```
int32_t value;
Py_ssize_t bytes = PyLong_AsNativeBytes(pylong, &value, sizeof(value), -1);
if (bytes < 0) {
    // Failed. A Python exception was set with the reason.
    return NULL;
}
else if (bytes <= (Py_ssize_t)sizeof(value)) {
    // Success!
}
else {
    // Overflow occurred, but 'value' contains the truncated
    // lowest bits of pylong.
}
```

Passing zero to *n_bytes* will return the size of a buffer that would be large enough to hold the value. This may be larger than technically necessary, but not unreasonably so. If *n_bytes=0*, *buffer* may be `NULL`.

> **Nota**
>
> Passing *n_bytes=0* to this function is not an accurate way to determine the bit length of the value.

To get at the entire Python value of an unknown size, the function can be called twice: first to determine the buffer size, then to fill it:

```
// Ask how much space we need.
Py_ssize_t expected = PyLong_AsNativeBytes(pylong, NULL, 0, -1);
if (expected < 0) {
    // Failed. A Python exception was set with the reason.
    return NULL;
}
assert(expected != 0);  // Impossible per the API definition.
uint8_t *bignum = malloc(expected);
if (!bignum) {
    PyErr_SetString(PyExc_MemoryError, "bignum malloc failed.");
    return NULL;
}
// Safely get the entire value.
Py_ssize_t bytes = PyLong_AsNativeBytes(pylong, bignum, expected, -1);
if (bytes < 0) {  // Exception has been set.
    free(bignum);
    return NULL;
}
else if (bytes > expected) {  // This should not be possible.
    PyErr_SetString(PyExc_RuntimeError,
        "Unexpected bignum truncation after a size check.");
    free(bignum);
    return NULL;
}
// The expected success given the above pre-check.
// ... use bignum ...
free(bignum);
```

flags is either `-1` (`Py_ASNATIVEBYTES_DEFAULTS`) to select defaults that behave most like a C cast, or a combination of the other flags in the table below. Note that `-1` cannot be combined with other flags.

Currently, `-1` corresponds to `Py_ASNATIVEBYTES_NATIVE_ENDIAN | Py_ASNATIVEBYTES_UNSIGNED_BUFFER`.

Sinalizador	Valor
Py_ASNATIVEBYTES_DEFAULTS	-1
Py_ASNATIVEBYTES_BIG_ENDIAN	0
Py_ASNATIVEBYTES_LITTLE_ENDIAN	1
Py_ASNATIVEBYTES_NATIVE_ENDIAN	3
Py_ASNATIVEBYTES_UNSIGNED_BUFFER	4
Py_ASNATIVEBYTES_REJECT_NEGATIVE	8
Py_ASNATIVEBYTES_ALLOW_INDEX	16

Specifying `Py_ASNATIVEBYTES_NATIVE_ENDIAN` will override any other endian flags. Passing `2` is reserved.

By default, sufficient buffer will be requested to include a sign bit. For example, when converting 128 with *n_bytes=1*, the function will return 2 (or more) in order to store a zero sign bit.

If `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` is specified, a zero sign bit will be omitted from size calculations. This allows, for example, 128 to fit in a single-byte buffer. If the destination buffer is later treated as signed, a positive input value may become negative. Note that the flag does not affect handling of negative values: for those, space for a sign bit is always requested.

Specifying `Py_ASNATIVEBYTES_REJECT_NEGATIVE` causes an exception to be set if *pylong* is negative. Without this flag, negative values will be copied provided there is enough space for at least one sign bit, regardless of whether `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` was specified.

If `Py_ASNATIVEBYTES_ALLOW_INDEX` is specified and a non-integer value is passed, its `__index__()` method will be called first. This may result in Python code executing and other threads being allowed to run, which could cause changes to other objects or values in use. When *flags* is `-1`, this option is not set, and non-integer values will raise `TypeError`.

Nota

With the default *flags* (`-1`, or *UNSIGNED_BUFFER* without *REJECT_NEGATIVE*), multiple Python integers can map to a single value without overflow. For example, both `255` and `-1` fit a single-byte buffer and set all its bits. This matches typical C cast behavior.

Adicionado na versão 3.13.

PyObject ***PyLong_GetInfo**(void)

Parte da ABI Estável. On success, return a read only *named tuple*, that holds information about Python's internal representation of integers. See `sys.int_info` for description of individual fields.

Em caso de falha, retorna `NULL` com uma exceção definida.

Adicionado na versão 3.1.

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

> Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Return 1 if *op* is compact, 0 otherwise.

This function makes it possible for performance-critical code to implement a "fast path" for small integers. For compact values use `PyUnstable_Long_CompactValue()`; for others fall back to a `PyLong_As*` function or `PyLong_AsNativeBytes()`.

The speedup is expected to be negligible for most users.

Exactly what values are considered compact is an implementation detail and is subject to change.

Adicionado na versão 3.12.

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

> Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

If *op* is compact, as determined by `PyUnstable_Long_IsCompact()`, return its value.

Otherwise, the return value is undefined.

Adicionado na versão 3.12.

8.2.2 Objetos Booleanos

Os booleanos no Python são implementados como um subclasse de inteiros. Há apenas dois booleanos, `Py_False` e `Py_True`. Assim sendo, as funções de criação e a exclusão normais não se aplicam aos booleanos. No entanto, as seguintes macros estão disponíveis.

PyTypeObject **PyBool_Type**

Parte da ABI Estável. Este instância de `PyTypeObject` representa o tipo booleano em Python; é o mesmo objeto que `bool` na camada Python.

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

Retorna verdadeiro se *o* for do tipo `PyBool_Type`. Esta função sempre tem sucesso.

PyObject ***Py_False**

O objeto Python `False`. Este objeto não tem métodos e é *imortal*.

Alterado na versão 3.12: `Py_False` é *imortal*.

PyObject ***Py_True**

O objeto Python `True`. Este objeto não tem métodos e é *imortal*.

Alterado na versão 3.12: `Py_True` é *imortal*.

Py_RETURN_FALSE

Retorna `Py_False` de uma função.

Py_RETURN_TRUE

Retorna `Py_True` de uma função.

PyObject ***PyBool_FromLong**(long v)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna `Py_True` ou `Py_False`, dependendo do valor verdade de *v*.

8.2.3 Objetos de ponto flutuante

type **PyFloatObject**

Este subtipo de *PyObject* representa um objeto de ponto flutuante do Python.

PyTypeObject **PyFloat_Type**

Parte da ABI Estável. Esta instância do *PyTypeObject* representa o tipo de ponto flutuante do Python. Este é o mesmo objeto `float` na camada do Python.

int **PyFloat_Check**(*PyObject* *p)

Retorna true se seu argumento é um *PyFloatObject* ou um subtipo de *PyFloatObject*. Esta função sempre tem sucesso.

int **PyFloat_CheckExact**(*PyObject* *p)

Retorna true se seu argumento é um *PyFloatObject*, mas um subtipo de *PyFloatObject*. Esta função sempre tem sucesso.

PyObject ***PyFloat_FromString**(*PyObject* *str)

Retorna valor: Nova referência. Parte da ABI Estável. Cria um objeto *PyFloatObject* baseado em uma string de valor "str" ou `NULL` em falha.

PyObject ***PyFloat_FromDouble**(double v)

Retorna valor: Nova referência. Parte da ABI Estável. Cria um objeto *PyFloatObject* de *v* ou `NULL` em falha.

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

Parte da ABI Estável. Retorna uma representação C `double` do conteúdo de *pyfloat*. Se *pyfloat* não é um objeto de ponto flutuante do Python, mas possui o método `__float__()`, esse método será chamado primeiro para converter *pyfloat* em um ponto flutuante. Se `__float__()` não estiver definido, será usado `__index__()`. Este método retorna `-1.0` em caso de falha, portanto, deve-se chamar *PyErr_Occurred()* para verificar se há erros.

Alterado na versão 3.8: Usa `__index__()`, se disponível.

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

Retorna uma representação C `double` do conteúdo de *pyfloat*, mas sem verificação de erro.

PyObject ***PyFloat_GetInfo**(void)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna uma instância de structseq que contém informações sobre a precisão, os valores mínimo e máximo de um ponto flutuante. É um invólucro fino em torno do arquivo de cabeçalho `float.h`.

double **PyFloat_GetMax**()

Parte da ABI Estável. Retorna o ponto flutuante finito máximo representável *DBL_MAX* como `double` do C.

double **PyFloat_GetMin**()

Parte da ABI Estável. Retorna o ponto flutuante positivo mínimo normalizado *DBL_MIN* como `double` do C.

Pack and Unpack functions

The pack and unpack functions provide an efficient platform-independent way to store floating-point values as byte strings. The Pack routines produce a bytes string from a C `double`, and the Unpack routines produce a C `double` from such a bytes string. The suffix (2, 4 or 8) specifies the number of bytes in the bytes string.

On platforms that appear to use IEEE 754 formats these functions work by copying bits. On other platforms, the 2-byte format is identical to the IEEE 754 binary16 half-precision format, the 4-byte format (32-bit) is identical to the IEEE 754 binary32 single precision format, and the 8-byte format to the IEEE 754 binary64 double precision format, although the packing of INFs and NaNs (if such things exist on the platform) isn't handled correctly, and attempting to unpack a bytes string containing an IEEE INF or NaN will raise an exception.

On non-IEEE platforms with more precision, or larger dynamic range, than IEEE 754 supports, not all values can be packed; on non-IEEE platforms with less precision, or smaller dynamic range, not all values can be unpacked. What happens in such cases is partly accidental (alas).

Adicionado na versão 3.11.

Pack functions

The pack routines write 2, 4 or 8 bytes, starting at *p*. *le* is an `int` argument, non-zero if you want the bytes string in little-endian format (exponent last, at `p+1`, `p+3`, or `p+6` `p+7`), zero if you want big-endian format (exponent first, at *p*). The `PY_BIG_ENDIAN` constant can be used to use the native endian: it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value: `0` if all is OK, `-1` if error (and an exception is set, most likely `OverflowError`).

There are two problems on non-IEEE platforms:

- What this does is undefined if *x* is a NaN or infinity.
- `-0.0` and `+0.0` produce the same bytes string.

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

Pack a C double as the IEEE 754 binary16 half-precision format.

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

Pack a C double as the IEEE 754 binary32 single precision format.

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

Pack a C double as the IEEE 754 binary64 double precision format.

Unpack functions

The unpack routines read 2, 4 or 8 bytes, starting at *p*. *le* is an `int` argument, non-zero if the bytes string is in little-endian format (exponent last, at `p+1`, `p+3` or `p+6` and `p+7`), zero if big-endian (exponent first, at *p*). The `PY_BIG_ENDIAN` constant can be used to use the native endian: it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value: The unpacked double. On error, this is `-1.0` and *PyErr_Occurred()* is true (and an exception is set, most likely `OverflowError`).

Note that on a non-IEEE platform this will refuse to unpack a bytes string that represents a NaN or infinity.

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

Unpack the IEEE 754 binary16 half-precision format as a C double.

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

Unpack the IEEE 754 binary32 single precision format as a C double.

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

Unpack the IEEE 754 binary64 double precision format as a C double.

8.2.4 Objetos números complexos

Os objetos números complexos do Python são implementados como dois tipos distintos quando visualizados na API C: um é o objeto Python exposto aos programas Python e o outro é uma estrutura C que representa o valor real do número complexo. A API fornece funções para trabalhar com ambos.

Números complexos como estruturas C.

Observe que as funções que aceitam essas estruturas como parâmetros e as retornam como resultados o fazem *por valor* em vez de desreferenciá-las por meio de ponteiros. Isso é consistente em toda a API.

type **Py_complex**

A estrutura C que corresponde à parte do valor de um objeto de número complexo Python. A maioria das funções para lidar com objetos de números complexos usa estruturas desse tipo como valores de entrada ou saída, conforme apropriado.

double **real**

double **imag**

A estrutura é definida como:

```
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* num)

Retorna a negação do número complexo *num*, 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*

Se *num* for nulo e *exp* não for um número real positivo, este método retorna zero e define `errno` para `EDOM`.

Números complexos como objetos Python

type **PyComplexObject**

Este subtipo de *PyObject* representa um objeto Python de número complexo.

PyTypeObject **PyComplex_Type**

Parte da ABI Estável. Esta instância de *PyTypeObject* representa o tipo de número complexo Python. É o mesmo objeto que `complex` na camada Python.

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

Retorna true se seu argumento é um *PyComplexObject* ou um subtipo de *PyComplexObject*. Esta função sempre tem sucesso.

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

Retorna true se seu argumento é um *PyComplexObject*, mas não um subtipo de *PyComplexObject*. Esta função sempre tem sucesso.

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

Retorna valor: Nova referência. Cria um novo objeto de número complexo Python a partir de um valor C *Py_complex*. Retorna `NULL` com uma exceção definida ao ocorrer um erro.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um novo objeto *PyComplexObject* de *real* e *imag*. Retorna `NULL` com uma exceção definida ao ocorrer um erro.

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

Parte da ABI Estável. Retorna a parte real de *op* como um `double` C.

Se *op* não é um objeto de número complexo Python, mas tem um método `__complex__()`, este método será primeiro chamado para converter *op* em um objeto de número complexo Python. Se `__complex__()` não estiver definido, ele volta a chamar *PyFloat_AsDouble()* e retorna seu resultado.

Em caso de falha, este método retorna `-1.0` com uma exceção definida, então deve-se chamar *PyErr_Occurred()* para verificar se há erros.

Alterado na versão 3.13: Usa `__complex__()`, se disponível.

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

Parte da ABI Estável. Retorna a parte imaginária de *op* como um `double` C.

Se *op* não é um objeto de número complexo Python, mas tem um método `__complex__()`, este método será primeiro chamado para converter *op* em um objeto de número complexo Python. Se `__complex__()` não estiver definido, ele volta a chamar *PyFloat_AsDouble()* e retorna `0.0` em caso de sucesso.

Em caso de falha, este método retorna `-1.0` com uma exceção definida, então deve-se chamar *PyErr_Occurred()* para verificar se há erros.

Alterado na versão 3.13: Usa `__complex__()`, se disponível.

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

Retorna o valor *Py_complex* do número complexo *op*.

Se *op* não é um objeto de número complexo Python, mas tem um método `__complex__()`, este método será primeiro chamado para converter *op* em um objeto de número complexo Python. Se `__complex__()` não for definido, então ele recorre a `__float__()`. Se `__float__()` não estiver definido, então ele volta para `__index__()`.

Em caso de falha, este método retorna *Py_complex* com *real* definido para `-1.0` e com uma exceção definida, então deve-se chamar *PyErr_Occurred()* para verificar se há erros.

Alterado na versão 3.8: Usa `__index__()`, se disponível.

8.3 Objetos Sequência

Operações genéricas em objetos de sequência foram discutidas no capítulo anterior; Esta seção lida com os tipos específicos de objetos sequência que são intrínsecos à linguagem Python.

8.3.1 Objetos Bytes

Estas funções levantam `TypeError` quando se espera um parâmetro bytes e são chamados com um parâmetro que não é bytes.

type **PyBytesObject**

Esta é uma instância de *PyObject* representando o objeto bytes do Python.

PyTypeObject **PyBytes_Type**

Parte da ABI Estável. Esta instância de *PyTypeObject* representa o tipo de bytes Python; é o mesmo objeto que `bytes` na camada de Python.

int **PyBytes_Check**(*PyObject* *o)

Retorna verdadeiro se o objeto *o* for um objeto bytes ou se for uma instância de um subtipo do tipo bytes. Esta função sempre tem sucesso.

int **PyBytes_CheckExact**(*PyObject* *o)

Retorna verdadeiro se o objeto *o* for um objeto bytes, mas não uma instância de um subtipo do tipo bytes. Esta função sempre tem sucesso.

PyObject ***PyBytes_FromString**(const char *v)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um novo objeto de bytes com uma cópia da string *v* como valor em caso de sucesso e `NULL` em caso de falha. O parâmetro *v* não deve ser `NULL` e isso não será verificado.

PyObject ***PyBytes_FromStringAndSize**(const char *v, *Py_ssize_t* len)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um novo objeto de bytes com uma cópia da string *v* como valor e comprimento *len* em caso de sucesso e `NULL` em caso de falha. Se *v* for `NULL`, o conteúdo do objeto bytes não será inicializado.

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

Retorna valor: Nova referência. Parte da ABI Estável. Leva uma string tipo `printf()` do C *format* e um número variável de argumentos, calcula o tamanho do objeto bytes do Python resultante e retorna um objeto bytes com os valores formatados nela. Os argumentos da variável devem ser tipos C e devem corresponder exatamente aos caracteres de formato na string *format*. Os seguintes formatos de caracteres são permitidos:

Caracteres Formatados	Tipo	Comentário
`%%`	*n/d*	O caractere literal %.
`%c`	int	Um único byte, representado como um C int.
`%d`	int	Equivalente a `printf("%d")`.[1]
`%u`	unsigned int	Equivalente a `printf("%u")`.[1]
`%ld`	long	Equivalente a `printf("%ld")`.[1]
`%lu`	unsigned long	Equivalente a `printf("%lu")`.[1]
`%zd`	*Py_ssize_t*	Equivalente a `printf("%zd")`.[1]
`%zu`	size_t	Equivalente a `printf("%zu")`.[1]
`%i`	int	Equivalente a `printf("%i")`.[1]
`%x`	int	Equivalente a `printf("%x")`.[1]
`%s`	const char*	Uma matriz de caracteres C com terminação nula.
`%p`	const 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.

Um caractere de formato não reconhecido faz com que todo o resto da string de formato seja copiado como é para o objeto resultante e todos os argumentos extras sejam descartados.

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

Retorna valor: Nova referência. Parte da ABI Estável. Idêntico a *PyBytes_FromFormat()* exceto que é preciso exatamente dois argumentos.

PyObject ***PyBytes_FromObject**(*PyObject* *o)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna a representação de bytes do objeto *o* que implementa o protocolo de buffer.

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

Parte da ABI Estável. Retorna o comprimento dos bytes em objeto bytes *o*.

Py_ssize_t **PyBytes_GET_SIZE**(*PyObject* *o)

Similar a *PyBytes_Size()*, mas sem verificação de erro.

char ***PyBytes_AsString**(*PyObject* *o)

Parte da ABI Estável. Retorna um ponteiro para o conteúdo de *o*. O ponteiro se refere ao buffer interno de *o*, que consiste em `len(o) + 1` bytes. O último byte no buffer é sempre nulo, independentemente de haver outros bytes nulos. Os dados não devem ser modificados de forma alguma, a menos que o objeto tenha sido criado usando `PyBytes_FromStringAndSize(NULL, size)`. Não deve ser desalocado. Se *o* não é um objeto de bytes, *PyBytes_AsString()* retorna `NULL` e levanta `TypeError`.

[1] Para especificadores de número inteiro (d, u, ld, lu, zd, zu, i, x): o sinalizador de conversão 0 tem efeito mesmo quando uma precisão é fornecida.

char *__PyBytes_AS_STRING__(*PyObject* *string)

Similar a `PyBytes_AsString()`, mas sem verificação de erro.

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

Parte da ABI Estável. Retorna os conteúdos terminados nulos do objeto *obj* através das variáveis de saída *buffer* e *length*. Retorna `0` em caso de sucesso.

Se *length* for `NULL`, o objeto bytes não poderá conter bytes nulos incorporados; se isso acontecer, a função retornará `-1` e a `ValueError` será levantado.

O buffer refere-se a um buffer interno de *obj*, que inclui um byte nulo adicional no final (não contado em *length*). Os dados não devem ser modificados de forma alguma, a menos que o objeto tenha sido criado apenas usando `PyBytes_FromStringAndSize(NULL, size)`. Não deve ser desalinhado. Se *obj* não é um objeto bytes, `PyBytes_AsStringAndSize()` retorna `-1` e levanta `TypeError`.

Alterado na versão 3.5: Anteriormente `TypeError` era levantado quando os bytes nulos incorporados eram encontrados no objeto bytes.

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

Parte da ABI Estável. Cria um novo objeto de bytes em **bytes* contendo o conteúdo de *newpart* anexado a *bytes*; o chamador será o proprietário da nova referência. A referência ao valor antigo de *bytes* será roubada. Se o novo objeto não puder ser criado, a antiga referência a *bytes* ainda será descartada e o valor de **bytes* será definido como `NULL`; a exceção apropriada será definida.

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

Parte da ABI Estável. "Crie um novo objeto bytes em **bytes* contendo o conteúdo de newpart anexado a bytes. Esta versão libera a *strong reference* (referência forte) para newpart (ou seja, decrementa a contagem de referências a ele)."

int ___PyBytes_Resize__(*PyObject* **bytes, *Py_ssize_t* newsize)

Redimensiona um objeto de bytes. *newsize* será o novo tamanho do objeto bytes. Você pode pensar nisso como se estivesse criando um novo objeto de bytes e destruindo o antigo, só que de forma mais eficiente. Passe o endereço de um objeto de bytes existente como um lvalue (pode ser gravado) e o novo tamanho desejado. Em caso de sucesso, **bytes* mantém o objeto de bytes redimensionados e `0` é retornado; o endereço em **bytes* pode diferir do seu valor de entrada. Se a realocação falhar, o objeto de bytes originais em **bytes* é desalocado, **bytes* é definido como `NULL`, `MemoryError` é definido e `-1` é retornado.

8.3.2 Objetos byte array

type __PyByteArrayObject__

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

PyTypeObject __PyByteArray_Type__

Parte da ABI Estável. Essa instância de `PyTypeObject` representa um tipo Python bytearray; é o mesmo objeto que o `bytearray` na camada Python.

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. Esta função sempre tem sucesso.

int __PyByteArray_CheckExact__(*PyObject* *o)

Retorna verdadeiro se o objeto *o* for um objeto bytearray, mas não uma instância de um subtipo do tipo bytearray. Esta função sempre tem sucesso.

Funções diretas da API

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um novo objeto bytearray, *o*, que implementa o *protocolo de buffer*.

Em caso de falha, retorna `NULL` com uma exceção definida.

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

Retorna valor: Nova referência. Parte da ABI Estável. Cria um novo objeto bytearray a partir de *string* e seu comprimento, *len*.

Em caso de falha, retorna `NULL` com uma exceção definida.

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

Retorna valor: Nova referência. Parte da ABI Estável. Concatena os bytearrays *a* e *b* e retorna um novo bytearray com o resultado.

Em caso de falha, retorna `NULL` com uma exceção definida.

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

Parte da ABI Estável. Retorna o tamanho de *bytearray* após verificar se há um ponteiro `NULL`.

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

Parte da ABI Estável. Retorna o conteúdo de *bytearray* como uma matriz de caracteres após verificar um ponteiro `NULL`. O array retornado sempre tem um byte nulo extra acrescentado.

int **PyByteArray_Resize**(*PyObject* *bytearray, *Py_ssize_t* len)

Parte da ABI Estável. 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)

Similar a *`PyByteArray_AsString()`*, mas sem verificação de erro.

Py_ssize_t **PyByteArray_GET_SIZE**(*PyObject* *bytearray)

Similar a *`PyByteArray_Size()`*, mas sem verificação de erro.

8.3.3 Objetos Unicode e Codecs

Unicode Objects

Since the implementation of **PEP 393** in Python 3.3, Unicode objects internally use a variety of representations, in order to allow handling the complete range of Unicode characters while staying memory efficient. There are special cases for strings where all code points are below 128, 256, or 65536; otherwise, code points must be below 1114112 (which is the full Unicode range).

UTF-8 representation is created on demand and cached in the Unicode object.

Nota

The *`Py_UNICODE`* representation has been removed since Python 3.12 with deprecated APIs. See **PEP 623** for more information.

Unicode Type

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

type **Py_UCS4**

type **Py_UCS2**

type **Py_UCS1**

Parte da ABI Estável. These types are typedefs for unsigned integer types wide enough to contain characters of 32 bits, 16 bits and 8 bits, respectively. When dealing with single Unicode characters, use *Py_UCS4*.

Adicionado na versão 3.3.

type **Py_UNICODE**

This is a typedef of `wchar_t`, which is a 16-bit type or 32-bit type depending on the platform.

Alterado na versão 3.3: In previous versions, this was a 16-bit type or a 32-bit type depending on whether you selected a "narrow" or "wide" Unicode version of Python at build time.

Deprecated since version 3.13, will be removed in version 3.15.

type **PyASCIIObject**

type **PyCompactUnicodeObject**

type **PyUnicodeObject**

These subtypes of *PyObject* represent a Python Unicode object. In almost all cases, they shouldn't be used directly, since all API functions that deal with Unicode objects take and return *PyObject* pointers.

Adicionado na versão 3.3.

PyTypeObject **PyUnicode_Type**

Parte da ABI Estável. This instance of *PyTypeObject* represents the Python Unicode type. It is exposed to Python code as `str`.

The following APIs are C macros and static inlined functions for fast checks and access to internal read-only data of Unicode objects:

int **PyUnicode_Check**(*PyObject* *obj)

Return true if the object *obj* is a Unicode object or an instance of a Unicode subtype. This function always succeeds.

int **PyUnicode_CheckExact**(*PyObject* *obj)

Return true if the object *obj* is a Unicode object, but not an instance of a subtype. This function always succeeds.

int **PyUnicode_READY**(*PyObject* *unicode)

Returns `0`. This API is kept only for backward compatibility.

Adicionado na versão 3.3.

Descontinuado desde a versão 3.10: This API does nothing since Python 3.12.

Py_ssize_t **PyUnicode_GET_LENGTH**(*PyObject* *unicode)

Return the length of the Unicode string, in code points. *unicode* has to be a Unicode object in the "canonical" representation (not checked).

Adicionado na versão 3.3.

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

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

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

Return a pointer to the canonical representation cast to UCS1, UCS2 or UCS4 integer types for direct character access. No checks are performed if the canonical representation has the correct character size; use *PyUnicode_KIND()* to select the right function.

Adicionado na versão 3.3.

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND

PyUnicode_4BYTE_KIND

Return values of the *PyUnicode_KIND()* macro.

Adicionado na versão 3.3.

Alterado na versão 3.12: `PyUnicode_WCHAR_KIND` has been removed.

int **PyUnicode_KIND**(*PyObject* *unicode)

Return one of the PyUnicode kind constants (see above) that indicate how many bytes per character this Unicode object uses to store its data. *unicode* has to be a Unicode object in the "canonical" representation (not checked).

Adicionado na versão 3.3.

void ***PyUnicode_DATA**(*PyObject* *unicode)

Return a void pointer to the raw Unicode buffer. *unicode* has to be a Unicode object in the "canonical" representation (not checked).

Adicionado na versão 3.3.

void **PyUnicode_WRITE**(int kind, void *data, *Py_ssize_t* index, *Py_UCS4* value)

Write into a canonical representation *data* (as obtained with *PyUnicode_DATA()*). This function performs no sanity checks, and is intended for usage in loops. The caller should cache the *kind* value and *data* pointer as obtained from other calls. *index* is the index in the string (starts at 0) and *value* is the new code point value which should be written to that location.

Adicionado na versão 3.3.

Py_UCS4 **PyUnicode_READ**(int kind, void *data, *Py_ssize_t* index)

Read a code point from a canonical representation *data* (as obtained with *PyUnicode_DATA()*). No checks or ready calls are performed.

Adicionado na versão 3.3.

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

Read a character from a Unicode object *unicode*, which must be in the "canonical" representation. This is less efficient than *PyUnicode_READ()* if you do multiple consecutive reads.

Adicionado na versão 3.3.

Py_UCS4 **PyUnicode_MAX_CHAR_VALUE**(*PyObject* *unicode)

Return the maximum code point that is suitable for creating another string based on *unicode*, which must be in the "canonical" representation. This is always an approximation but more efficient than iterating over the string.

Adicionado na versão 3.3.

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

Parte da ABI Estável. Return `1` if the string is a valid identifier according to the language definition, section identifiers. Return `0` otherwise.

Alterado na versão 3.9: The function does not call *Py_FatalError()* anymore if the string is not ready.

Unicode Character Properties

Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration.

int **Py_UNICODE_ISSPACE**(*Py_UCS4* ch)

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

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

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

int **Py_UNICODE_ISUPPER**(*Py_UCS4* ch)
Return `1` or `0` depending on whether *ch* is an uppercase character.

int **Py_UNICODE_ISTITLE**(*Py_UCS4* ch)
Return `1` or `0` depending on whether *ch* is a titlecase character.

int **Py_UNICODE_ISLINEBREAK**(*Py_UCS4* ch)
Return `1` or `0` depending on whether *ch* is a linebreak character.

int **Py_UNICODE_ISDECIMAL**(*Py_UCS4* ch)
Return `1` or `0` depending on whether *ch* is a decimal character.

int **Py_UNICODE_ISDIGIT**(*Py_UCS4* ch)
Return `1` or `0` depending on whether *ch* is a digit character.

int **Py_UNICODE_ISNUMERIC**(*Py_UCS4* ch)
Return `1` or `0` depending on whether *ch* is a numeric character.

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

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

int **Py_UNICODE_ISPRINTABLE**(*Py_UCS4* ch)
Return `1` or `0` depending on whether *ch* is a printable character, in the sense of `str.isprintable()`.

These APIs can be used for fast direct character conversions:

Py_UCS4 **Py_UNICODE_TOLOWER**(*Py_UCS4* ch)
Return the character *ch* converted to lower case.

Py_UCS4 **Py_UNICODE_TOUPPER**(*Py_UCS4* ch)
Return the character *ch* converted to upper case.

Py_UCS4 **Py_UNICODE_TOTITLE**(*Py_UCS4* ch)
Return the character *ch* converted to title case.

int **Py_UNICODE_TODECIMAL**(*Py_UCS4* ch)
Return the character *ch* converted to a decimal positive integer. Return `-1` if this is not possible. This function does not raise exceptions.

int **Py_UNICODE_TODIGIT**(*Py_UCS4* ch)
Return the character *ch* converted to a single digit integer. Return `-1` if this is not possible. This function does not raise exceptions.

double **Py_UNICODE_TONUMERIC**(*Py_UCS4* ch)
Return the character *ch* converted to a double. Return `-1.0` if this is not possible. This function does not raise exceptions.

These APIs can be used to work with surrogates:

int **Py_UNICODE_IS_SURROGATE**(*Py_UCS4* ch)
Check if *ch* is a surrogate (`0xD800 <= ch <= 0xDFFF`).

int **Py_UNICODE_IS_HIGH_SURROGATE**(*Py_UCS4* ch)
Check if *ch* is a high surrogate (`0xD800 <= ch <= 0xDBFF`).

int **Py_UNICODE_IS_LOW_SURROGATE**(*Py_UCS4* ch)
Check if *ch* is a low surrogate (`0xDC00 <= ch <= 0xDFFF`).

Py_UCS4 **Py_UNICODE_JOIN_SURROGATES**(*Py_UCS4* high, *Py_UCS4* low)
Join two surrogate code points and return a single *Py_UCS4* value. *high* and *low* are respectively the leading and trailing surrogates in a surrogate pair. *high* must be in the range [0xD800; 0xDBFF] and *low* must be in the range [0xDC00; 0xDFFF].

Creating and accessing Unicode strings

To create Unicode objects and access their basic sequence properties, use these APIs:

PyObject ***PyUnicode_New**(*Py_ssize_t* size, *Py_UCS4* maxchar)

Retorna valor: Nova referência. Create a new Unicode object. *maxchar* should be the true maximum code point to be placed in the string. As an approximation, it can be rounded up to the nearest value in the sequence 127, 255, 65535, 1114111.

This is the recommended way to allocate a new Unicode object. Objects created using this function are not resizable.

On error, set an exception and return `NULL`.

Adicionado na versão 3.3.

PyObject ***PyUnicode_FromKindAndData**(int kind, const void *buffer, *Py_ssize_t* size)

Retorna valor: Nova referência. Cria um novo objeto Unicode com o tipo *type* fornecido (os valores possíveis são *PyUnicode_1BYTE_KIND* etc., conforme retornado por *PyUnicode_KIND()*). O *buffer* deve apontar para um array de unidades com *size* de 1, 2 ou 4 bytes por caractere, conforme fornecido pelo tipo.

If necessary, the input *buffer* is copied and transformed into the canonical representation. For example, if the *buffer* is a UCS4 string (*PyUnicode_4BYTE_KIND*) and it consists only of codepoints in the UCS1 range, it will be transformed into UCS1 (*PyUnicode_1BYTE_KIND*).

Adicionado na versão 3.3.

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

Retorna valor: Nova referência. Parte da ABI Estável. Create a Unicode object from the char buffer *str*. The bytes will be interpreted as being UTF-8 encoded. The buffer is copied into the new object. The return value might be a shared object, i.e. modification of the data is not allowed.

This function raises `SystemError` when:

- *size* < 0,
- *str* is `NULL` and *size* > 0

Alterado na versão 3.12: *str* == `NULL` with *size* > 0 is not allowed anymore.

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

Retorna valor: Nova referência. Parte da ABI Estável. Create a Unicode object from a UTF-8 encoded null-terminated char buffer *str*.

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

Retorna valor: Nova referência. Parte da ABI Estável. Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python Unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* ASCII-encoded string.

Um especificador de conversão contém dois ou mais caracteres e tem os seguintes componentes, que devem aparecer nesta ordem:

1. O caractere `'%'`, que determina o início do especificador.
2. Flags de conversão (opcional), que afetam o resultado de alguns tipos de conversão.
3. Minimum field width (optional). If specified as an `'*'` (asterisk), the actual width is given in the next argument, which must be of type `int`, and the object to convert comes after the minimum field width and optional precision.
4. Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'` (an asterisk), the actual precision is given in the next argument, which must be of type `int`, and the value to convert comes after the precision.
5. Modificador de Comprimento(opcional).
6. Tipos de Conversão

Os caracteres flags de conversão são:

Sinalizador	Significado
`0`	A conversão será preenchida por zeros para valores numéricos.
`-`	The converted value is left adjusted (overrides the `0` flag if both are given).

The length modifiers for following integer conversions (`d`, `i`, `o`, `u`, `x`, or `X`) specify the type of the argument (`int` by default):

Modifier	Tipos
`l`	`long` or `unsigned long`
`ll`	`long long` or `unsigned long long`
`j`	`intmax_t` or `uintmax_t`
`z`	`size_t` or `ssize_t`
`t`	`ptrdiff_t`

The length modifier `l` for following conversions `s` or `V` specify that the type of the argument is `const wchar_t*`.

The conversion specifiers are:

Conversion Specifier	Tipo	Comentário
`%`	*n/d*	The literal `%` character.
`d`, `i`	Specified by the length modifier	The decimal representation of a signed C integer.
`u`	Specified by the length modifier	The decimal representation of an unsigned C integer.
`o`	Specified by the length modifier	The octal representation of an unsigned C integer.
`x`	Specified by the length modifier	The hexadecimal representation of an unsigned C integer (lowercase).
`X`	Specified by the length modifier	The hexadecimal representation of an unsigned C integer (uppercase).
`c`	`int`	A single character.
`s`	`const char*` or `const wchar_t*`	Uma matriz de caracteres C com terminação nula.
`p`	`const void*`	The hex representation of a C pointer. Mostly equivalent to `printf("%p")` except that it is guaranteed to start with the literal `0x` regardless of what the platform's `printf` yields.
`A`	*PyObject**	The result of calling `ascii()`.
`U`	*PyObject**	A Unicode object.
`V`	*PyObject**, `const char*` or `const wchar_t*`	A Unicode object (which may be `NULL`) and a null-terminated C character array as a second parameter (which will be used, if the first parameter is `NULL`).
`S`	*PyObject**	The result of calling *PyObject_Str()*.
`R`	*PyObject**	The result of calling *PyObject_Repr()*.
`T`	*PyObject**	Get the fully qualified name of an object type; call *PyType_GetFullyQualifiedName()*.
`#T`	*PyObject**	Similar to `T` format, but use a colon (`:`) as separator between the module name and the qualified name.
`N`	*PyTypeObject**	Get the fully qualified name of a type; call *PyType_GetFullyQualifiedName()*.
`#N`	*PyTypeObject**	Similar to `N` format, but use a colon (`:`) as separator between the module name and the qualified name.

Nota

The width formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes or `wchar_t` items (if the length modifier `l` is used) for `"%s"` and `"%V"` (if the `PyObject*` argument is `NULL`), and a number of characters for `"%A"`, `"%U"`, `"%S"`, `"%R"` and `"%V"` (if the `PyObject*` argument is not `NULL`).

Nota

Unlike to C `printf()` the `0` flag has effect even when a precision is given for integer conversions (`d`, `i`, `u`, `o`, `x`, or `X`).

Alterado na versão 3.2: Suporte adicionado para `"%lld"` e `"%llu"`.

Alterado na versão 3.3: Support for `"%li"`, `"%lli"` and `"%zi"` added.

Alterado na versão 3.4: Support width and precision formatter for `"%s"`, `"%A"`, `"%U"`, `"%V"`, `"%S"`, `"%R"`

added.

Alterado na versão 3.12: Support for conversion specifiers `o` and `X`. Support for length modifiers `j` and `t`. Length modifiers are now applied to all integer conversions. Length modifier `l` is now applied to conversion specifiers `s` and `V`. Support for variable width and precision `*`. Support for flag `-`.

An unrecognized format character now sets a `SystemError`. In previous versions it caused all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

Alterado na versão 3.13: Support for `%T`, `%#T`, `%N` and `%#N` formats added.

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

Retorna valor: Nova referência. Parte da ABI Estável. Identical to *PyUnicode_FromFormat()* except that it takes exactly two arguments.

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

Retorna valor: Nova referência. Parte da ABI Estável. Copy an instance of a Unicode subtype to a new true Unicode object if necessary. If *obj* is already a true Unicode object (not a subtype), return a new *strong reference* to the object.

Objects other than Unicode or its subtypes will cause a `TypeError`.

PyObject ***PyUnicode_FromEncodedObject**(*PyObject* *obj, const char *encoding, const char *errors)

Retorna valor: Nova referência. Parte da ABI Estável. Decode an encoded object *obj* to a Unicode object.

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

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

The API returns `NULL` if there was an error. The caller is responsible for decref'ing the returned objects.

const char ***PyUnicode_GetDefaultEncoding**(void)

Parte da ABI Estável. Return the name of the default string encoding, `"utf-8"`. See `sys.getdefaultencoding()`.

The returned string does not need to be freed, and is valid until interpreter shutdown.

Py_ssize_t **PyUnicode_GetLength**(*PyObject* *unicode)

Parte da ABI Estável *desde a versão 3.7.* Return the length of the Unicode object, in code points.

On error, set an exception and return `-1`.

Adicionado na versão 3.3.

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

Copy characters from one Unicode object into another. This function performs character conversion when necessary and falls back to `memcpy()` if possible. Returns `-1` and sets an exception on error, otherwise returns the number of copied characters.

Adicionado na versão 3.3.

Py_ssize_t **PyUnicode_Fill**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* length, *Py_UCS4* fill_char)

Fill a string with a character: write *fill_char* into `unicode[start:start+length]`.

Fail if *fill_char* is bigger than the string maximum character, or if the string has more than 1 reference.

Return the number of written character, or return `-1` and raise an exception on error.

Adicionado na versão 3.3.

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

Parte da ABI Estável *desde a versão 3.7.* Write a character to a string. The string must have been created through *PyUnicode_New()*. Since Unicode strings are supposed to be immutable, the string must not be shared, or have been hashed yet.

This function checks that *unicode* is a Unicode object, that the index is not out of bounds, and that the object can be modified safely (i.e. that it its reference count is one).

Return `0` on success, `-1` on error with an exception set.

Adicionado na versão 3.3.

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

Parte da ABI Estável *desde a versão 3.7.* Read a character from a string. This function checks that *unicode* is a Unicode object and the index is not out of bounds, in contrast to *PyUnicode_READ_CHAR()*, which performs no error checking.

Return character on success, `-1` on error with an exception set.

Adicionado na versão 3.3.

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

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.7.* Return a substring of *unicode*, from character index *start* (included) to character index *end* (excluded). Negative indices are not supported. On error, set an exception and return `NULL`.

Adicionado na versão 3.3.

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

Parte da ABI Estável *desde a versão 3.7.* Copy the string *unicode* into a UCS4 buffer, including a null character, if *copy_null* is set. Returns `NULL` and sets an exception on error (in particular, a `SystemError` if *buflen* is smaller than the length of *unicode*). *buffer* is returned on success.

Adicionado na versão 3.3.

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

Parte da ABI Estável *desde a versão 3.7.* Copy the string *unicode* into a new UCS4 buffer that is allocated using *PyMem_Malloc()*. If this fails, `NULL` is returned with a `MemoryError` set. The returned buffer always has an extra null code point appended.

Adicionado na versão 3.3.

Locale Encoding

The current locale encoding can be used to decode text from the operating system.

PyObject ***PyUnicode_DecodeLocaleAndSize**(const char *str, *Py_ssize_t* length, const char *errors)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.7.* Decode a string from UTF-8 on Android and VxWorks, or from the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The decoder uses `"strict"` error handler if *errors* is `NULL`. *str* must end with a null character but cannot contain embedded null characters.

Use *PyUnicode_DecodeFSDefaultAndSize()* to decode a string from the *filesystem encoding and error handler*.

This function ignores the Python UTF-8 Mode.

Ver também

The *Py_DecodeLocale()* function.

Adicionado na versão 3.3.

Alterado na versão 3.7: The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, *Py_DecodeLocale()* was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

PyObject ***PyUnicode_DecodeLocale**(const char *str, const char *errors)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.7.* Similar to `PyUnicode_DecodeLocaleAndSize()`, but compute the string length using `strlen()`.

Adicionado na versão 3.3.

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

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.7.* Encode a Unicode object to UTF-8 on Android and VxWorks, or to the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The encoder uses `"strict"` error handler if *errors* is `NULL`. Return a `bytes` object. *unicode* cannot contain embedded null characters.

Use `PyUnicode_EncodeFSDefault()` to encode a string to the *filesystem encoding and error handler*.

This function ignores the Python UTF-8 Mode.

Ver também

The `Py_EncodeLocale()` function.

Adicionado na versão 3.3.

Alterado na versão 3.7: The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, `Py_EncodeLocale()` was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

File System Encoding

Functions encoding to and decoding from the *filesystem encoding and error handler* (**PEP 383** and **PEP 529**).

To encode file names to `bytes` during argument parsing, the `"O&"` converter should be used, passing `PyUnicode_FSConverter()` as the conversion function:

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

Parte da ABI Estável. *PyArg_Parse* converter*: encode `str` objects – obtained directly or through the `os.PathLike` interface – to `bytes` using `PyUnicode_EncodeFSDefault()`; `bytes` objects are output as-is. *result* must be an address of a C variable of type `PyObject*` (or `PyBytesObject*`). On success, set the variable to a new *strong reference* to a *bytes object* which must be released when it is no longer used and return a non-zero value (`Py_CLEANUP_SUPPORTED`). Embedded null bytes are not allowed in the result. On failure, return `0` with an exception set.

If *obj* is `NULL`, the function releases a strong reference stored in the variable referred by *result* and returns `1`.

Adicionado na versão 3.1.

Alterado na versão 3.6: Aceita um *objeto caminho ou similar*.

To decode file names to `str` during argument parsing, the `"O&"` converter should be used, passing `PyUnicode_FSDecoder()` as the conversion function:

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

Parte da ABI Estável. *PyArg_Parse* converter*: decode `bytes` objects – obtained either directly or indirectly through the `os.PathLike` interface – to `str` using `PyUnicode_DecodeFSDefaultAndSize()`; `str` objects are output as-is. *result* must be an address of a C variable of type `PyObject*` (or `PyUnicodeObject*`). On success, set the variable to a new *strong reference* to a *Unicode object* which must be released when it is no longer used and return a non-zero value (`Py_CLEANUP_SUPPORTED`). Embedded null characters are not allowed in the result. On failure, return `0` with an exception set.

If *obj* is `NULL`, release the strong reference to the object referred to by *result* and return `1`.

Adicionado na versão 3.2.

Alterado na versão 3.6: Aceita um *objeto caminho ou similar*.

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

Retorna valor: Nova referência. Parte da ABI Estável. Decode a string from the *filesystem encoding and error handler*.

If you need to decode a string from the current locale encoding, use `PyUnicode_DecodeLocaleAndSize()`.

Ver também

The `Py_DecodeLocale()` function.

Alterado na versão 3.6: The *filesystem error handler* is now used.

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

Retorna valor: Nova referência. Parte da ABI Estável. Decode a null-terminated string from the *filesystem encoding and error handler*.

If the string length is known, use `PyUnicode_DecodeFSDefaultAndSize()`.

Alterado na versão 3.6: The *filesystem error handler* is now used.

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

Retorna valor: Nova referência. Parte da ABI Estável. Encode a Unicode object to the *filesystem encoding and error handler*, and return `bytes`. Note that the resulting `bytes` object can contain null bytes.

If you need to encode a string to the current locale encoding, use `PyUnicode_EncodeLocale()`.

Ver também

The `Py_EncodeLocale()` function.

Adicionado na versão 3.2.

Alterado na versão 3.6: The *filesystem error handler* is now used.

wchar_t Support

`wchar_t` support for platforms which support it:

PyObject ***PyUnicode_FromWideChar**(const wchar_t *wstr, *Py_ssize_t* size)

Retorna valor: Nova referência. Parte da ABI Estável. Create a Unicode object from the `wchar_t` buffer *wstr* of the given *size*. Passing `-1` as the *size* indicates that the function must itself compute the length, using `wcslen()`. Return `NULL` on failure.

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

Parte da ABI Estável. Copy the Unicode object contents into the `wchar_t` buffer *wstr*. At most *size* `wchar_t` characters are copied (excluding a possibly trailing null termination character). Return the number of `wchar_t` characters copied or `-1` in case of an error.

When *wstr* is `NULL`, instead return the *size* that would be required to store all of *unicode* including a terminating null.

Note that the resulting `wchar_t*` string may or may not be null-terminated. It is the responsibility of the caller to make sure that the `wchar_t*` string is null-terminated in case this is required by the application. Also, note that the `wchar_t*` string might contain null characters, which would cause the string to be truncated when used with most C functions.

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

Parte da ABI Estável *desde a versão 3.7.* Convert the Unicode object to a wide character string. The output string always ends with a null character. If *size* is not `NULL`, write the number of wide characters (excluding the trailing null termination character) into **size*. Note that the resulting `wchar_t` string might contain null

characters, which would cause the string to be truncated when used with most C functions. If *size* is `NULL` and the `wchar_t*` string contains null characters a `ValueError` is raised.

Returns a buffer allocated by *PyMem_New* (use *PyMem_Free()* to free it) on success. On error, returns `NULL` and **size* is undefined. Raises a `MemoryError` if memory allocation is failed.

Adicionado na versão 3.2.

Alterado na versão 3.7: Raises a `ValueError` if *size* is `NULL` and the `wchar_t*` string contains null characters.

Built-in Codecs

Python provides a set of built-in codecs which are written in C for speed. All of these codecs are directly usable via the following functions.

Many of the following APIs take two arguments encoding and errors, and they have the same semantics as the ones of the built-in `str()` string object constructor.

Setting encoding to `NULL` causes the default encoding to be used which is UTF-8. The file system calls should use *PyUnicode_FSConverter()* for encoding file names. This uses the *filesystem encoding and error handler* internally.

Error handling is set by errors which may also be set to `NULL` meaning to use the default handling defined for the codec. Default error handling for all built-in codecs is "strict" (`ValueError` is raised).

The codecs all use a similar interface. Only deviations from the following generic ones are documented for simplicity.

Generic Codecs

These are the generic codec APIs:

PyObject ***PyUnicode_Decode**(const char *str, *Py_ssize_t* size, const char *encoding, const char *errors)

Retorna valor: Nova referência. Parte da ABI Estável. Create a Unicode object by decoding *size* bytes of the encoded string *str*. *encoding* and *errors* have the same meaning as the parameters of the same name in the `str()` built-in function. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsEncodedString**(*PyObject* *unicode, const char *encoding, const char *errors)

Retorna valor: Nova referência. Parte da ABI Estável. Encode a Unicode object and return the result as Python bytes object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode `encode()` method. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

UTF-8 Codecs

These are the UTF-8 codec APIs:

PyObject ***PyUnicode_DecodeUTF8**(const char *str, *Py_ssize_t* size, const char *errors)

Retorna valor: Nova referência. Parte da ABI Estável. Create a Unicode object by decoding *size* bytes of the UTF-8 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeUTF8Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Retorna valor: Nova referência. Parte da ABI Estável. If *consumed* is `NULL`, behave like *PyUnicode_DecodeUTF8()*. If *consumed* is not `NULL`, trailing incomplete UTF-8 byte sequences will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF8String**(*PyObject* *unicode)

Retorna valor: Nova referência. Parte da ABI Estável. Encode a Unicode object using UTF-8 and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

The function fails if the string contains surrogate code points (`U+D800` - `U+DFFF`).

const char *__PyUnicode_AsUTF8AndSize__(*PyObject* *unicode, *Py_ssize_t* *size)

Parte da ABI Estável *desde a versão 3.10.* Return a pointer to the UTF-8 encoding of the Unicode object, and store the size of the encoded representation (in bytes) in *size*. The *size* argument can be `NULL`; in this case no size will be stored. The returned buffer always has an extra null byte appended (not included in *size*), regardless of whether there are any other null code points.

On error, set an exception, set *size* to `-1` (if it's not NULL) and return `NULL`.

The function fails if the string contains surrogate code points (`U+D800` - `U+DFFF`).

This caches the UTF-8 representation of the string in the Unicode object, and subsequent calls will return a pointer to the same buffer. The caller is not responsible for deallocating the buffer. The buffer is deallocated and pointers to it become invalid when the Unicode object is garbage collected.

Adicionado na versão 3.3.

Alterado na versão 3.7: The return type is now `const char *` rather of `char *`.

Alterado na versão 3.10: This function is a part of the *limited API*.

const char *__PyUnicode_AsUTF8__(*PyObject* *unicode)

As *PyUnicode_AsUTF8AndSize()*, but does not store the size.

> **Aviso**
>
> This function does not have any special behavior for null characters embedded within *unicode*. As a result, strings containing null characters will remain in the returned string, which some C functions might interpret as the end of the string, leading to truncation. If truncation is an issue, it is recommended to use *PyUnicode_AsUTF8AndSize()* instead.

Adicionado na versão 3.3.

Alterado na versão 3.7: The return type is now `const char *` rather of `char *`.

UTF-32 Codecs

These are the UTF-32 codec APIs:

PyObject *__PyUnicode_DecodeUTF32__(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

Retorna valor: Nova referência. Parte da ABI Estável. Decode *size* bytes from a UTF-32 encoded buffer string and return the corresponding Unicode object. *errors* (if non-`NULL`) defines the error handling. It defaults to "strict".

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

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

If `*byteorder` is zero, and the first four bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output.

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

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

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

PyObject *__PyUnicode_DecodeUTF32Stateful__(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

Retorna valor: Nova referência. Parte da ABI Estável. If *consumed* is `NULL`, behave like *PyUnicode_DecodeUTF32()*. If *consumed* is not `NULL`, *PyUnicode_DecodeUTF32Stateful()* will not

treat trailing incomplete UTF-32 byte sequences (such as a number of bytes not divisible by four) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF32String**(*PyObject* *unicode)

Retorna valor: Nova referência. Parte da ABI Estável. Return a Python byte string using the UTF-32 encoding in native byte order. The string always starts with a BOM mark. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

UTF-16 Codecs

These are the UTF-16 codec APIs:

PyObject ***PyUnicode_DecodeUTF16**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

Retorna valor: Nova referência. Parte da ABI Estável. Decode *size* bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. *errors* (if non-`NULL`) defines the error handling. It defaults to "strict".

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

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

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

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

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

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

PyObject ***PyUnicode_DecodeUTF16Stateful**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

Retorna valor: Nova referência. Parte da ABI Estável. If *consumed* is `NULL`, behave like *PyUnicode_DecodeUTF16()*. If *consumed* is not `NULL`, *PyUnicode_DecodeUTF16Stateful()* will not treat trailing incomplete UTF-16 byte sequences (such as an odd number of bytes or a split surrogate pair) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF16String**(*PyObject* *unicode)

Retorna valor: Nova referência. Parte da ABI Estável. Return a Python byte string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

UTF-7 Codecs

These are the UTF-7 codec APIs:

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

Retorna valor: Nova referência. Parte da ABI Estável. Create a Unicode object by decoding *size* bytes of the UTF-7 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeUTF7Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Retorna valor: Nova referência. Parte da ABI Estável. If *consumed* is `NULL`, behave like *PyUnicode_DecodeUTF7()*. If *consumed* is not `NULL`, trailing incomplete UTF-7 base-64 sections will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

Unicode-Escape Codecs

These are the "Unicode Escape" codec APIs:

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

Retorna valor: Nova referência. Parte da ABI Estável. Create a Unicode object by decoding *size* bytes of the Unicode-Escape encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsUnicodeEscapeString**(*PyObject* *unicode)

Retorna valor: Nova referência. Parte da ABI Estável. Encode a Unicode object using Unicode-Escape and return the result as a bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

Raw-Unicode-Escape Codecs

These are the "Raw Unicode Escape" codec APIs:

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

Retorna valor: Nova referência. Parte da ABI Estável. Create a Unicode object by decoding *size* bytes of the Raw-Unicode-Escape encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsRawUnicodeEscapeString**(*PyObject* *unicode)

Retorna valor: Nova referência. Parte da ABI Estável. Encode a Unicode object using Raw-Unicode-Escape and return the result as a bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

Latin-1 Codecs

These are the Latin-1 codec APIs: Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

PyObject ***PyUnicode_DecodeLatin1**(const char *str, *Py_ssize_t* size, const char *errors)

Retorna valor: Nova referência. Parte da ABI Estável. Create a Unicode object by decoding *size* bytes of the Latin-1 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsLatin1String**(*PyObject* *unicode)

Retorna valor: Nova referência. Parte da ABI Estável. Encode a Unicode object using Latin-1 and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

ASCII Codecs

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

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

Retorna valor: Nova referência. Parte da ABI Estável. Create a Unicode object by decoding *size* bytes of the ASCII encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsASCIIString**(*PyObject* *unicode)

Retorna valor: Nova referência. Parte da ABI Estável. Encode a Unicode object using ASCII and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

Character Map Codecs

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the `encodings` package). The codec uses mappings to encode and decode characters. The mapping objects provided must support the `__getitem__()` mapping interface; dictionaries and sequences work well.

These are the mapping codec APIs:

PyObject ***PyUnicode_DecodeCharmap**(const char *str, *Py_ssize_t* length, *PyObject* *mapping, const char *errors)

Retorna valor: Nova referência. Parte da ABI Estável. Create a Unicode object by decoding *size* bytes of the encoded string *str* using the given *mapping* object. Return `NULL` if an exception was raised by the codec.

If *mapping* is `NULL`, Latin-1 decoding will be applied. Else *mapping* must map bytes ordinals (integers in the range from 0 to 255) to Unicode strings, integers (which are then interpreted as Unicode ordinals) or `None`. Unmapped data bytes – ones which cause a `LookupError`, as well as ones which get mapped to `None`, `0xFFFE` or `'\ufffe'`, are treated as undefined mappings and cause an error.

PyObject ***PyUnicode_AsCharmapString**(*PyObject* *unicode, *PyObject* *mapping)

Retorna valor: Nova referência. Parte da ABI Estável. Encode a Unicode object using the given *mapping* object and return the result as a bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

The *mapping* object must map Unicode ordinal integers to bytes objects, integers in the range from 0 to 255 or `None`. Unmapped character ordinals (ones which cause a `LookupError`) as well as mapped to `None` are treated as "undefined mapping" and cause an error.

The following codec API is special in that maps Unicode to Unicode.

PyObject ***PyUnicode_Translate**(*PyObject* *unicode, *PyObject* *table, const char *errors)

Retorna valor: Nova referência. Parte da ABI Estável. Translate a string by applying a character mapping table to it and return the resulting Unicode object. Return `NULL` if an exception was raised by the codec.

The mapping table must map Unicode ordinal integers to Unicode ordinal integers or `None` (causing deletion of the character).

Mapping tables need only provide the `__getitem__()` interface; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a `LookupError`) are left untouched and are copied as-is.

errors has the usual meaning for codecs. It may be `NULL` which indicates to use the default error handling.

MBCS codecs for Windows

These are the MBCS codec APIs. They are currently only available on Windows and use the Win32 MBCS converters to implement the conversions. Note that MBCS (or DBCS) is a class of encodings, not just one. The target encoding is defined by the user settings on the machine running the codec.

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

Retorna valor: Nova referência. Parte da ABI Estável *on Windows desde a versão 3.7.* Create a Unicode object by decoding *size* bytes of the MBCS encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeMBCSStateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Retorna valor: Nova referência. Parte da ABI Estável *on Windows desde a versão 3.7.* If *consumed* is `NULL`, behave like *`PyUnicode_DecodeMBCS()`*. If *consumed* is not `NULL`, *`PyUnicode_DecodeMBCSStateful()`* will not decode trailing lead byte and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_DecodeCodePageStateful**(int code_page, const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Retorna valor: Nova referência. Parte da ABI Estável *on Windows desde a versão 3.7.* Similar to *`PyUnicode_DecodeMBCSStateful()`*, except uses the code page specified by *code_page*.

PyObject ***PyUnicode_AsMBCSString**(*PyObject* *unicode)

Retorna valor: Nova referência. Parte da ABI Estável *on Windows desde a versão 3.7.* Encode a Unicode object using MBCS and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_EncodeCodePage**(int code_page, *PyObject* *unicode, const char *errors)

Retorna valor: Nova referência. Parte da ABI Estável *on Windows desde a versão 3.7.* Encode the Unicode object using the specified code page and return a Python bytes object. Return `NULL` if an exception was raised by the codec. Use `CP_ACP` code page to get the MBCS encoder.

Adicionado na versão 3.3.

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)

Retorna valor: Nova referência. Parte da ABI Estável. Concat two strings giving a new Unicode string.

PyObject ***PyUnicode_Split**(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

Retorna valor: Nova referência. Parte da ABI Estável. 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.

On error, return `NULL` with an exception set.

Equivalent to `str.split()`.

PyObject ***PyUnicode_RSplit**(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

Retorna valor: Nova referência. Parte da ABI Estável. Similar to *PyUnicode_Split()*, but splitting will be done beginning at the end of the string.

On error, return `NULL` with an exception set.

Equivalent to `str.rsplit()`.

PyObject ***PyUnicode_Splitlines**(*PyObject* *unicode, int keepends)

Retorna valor: Nova referência. Parte da ABI Estável. Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If *keepends* is `0`, the Line break characters are not included in the resulting strings.

PyObject ***PyUnicode_Partition**(*PyObject* *unicode, *PyObject* *sep)

Retorna valor: Nova referência. Parte da ABI Estável. Split a Unicode string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

sep must not be empty.

On error, return `NULL` with an exception set.

Equivalent to `str.partition()`.

PyObject ***PyUnicode_RPartition**(*PyObject* *unicode, *PyObject* *sep)

Retorna valor: Nova referência. Parte da ABI Estável. Similar to *PyUnicode_Partition()*, but split a Unicode string at the last occurrence of *sep*. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

sep must not be empty.

On error, return `NULL` with an exception set.

Equivalent to `str.rpartition()`.

PyObject ***PyUnicode_Join**(*PyObject* *separator, *PyObject* *seq)

Retorna valor: Nova referência. Parte da ABI Estável. Join a sequence of strings using the given *separator* and return the resulting Unicode string.

Py_ssize_t **PyUnicode_Tailmatch**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Parte da ABI Estável. Return `1` if *substr* matches `unicode[start:end]` at the given tail end (*direction* == `-1` means to do a prefix match, *direction* == `1` a suffix match), `0` otherwise. Return `-1` if an error occurred.

Py_ssize_t **PyUnicode_Find**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Parte da ABI Estável. Return the first position of *substr* in `unicode[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Py_ssize_t **PyUnicode_FindChar**(*PyObject* *unicode, *Py_UCS4* ch, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Parte da ABI Estável *desde a versão 3.7.* Return the first position of the character *ch* in `unicode[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Adicionado na versão 3.3.

Alterado na versão 3.7: *start* and *end* are now adjusted to behave like `unicode[start:end]`.

Py_ssize_t **PyUnicode_Count**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end)

Parte da ABI Estável. Return the number of non-overlapping occurrences of *substr* in `unicode[start:end]`. Return `-1` if an error occurred.

PyObject ***PyUnicode_Replace**(*PyObject* *unicode, *PyObject* *substr, *PyObject* *replstr, *Py_ssize_t* maxcount)

Retorna valor: Nova referência. Parte da ABI Estável. Replace at most *maxcount* occurrences of *substr* in *unicode* with *replstr* and return the resulting Unicode object. *maxcount* == `-1` means replace all occurrences.

int **PyUnicode_Compare**(*PyObject* *left, *PyObject* *right)

Parte da ABI Estável. Compare two strings and return `-1`, `0`, `1` for less than, equal, and greater than, respectively.

This function returns `-1` upon failure, so one should call *`PyErr_Occurred()`* to check for errors.

int **PyUnicode_EqualToUTF8AndSize**(*PyObject* *unicode, const char *string, *Py_ssize_t* size)

Parte da ABI Estável *desde a versão 3.13.* Compare a Unicode object with a char buffer which is interpreted as being UTF-8 or ASCII encoded and return true (`1`) if they are equal, or false (`0`) otherwise. If the Unicode object contains surrogate code points (`U+D800` - `U+DFFF`) or the C string is not valid UTF-8, false (`0`) is returned.

This function does not raise exceptions.

Adicionado na versão 3.13.

int **PyUnicode_EqualToUTF8**(*PyObject* *unicode, const char *string)

Parte da ABI Estável *desde a versão 3.13.* Similar to *`PyUnicode_EqualToUTF8AndSize()`*, but compute *string* length using `strlen()`. If the Unicode object contains null characters, false (`0`) is returned.

Adicionado na versão 3.13.

int **PyUnicode_CompareWithASCIIString**(*PyObject* *unicode, const char *string)

Parte da ABI Estável. Compare a Unicode object, *unicode*, with *string* and return `-1`, `0`, `1` for less than, equal, and greater than, respectively. It is best to pass only ASCII-encoded strings, but the function interprets the input string as ISO-8859-1 if it contains non-ASCII characters.

This function does not raise exceptions.

PyObject ***PyUnicode_RichCompare**(*PyObject* *left, *PyObject* *right, int op)

Retorna valor: Nova referência. Parte da ABI Estável. Rich compare two Unicode strings and return one of the following:

- `NULL` in case an exception was raised

- *Py_True* or *Py_False* for successful comparisons
- *Py_NotImplemented* in case the type combination is unknown

Possible values for *op* are *Py_GT*, *Py_GE*, *Py_EQ*, *Py_NE*, *Py_LT*, and *Py_LE*.

PyObject ***PyUnicode_Format**(*PyObject* *format, *PyObject* *args)

Retorna valor: Nova referência. Parte da ABI Estável. Return a new string object from *format* and *args*; this is analogous to `format % args`.

int **PyUnicode_Contains**(*PyObject* *unicode, *PyObject* *substr)

Parte da ABI Estável. Check whether *substr* is contained in *unicode* and return true or false accordingly.

substr has to coerce to a one element Unicode string. `-1` is returned if there was an error.

void **PyUnicode_InternInPlace**(*PyObject* **p_unicode)

Parte da ABI Estável. Intern the argument **p_unicode* in place. The argument must be the address of a pointer variable pointing to a Python Unicode string object. If there is an existing interned string that is the same as **p_unicode*, it sets **p_unicode* to it (releasing the reference to the old string object and creating a new *strong reference* to the interned string object), otherwise it leaves **p_unicode* alone and interns it.

(Clarification: even though there is a lot of talk about references, think of this function as reference-neutral. You must own the object you pass in; after the call you no longer own the passed-in reference, but you newly own the result.)

This function never raises an exception. On error, it leaves its argument unchanged without interning it.

Instances of subclasses of `str` may not be interned, that is, *PyUnicode_CheckExact*`(`**p_unicode*`)` must be true. If it is not, then – as with any other error – the argument is left unchanged.

Note that interned strings are not "immortal". You must keep a reference to the result to benefit from interning.

PyObject ***PyUnicode_InternFromString**(const char *str)

Retorna valor: Nova referência. Parte da ABI Estável. A combination of *PyUnicode_FromString()* and *PyUnicode_InternInPlace()*, meant for statically allocated strings.

Return a new ("owned") reference to either a new Unicode string object that has been interned, or an earlier interned string object with the same value.

Python may keep a reference to the result, or make it *immortal*, preventing it from being garbage-collected promptly. For interning an unbounded number of different strings, such as ones coming from user input, prefer calling *PyUnicode_FromString()* and *PyUnicode_InternInPlace()* directly.

Detalhes da implementação do CPython: Strings interned this way are made *immortal*.

8.3.4 Objeto tupla

type **PyTupleObject**

Este subtipo de *PyObject* representa um objeto tupla em Python.

PyTypeObject **PyTuple_Type**

Parte da ABI Estável. Esta instância de *PyTypeObject* representa o tipo tupla de Python; é o mesmo objeto que `tuple` na camada Python.

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

Retorna verdadeiro se *p* é um objeto tupla ou uma instância de um subtipo do tipo tupla. Esta função sempre tem sucesso.

int **PyTuple_CheckExact**(*PyObject* *p)

Retorna verdadeiro se *p* é um objeto tupla, mas não uma instância de um subtipo do tipo tupla. Esta função sempre tem sucesso.

PyObject ***PyTuple_New**(*Py_ssize_t* len)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um novo objeto tupla de tamanho *len*, ou `NULL` com uma exceção definida em caso de falha.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um novo objeto tupla de tamanho *n*, ou `NULL` com uma exceção em caso de falha. Os valores da tupla são inicializados para os *n* argumentos C subsequentes apontando para objetos Python. `PyTuple_Pack(2, a, b)` é equivalente a `Py_BuildValue("(OO)", a, b)`.

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

Parte da ABI Estável. Pega um ponteiro para um objeto tupla e retorna o tamanho dessa tupla. No erro, retorna `-1` e com uma exceção definida.

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

Como *PyTuple_Size()*, mas sem verificação de erro.

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

Retorna valor: Referência emprestada. Parte da ABI Estável. Retorna o objeto na posição *pos* na tupla apontada por *p*. Se *pos* estiver fora dos limites, retorna `NULL` e define uma exceção `IndexError`.

A referência retornada é emprestada da tupla *p* (ou seja: ela só é válida enquanto você mantém uma referência a *p*). Para obter uma *referência forte*, use *Py_NewRef(PyTuple_GetItem(...))* ou *PySequence_GetItem()*.

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

Retorna valor: Referência emprestada. Como *PyTuple_GetItem()*, mas faz nenhuma verificação de seus argumentos.

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

Retorna valor: Nova referência. Parte da ABI Estável. Retorna a fatia da tupla apontada por *p* entre *low* e *high*, ou `NULL` com uma exceção definida em caso de falha.

Isso é o equivalente da expressão Python `p[low:high]`. A indexação do final da tupla não é suportada.

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

Parte da ABI Estável. Insere uma referência ao objeto *o* na posição *pos* da tupla apontada por *p*. Retorna `0` em caso de sucesso. Se *pos* estiver fora dos limites, retorne `-1` e define uma exceção `IndexError`.

> **Nota**
>
> Esta função "rouba" uma referência a *o* e descarta uma referência a um item já na tupla na posição afetada.

void **PyTuple_SET_ITEM**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Como *PyTuple_SetItem()*, mas não verifica erros e deve *apenas* ser usado para preencher novas tuplas.

A verificação de limites é realizada como uma asserção se o Python for construído em modo de depuração ou `com asserções`.

> **Nota**
>
> Esta função "rouba" uma referência para *o* e, ao contrário de *PyTuple_SetItem()*, *não* descarta uma referência para nenhum item que esteja sendo substituído; qualquer referência na tupla na posição *pos* será perdida.

int **_PyTuple_Resize**(*PyObject* **p, *Py_ssize_t* newsize)

Pode ser usado para redimensionar uma tupla. *newsize* será o novo comprimento da tupla. Como as tuplas são *supostamente* imutáveis, isso só deve ser usado se houver apenas uma referência ao objeto. *Não* use isto se a tupla já for conhecida por alguma outra parte do código. A tupla sempre aumentará ou diminuirá no final. Pense nisso como destruir a tupla antiga e criar uma nova, mas com mais eficiência. Retorna `0` em caso de sucesso. O código do cliente nunca deve presumir que o valor resultante de `*p` será o mesmo de antes de chamar esta função. Se o objeto referenciado por `*p` for substituído, o `*p` original será destruído. Em caso de falha, retorna `-1` e define `*p` para `NULL`, e levanta `MemoryError` ou `SystemError`.

8.3.5 Objetos sequência de estrutura

Objetos sequência de estrutura são o equivalente em C dos objetos `namedtuple()`, ou seja, uma sequência cujos itens também podem ser acessados por meio de atributos. Para criar uma sequência de estrutura, você primeiro precisa criar um tipo de sequência de estrutura específico.

PyTypeObject ***PyStructSequence_NewType**(*PyStructSequence_Desc* *desc)

Retorna valor: Nova referência. *Parte da* ABI Estável. Cria um novo tipo de sequência de estrutura a partir dos dados em *desc*, descrito abaixo. Instâncias do tipo resultante podem ser criadas com *PyStructSequence_New()*.

Retorna `NULL` com uma exceção definida em caso de falha.

void **PyStructSequence_InitType**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

Inicializa um tipo de sequência de estrutura *type* de *desc* no lugar.

int **PyStructSequence_InitType2**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

Como *PyStructSequence_InitType()*, mas retorna `0` em caso de sucesso e `-1` com uma exceção definida em caso de falha.

Adicionado na versão 3.4.

type **PyStructSequence_Desc**

Parte da ABI Estável *(incluindo todos os membros).* Contém as metainformações de um tipo de sequência de estrutura a ser criado.

const char ***name**

Nome totalmente qualificado do tipo; codificado em UTF-8 terminado em nulo. O nome deve conter o nome do módulo.

const char ***doc**

Ponteiro para docstring para o tipo ou `NULL` para omitir

PyStructSequence_Field ***fields**

Ponteiro para um vetor terminado em `NULL` com nomes de campos do novo tipo

int **n_in_sequence**

Número de campos visíveis para o lado Python (se usado como tupla)

type **PyStructSequence_Field**

Parte da ABI Estável *(incluindo todos os membros).* Descreve um campo de uma sequência de estrutura. Como uma sequência de estrutura é modelada como uma tupla, todos os campos são digitados como *PyObject**. O índice no vetor *fields* do *PyStructSequence_Desc* determina qual campo da sequência de estrutura é descrito.

const char ***name**

Nome do campo ou `NULL` para terminar a lista de campos nomeados; definida para *PyStructSequence_UnnamedField* para deixar sem nome.

const char ***doc**

Campo docstring ou `NULL` para omitir.

const char *const **PyStructSequence_UnnamedField**

Parte da ABI Estável *desde a versão 3.11.* Valor especial para um nome de campo para deixá-lo sem nome.

Alterado na versão 3.9: O tipo foi alterado de `char *`.

PyObject ***PyStructSequence_New**(*PyTypeObject* *type)

Retorna valor: Nova referência. *Parte da* ABI Estável. Cria um instância de *type*, que deve ser criada com *PyStructSequence_NewType()*.

Retorna `NULL` com uma exceção definida em caso de falha.

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

Retorna valor: Referência emprestada. *Parte da* ABI Estável. Retorna o objeto na posição *pos* na sequência de estrutura apontada por *p*.

A verificação de limites é realizada como uma asserção se o Python for construído em modo de depuração ou `com asserções`.

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

Retorna valor: Referência emprestada. Apelido para *PyStructSequence_GetItem()*.

Alterado na versão 3.13: Agora implementada como um apelido para *PyStructSequence_GetItem()*.

void **PyStructSequence_SetItem**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Parte da ABI Estável. Define o campo no índice *pos* da sequência de estrutura *p* para o valor *o*. Como *PyTuple_SET_ITEM()*, isto só deve ser usado para preencher novas instâncias.

A verificação de limites é realizada como uma asserção se o Python for construído em modo de depuração ou `com asserções`.

> **Nota**
>
> Esta função "rouba" uma referência a *o*.

void **PyStructSequence_SET_ITEM**(*PyObject* *p, *Py_ssize_t* *pos, *PyObject* *o)

Apelido para *PyStructSequence_SetItem()*.

Alterado na versão 3.13: Agora implementada como um apelido para *PyStructSequence_SetItem()*.

8.3.6 Objeto List

type **PyListObject**

Este subtipo de *PyObject* representa um objeto de lista Python.

PyTypeObject **PyList_Type**

Parte da ABI Estável. Esta instância de *PyTypeObject* representa o tipo de lista Python. Este é o mesmo objeto que `list` na camada Python.

int **PyList_Check**(*PyObject* *p)

Retorna verdadeiro se *p* é um objeto lista ou uma instância de um subtipo do tipo lista. Esta função sempre tem sucesso.

int **PyList_CheckExact**(*PyObject* *p)

Retorna verdadeiro se *p* é um objeto lista, mas não uma instância de um subtipo do tipo lista. Esta função sempre tem sucesso.

PyObject ***PyList_New**(*Py_ssize_t* len)

Retorna valor: Nova referência. *Parte da* ABI Estável. Retorna uma nova lista de comprimento *len* em caso de sucesso, ou `NULL` em caso de falha.

> **Nota**
>
> Se *len* for maior que zero, os itens do objeto da lista retornada são definidos como `NULL`. Portanto, você não pode usar funções API abstratas, como *PySequence_SetItem()* ou expor o objeto ao código Python antes de definir todos os itens para um objeto real com *PyList_SetItem()* ou *PyList_SET_ITEM()*. As seguintes APIs são seguras desde que antes as listas tenham sido inicializadas: *PyList_SetItem()* e *PyList_SET_ITEM()*.

Py_ssize_t **PyList_Size**(*PyObject* *list)

Parte da ABI Estável. Retorna o comprimento do objeto de lista em *list*; isto é equivalente a `len(list)` em um objeto lista.

Py_ssize_t **PyList_GET_SIZE**(*PyObject* *list)

Similar a *PyList_Size()*, mas sem verificação de erro.

PyObject ***PyList_GetItemRef**(*PyObject* *list, *Py_ssize_t* index)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.13.* Retorna o objeto na posição *index* na lista apontada por *list*. A posição deve ser não negativa; não há suporte à indexação do final da lista. Se *index* estiver fora dos limites (`<0 or >=len(list)`), retorna `NULL` e levanta uma exceção `IndexError`.

Adicionado na versão 3.13.

PyObject ***PyList_GetItem**(*PyObject* *list, *Py_ssize_t* index)

Retorna valor: Referência emprestada. Parte da ABI Estável. Como *PyList_GetItemRef()*, mas retorna uma *referência emprestada* em vez de uma *referência forte*.

PyObject ***PyList_GET_ITEM**(*PyObject* *list, *Py_ssize_t* i)

Retorna valor: Referência emprestada. Similar a *PyList_GetItem()*, mas sem verificação de erro.

int **PyList_SetItem**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

Parte da ABI Estável. Define o item no índice *index* na lista como *item*. Retorna `0` em caso de sucesso. Se *index* estiver fora dos limites, retorna `-1` e levanta uma exceção `IndexError`.

> **Nota**
>
> Esta função "rouba" uma referência para o *item* e descarta uma referência para um item já presente na lista na posição afetada.

void **PyList_SET_ITEM**(*PyObject* *list, *Py_ssize_t* i, *PyObject* *o)

Forma macro de *PyList_SetItem()* sem verificação de erro. Este é normalmente usado apenas para preencher novas listas onde não há conteúdo anterior.

A verificação de limites é realizada como uma asserção se o Python for construído em modo de depuração ou `com asserções`.

> **Nota**
>
> Esta macro "rouba" uma referência para o *item* e, ao contrário de *PyList_SetItem()*, *não* descarta uma referência para nenhum item que esteja sendo substituído; qualquer referência em *list* será perdida.

int **PyList_Insert**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

Parte da ABI Estável. Insere o item *item* na lista *list* na frente do índice *index*. Retorna `0` se for bem-sucedido; retorna `-1` e levanta uma exceção se malsucedido. Análogo a `list.insert(index, item)`.

int **PyList_Append**(*PyObject* *list, *PyObject* *item)

Parte da ABI Estável. Adiciona o item *item* ao final da lista *list*. Retorna `0` se for bem-sucedido; retorna `-1` e levanta uma exceção se malsucedido. Análogo a `list.insert(index, item)`.

PyObject ***PyList_GetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna uma lista dos objetos em *list* contendo os objetos *entre low* e *alto*. Retorne `NULL` e levanta uma exceção se malsucedido. Análogo a `list[low:high]`. Não há suporte à indexação do final da lista.

int **PyList_SetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high, *PyObject* *itemlist)

Parte da ABI Estável. Define a fatia de *list* entre *low* e *high* para o conteúdo de *itemlist*. Análogo a `list[low:high] = itemlist`. *itemlist* pode ser `NULL`, indicando a atribuição de uma lista vazia (exclusão de fatia). Retorna `0` em caso de sucesso, `-1` em caso de falha. Não há suporte à indexação do final da lista.

int **PyList_Extend**(*PyObject* *list, *PyObject* *iterable)

Estende *list* com o conteúdo de *iterable*. Isto é o mesmo que `PyList_SetSlice(list, PY_SSIZE_T_MAX, PY_SSIZE_T_MAX, iterable)` e análogo a `list.extend(iterable)` ou `list += iterable`.

Levanta uma exceção e retorna `-1` se *list* não for um objeto `list`. Retorna 0 em caso de sucesso.

Adicionado na versão 3.13.

int **PyList_Clear**(*PyObject* *list)

Remove todos os itens da *lista*. Isto é o mesmo que `PyList_SetSlice(list, 0, PY_SSIZE_T_MAX, NULL)` e análogo a `list.clear()` ou `del list[:]`.

Levanta uma exceção e retorna `-1` se *list* não for um objeto `list`. Retorna 0 em caso de sucesso.

Adicionado na versão 3.13.

int **PyList_Sort**(*PyObject* *list)

Parte da ABI Estável. Ordena os itens de *list* no mesmo lugar. Retorna `0` em caso de sucesso, e `-1` em caso de falha. Isso é o equivalente de `list.sort()`.

int **PyList_Reverse**(*PyObject* *list)

Parte da ABI Estável. Inverte os itens 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)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um novo objeto tupla contendo os conteúdos de *list*; equivale a `tuple(list)`.

8.4 Coleções

8.4.1 Objetos dicionários

type **PyDictObject**

Este subtipo do *PyObject* representa um objeto dicionário Python.

PyTypeObject **PyDict_Type**

Parte da ABI Estável. Esta instância do *PyTypeObject* representa o tipo do dicionário Python. Este é o mesmo objeto `dict` na camada do Python.

int **PyDict_Check**(*PyObject* *p)

Retorna verdadeiro se *p* é um objeto dicionário ou uma instância de um subtipo do tipo dicionário. Esta função sempre tem sucesso.

int **PyDict_CheckExact**(*PyObject* *p)

Retorna verdadeiro se *p* é um objeto dicionário, mas não uma instância de um subtipo do tipo dicionário. Esta função sempre tem sucesso.

PyObject ***PyDict_New**()

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um novo dicionário vazio ou `NULL` em caso de falha.

PyObject ***PyDictProxy_New**(*PyObject* *mapping)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um objeto `types.MappingProxyType` para um mapeamento que reforça o comportamento somente leitura. Isso normalmente é usado para criar uma visão para evitar a modificação do dicionário para tipos de classes não dinâmicas.

void **PyDict_Clear**(*PyObject* *p)

Parte da ABI Estável. Esvazia um dicionário existente de todos os pares chave-valor.

int **PyDict_Contains**(*PyObject* *p, *PyObject* *key)

Parte da ABI Estável. Determina se o dicionário *p* contém *key*. Se um item em *p* corresponder à *key*, retorna `1`, caso contrário, retorna `0`. Em caso de erro, retorna `-1`. Isso é equivalente à expressão Python `key in p`.

int **PyDict_ContainsString**(*PyObject* *p, const char *key)

É o mesmo que *PyDict_Contains()*, mas *key* é especificada como uma string de bytes `const char*` codificada em UTF-8, em vez de um *PyObject**.

Adicionado na versão 3.13.

PyObject ***PyDict_Copy**(*PyObject* *p)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um novo dicionário que contém o mesmo chave-valor como *p*.

int **PyDict_SetItem**(*PyObject* *p, *PyObject* *key, *PyObject* *val)

Parte da ABI Estável. Insere *val* no dicionário *p* com a tecla *key*. *key* deve ser *hasheável*; se não for, `TypeError` será levantada. Retorna `0` em caso de sucesso ou `-1` em caso de falha. Esta função *não* rouba uma referência a *val*.

int **PyDict_SetItemString**(*PyObject* *p, const char *key, *PyObject* *val)

Parte da ABI Estável. É o mesmo que *PyDict_SetItem()*, mas *key* é especificada como uma string de bytes `const char*` codificada em UTF-8, em vez de um *PyObject**.

int **PyDict_DelItem**(*PyObject* *p, *PyObject* *key)

Parte da ABI Estável. Remove a entrada no dicionário *p* com a chave *key*. *key* deve ser *hasheável*; se não for, `TypeError` é levantada. Se *key* não estiver no dicionário, `KeyError` é levantada. Retorna `0` em caso de sucesso ou `-1` em caso de falha.

int **PyDict_DelItemString**(*PyObject* *p, const char *key)

Parte da ABI Estável. É o mesmo que *PyDict_DelItem()*, mas *key* é especificada como uma string de bytes `const char*` codificada em UTF-8, em vez de um *PyObject**.

int **PyDict_GetItemRef**(*PyObject* *p, *PyObject* *key, *PyObject* **result)

Parte da ABI Estável *desde a versão 3.13.* Retorna uma nova *referência forte* para o objeto dicionário *p* que possui uma chave *key*:

- Se a chave estiver presente, define **result* como uma nova *referência forte* para o valor e retorna `1`.
- Se a chave estiver ausente, define **result* como `NULL` e retorna `0`.
- Em caso de erro, levanta uma exceção e retorna `-1`.

Adicionado na versão 3.13.

Veja também a função *PyObject_GetItem()*.

PyObject ***PyDict_GetItem**(*PyObject* *p, *PyObject* *key)

Retorna valor: Referência emprestada. Parte da ABI Estável. Retorna um *referência emprestada* para o objeto do dicionário *p* que possui uma chave *key*. Retorna `NULL` se a chave *key* não estiver presente, mas *sem* definir uma exceção.

> **Nota**
>
> Exceções que ocorrem ao chamar os métodos `__hash__()` e `__eq__()` são ignoradas silenciosamente. Ao invés disso, use a função *PyDict_GetItemWithError()*.

Alterado na versão 3.10: Chamar esta API sem *GIL* retida era permitido por motivos históricos. Não é mais permitido.

PyObject ***PyDict_GetItemWithError**(*PyObject* *p, *PyObject* *key)

Retorna valor: Referência emprestada. *Parte da* ABI Estável. Variante de `PyDict_GetItem()` que não suprime exceções. Retorna `NULL` **com** uma exceção definida se uma exceção ocorreu. Retorna `NULL` ** sem ** uma exceção definida se a chave não estiver presente.

PyObject ***PyDict_GetItemString**(*PyObject* *p, const char *key)

Retorna valor: Referência emprestada. *Parte da* ABI Estável. É o mesmo que `PyDict_GetItem()`, mas *key* é especificada como uma string de bytes `const char*` codificada em UTF-8, em vez de um `PyObject*`.

> **Nota**
>
> Exceções que ocorrem ao chamar os métodos `__hash__()` e `__eq__()` ou ao criar objetos temporários da classe `str` são ignoradas silenciosamente. Ao invés disso, prefira usar a função `PyDict_GetItemWithError()` com sua própria *key* de `PyUnicode_FromString()`.

int **PyDict_GetItemStringRef**(*PyObject* *p, const char *key, *PyObject* **result)

Parte da ABI Estável *desde a versão 3.13.* Similar a `PyDict_GetItemRef()`, mas *key* é especificada como uma string de bytes `const char*` codificada em UTF-8, em vez de um `PyObject*`.

Adicionado na versão 3.13.

PyObject ***PyDict_SetDefault**(*PyObject* *p, *PyObject* *key, *PyObject* *defaultobj)

Retorna valor: Referência emprestada. Isso é o mesmo que o `dict.setdefault()` de nível Python. Se presente, ele retorna o valor correspondente a *key* do dicionário *p*. Se a chave não estiver no dict, ela será inserida com o valor *defaultobj* e *defaultobj* será retornado. Esta função avalia a função hash de *key* apenas uma vez, em vez de avaliá-la independentemente para a pesquisa e a inserção.

Adicionado na versão 3.4.

int **PyDict_SetDefaultRef**(*PyObject* *p, *PyObject* *key, *PyObject* *default_value, *PyObject* **result)

Insere *default_value* no dicionário *p* com uma chave *key* se a chave ainda não estiver presente no dicionário. Se *result* não for `NULL`, então **result* é definido como uma *referência forte* para *default_value*, se a chave não estiver presente, ou para o valor existente, se *key* já estava presente no dicionário. Retorna `1` se a chave estava presente e *default_value* não foi inserido, ou `0` se a chave não estava presente e *default_value* foi inserido. Em caso de falha, retorna `-1`, define uma exceção e define `*result` como `NULL`.

Para maior clareza: se você tiver uma referência forte para *default_value* antes de chamar esta função, então depois que ela retornar, você terá uma referência forte para *default_value* e **result* (se não for `NULL`). Estes podem referir-se ao mesmo objeto: nesse caso você mantém duas referências separadas para ele.

Adicionado na versão 3.13.

int **PyDict_Pop**(*PyObject* *p, *PyObject* *key, *PyObject* **result)

Remove *key* do dicionário *p* e, opcionalmente, retorna o valor removido. Não levanta `KeyError` se a chave estiver ausente.

- Se a chave estiver presente, define **result* como uma nova referência para o valor se *result* não for `NULL` e retorna `1`.
- Se a chave estiver ausente, define **result* como `NULL` se *result* não for `NULL` e retorna `0`.
- Em caso de erro, levanta uma exceção e retorna `-1`.

Similar a `dict.pop()`, mas sem o valor padrão e sem levantar `KeyError` se a chave estiver ausente.

Adicionado na versão 3.13.

int **PyDict_PopString**(*PyObject* *p, const char *key, *PyObject* **result)

Similar a `PyDict_Pop()`, mas *key* é especificada como uma string de bytes `const char*` codificada em UTF-8, em vez de um `PyObject*`.

Adicionado na versão 3.13.

PyObject ***PyDict_Items**(*PyObject* *p)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um `PyListObject` contendo todos os itens do dicionário.

PyObject ***PyDict_Keys**(*PyObject* *p)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um `PyListObject` contendo todas as chaves do dicionário.

PyObject ***PyDict_Values**(*PyObject* *p)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um `PyListObject` contendo todos os valores do dicionário *p*.

Py_ssize_t **PyDict_Size**(*PyObject* *p)

Parte da ABI Estável. Retorna o número de itens no dicionário. Isso é equivalente a `len(p)` em um dicionário.

int **PyDict_Next**(*PyObject* *p, *Py_ssize_t* *ppos, *PyObject* **pkey, *PyObject* **pvalue)

Parte da ABI Estável. Itera todos os pares de valores-chave no dicionário *p*. O `Py_ssize_t` referido por *ppos* deve ser inicializado para `0` antes da primeira chamada para esta função para iniciar a iteração; a função retorna true para cada par no dicionário e false quando todos os pares forem relatados. Os parâmetros *pkey* e *pvalue* devem apontar para variáveis de `PyObject*` que serão preenchidas com cada chave e valor, respectivamente, ou podem ser `NULL`. Todas as referências retornadas por meio deles são emprestadas. *ppos* não deve ser alterado durante a iteração. Seu valor representa deslocamentos dentro da estrutura do dicionário interno e, como a estrutura é esparsa, os deslocamentos não são consecutivos.

Por exemplo:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* fazer algo de interessante com os valores... */
    ...
}
```

O dicionário *p* não deve sofrer mutação durante a iteração. É seguro modificar os valores das chaves à medida que você itera no dicionário, mas apenas enquanto o conjunto de chaves não mudar. Por exemplo:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    long i = PyLong_AsLong(value);
    if (i == -1 && PyErr_Occurred()) {
        return -1;
    }
    PyObject *o = PyLong_FromLong(i + 1);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
    Py_DECREF(o);
}
```

A função não é segura para thread na construção com *threads livres* sem sincronização externa. Você pode usar `Py_BEGIN_CRITICAL_SECTION` para travar o dicionário enquanto itera sobre ele:

```
Py_BEGIN_CRITICAL_SECTION(self->dict);
while (PyDict_Next(self->dict, &pos, &key, &value)) {
    ...
}
Py_END_CRITICAL_SECTION();
```

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

Parte da ABI Estável. Itera sobre o objeto de mapeamento *b* adicionando pares de valores-chave ao dicionário *a*. *b* pode ser um dicionário, ou qualquer objeto que suporte `PyMapping_Keys()` e `PyObject_GetItem()`. Se *override* for verdadeiro, os pares existentes em *a* serão substituídos se uma chave correspondente for encontrada em *b*, caso contrário, os pares serão adicionados apenas se não houver uma chave correspondente em *a*. Retorna `0` em caso de sucesso ou `-1` se uma exceção foi levantada.

int **PyDict_Update**(*PyObject* *a, *PyObject* *b)

Parte da ABI Estável. É o mesmo que `PyDict_Merge(a, b, 1)` em C, e é semelhante a `a.update(b)` em Python, exceto que `PyDict_Update()` não cai na iteração em uma sequência de pares de valores de chave se o segundo argumento não tiver o atributo "keys". Retorna `0` em caso de sucesso ou `-1` se uma exceção foi levantada.

int **PyDict_MergeFromSeq2**(*PyObject* *a, *PyObject* *seq2, int override)

Parte da ABI Estável. Atualiza ou mescla no dicionário *a*, a partir dos pares de chave-valor em *seq2*. *seq2* deve ser um objeto iterável produzindo objetos iteráveis de comprimento 2, vistos como pares chave-valor. No caso de chaves duplicadas, a última vence se *override* for verdadeiro, caso contrário, a primeira vence. Retorne `0` em caso de sucesso ou `-1` se uma exceção foi levantada. Python equivalente (exceto para o valor de retorno):

```
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value
```

int **PyDict_AddWatcher**(*PyDict_WatchCallback* callback)

Registra *callback* como um observador de dicionário. Retorna um ID inteiro não negativo que deve ser passado para futuras chamadas a `PyDict_Watch()`. Em caso de erro (por exemplo, não há mais IDs de observador disponíveis), retorna `-1` e define uma exceção.

Adicionado na versão 3.12.

int **PyDict_ClearWatcher**(int watcher_id)

Limpa o observador identificado por *watcher_id* retornado anteriormente de `PyDict_AddWatcher()`. Retorna `0` em caso de sucesso, `-1` em caso de erro (por exemplo, se o *watcher_id* fornecido nunca foi registrado).

Adicionado na versão 3.12.

int **PyDict_Watch**(int watcher_id, *PyObject* *dict)

Marca o dicionário *dict* como observado. A função de retorno concedida a *watcher_id* por `PyDict_AddWatcher()` será chamada quando *dict* for modificado ou desalocado. Retorna `0` em caso de sucesso ou `-1` em caso de erro.

Adicionado na versão 3.12.

int **PyDict_Unwatch**(int watcher_id, *PyObject* *dict)

Marca o dicionário *dict* como não mais observado. A função de retorno concedida a *watcher_id* por `PyDict_AddWatcher()` será chamada quando *dict* for modificado ou desalocado. O dicionário deve ter sido observado anteriormente por este observador. Retorna `0` em caso de sucesso ou `-1` em caso de erro.

Adicionado na versão 3.12.

type **PyDict_WatchEvent**

Enumeração de possíveis eventos de observador de dicionário: `PyDict_EVENT_ADDED`, `PyDict_EVENT_MODIFIED`, `PyDict_EVENT_DELETED`, `PyDict_EVENT_CLONED`, `PyDict_EVENT_CLEARED` ou `PyDict_EVENT_DEALLOCATED`.

Adicionado na versão 3.12.

typedef int (***PyDict_WatchCallback**)(*PyDict_WatchEvent* event, *PyObject* *dict, *PyObject* *key, *PyObject* *new_value)

Tipo de uma função de retorno de chamada de observador de dicionário.

Se *event* for `PyDict_EVENT_CLEARED` ou `PyDict_EVENT_DEALLOCATED`, tanto *key* quanto *new_value* serão `NULL`. Se *event* for `PyDict_EVENT_ADDED` ou `PyDict_EVENT_MODIFIED`, *new_value* será o novo valor de *key*. Se *event* for `PyDict_EVENT_DELETED`, *key* estará sendo excluída do dicionário e *new_value* será `NULL`.

`PyDict_EVENT_CLONED` ocorre quando *dict* estava anteriormente vazio e outro dict é mesclado a ele. Para manter a eficiência dessa operação, os eventos `PyDict_EVENT_ADDED` por chave não são emitidos nesse caso; em vez disso, um único `PyDict_EVENT_CLONED` é emitido e *key* será o dicionário de origem.

A função de retorno pode inspecionar, mas não deve modificar o *dict*; isso pode ter efeitos imprevisíveis, inclusive recursão infinita. Não acione a execução do código Python na função de retorno, pois isso poderia modificar o dict como um efeito colateral.

Se *event* for `PyDict_EVENT_DEALLOCATED`, a obtenção de uma nova referência na função de retorno para o dicionário prestes a ser destruído o ressuscitará e impedirá que ele seja liberado nesse momento. Quando o objeto ressuscitado for destruído mais tarde, quaisquer funções de retorno do observador ativos naquele momento serão chamados novamente.

As funções de retorno ocorrem antes que a modificação notificada no *dict* ocorra, de modo que o estado anterior do *dict* possa ser inspecionado.

Se a função de retorno definir uma exceção, ela deverá retornar `-1`; essa exceção será impressa como uma exceção não reprovável usando *PyErr_WriteUnraisable()*. Caso contrário, deverá retornar `0`.

É possível que já exista uma exceção pendente definida na entrada da função de retorno. Nesse caso, a função de retorno deve retornar `0` com a mesma exceção ainda definida. Isso significa que a função de retorno não pode chamar nenhuma outra API que possa definir uma exceção, a menos que salve e limpe o estado da exceção primeiro e o restaure antes de retornar.

Adicionado na versão 3.12.

8.4.2 Objeto Set

Esta seção detalha a API pública para os objetos `set` e `frozenset`. Qualquer funcionalidade não listada abaixo é melhor acessada usando o protocolo de objeto abstrato (incluindo *PyObject_CallMethod()*, *PyObject_RichCompareBool()*, *PyObject_Hash()*, *PyObject_Repr()*, *PyObject_IsTrue()*, *PyObject_Print()* e *PyObject_GetIter()*) ou o protocolo abstrato de número (incluindo *PyNumber_And()*, *PyNumber_Subtract()*, *PyNumber_Or()*, *PyNumber_Xor()*, *PyNumber_InPlaceAnd()*, *PyNumber_InPlaceSubtract()*, *PyNumber_InPlaceOr()* e *PyNumber_InPlaceXor()*).

type **PySetObject**

Este subtipo de *PyObject* é usado para manter os dados internos para ambos os objetos `set` e `frozenset`. É como um *PyDictObject* em que tem um tamanho fixo para conjuntos pequenos (muito parecido com o armazenamento de tupla) e apontará para um bloco de memória de tamanho variável separado para conjuntos de tamanho médio e grande (muito parecido com lista armazenamento). Nenhum dos campos desta estrutura deve ser considerado público e todos estão sujeitos a alterações. Todo o acesso deve ser feito por meio da API documentada, em vez de manipular os valores na estrutura.

PyTypeObject **PySet_Type**

Parte da ABI Estável. Essa é uma instância de *PyTypeObject* representando o tipo Python `set`

PyTypeObject **PyFrozenSet_Type**

Parte da ABI Estável. Esta é uma instância de *PyTypeObject* representando o tipo Python `frozenset`.

As macros de verificação de tipo a seguir funcionam em ponteiros para qualquer objeto Python. Da mesma forma, as funções construtoras funcionam com qualquer objeto Python iterável.

int **PySet_Check**(*PyObject* *p)

Retorna verdadeiro se *p* for um objeto `set` ou uma instância de um subtipo. Esta função sempre tem sucesso.

int **PyFrozenSet_Check**(*PyObject* *p)

Retorna verdadeiro se *p* for um objeto `frozenset` ou uma instância de um subtipo. Esta função sempre tem sucesso.

int **PyAnySet_Check**(*PyObject* *p)

Retorna verdadeiro se *p* for um objeto `set`, um objeto `frozenset` ou uma instância de um subtipo. Esta função sempre tem sucesso.

int **PySet_CheckExact**(*PyObject* *p)

Retorna verdadeiro se *p* for um objeto `set`, mas não uma instância de um subtipo. Esta função sempre tem sucesso.

Adicionado na versão 3.10.

int **PyAnySet_CheckExact**(*PyObject* *p)

Retorna verdadeiro se *p* for um objeto `set` ou um objeto `frozenset`, mas não uma instância de um subtipo. Esta função sempre tem sucesso.

int **PyFrozenSet_CheckExact**(*PyObject* *p)

Retorna verdadeiro se *p* for um objeto `frozenset`, mas não uma instância de um subtipo. Esta função sempre tem sucesso.

PyObject ***PySet_New**(*PyObject* *iterable)

Retorna valor: Nova referência. *Parte da* ABI Estável. Retorna uma nova `set` contendo objetos retornados pelo iterável *iterable*. O *iterable* pode ser `NULL` para criar um novo conjunto vazio. Retorna o novo conjunto em caso de sucesso ou `NULL` em caso de falha. Levanta `TypeError` se *iterable* não for realmente iterável. O construtor também é útil para copiar um conjunto (`c=set(s)`).

PyObject ***PyFrozenSet_New**(*PyObject* *iterable)

Retorna valor: Nova referência. *Parte da* ABI Estável. Retorna uma nova `frozenset` contendo objetos retornados pelo iterável *iterable*. O *iterable* pode ser `NULL` para criar um novo frozenset vazio. Retorna o novo conjunto em caso de sucesso ou `NULL` em caso de falha. Levanta `TypeError` se *iterable* não for realmente iterável.

As seguintes funções e macros estão disponíveis para instâncias de `set` ou `frozenset` ou instâncias de seus subtipos.

Py_ssize_t **PySet_Size**(*PyObject* *anyset)

Parte da ABI Estável. Retorna o comprimento de um objeto `set` ou `frozenset`. Equivalente a `len(anyset)`. Levanta um `SystemError` se *anyset* não for um `set`, `frozenset`, ou uma instância de um subtipo.

Py_ssize_t **PySet_GET_SIZE**(*PyObject* *anyset)

Forma macro de *PySet_Size()* sem verificação de erros.

int **PySet_Contains**(*PyObject* *anyset, *PyObject* *key)

Parte da ABI Estável. Retorna `1` se encontrado, `0` se não encontrado, e `-1` se um erro é encontrado. Ao contrário do método Python `__contains__()`, esta função não converte automaticamente conjuntos não hasheáveis em frozensets temporários. Levanta um `TypeError` se a *key* não for hasheável. Levanta `SystemError` se *anyset* não é um `set`, `frozenset`, ou uma instância de um subtipo.

int **PySet_Add**(*PyObject* *set, *PyObject* *key)

Parte da ABI Estável. Adiciona *key* a uma instância de `set`. Também funciona com instâncias de `frozenset` (como *PyTuple_SetItem()*, ele pode ser usado para preencher os valores de novos conjuntos de congelamentos antes que eles sejam expostos a outro código). Retorna `0` em caso de sucesso ou `-1` em caso de falha. Levanta um `TypeError` se a *key* não for hasheável. Levanta uma `MemoryError` se não houver espaço para crescer. Levanta uma `SystemError` se *set* não for uma instância de `set` ou seu subtipo.

As seguintes funções estão disponíveis para instâncias de `set` ou seus subtipos, mas não para instâncias de `frozenset` ou seus subtipos.

int **PySet_Discard**(*PyObject* *set, *PyObject* *key)

Parte da ABI Estável. Retorna `1` se encontrado e removido, `0` se não encontrado (nenhuma ação realizada) e `-1` se um erro for encontrado. Não levanta `KeyError` para chaves ausentes. Levanta uma `TypeError` se a *key* não for hasheável. Ao contrário do método Python `discard()`, esta função não converte automaticamente conjuntos não hasheáveis em frozensets temporários. Levanta `SystemError` se *set* não é uma instância de `set` ou seu subtipo.

PyObject ***PySet_Pop**(*PyObject* *set)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna uma nova referência a um objeto arbitrário no *set* e remove o objeto do *set*. Retorna `NULL` em caso de falha. Levanta `KeyError` se o conjunto estiver vazio. Levanta uma `SystemError` se *set* não for uma instância de `set` ou seu subtipo.

int **PySet_Clear**(*PyObject* *set)

Parte da ABI Estável. Esvazia um conjunto existente de todos os elementos. Retorna `0` em caso de sucesso. Retorna `-1` e levanta `SystemError` se *set* não for uma instância de `set` ou seu subtipo.

8.5 Objetos Função

8.5.1 Objetos Função

Existem algumas funções específicas para as funções do Python.

type **PyFunctionObject**

A estrutura C usada para funções.

PyTypeObject **PyFunction_Type**

Esta é uma instância de *PyTypeObject* e representa o tipo de função Python. Está exposta a programadores Python como `types.FunctionType`.

int **PyFunction_Check**(*PyObject* *o)

Retorna verdadeiro se *o* for um objeto função (tem tipo *PyFunction_Type*). O parâmetro não deve ser `NULL`. Esta função sempre obtém sucesso.

PyObject ***PyFunction_New**(*PyObject* *code, *PyObject* *globals)

Retorna valor: Nova referência. Retorna um novo objeto função associado ao objeto código *code*. *globals* deve ser um dicionário com as variáveis globais acessíveis à função.

O nome e *docstring* da função são adquiridos pelo objeto código. O atributo `__module__` é adquirido por meio de *globals*. Os valores-padrão de argumentos, anotações, e fechamento são definidos como `NULL`. O atributo `__qualname__` é definido com o mesmo valor do campo `co_qualname` de um objeto código.

PyObject ***PyFunction_NewWithQualName**(*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)

Retorna valor: Nova referência. Similar a *PyFunction_New()*, mas também permite definir o atributo `__qualname__` do objeto função. *qualname* deve ser um objeto Unicode ou `NULL`. Se `NULL`, o atributo `__qualname__` é definido com o mesmo valor do campo `co_qualname` do objeto código.

Adicionado na versão 3.3.

PyObject ***PyFunction_GetCode**(*PyObject* *op)

Retorna valor: Referência emprestada. Retorna o objeto código associado ao objeto função *op*.

PyObject ***PyFunction_GetGlobals**(*PyObject* *op)

Retorna valor: Referência emprestada. Retorna o dicionário global associado ao objeto função *op*.

PyObject ***PyFunction_GetModule**(*PyObject* *op)

Retorna valor: Referência emprestada. Retorna uma *referência emprestada* ao atributo `__module__` do objeto função *op*. Pode ser *NULL*.

Normalmente, trata-se de um `string` contendo o nome do módulo, mas pode ser definido como qualquer outro objeto pelo código Python.

PyObject ***PyFunction_GetDefaults**(*PyObject* *op)

Retorna valor: Referência emprestada. Retorna os valores-padrão de argumentos do objeto função *op*. Pode ser uma tupla de argumentos ou `NULL`.

int **PyFunction_SetDefaults**(*PyObject* *op, *PyObject* *defaults)

Define os valores-padrão dos argumentos do objeto função *op*. *defaults* deve ser `Py_None` ou uma tupla.

Levanta `SystemError` e retorna `-1` em caso de falha.

void **PyFunction_SetVectorcall**(*PyFunctionObject* *func, *vectorcallfunc* vectorcall)

Define o campo *vectorcall* de um objeto função *func*.

Atenção: extensões que usam essa API devem preservar o comportamento inalterado (padrão) de uma função *vectorcall*!

Adicionado na versão 3.12.

PyObject ***PyFunction_GetClosure**(*PyObject* *op)

Retorna valor: Referência emprestada. Retorna o fechamento associado ao objeto função *op*. Pode ser `NULL` ou uma tupla de objetos célula.

int **PyFunction_SetClosure**(*PyObject* *op, *PyObject* *closure)

Define o fechamento associado ao objeto função *op*. *closure* deve ser `Py_None` ou uma tupla de objetos de célula.

Levanta `SystemError` e retorna `-1` em caso de falha.

PyObject ***PyFunction_GetAnnotations**(*PyObject* *op)

Retorna valor: Referência emprestada. Retorna as anotações do objeto função *op*. Este pode ser um dicionário mutável ou `NULL`.

int **PyFunction_SetAnnotations**(*PyObject* *op, *PyObject* *annotations)

Define as anotações para o objeto função *op*. *annotations* deve ser um dicionário ou `Py_None`.

Levanta `SystemError` e retorna `-1` em caso de falha.

int **PyFunction_AddWatcher**(*PyFunction_WatchCallback* callback)

Registra *callback* como uma sentinela de função para o interpretador atual. Retorna um ID que pode ser passado para *PyFunction_ClearWatcher()*. Em caso de erro (por exemplo, sem novos IDs de sentinelas disponíveis), retorna `-1` e define uma exceção.

Adicionado na versão 3.12.

int **PyFunction_ClearWatcher**(int watcher_id)

Cancela o registro da sentinela identificada pelo *watcher_id* retornado por *PyFunction_AddWatcher()* para o interpretador atual. Retorna `0` em caso de sucesso, ou `-1` e define uma exceção em caso de erro (por exemplo, ao receber um *watcher_id* desconhecido.)

Adicionado na versão 3.12.

type **PyFunction_WatchEvent**

Enumeration of possible function watcher events:

- `PyFunction_EVENT_CREATE`
- `PyFunction_EVENT_DESTROY`
- `PyFunction_EVENT_MODIFY_CODE`
- `PyFunction_EVENT_MODIFY_DEFAULTS`
- `PyFunction_EVENT_MODIFY_KWDEFAULTS`

Adicionado na versão 3.12.

typedef int (***PyFunction_WatchCallback**)(*PyFunction_WatchEvent* event, *PyFunctionObject* *func, *PyObject* *new_value)

Tipo de uma função de retorno de sentinela de função.

Se *event* for `PyFunction_EVENT_CREATE` ou `PyFunction_EVENT_DESTROY`, *new_value* será `NULL`. Senão, *new_value* portará uma *referência emprestada* ao novo valor prestes a ser guardado em *func* para o atributo que está sendo modificado.

A função de retorno poderá somente inspecionar, e não modificar *func*. Caso contrário, poderíamos ter efeitos imprevisíveis, incluindo recursão infinita.

Se *event* for `PyFunction_EVENT_CREATE`, a função de retorno será invocada após *func* ter sido completamente inicializada. Caso contrário, a função de retorno será invocada antes de modificar *func*, então o estado anterior de *func* poderá ser inspecionado. O ambiente de execução pode otimizar a criação de objetos função, quando possível, ao ignorá-las. Nesses casos, nenhum evento será emitido. Apesar de decisões de otimização criarem diferenças de comportamento em tempo de execução, elas não mudam a semântica do código Python sendo executado.

Se *event* for `PyFunction_EVENT_DESTROY`, então obter uma referência dentro da função de retorno para a função prestes a ser destruída irá revivê-la, impedindo que esta função seja liberada nesse tempo. Quando o objeto revivido for destruído, quaisquer funções de retorno sentinelas ativas nesse momento poderão ser chamadas novamente.

Se a função de retorno definir uma exceção, ela deverá retornar `-1`. Essa exceção será exibida como uma exceção não levantável usando *`PyErr_WriteUnraisable()`*. Caso contrário, deverá retornar `0`.

É possível que já exista uma exceção pendente definida na entrada da função de retorno. Nesse caso, a função de retorno deve retornar `0` com a mesma exceção ainda definida. Isso significa que a função de retorno não pode chamar nenhuma outra API que possa definir uma exceção, a menos que salve e limpe o estado da exceção primeiro e restaure a exceção antes de retornar.

Adicionado na versão 3.12.

8.5.2 Objetos de Método de Instância

Um método de instância é um invólucro para um *`PyCFunction`* e a nova maneira de vincular um *`PyCFunction`* a um objeto classe. Ele substitui a chamada anterior `PyMethod_New(func, NULL, class)`.

PyTypeObject **PyInstanceMethod_Type**

Esta instância de *`PyTypeObject`* representa o tipo de método de instância Python. Não é exposto a programas Python.

int **PyInstanceMethod_Check**(*PyObject* *o)

Retorna verdadeiro se *o* é um objeto de método de instância (tem tipo *`PyInstanceMethod_Type`*). O parâmetro não deve ser `NULL`. Esta função sempre tem sucesso.

PyObject ***PyInstanceMethod_New**(*PyObject* *func)

Retorna valor: Nova referência. Retorna um novo objeto de método de instância, com *func* sendo qualquer objeto chamável. *func* é a função que será chamada quando o método de instância for chamado.

PyObject ***PyInstanceMethod_Function**(*PyObject* *im)

Retorna valor: Referência emprestada. Retorna o objeto função associado ao método de instância *im*.

PyObject ***PyInstanceMethod_GET_FUNCTION**(*PyObject* *im)

Retorna valor: Referência emprestada. Versão macro de *`PyInstanceMethod_Function()`* que evita a verificação de erros.

8.5.3 Objetos método

Métodos são objetos função vinculados. Os métodos são sempre associados a uma instância de uma classe definida pelo usuário. Métodos não vinculados (métodos vinculados a um objeto de classe) não estão mais disponíveis.

PyTypeObject **PyMethod_Type**

Esta instância de *PyTypeObject* representa o tipo de método Python. Isso é exposto a programas Python como `types.MethodType`.

int **PyMethod_Check**(*PyObject* *o)

Retorna verdadeiro se *o* é um objeto de método (tem tipo *PyMethod_Type*). O parâmetro não deve ser `NULL`. Esta função sempre tem sucesso.

PyObject ***PyMethod_New**(*PyObject* *func, *PyObject* *self)

Retorna valor: Nova referência. Retorna um novo objeto de método, com *func* sendo qualquer objeto chamável e *self* a instância à qual o método deve ser vinculado. *func* é a função que será chamada quando o método for chamado. *self* não deve ser `NULL`.

PyObject ***PyMethod_Function**(*PyObject* *meth)

Retorna valor: Referência emprestada. Retorna o objeto função associado ao método *meth*.

PyObject ***PyMethod_GET_FUNCTION**(*PyObject* *meth)

Retorna valor: Referência emprestada. Versão macro de *PyMethod_Function()* que evita a verificação de erros.

PyObject ***PyMethod_Self**(*PyObject* *meth)

Retorna valor: Referência emprestada. Retorna a instância associada com o método *meth*.

PyObject ***PyMethod_GET_SELF**(*PyObject* *meth)

Retorna valor: Referência emprestada. Versão macro de *PyMethod_Self()* que evita a verificação de erros.

8.5.4 Objeto célula

Objetos "cell" são usados para implementar variáveis referenciadas por múltiplos escopos. Para cada variável, um objeto 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.

type **PyCellObject**

A estrutura C usada para objetos célula.

PyTypeObject **PyCell_Type**

O objeto de tipo correspondente aos objetos célula.

int **PyCell_Check**(*PyObject* *ob)

Retorna verdadeiro se *ob* for um objeto célula; *ob* não deve ser `NULL`. Esta função sempre tem sucesso.

PyObject ***PyCell_New**(*PyObject* *ob)

Retorna valor: Nova referência. Cria e retorna um novo objeto célula contendo o valor *ob*. O parâmetro pode ser `NULL`.

PyObject ***PyCell_Get**(*PyObject* *cell)

Retorna valor: Nova referência. Retorna o conteúdo da célula *cell*, que pode ser `NULL`. Se *cell* não for um objeto célula, retorna `NULL` com um conjunto de exceções.

PyObject ***PyCell_GET**(*PyObject* *cell)

Retorna valor: Referência emprestada. Retorna o conteúdo da célula *cell*, mas sem verificar se *cell* não é `NULL` e um objeto célula.

int **PyCell_Set**(*PyObject* *cell, *PyObject* *value)

Define o conteúdo do objeto da célula *cell* como *value*. Isso libera a referência a qualquer conteúdo atual da célula. *value* pode ser `NULL`. *cell* não pode ser `NULL`.

Em caso de sucesso, retorna `0`. Se *cell* não for um objeto célula, define uma exceção e retorna `-1`.

void **PyCell_SET**(*PyObject* *cell, *PyObject* *value)

Define o valor do objeto da célula *cell* como *value*. Nenhuma contagem de referência é ajustada e nenhuma verificação é feita quanto à segurança; *cell* não pode ser `NULL` e deve ser um objeto célula.

8.5.5 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.

type **PyCodeObject**

A estrutura C dos objetos usados para descrever objetos código. Os campos deste tipo estão sujeitos a alterações a qualquer momento.

PyTypeObject **PyCode_Type**

Esta é uma instância de `PyTypeObject` representando o objeto código Python.

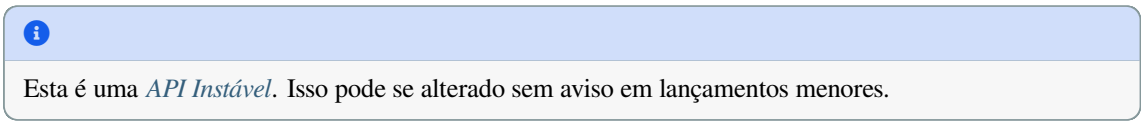
int **PyCode_Check**(*PyObject* *co)

Retorna verdadeiro se *co* for um objeto código. Esta função sempre tem sucesso.

Py_ssize_t **PyCode_GetNumFree**(*PyCodeObject* *co)

Retorna o número de *variáveis livres (de clausura)* em um objeto código.

int **PyUnstable_Code_GetFirstFree**(*PyCodeObject* *co)

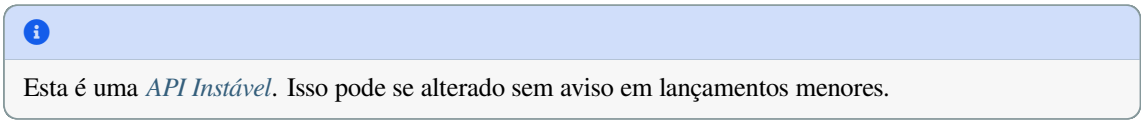


Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Retorna a posição da primeira *variável livre (de clausura)* em um objeto código.

Alterado na versão 3.13: Renomeado de `PyCode_GetFirstFree` como parte da *API C Instável*. O nome antigo foi descontinuado, mas continuará disponível até que a assinatura mude novamente.

PyCodeObject ***PyUnstable_Code_New**(int argcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)



Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Devolve um novo objeto código. Se precisar de um objeto código vazio para criar um quadro, use `PyCode_NewEmpty()`

Como a definição de bytecode muda constantemente, chamar `PyUnstable_Code_New()` diretamente pode vinculá-lo a uma versão de Python específica.

Os vários argumentos desta função são inter-dependentes de maneiras complexas, significando que mesmo alterações discretas de valor tem chances de resultar em execução incorreta ou erros fatais de VM. Tenha extremo cuidado ao usar esta função.

Alterado na versão 3.11: Adicionou os parâmetros `qualname` e `exceptiontable`

Alterado na versão 3.12: Renomeado de `PyCode_New` como parte da *API C Instável*. O nome antigo foi descontinuado, mas continuará disponível até que a assinatura mude novamente.

PyCodeObject ***PyUnstable_Code_NewWithPosOnlyArgs**(int argcount, int posonlyargcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)

Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Similar a `PyUnstable_Code_New()`, mas com um "posonlyargcount" extra para argumentos somente-posicionais. As mesmas ressalvas que se aplicam a `PyUnstable_Code_New` também se aplicam a essa função.

Adicionado na versão 3.8: Como `PyCode_NewWithPosOnlyArgs`

Alterado na versão 3.11: Adicionados os parâmetros `qualname` e `exceptiontable`

Alterado na versão 3.12: Renomeado para `PyUnstable_Code_NewWithPosOnlyArgs`. O nome antigo foi descontinuado, mas continuará disponível até que a assinatura mude novamente.

PyCodeObject ***PyCode_NewEmpty**(const char *filename, const char *funcname, int firstlineno)

Retorna valor: Nova referência. Retorna um novo objeto código vazio com o nome de arquivo, nome da função e número da primeira linha especificados. O objeto código resultante irá levantar uma `Exception` se executado.

int **PyCode_Addr2Line**(*PyCodeObject* *co, int byte_offset)

Retorna o número da linha da instrução que ocorre em ou antes de `byte_offset` e termina depois disso. Se você só precisa do número da linha de um quadro, use `PyFrame_GetLineNumber()`.

Para iterar eficientemente sobre os números de linha em um objeto código, use **a API descrita na PEP 626** .

int **PyCode_Addr2Location**(*PyObject* *co, int byte_offset, int *start_line, int *start_column, int *end_line, int *end_column)

Define os ponteiros `int` passados para a linha do código-fonte e os números da coluna para a instrução em `byte_offset`. Define o valor como `0` quando as informações não estão disponíveis para nenhum elemento em particular.

Retorna `1` se a função for bem-sucedida e 0 caso contrário.

Adicionado na versão 3.11.

PyObject ***PyCode_GetCode**(*PyCodeObject* *co)

Equivalente ao código Python `getattr(co, 'co_code')`. Retorna uma referência forte a um `PyBytesObject` representando o bytecode em um objeto código. Em caso de erro, `NULL` é retornado e uma exceção é levantada.

Este `PyBytesObject` pode ser criado sob demanda pelo interpretador e não representa necessariamente o bytecode realmente executado pelo CPython. O caso de uso primário para esta função são depuradores e criadores de perfil.

Adicionado na versão 3.11.

PyObject ***PyCode_GetVarnames**(*PyCodeObject* *co)

Equivalente ao código Python `getattr(co, 'co_varnames')`. Retorna uma nova referência a um `PyTupleObject` contendo os nomes das variáveis locais. Em caso de erro, `NULL` é retornado e uma exceção é levantada.

Adicionado na versão 3.11.

PyObject ***PyCode_GetCellvars**(*PyCodeObject* *co)

Equivalente ao código Python `getattr(co, 'co_cellvars')`. Retorna uma nova referência a um *PyTupleObject* contendo os nomes das variáveis locais referenciadas por funções aninhadas. Em caso de erro, `NULL` é retornado e uma exceção é levantada.

Adicionado na versão 3.11.

PyObject ***PyCode_GetFreevars**(*PyCodeObject* *co)

Equivalente ao código Python `getattr(co, 'co_freevars')`. Retorna uma nova referência a um *PyTupleObject* contendo os nomes das *variáveis livres (de clausura)*. Em caso de erro, `NULL` é retornado e uma exceção é levantada.

Adicionado na versão 3.11.

int **PyCode_AddWatcher**(*PyCode_WatchCallback* callback)

Registra *callback* como um observador do objeto código para o interpretador atual. Devolve um ID que pode ser passado para *PyCode_ClearWatcher()*. Em caso de erro (por exemplo, não há IDs de observadores disponíveis), devolve `-1` e define uma exceção.

Adicionado na versão 3.12.

int **PyCode_ClearWatcher**(int watcher_id)

Libera o observador identificado por *watcher_id* anteriormente retornado por *PyCode_AddWatcher()* para o interpretador atual. Retorna `0` em caso de sucesso ou `-1` em caso de erro e levanta uma exceção (ex., se o *watcher_id* dado não foi registrado.)

Adicionado na versão 3.12.

type **PyCodeEvent**

Enumeração dos possíveis eventos de observador do objeto código: `PY_CODE_EVENT_CREATE` - `PY_CODE_EVENT_DESTROY`

Adicionado na versão 3.12.

typedef int (***PyCode_WatchCallback**)(*PyCodeEvent* event, *PyCodeObject* *co)

Tipo de uma função de callback de observador de objeto código.

Se *evento* é `PY_CODE_EVENT_CREATE`, então a função de retorno é invocada após *co'ter sido completamente inicializado. Senão, a função de retorno é invocada antes que a destruição de 'co* ocorra, para que o estado anterior de *co* possa ser inspecionado.

Se *evento* for `PY_CODE_EVENT_DESTROY`, obter uma referência para a função de retorno do objeto-a-ser--destruído irá reativá-lo e impedirá que o objeto seja liberado. Quando o objeto reativado é posteriormente destruído, qualquer observador de funções de retorno ativos naquele momento serão chamados novamente.

Usuários desta API não devem depender de detalhes internos de implementação em tempo de execução. Tais detalhes podem incluir, mas não estão limitados a: o ordem e o momento exatos da criação e destruição de objetos código. Enquanto alterações nestes detalhes podem resultar em diferenças que são visíveis para os observadores (incluindo se uma função de retorno é chamada ou não), isso não muda a semântica do código Python executado.

Se a função de retorno definir uma exceção, ela deverá retornar `-1`; essa exceção será impressa como uma exceção não reprovável usando *PyErr_WriteUnraisable()*. Caso contrário, deverá retornar `0`.

É possível que já exista uma exceção pendente definida na entrada da função de retorno. Nesse caso, a função de retorno deve retornar `0` com a mesma exceção ainda definida. Isso significa que a função de retorno não pode chamar nenhuma outra API que possa definir uma exceção, a menos que salve e limpe o estado da exceção primeiro e o restaure antes de retornar.

Adicionado na versão 3.12.

8.5.6 Informação adicional

Para suportar extensões de baixo nível de avaliação de quadro (frame), tais como compiladores "just-in-time", é possível anexar dados arbitrários adicionais a objetos código.

Estas funções são parte da camada instável da API C: Essa funcionalidade é um detalhe de implementação do CPython, e a API pode mudar sem avisos de descontinuidade.

Py_ssize_t **PyUnstable_Eval_RequestCodeExtraIndex**(*freefunc* free)

> Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Retorna um novo e opaco valor de índice usado para adicionar dados a objetos código.

Geralmente, você chama esta função apenas uma vez (por interpretador) e usa o resultado com `PyCode_GetExtra` e `PyCode_SetExtra` para manipular dados em objetos código individuais.

Se *free* não for `NULL`: quando o objeto código é desalocado, *free* será chamado em dados não-`NULL` armazenados sob o novo índice. Use `Py_DecRef()` quando armazenar `PyObject`.

Adicionado na versão 3.6: como `_PyEval_RequestCodeExtraIndex`

Alterado na versão 3.12: Renomeado para `PyUnstable_Eval_RequestCodeExtraIndex`. O nome antigo nome privado foi descontinuado, mas continuará disponível até a mudança da API.

int **PyUnstable_Code_GetExtra**(*PyObject* *code, *Py_ssize_t* index, void **extra)

> Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Define *extra* para os dados adicionais armazenados sob o novo índice dado. Retorna 0 em caso de sucesso. Define uma exceção e retorna -1 em caso de erro.

Se nenhum dado foi determinado sob o índice, define *extra* como `NULL` e retorna 0 sem definir nenhuma exceção.

Adicionado na versão 3.6: como `_PyCode_GetExtra`

Alterado na versão 3.12: Renomeado para `PyUnstable_Code_GetExtra`. O nome antigo privado foi descontinuado, mas continuará disponível até a mudança da API.

int **PyUnstable_Code_SetExtra**(*PyObject* *code, *Py_ssize_t* index, void *extra)

> Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Define os dados extras armazenados sob o índice dado a *extra*. Retorna 0 em caso de sucesso. Define uma exceção e retorna -1 em caso de erro.

Adicionado na versão 3.6: como `_PyCode_SetExtra`

Alterado na versão 3.12: Renomeado para `PyUnstable_Code_SetExtra`. O nome antigo privado foi descontinuado, mas continuará disponível até a mudança da API.

8.6 Outros Objetos

8.6.1 Objetos arquivos

Essas APIs são uma emulação mínima da API C do Python 2 para objetos arquivo embutidos, que costumavam depender do suporte de E/S em buffer (`FILE*`) da biblioteca C padrão. No Python 3, arquivos e streams usam o novo módulo `io`, que define várias camadas sobre a E/S sem buffer de baixo nível do sistema operacional. As funções descritas a seguir são invólucros de conveniência para o C sobre essas novas APIs e são destinadas principalmente para relatórios de erros internos no interpretador; código de terceiros é recomendado para acessar as APIs de `io`.

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

Retorna valor: Nova referência. Parte da ABI Estável. Cria um objeto arquivo Python a partir do descritor de arquivo de um arquivo já aberto *fd*. Os argumentos *name*, *encoding*, *errors* and *newline* podem ser `NULL` para usar os padrões; *buffering* pode ser *-1* para usar o padrão. *name* é ignorado e mantido para compatibilidade com versões anteriores. Retorna `NULL` em caso de falha. Para uma descrição mais abrangente dos argumentos, consulte a documentação da função `io.open()`.

> **Aviso**
>
> Como os streams do Python têm sua própria camada de buffer, combiná-los com os descritores de arquivo no nível do sistema operacional pode produzir vários problemas (como ordenação inesperada de dados).

Alterado na versão 3.2: Ignora atributo *name*.

int **PyObject_AsFileDescriptor**(*PyObject* *p)

Parte da ABI Estável. Retorna o descritor de arquivo associado a *p* como um `int`. Se o objeto for um inteiro, seu valor será retornado. Caso contrário, o método `fileno()` do objeto será chamado se existir; o método deve retornar um inteiro, que é retornado como o valor do descritor de arquivo. Define uma exceção e retorna `-1` em caso de falha.

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

Retorna valor: Nova referência. Parte da ABI Estável. Equivalente a `p.readline([n])`, esta função lê uma linha do objeto *p*. *p* pode ser um objeto arquivo ou qualquer objeto com um método `readline()`. Se *n* for `0`, exatamente uma linha é lida, independentemente do comprimento da linha. Se *n* for maior que `0`, não mais do que *n* bytes serão lidos do arquivo; uma linha parcial pode ser retornada. Em ambos os casos, uma string vazia é retornada se o final do arquivo for alcançado imediatamente. Se *n* for menor que `0`, entretanto, uma linha é lida independentemente do comprimento, mas `EOFError` é levantada se o final do arquivo for alcançado imediatamente.

int **PyFile_SetOpenCodeHook**(*Py_OpenCodeHookFunction* handler)

Substitui o comportamento normal de `io.open_code()` para passar seu parâmetro por meio do manipulador fornecido.

O *handler* é uma função do tipo:

typedef *PyObject* *(***Py_OpenCodeHookFunction**)(*PyObject**, void*)

Equivalente de `PyObject *(*)(PyObject *path, void *userData)`, sendo *path* garantido como sendo `PyUnicodeObject`.

O ponteiro *userData* é passado para a função de gancho. Como as funções de gancho podem ser chamadas de diferentes tempos de execução, esse ponteiro não deve se referir diretamente ao estado do Python.

Como este gancho é usado intencionalmente durante a importação, evite importar novos módulos durante sua execução, a menos que eles estejam congelados ou disponíveis em `sys.modules`.

Uma vez que um gancho foi definido, ele não pode ser removido ou substituído, e chamadas posteriores para `PyFile_SetOpenCodeHook()` irão falhar. Em caso de falha, a função retorna -1 e define uma exceção se o interpretador foi inicializado.

É seguro chamar esta função antes `Py_Initialize()`.

Levanta um evento de auditoria `setopencodehook` sem argumentos.

Adicionado na versão 3.8.

int **PyFile_WriteObject**(*PyObject* *obj, *PyObject* *p, int flags)

Parte da ABI Estável. Escreve o objeto *obj* no objeto arquivo *p*. O único sinalizador suportado para *flags* é `Py_PRINT_RAW`; se fornecido, o `str()` do objeto é escrito em vez de `repr()`. Retorna `0` em caso de sucesso ou `-1` em caso de falha; a exceção apropriada será definida.

int **PyFile_WriteString**(const char *s, *PyObject* *p)

Parte da ABI Estável. Escreve a string *s* no objeto arquivo *p*. Retorna `0` em caso de sucesso ou `-1` em caso de falha; a exceção apropriada será definida.

8.6.2 Objetos do Módulo

PyTypeObject **PyModule_Type**

Parte da ABI Estável. Esta instância de `PyTypeObject` representa o tipo de módulo Python. Isso é exposto a programas Python como `types.ModuleType`.

int **PyModule_Check**(*PyObject* *p)

Retorna true se *p* for um objeto de módulo ou um subtipo de um objeto de módulo. Esta função sempre é bem-sucedida.

int **PyModule_CheckExact**(*PyObject* *p)

Retorna true se *p* for um objeto de módulo, mas não um subtipo de `PyModule_Type`. Essa função é sempre bem-sucedida.

PyObject ***PyModule_NewObject**(*PyObject* *name)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.7.* Retorna um novo objeto de módulo com `module.__name__` definido como *name*. Os atributos `__name__`, `__doc__`, `__package__` e `__loader__` do módulo são preenchidos (todos, exceto `__name__`, são definidos como `None`). O chamador é responsável por definir um atributo `__file__`.

Retorna `NULL` com uma exceção definida em caso de erro.

Adicionado na versão 3.3.

Alterado na versão 3.4: `__package__` e `__loader__` agora estão definidos como `None`.

PyObject ***PyModule_New**(const char *name)

Retorna valor: Nova referência. Parte da ABI Estável. Semelhante a `PyModule_NewObject()`, mas o nome é uma string codificada em UTF-8 em vez de um objeto Unicode.

PyObject ***PyModule_GetDict**(*PyObject* *module)

Retorna valor: Referência emprestada. Parte da ABI Estável. Retorna o objeto dicionário que implementa o espaço de nomes de *module*; este objeto é o mesmo que o atributo `__dict__` do objeto de módulo. Se *module* não for um objeto de módulo (ou um subtipo de um objeto de módulo), `SystemError` é levantada e `NULL` é retornado.

É recomendado que as extensões usem outras funções `PyModule_*` e `PyObject_*` em vez de manipular diretamente o `__dict__` de um módulo.

PyObject ***PyModule_GetNameObject**(*PyObject* *module)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.7.* Retorna o valor `__name__` do *module*. Se o módulo não fornecer um, ou se não for uma string, `SystemError` é levantada e `NULL` é retornado.

Adicionado na versão 3.3.

const char ***PyModule_GetName**(*PyObject* *module)

Parte da ABI Estável. Semelhante a `PyModule_GetNameObject()` mas retorna o nome codificado em `'utf-8'`

void *__PyModule_GetState__(*PyObject* *module)

Parte da ABI Estável. Retorna o "estado" do módulo, ou seja, um ponteiro para o bloco de memória alocado no momento de criação do módulo, ou `NULL`. Ver *PyModuleDef.m_size*.

PyModuleDef *__PyModule_GetDef__(*PyObject* *module)

Parte da ABI Estável. Retorna um ponteiro para a estrutura *PyModuleDef* da qual o módulo foi criado, ou `NULL` se o módulo não foi criado de uma definição.

PyObject *__PyModule_GetFilenameObject__(*PyObject* *module)

Retorna valor: Nova referência. *Parte da* ABI Estável. Retorna o nome do arquivo do qual o *module* foi carregado usando o atributo `__file__` do *module*. Se não estiver definido, ou se não for uma string unicode, levanta `SystemError` e retorna `NULL`; Caso contrário, retorna uma referência a um objeto Unicode.

Adicionado na versão 3.2.

const char *__PyModule_GetFilename__(*PyObject* *module)

Parte da ABI Estável. Semelhante a *PyModule_GetFilenameObject()* mas retorna o nome do arquivo codificado em 'utf-8'.

Descontinuado desde a versão 3.2: *PyModule_GetFilename()*: levanta `UnicodeEncodeError` quando há nomes de arquivos não codificáveis, use *PyModule_GetFilenameObject()*.

Inicializando módulos C

Objetos de módulos são geralmente criados a partir de módulos de extensão (bibliotecas compartilhadas que exportam uma função de inicialização), ou módulos compilados (onde a função de inicialização é adicionada usando *PyImport_AppendInittab()*). Ver building ou extending-with-embedding para mais detalhes.

A função de inicialização pode passar uma instância de definição de módulo para *PyModule_Create()* e retornar o objeto de módulo resultante ou solicitar "inicialização multifásica" retornando a própria estrutura de definição.

type __PyModuleDef__

Parte da ABI Estável *(incluindo todos os membros).* A estrutura de definição de módulo, que contém todas as informações necessária para criar um objeto de módulo. Geralmente, há apenas uma variável inicializada estaticamente desse tipo para cada módulo.

PyModuleDef_Base **m_base**

Sempre inicializa este membro para `PyModuleDef_HEAD_INIT`.

const char ***m_name**

Nome para o novo módulo.

const char ***m_doc**

Docstring para o módulo; geralmente uma variável docstring criada com *PyDoc_STRVAR* é usada.

Py_ssize_t **m_size**

O estado do módulo pode ser mantido em uma área de memória por módulo que pode ser recuperada com *PyModule_GetState()*, em vez de em globais estáticos. Isso torna os módulos seguros para uso em vários subinterpretadores.

Esta área de memória é alocada com base em *m_size* na criação do módulo e liberada quando o objeto do módulo é desalocado, após a função *m_free* ter sido chamada, se presente.

Definir `m_size` como `-1` significa que o módulo não oferece suporte a subinterpretadores, porque ele tem estado global.

Defini-lo como um valor não negativo significa que o módulo pode ser reinicializado e especifica a quantidade adicional de memória necessária para seu estado. `m_size` não negativo é necessário para inicialização multifásica.

Ver **PEP 3121** para mais detalhes.

PyMethodDef ***m_methods**

Um ponteiro para uma tabela de funções de nível de módulo, descritas por valores *PyMethodDef*. Pode ser NULL se nenhuma função estiver presente.

PyModuleDef_Slot ***m_slots**

Uma matriz de definições de slot para inicialização multifásica, terminada por uma entrada {0, NULL}. Ao usar inicialização monofásica, *m_slots* deve ser NULL.

Alterado na versão 3.5: Antes da versão 3.5, esse membro era sempre definido como NULL e era definido como:

inquiry **m_reload**

traverseproc **m_traverse**

Uma função de travessia para chamar durante a travessia do GC do objeto do módulo, ou NULL se não for necessário.

Esta função não é mais chamada se o estado do módulo foi solicitado, mas ainda não está alocado. Este é o caso imediatamente após o módulo ser criado e antes de o módulo ser executado (função *Py_mod_exec*). Mais precisamente, esta função não é chamada se *m_size* for maior que 0 e o estado do módulo (como retornado por *PyModule_GetState()*) for NULL.

Alterado na versão 3.9: Não é mais chamado antes que o estado do módulo seja alocado.

inquiry **m_clear**

Uma função de limpeza para chamar durante a limpeza do GC do objeto do módulo, ou NULL se não for necessário.

Esta função não é mais chamada se o estado do módulo foi solicitado, mas ainda não está alocado. Este é o caso imediatamente após o módulo ser criado e antes de o módulo ser executado (função *Py_mod_exec*). Mais precisamente, esta função não é chamada se *m_size* for maior que 0 e o estado do módulo (como retornado por *PyModule_GetState()*) for NULL.

Assim como *PyTypeObject.tp_clear*, esta função não é *sempre* chamada antes de um módulo ser desalocado. Por exemplo, quando a contagem de referências é suficiente para determinar que um objeto não é mais usado, o coletor de lixo cíclico não é envolvido e *m_free* é chamado diretamente.

Alterado na versão 3.9: Não é mais chamado antes que o estado do módulo seja alocado.

freefunc **m_free**

Uma função para ser chamada durante a desalocação do objeto do módulo, ou NULL se não for necessário.

Esta função não é mais chamada se o estado do módulo foi solicitado, mas ainda não está alocado. Este é o caso imediatamente após o módulo ser criado e antes de o módulo ser executado (função *Py_mod_exec*). Mais precisamente, esta função não é chamada se *m_size* for maior que 0 e o estado do módulo (como retornado por *PyModule_GetState()*) for NULL.

Alterado na versão 3.9: Não é mais chamado antes que o estado do módulo seja alocado.

inicialização de fase única

A função de inicialização do módulo pode criar e retornar o objeto do módulo diretamente. Isso é chamado de "inicialização de fase única" e usa uma das duas funções de criação de módulo a seguir:

PyObject ***PyModule_Create**(*PyModuleDef* *def)

Retorna valor: Nova referência. Cria um novo objeto de módulo, dada a definição em *def*. Isso se comporta como *PyModule_Create2()* com *module_api_version* definido como PYTHON_API_VERSION

PyObject ***PyModule_Create2**(*PyModuleDef* *def, int module_api_version)

Retorna valor: Nova referência. Parte da ABI Estável. Create a new module object, given the definition in *def*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a RuntimeWarning is emitted.

Retorna NULL com uma exceção definida em caso de erro.

> **Nota**
>
> A maioria dos usos dessa função deve ser feita com `PyModule_Create()`; use-o apenas se tiver certeza de que precisa.

Before it is returned from in the initialization function, the resulting module object is typically populated using functions like `PyModule_AddObjectRef()`.

Inicialização multifásica

An alternate way to specify extensions is to request "multi-phase initialization". Extension modules created this way behave more like Python modules: the initialization is split between the *creation phase*, when the module object is created, and the *execution phase*, when it is populated. The distinction is similar to the `__new__()` and `__init__()` methods of classes.

Unlike modules created using single-phase initialization, these modules are not singletons: if the *sys.modules* entry is removed and the module is re-imported, a new module object is created, and the old module is subject to normal garbage collection – as with Python modules. By default, multiple modules created from the same definition should be independent: changes to one should not affect the others. This means that all state should be specific to the module object (using e.g. using `PyModule_GetState()`), or its contents (such as the module's `__dict__` or individual classes created with `PyType_FromSpec()`).

All modules created using multi-phase initialization are expected to support *sub-interpreters*. Making sure multiple modules are independent is typically enough to achieve this.

To request multi-phase initialization, the initialization function (PyInit_modulename) returns a `PyModuleDef` instance with non-empty `m_slots`. Before it is returned, the `PyModuleDef` instance must be initialized with the following function:

PyObject ***PyModuleDef_Init**(*PyModuleDef* *def)

Retorna valor: Referência emprestada. Parte da ABI Estável *desde a versão 3.5.* Garante que uma definição de módulo é um objeto Python devidamente inicializado que reporta corretamente seu tipo e contagem de referências.

Returns *def* cast to `PyObject*`, or `NULL` if an error occurred.

Adicionado na versão 3.5.

The *m_slots* member of the module definition must point to an array of `PyModuleDef_Slot` structures:

type **PyModuleDef_Slot**

int **slot**

Um ID de lot, escolhido a partir dos valores disponíveis explicados abaixo.

void ***value**

Valor do slot, cujo significado depende do ID do slot.

Adicionado na versão 3.5.

The *m_slots* array must be terminated by a slot with id 0.

Os tipos de slot disponíveis são:

Py_mod_create

Specifies a function that is called to create the module object itself. The *value* pointer of this slot must point to a function of the signature:

PyObject ***create_module**(*PyObject* *spec, *PyModuleDef* *def)

The function receives a `ModuleSpec` instance, as defined in **PEP 451**, and the module definition. It should return a new module object, or set an error and return `NULL`.

This function should be kept minimal. In particular, it should not call arbitrary Python code, as trying to import the same module again may result in an infinite loop.

Múltiplos slots `Py_mod_create` podem não estar especificados em uma definição de módulo.

If `Py_mod_create` is not specified, the import machinery will create a normal module object using *PyModule_New()*. The name is taken from *spec*, not the definition, to allow extension modules to dynamically adjust to their place in the module hierarchy and be imported under different names through symlinks, all while sharing a single module definition.

There is no requirement for the returned object to be an instance of *PyModule_Type*. Any type can be used, as long as it supports setting and getting import-related attributes. However, only `PyModule_Type` instances may be returned if the `PyModuleDef` has non-`NULL` `m_traverse`, `m_clear`, `m_free`; non-zero `m_size`; or slots other than `Py_mod_create`.

Py_mod_exec

Specifies a function that is called to *execute* the module. This is equivalent to executing the code of a Python module: typically, this function adds classes and constants to the module. The signature of the function is:

int **exec_module**(*PyObject* *module)

Se vários slots `Py_mod_exec` forem especificados, eles serão processados na ordem em que aparecem no vetor *m_slots*.

Py_mod_multiple_interpreters

Specifies one of the following values:

Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED

The module does not support being imported in subinterpreters.

Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED

The module supports being imported in subinterpreters, but only when they share the main interpreter's GIL. (See isolating-extensions-howto.)

Py_MOD_PER_INTERPRETER_GIL_SUPPORTED

The module supports being imported in subinterpreters, even when they have their own GIL. (See isolating-extensions-howto.)

This slot determines whether or not importing this module in a subinterpreter will fail.

Multiple `Py_mod_multiple_interpreters` slots may not be specified in one module definition.

If `Py_mod_multiple_interpreters` is not specified, the import machinery defaults to `Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED`.

Adicionado na versão 3.12.

Py_mod_gil

Specifies one of the following values:

Py_MOD_GIL_USED

The module depends on the presence of the global interpreter lock (GIL), and may access global state without synchronization.

Py_MOD_GIL_NOT_USED

The module is safe to run without an active GIL.

This slot is ignored by Python builds not configured with `--disable-gil`. Otherwise, it determines whether or not importing this module will cause the GIL to be automatically enabled. See whatsnew313-free-threaded--cpython for more detail.

Multiple `Py_mod_gil` slots may not be specified in one module definition.

If `Py_mod_gil` is not specified, the import machinery defaults to `Py_MOD_GIL_USED`.

Adicionado na versão 3.13.

Ver **PEP 489** para obter mais detalhes sobre a inicialização multifásica.

Funções de criação de módulo de baixo nível

The following functions are called under the hood when using multi-phase initialization. They can be used directly, for example when creating module objects dynamically. Note that both `PyModule_FromDefAndSpec` and `PyModule_ExecDef` must be called to fully initialize a module.

PyObject ***PyModule_FromDefAndSpec**(*PyModuleDef* *def, *PyObject* *spec)

Retorna valor: Nova referência. Create a new module object, given the definition in *def* and the ModuleSpec *spec*. This behaves like *PyModule_FromDefAndSpec2()* with *module_api_version* set to `PYTHON_API_VERSION`.

Adicionado na versão 3.5.

PyObject ***PyModule_FromDefAndSpec2**(*PyModuleDef* *def, *PyObject* *spec, int module_api_version)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.7.* Create a new module object, given the definition in *def* and the ModuleSpec *spec*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

Retorna `NULL` com uma exceção definida em caso de erro.

> **Nota**
>
> Most uses of this function should be using *PyModule_FromDefAndSpec()* instead; only use this if you are sure you need it.

Adicionado na versão 3.5.

int **PyModule_ExecDef**(*PyObject* *module, *PyModuleDef* *def)

Parte da ABI Estável *desde a versão 3.7.* Process any execution slots (*Py_mod_exec*) given in *def*.

Adicionado na versão 3.5.

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

Parte da ABI Estável *desde a versão 3.7.* Set the docstring for *module* to *docstring*. This function is called automatically when creating a module from `PyModuleDef`, using either `PyModule_Create` or `PyModule_FromDefAndSpec`.

Adicionado na versão 3.5.

int **PyModule_AddFunctions**(*PyObject* *module, *PyMethodDef* *functions)

Parte da ABI Estável *desde a versão 3.7.* Add the functions from the `NULL` terminated *functions* array to *module*. Refer to the *PyMethodDef* documentation for details on individual entries (due to the lack of a shared module namespace, module level "functions" implemented in C typically receive the module as their first parameter, making them similar to instance methods on Python classes). This function is called automatically when creating a module from `PyModuleDef`, using either `PyModule_Create` or `PyModule_FromDefAndSpec`.

Adicionado na versão 3.5.

Support functions

The module initialization function (if using single phase initialization) or a function called from a module execution slot (if using multi-phase initialization), can use the following functions to help initialize the module state:

int **PyModule_AddObjectRef**(*PyObject* *module, const char *name, *PyObject* *value)

Parte da ABI Estável *desde a versão 3.10.* Add an object to *module* as *name*. This is a convenience function which can be used from the module's initialization function.

On success, return `0`. On error, raise an exception and return `-1`.

Exemplo de uso:

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    if (obj == NULL) {
        return -1;
    }
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_DECREF(obj);
    return res;
}
```

To be convenient, the function accepts `NULL` *value* with an exception set. In this case, return `-1` and just leave the raised exception unchanged.

O exemplo também pode ser escrito sem verificar explicitamente se *obj* é `NULL`:

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_XDECREF(obj);
    return res;
}
```

Note that `Py_XDECREF()` should be used instead of `Py_DECREF()` in this case, since *obj* can be `NULL`.

The number of different *name* strings passed to this function should be kept small, usually by only using statically allocated strings as *name*. For names that aren't known at compile time, prefer calling *PyUnicode_FromString()* and *PyObject_SetAttr()* directly. For more details, see *PyUnicode_InternFromString()*, which may be used internally to create a key object.

Adicionado na versão 3.10.

int **PyModule_Add**(*PyObject* *module, const char *name, *PyObject* *value)

Parte da ABI Estável *desde a versão 3.13.* Similar to *PyModule_AddObjectRef()*, but "steals" a reference to *value*. It can be called with a result of function that returns a new reference without bothering to check its result or even saving it to a variable.

Exemplo de uso:

```
if (PyModule_Add(module, "spam", PyBytes_FromString(value)) < 0) {
    goto error;
}
```

Adicionado na versão 3.13.

int **PyModule_AddObject**(*PyObject* *module, const char *name, *PyObject* *value)

Parte da ABI Estável. Similar to *PyModule_AddObjectRef()*, but steals a reference to *value* on success (if it returns `0`).

The new *PyModule_Add()* or *PyModule_AddObjectRef()* functions are recommended, since it is easy to introduce reference leaks by misusing the *PyModule_AddObject()* function.

Nota

Unlike other functions that steal references, `PyModule_AddObject()` only releases the reference to *value* **on success**.

This means that its return value must be checked, and calling code must `Py_XDECREF()` *value* manually on error.

Exemplo de uso:

```
PyObject *obj = PyBytes_FromString(value);
if (PyModule_AddObject(module, "spam", obj) < 0) {
    // If 'obj' is not NULL and PyModule_AddObject() failed,
    // 'obj' strong reference must be deleted with Py_XDECREF().
    // If 'obj' is NULL, Py_XDECREF() does nothing.
    Py_XDECREF(obj);
    goto error;
}
// PyModule_AddObject() stole a reference to obj:
// Py_XDECREF(obj) is not needed here.
```

Descontinuado desde a versão 3.13: `PyModule_AddObject()` is *soft deprecated*.

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

Parte da ABI Estável. Add an integer constant to *module* as *name*. This convenience function can be used from the module's initialization function. Return `-1` with an exception set on error, `0` on success.

This is a convenience function that calls `PyLong_FromLong()` and `PyModule_AddObjectRef()`; see their documentation for details.

int **PyModule_AddStringConstant**(*PyObject* *module, const char *name, const char *value)

Parte da ABI Estável. 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` with an exception set on error, `0` on success.

This is a convenience function that calls `PyUnicode_InternFromString()` and `PyModule_AddObjectRef()`; see their documentation for details.

PyModule_AddIntMacro(module, macro)

Add an int constant to *module*. The name and the value are taken from *macro*. For example `PyModule_AddIntMacro(module, AF_INET)` adds the int constant *AF_INET* with the value of *AF_INET* to *module*. Return `-1` with an exception set on error, `0` on success.

PyModule_AddStringMacro(module, macro)

Add a string constant to *module*.

int **PyModule_AddType**(*PyObject* *module, *PyTypeObject* *type)

Parte da ABI Estável *desde a versão 3.10.* Add a type object to *module*. The type object is finalized by calling internally `PyType_Ready()`. The name of the type object is taken from the last component of `tp_name` after dot. Return `-1` with an exception set on error, `0` on success.

Adicionado na versão 3.9.

int **PyUnstable_Module_SetGIL**(*PyObject* *module, void *gil)

Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Indicate that *module* does or does not support running without the global interpreter lock (GIL), using one of the values from `Py_mod_gil`. It must be called during *module*'s initialization function. If this function is not called during module initialization, the import machinery assumes the module does not support running without the GIL. This function is only available in Python builds configured with `--disable-gil`. Return `-1` with an exception set on error, `0` on success.

Adicionado na versão 3.13.

Pesquisa por módulos

Single-phase initialization creates singleton modules that can be looked up in the context of the current interpreter. This allows the module object to be retrieved later with only a reference to the module definition.

These functions will not work on modules created using multi-phase initialization, since multiple such modules can be created from a single definition.

PyObject ***PyState_FindModule**(*PyModuleDef* *def)

Retorna valor: Referência emprestada. Parte da ABI Estável. Returns the module object that was created from *def* for the current interpreter. This method requires that the module object has been attached to the interpreter state with *PyState_AddModule()* beforehand. In case the corresponding module object is not found or has not been attached to the interpreter state yet, it returns `NULL`.

int **PyState_AddModule**(*PyObject* *module, *PyModuleDef* *def)

Parte da ABI Estável *desde a versão 3.3.* Attaches the module object passed to the function to the interpreter state. This allows the module object to be accessible via *PyState_FindModule()*.

Only effective on modules created using single-phase initialization.

Python calls `PyState_AddModule` automatically after importing a module, so it is unnecessary (but harmless) to call it from module initialization code. An explicit call is needed only if the module's own init code subsequently calls `PyState_FindModule`. The function is mainly intended for implementing alternative import mechanisms (either by calling it directly, or by referring to its implementation for details of the required state updates).

The caller must hold the GIL.

Retorna `-1` com uma exceção definida em caso de erro, `0` em caso de sucesso.

Adicionado na versão 3.3.

int **PyState_RemoveModule**(*PyModuleDef* *def)

Parte da ABI Estável *desde a versão 3.3.* Removes the module object created from *def* from the interpreter state. Return `-1` with an exception set on error, `0` on success.

The caller must hold the GIL.

Adicionado na versão 3.3.

8.6.3 Objetos Iteradores

O Python fornece dois objetos iteradores de propósito geral. O primeiro, um iterador de sequência, trabalha com uma sequência arbitrária suportando o método `__getitem__()`. O segundo trabalha com um objeto chamável e um valor de sentinela, chamando o chamável para cada item na sequência e finalizando a iteração quando o valor de sentinela é retornado.

PyTypeObject **PySeqIter_Type**

Parte da ABI Estável. Objeto de tipo para objetos iteradores retornados por *PySeqIter_New()* e a forma de um argumento da função embutida `iter()` para os tipos de sequência embutidos.

int **PySeqIter_Check**(*PyObject* *op)

Retorna true se o tipo de *op* for *PySeqIter_Type*. Esta função sempre é bem-sucedida.

PyObject ***PySeqIter_New**(*PyObject* *seq)

Retorna valor: Nova referência. Parte da ABI Estável. 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.

PyTypeObject **PyCallIter_Type**

Parte da ABI Estável. Objeto de tipo para objetos iteradores retornados por *PyCallIter_New()* e a forma de dois argumentos da função embutida `iter()`.

int **PyCallIter_Check**(*PyObject* *op)

Retorna true se o tipo de *op* for `PyCallIter_Type`. Esta função sempre é bem-sucedida.

PyObject ***PyCallIter_New**(*PyObject* *callable, *PyObject* *sentinel)

Retorna valor: Nova referência. *Parte da* ABI Estável. 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.

8.6.4 Objetos Descritores

"Descritores" são objetos que descrevem algum atributo de um objeto. Eles são encontrados no dicionário de objetos de tipo.

PyTypeObject **PyProperty_Type**

Parte da ABI Estável. O tipo de objeto para os tipos de descritores embutidos.

PyObject ***PyDescr_NewGetSet**(*PyTypeObject* *type, struct *PyGetSetDef* *getset)

Retorna valor: Nova referência. *Parte da* ABI Estável.

PyObject ***PyDescr_NewMember**(*PyTypeObject* *type, struct *PyMemberDef* *meth)

Retorna valor: Nova referência. *Parte da* ABI Estável.

PyObject ***PyDescr_NewMethod**(*PyTypeObject* *type, struct *PyMethodDef* *meth)

Retorna valor: Nova referência. *Parte da* ABI Estável.

PyObject ***PyDescr_NewWrapper**(*PyTypeObject* *type, struct wrapperbase *wrapper, void *wrapped)

Retorna valor: Nova referência.

PyObject ***PyDescr_NewClassMethod**(*PyTypeObject* *type, *PyMethodDef* *method)

Retorna valor: Nova referência. *Parte da* ABI Estável.

int **PyDescr_IsData**(*PyObject* *descr)

Retorna não-zero se os objetos descritores *descr* descrevem um atributo de dados, ou `0` se os mesmos descrevem um método. *descr* deve ser um objeto descritor; não há verificação de erros.

PyObject ***PyWrapper_New**(*PyObject**, *PyObject**)

Retorna valor: Nova referência. *Parte da* ABI Estável.

8.6.5 Objetos Slice

PyTypeObject **PySlice_Type**

Parte da ABI Estável. Tipo de objeto para objetos fatia. Isso é o mesmo que `slice` na camada Python.

int **PySlice_Check**(*PyObject* *ob)

Retorna true se *ob* for um objeto fatia; *ob* não deve ser `NULL`. Esta função sempre tem sucesso.

PyObject ***PySlice_New**(*PyObject* *start, *PyObject* *stop, *PyObject* *step)

Retorna valor: Nova referência. *Parte da* ABI Estável. Retorna um novo objeto fatia com os valores fornecidos. Os parâmetros *start*, *stop* e *step* são usados como os valores dos atributos do objeto fatia com os mesmos nomes. Qualquer um dos valores pode ser `NULL`, caso em que `None` será usado para o atributo correspondente.

Retorna `NULL` com uma exceção definida se o novo objeto não puder ser alocado.

int **PySlice_GetIndices**(*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)

Parte da ABI Estável. Recupera os índices de início, parada e intermediário do objeto fatia *slice*, presumindo uma sequência de comprimento *length*. Trata índices maiores que *length* como erros.

Retorna `0` em caso de sucesso e `-1` em caso de erro sem exceção definida (a menos que um dos índices não fosse `None` e falhou ao ser convertido para um inteiro, neste caso `-1` é retornado com uma exceção definida).

Você provavelmente não deseja usar esta função.

Alterado na versão 3.2: O tipo de parâmetro para o parâmetro *slice* era antes de `PySliceObject*`.

int **PySlice_GetIndicesEx**(*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step, *Py_ssize_t* *slicelength)

Parte da ABI Estável. Substituição utilizável para *PySlice_GetIndices()*. Recupera os índices de início, parada e intermediário do objeto fatia *slice* presumindo uma sequência de comprimento *length* e armazena o comprimento da fatia em *slicelength*. Índices fora dos limites são cortados de maneira consistente com o tratamento de fatias normais.

Retorna `0` em caso de sucesso e `-1` em caso de erro com uma exceção definida.

Nota

Esta função não é considerada segura para sequências redimensionáveis. Sua invocação deve ser substituída por uma combinação de *PySlice_Unpack()* e *PySlice_AdjustIndices()* sendo

```
if (PySlice_GetIndicesEx(slice, length, &start, &stop, &step, &slicelength)
↪< 0) {
    // retorna erro
}
```

substituído por

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {
    // retorna erro
}
slicelength = PySlice_AdjustIndices(length, &start, &stop, step);
```

Alterado na versão 3.2: O tipo de parâmetro para o parâmetro *slice* era antes de `PySliceObject*`.

Alterado na versão 3.6.1: Se `Py_LIMITED_API` não estiver definido ou estiver definido com um valor entre `0x03050400` e `0x03060000` (não incluso) ou `0x03060100` ou mais alto, `PySlice_GetIndicesEx()` é implementado como uma macro usando `PySlice_Unpack()` e `PySlice_AdjustIndices()`. Os argumentos *start*, *stop* e *step* são avaliados mais de uma vez.

Descontinuado desde a versão 3.6.1: Se `Py_LIMITED_API` estiver definido para um valor menor que `0x03050400` ou entre `0x03060000` e `0x03060100` (não incluso), `PySlice_GetIndicesEx()` é uma função descontinuada.

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

Parte da ABI Estável *desde a versão 3.7.* Extrai os membros de dados de início, parada e intermediário de um objeto fatia como C inteiros. Reduz silenciosamente os valores maiores do que `PY_SSIZE_T_MAX` para `PY_SSIZE_T_MAX`, aumenta silenciosamente os valores de início e parada menores que `PY_SSIZE_T_MIN` para `PY_SSIZE_T_MIN`, e silenciosamente aumenta os valores de intermediário menores que `-PY_SSIZE_T_MAX` para `-PY_SSIZE_T_MAX`.

Retorna `-1` com uma exceção definida em caso de erro, `0` em caso de sucesso.

Adicionado na versão 3.6.1.

Py_ssize_t **PySlice_AdjustIndices**(*Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* step)

Parte da ABI Estável *desde a versão 3.7.* Ajusta os índices de fatias inicial/final presumindo uma sequência do comprimento especificado. Índices fora dos limites são cortados de maneira consistente com o tratamento de fatias normais.

Retorna o comprimento da fatia. Sempre bem-sucedido. Não chama o código Python.

Adicionado na versão 3.6.1.

Objeto Ellipsis

PyTypeObject **PyEllipsis_Type**

Parte da ABI Estável. O tipo do objeto Python `Ellipsis`. O mesmo que `types.EllipsisType` na camada Python.

PyObject ***Py_Ellipsis**

O objeto Python `Ellipsis`. Este objeto não tem métodos. Como *Py_None*, é um objeto singleton *imortal*.

Alterado na versão 3.12: *Py_Ellipsis* é imortal.

8.6.6 Objetos MemoryView

Um objeto `memoryview` expõe a *interface de buffer* a nível de C como um objeto Python que pode ser passado como qualquer outro objeto.

PyObject ***PyMemoryView_FromObject**(*PyObject* *obj)

Retorna valor: Nova referência. Parte da ABI Estável. Cria um objeto memoryview a partir de um objeto que fornece a interface do buffer. Se *obj* tiver suporte a exportações de buffer graváveis, o objeto memoryview será de leitura/gravação; caso contrário, poderá ser somente leitura ou leitura/gravação, a critério do exportador.

PyBUF_READ

Sinalizador para solicitar um buffer de somente leitura.

PyBUF_WRITE

Sinalizador para solicitar um buffer gravável.

PyObject ***PyMemoryView_FromMemory**(char *mem, *Py_ssize_t* size, int flags)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.7.* Cria um objeto memoryview usando *mem* como o buffer subjacente. *flags* pode ser um dos seguintes *PyBUF_READ* ou *PyBUF_WRITE*.

Adicionado na versão 3.3.

PyObject ***PyMemoryView_FromBuffer**(const *Py_buffer* *view)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.11.* Cria um objeto de memoryview envolvendo a estrutura de buffer *view* fornecida. Para buffers de bytes simples, *PyMemoryView_FromMemory()* é a função preferida.

PyObject ***PyMemoryView_GetContiguous**(*PyObject* *obj, int buffertype, char order)

Retorna valor: Nova referência. Parte da ABI Estável. Cria um objeto memoryview para um pedaço *contíguo* de memória (na ordem 'C' ou 'F'ortran, representada por *order*) a partir de um objeto que define a interface do buffer. Se a memória for contígua, o objeto memoryview apontará para a memória original. Caso contrário, é feita uma cópia e a visualização da memória aponta para um novo objeto bytes.

buffertype pode ser um entre *PyBUF_READ* ou *PyBUF_WRITE*.

int **PyMemoryView_Check**(*PyObject* *obj)

Retorna true se o objeto *obj* for um objeto memoryview. Atualmente, não é permitido criar subclasses de `memoryview`. Esta função sempre tem sucesso.

Py_buffer ***PyMemoryView_GET_BUFFER**(*PyObject* *mview)

Retorna um ponteiro para a cópia privada da memória do buffer do exportador. *mview* **deve** ser uma instância de memoryview; Se essa macro não verificar seu tipo, faça você mesmo ou corre o risco de travar.

PyObject ***PyMemoryView_GET_BASE**(*PyObject* *mview)

Retorna um ponteiro para o objeto de exportação no qual a memória é baseada ou `NULL` se a memória tiver sido criada por uma das funções *PyMemoryView_FromMemory()* ou *PyMemoryView_FromBuffer()*. *mview* **deve** ser uma instância de memoryview.

8.6.7 Objetos referência fraca

O Python oferece suporte a *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 intermediário ao objeto original tanto quanto ele pode.

int **PyWeakref_Check**(*PyObject* *ob)

Retorna não zero se *ob* for um objeto referência ou um objeto intermediário. Esta função sempre tem sucesso.

int **PyWeakref_CheckRef**(*PyObject* *ob)

Retorna não zero se *ob* for um objeto referência. Esta função sempre tem sucesso.

int **PyWeakref_CheckProxy**(*PyObject* *ob)

Retorna não zero se *ob* for um objeto intermediário. Esta função sempre tem sucesso.

PyObject ***PyWeakref_NewRef**(*PyObject* *ob, *PyObject* *callback)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um objeto de referência fraco para o objeto *ob*. Isso sempre retornará uma nova referência, mas não é garantido para criar um novo objeto; um objeto de referência existente pode ser retornado. O segundo parâmetro, *callback*, pode ser um objeto chamável que recebe notificação quando *ob* for lixo coletado; ele deve aceitar um único parâmetro, que será o objeto de referência fraco propriamente dito. *callback* também pode ser `None` ou `NULL`. Se *ob* não for um objeto fracamente referenciável, ou se *callback* não for um chamável, `None`, ou `NULL`, isso retornará `NULL` e levantará a `TypeError`.

PyObject ***PyWeakref_NewProxy**(*PyObject* *ob, *PyObject* *callback)

Retorna valor: Nova referência. Parte da ABI Estável. Retorna um objeto de proxy de referência fraca para o objeto *ob*. Isso sempre retornará uma nova referência, mas não é garantido para criar um novo objeto; um objeto de proxy existente pode ser retornado. O segundo parâmetro, *callback*, pode ser um objeto chamável que recebe notificação quando *ob* for lixo coletado; ele deve aceitar um único parâmetro, que será o objeto de referência fraco propriamente dito. *callback* também pode ser `None` ou `NULL`. Se *ob* não for um objeto referência fraca, ou se *callback* não for um chamável, `None`, ou `NULL`, isso retornará `NULL` e levantará a `TypeError`.

int **PyWeakref_GetRef**(*PyObject* *ref, *PyObject* **pobj)

Parte da ABI Estável *desde a versão 3.13.* Obtém uma *referência forte* para o objeto referenciado a partir de uma referência fraca, *ref*, em **pobj*.

- Em caso de sucesso, define **pobj* como uma nova *referência forte* para o objeto referenciado e retorna 1.
- Se a referência estiver quebrada, define **pobj* como `NULL` e retorna 0.
- Em caso de erro, levanta uma exceção e retorna -1.

Adicionado na versão 3.13.

PyObject ***PyWeakref_GetObject**(*PyObject* *ref)

Retorna valor: Referência emprestada. Parte da ABI Estável. Retorna uma *referência emprestada* ao objeto referenciado a partir de uma referência fraca, *ref*. Se o referente não estiver mais em tempo real, retorna `Py_None`.

> **Nota**
>
> Esta função retorna uma *referência emprestada* para o objeto referenciado. Isso significa que você deve sempre chamar *Py_INCREF()* no objeto, exceto quando ele não puder ser destruído antes do último uso da referência emprestada.

Deprecated since version 3.13, will be removed in version 3.15: Usa *PyWeakref_GetRef()*.

PyObject ***PyWeakref_GET_OBJECT**(*PyObject* *ref)

Retorna valor: Referência emprestada. Semelhante a *PyWeakref_GetObject()*, mas não verifica erros.

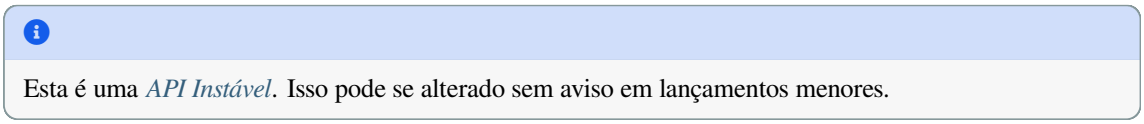
Deprecated since version 3.13, will be removed in version 3.15: Usa *PyWeakref_GetRef()*.

void **PyObject_ClearWeakRefs**(*PyObject* *object)

Parte da ABI Estável. Esta função é chamada pelo tratador `tp_dealloc` para limpar referências fracas.

Isso itera pelas referências fracas para *object* e chama retornos de chamada para as referências que possuem um. Ele retorna quando todos os retornos de chamada foram tentados.

void **PyUnstable_Object_ClearWeakRefsNoCallbacks**(*PyObject* *object)



Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Limpa as referências fracas para *object* sem chamar as funções de retorno

Esta função é chamada pelo manipulador `tp_dealloc` para tipos com finalizadores (por exemplo, `__del__()`). O manipulador para esses objetos primeiro chama `PyObject_ClearWeakRefs()` para limpar referências fracas e chamar suas funções de retorno, depois o finalizador e, finalmente, esta função para limpar quaisquer referências fracas que possam ter sido criadas pelo finalizador.

Na maioria das circunstâncias, é mais apropriado usar `PyObject_ClearWeakRefs()` para limpar referências fracas em vez desta função.

Adicionado na versão 3.13.

8.6.8 Capsules

Consulte using-capsules para obter mais informações sobre o uso desses objetos.

Adicionado na versão 3.1.

type **PyCapsule**

Este subtipo de `PyObject` representa um valor opaco, útil para módulos de extensão C que precisam passar um valor opaco (como ponteiro `void*`) através do código Python para outro código C . É frequentemente usado para disponibilizar um ponteiro de função C definido em um módulo para outros módulos, para que o mecanismo de importação regular possa ser usado para acessar APIs C definidas em módulos carregados dinamicamente.

type **PyCapsule_Destructor**

Parte da ABI Estável. O tipo de um retorno de chamada destruidor para uma cápsula. Definido como:

```
typedef void (*PyCapsule_Destructor)(PyObject *);
```

Veja `PyCapsule_New()` para a semântica dos retornos de chamada PyCapsule_Destructor.

int **PyCapsule_CheckExact**(*PyObject* *p)

Retorna true se seu argumento é um `PyCapsule`. Esta função sempre tem sucesso.

PyObject ***PyCapsule_New**(void *pointer, const char *name, *PyCapsule_Destructor* destructor)

Retorna valor: Nova referência. Parte da ABI Estável. Cria um `PyCapsule` que encapsula o *ponteiro*. O argumento *pointer* pode não ser `NULL`.

Em caso de falha, define uma exceção e retorna `NULL`.

A string *name* pode ser `NULL` ou um ponteiro para uma string C válida. Se não for `NULL`, essa string deverá sobreviver à cápsula. (Embora seja permitido liberá-lo dentro do *descructor*.)

Se o argumento *destructor* não for `NULL`, ele será chamado com a cápsula como argumento quando for destruído.

Se esta cápsula for armazenada como um atributo de um módulo, o *name* deve ser especificado como `modulename.attributename`. Isso permitirá que outros módulos importem a cápsula usando `PyCapsule_Import()`.

void *__PyCapsule_GetPointer__(*PyObject* *capsule, const char *name)

Parte da ABI Estável. Recupera o *pointer* armazenado na cápsula. Em caso de falha, define uma exceção e retorna `NULL`.

O parâmetro *name* deve ser comparado exatamente com o nome armazenado na cápsula. Se o nome armazenado na cápsula for `NULL`, o *name* passado também deve ser `NULL`. Python usa a função C `strcmp()` para comparar nomes de cápsulas.

PyCapsule_Destructor __PyCapsule_GetDestructor__(*PyObject* *capsule)

Parte da ABI Estável. Retorna o destruidor atual armazenado na cápsula. Em caso de falha, define uma exceção e retorna `NULL`.

É legal para uma cápsula ter um destruidor `NULL`. Isso torna um código de retorno `NULL` um tanto ambíguo; use *`PyCapsule_IsValid()`* ou *`PyErr_Occurred()`* para desambiguar.

void *__PyCapsule_GetContext__(*PyObject* *capsule)

Parte da ABI Estável. Retorna o contexto atual armazenado na cápsula. Em caso de falha, define uma exceção e retorna `NULL`.

É legal para uma cápsula ter um contexto `NULL`. Isso torna um código de retorno `NULL` um tanto ambíguo; use *`PyCapsule_IsValid()`* ou *`PyErr_Occurred()`* para desambiguar.

const char *__PyCapsule_GetName__(*PyObject* *capsule)

Parte da ABI Estável. Retorna o nome atual armazenado na cápsula. Em caso de falha, define uma exceção e retorna `NULL`.

É legal para uma cápsula ter um nome `NULL`. Isso torna um código de retorno `NULL` um tanto ambíguo; use *`PyCapsule_IsValid()`* ou *`PyErr_Occurred()`* para desambiguar.

void *__PyCapsule_Import__(const char *name, int no_block)

Parte da ABI Estável. Importa um ponteiro para um objeto C de um atributo de cápsula em um módulo. O parâmetro *name* deve especificar o nome completo do atributo, como em `module.attribute`. O *name* armazenado na cápsula deve corresponder exatamente a essa string.

Retorna o ponteiro interno *pointer* da cápsula com sucesso. Em caso de falha, define uma exceção e retorna `NULL`.

Alterado na versão 3.3: *no_block* não tem mais efeito.

int __PyCapsule_IsValid__(*PyObject* *capsule, const char *name)

Parte da ABI Estável. Determina se *capsule* é ou não uma cápsula válida. Uma cápsula válida é diferente de `NULL`, passa *`PyCapsule_CheckExact()`*, possui um ponteiro diferente de `NULL` armazenado e seu nome interno corresponde ao parâmetro *name*. (Consulte *`PyCapsule_GetPointer()`* para obter informações sobre como os nomes das cápsulas são comparados.)

Em outras palavras, se *`PyCapsule_IsValid()`* retornar um valor verdadeiro, as chamadas para qualquer um dos acessadores (qualquer função que comece com `PyCapsule_Get`) terão êxito garantido.

Retorna um valor diferente de zero se o objeto for válido e corresponder ao nome passado. Retorna `0` caso contrário. Esta função não falhará.

int __PyCapsule_SetContext__(*PyObject* *capsule, void *context)

Parte da ABI Estável. Define o ponteiro de contexto dentro de *capsule* para *context*.

Retorna `0` em caso de sucesso. Retorna diferente de zero e define uma exceção em caso de falha.

int __PyCapsule_SetDestructor__(*PyObject* *capsule, *PyCapsule_Destructor* destructor)

Parte da ABI Estável. Define o destrutor dentro de *capsule* para *destructor*.

Retorna `0` em caso de sucesso. Retorna diferente de zero e define uma exceção em caso de falha.

int __PyCapsule_SetName__(*PyObject* *capsule, const char *name)

Parte da ABI Estável. Define o nome dentro de *capsule* como *name*. Se não for `NULL`, o nome deve sobreviver à cápsula. Se o *name* anterior armazenado na cápsula não era `NULL`, nenhuma tentativa será feita para liberá-lo.

Retorna `0` em caso de sucesso. Retorna diferente de zero e define uma exceção em caso de falha.

int **PyCapsule_SetPointer**(*PyObject* *capsule, void *pointer)

Parte da ABI Estável. Define o ponteiro nulo dentro de *capsule* para *pointer*. O ponteiro não pode ser `NULL`.

Retorna `0` em caso de sucesso. Retorna diferente de zero e define uma exceção em caso de falha.

8.6.9 Objetos Frame

type **PyFrameObject**

Parte da API Limitada *(como uma estrutura opaca).* A estrutura C dos objetos usados para descrever objetos frame.

Não há membros públicos nesta estrutura.

Alterado na versão 3.11: Os membros dessa estrutura foram removidos da API C pública. Consulte a entrada O Que há de Novo para detalhes.

As funções *PyEval_GetFrame()* e *PyThreadState_GetFrame()* podem ser utilizadas para obter um objeto frame.

Veja também *Reflexão*.

PyTypeObject **PyFrame_Type**

O tipo de objetos frame. É o mesmo objeto que `types.FrameType` na camada Python.

Alterado na versão 3.11: Anteriormente, este tipo só estava disponível após incluir `<frameobject.h>`.

int **PyFrame_Check**(*PyObject* *obj)

Retorna diferente de zero se *obj* é um objeto frame

Alterado na versão 3.11: Anteriormente, esta função só estava disponível após incluir `<frameobject.h>`.

PyFrameObject ***PyFrame_GetBack**(*PyFrameObject* *frame)

Retorna valor: Nova referência. Obtém o *frame* próximo ao quadro externo.

Retorna uma *referência forte* ou `NULL` se *frame* não tiver quadro externo.

Adicionado na versão 3.9.

PyObject ***PyFrame_GetBuiltins**(*PyFrameObject* *frame)

Retorna valor: Nova referência. Get the *frame*'s `f_builtins` attribute.

Retorna uma *referência forte*. O resultado não pode ser `NULL`.

Adicionado na versão 3.11.

PyCodeObject ***PyFrame_GetCode**(*PyFrameObject* *frame)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.10.* Obtém o código de *frame*.

Retorna uma *referência forte*.

O resultado (código do frame) não pode ser `NULL`.

Adicionado na versão 3.9.

PyObject ***PyFrame_GetGenerator**(*PyFrameObject* *frame)

Retorna valor: Nova referência. Obtém o gerador, corrotina ou gerador assíncrono que possui este frame, ou `NULL` se o frame não pertence a um gerador. Não levanta exceção, mesmo que o valor retornado seja `NULL`.

Retorna uma *referência forte*, ou `NULL`.

Adicionado na versão 3.11.

PyObject ***PyFrame_GetGlobals**(*PyFrameObject* *frame)

Retorna valor: Nova referência. Get the *frame*'s `f_globals` attribute.

Retorna uma *referência forte*. O resultado não pode ser `NULL`.

Adicionado na versão 3.11.

int **PyFrame_GetLasti**(*PyFrameObject* *frame)

Get the *frame*'s `f_lasti` attribute.

Retorna -1 se `frame.f_lasti` é `None`.

Adicionado na versão 3.11.

PyObject ***PyFrame_GetVar**(*PyFrameObject* *frame, *PyObject* *name)

Retorna valor: Nova referência. Get the variable *name* of *frame*.

- Return a *strong reference* to the variable value on success.
- Raise `NameError` and return `NULL` if the variable does not exist.
- Raise an exception and return `NULL` on error.

name type must be a `str`.

Adicionado na versão 3.12.

PyObject ***PyFrame_GetVarString**(*PyFrameObject* *frame, const char *name)

Retorna valor: Nova referência. Similar to `PyFrame_GetVar()`, but the variable name is a C string encoded in UTF-8.

Adicionado na versão 3.12.

PyObject ***PyFrame_GetLocals**(*PyFrameObject* *frame)

Retorna valor: Nova referência. Get the *frame*'s `f_locals` attribute. If the frame refers to an *optimized scope*, this returns a write-through proxy object that allows modifying the locals. In all other cases (classes, modules, `exec()`, `eval()`) it returns the mapping representing the frame locals directly (as described for `locals()`).

Retorna uma *referência forte*.

Adicionado na versão 3.11.

Alterado na versão 3.13: As part of **PEP 667**, return an instance of `PyFrameLocalsProxy_Type`.

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

Parte da ABI Estável *desde a versão 3.10.* Retorna o número da linha do *frame* atualmente em execução.

Frame Locals Proxies

Adicionado na versão 3.13.

The `f_locals` attribute on a frame object is an instance of a "frame-locals proxy". The proxy object exposes a write-through view of the underlying locals dictionary for the frame. This ensures that the variables exposed by `f_locals` are always up to date with the live local variables in the frame itself.

See **PEP 667** for more information.

PyTypeObject **PyFrameLocalsProxy_Type**

The type of frame `locals()` proxy objects.

int **PyFrameLocalsProxy_Check**(*PyObject* *obj)

Return non-zero if *obj* is a frame `locals()` proxy.

Internal Frames

Unless using **PEP 523**, you will not need this.

struct **_PyInterpreterFrame**

The interpreter's internal frame representation.

Adicionado na versão 3.11.

PyObject ***PyUnstable_InterpreterFrame_GetCode**(struct *_PyInterpreterFrame* *frame);

> Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Return a *strong reference* to the code object for the frame.

Adicionado na versão 3.12.

int **PyUnstable_InterpreterFrame_GetLasti**(struct *_PyInterpreterFrame* *frame);

> Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Return the byte offset into the last executed instruction.

Adicionado na versão 3.12.

int **PyUnstable_InterpreterFrame_GetLine**(struct *_PyInterpreterFrame* *frame);

> Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Return the currently executing line number, or -1 if there is no line number.

Adicionado na versão 3.12.

8.6.10 Objetos Geradores

Objetos geradores são o que o Python usa para implementar iteradores geradores. Eles são normalmente criados por iteração sobre uma função que produz valores, em vez de invocar explicitamente `PyGen_New()` ou `PyGen_NewWithQualName()`.

type **PyGenObject**

A estrutura C usada para objetos geradores.

PyTypeObject **PyGen_Type**

O objeto de tipo correspondendo a objetos geradores.

int **PyGen_Check**(*PyObject* *ob)

Retorna verdadeiro se *ob* for um objeto gerador; *ob* não deve ser `NULL`. Esta função sempre tem sucesso.

int **PyGen_CheckExact**(*PyObject* *ob)

Retorna verdadeiro se o tipo do *ob* é `PyGen_Type`; *ob* não deve ser `NULL`. Esta função sempre tem sucesso.

PyObject ***PyGen_New**(*PyFrameObject* *frame)

Retorna valor: Nova referência. Cria e retorna um novo objeto gerador com base no objeto *frame*. Uma referência a *quadro* é roubada por esta função. O argumento não deve ser `NULL`.

PyObject ***PyGen_NewWithQualName**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

Retorna valor: Nova referência. Cria e retorna um novo objeto gerador com base no objeto *frame*, com `__name__` e `__qualname__` definidos como *name* e *qualname*. Uma referência a *frame* é roubada por esta função. O argumento *frame* não deve ser `NULL`.

8.6.11 Objetos corrotina

Adicionado na versão 3.5.

Os objetos corrotina são aquelas funções declaradas com um retorno de palavra-chave `async`.

type **PyCoroObject**

A estrutura C utilizada para objetos corrotinas.

PyTypeObject **PyCoro_Type**

O tipo de objeto correspondente a objetos corrotina.

int **PyCoro_CheckExact**(*PyObject* *ob)

Retorna true se o tipo do *ob* é `PyCoro_Type`; *ob* não deve ser `NULL`. Esta função sempre tem sucesso.

PyObject ***PyCoro_New**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

Retorna valor: Nova referência. Cria e retorna um novo objeto de corrotina com base no objeto *frame*, com `__name__` e `__qualname__` definido como *name* e *qualname*. Uma referência a *frame* é roubada por esta função. O argumento *frame* não deve ser `NULL`.

8.6.12 Objetos de variáveis de contexto

Adicionado na versão 3.7.

Alterado na versão 3.7.1:

Nota

No Python 3.7.1, as assinaturas de todas as APIs C de variáveis de contexto foram **alteradas** para usar ponteiros `PyObject` em vez de `PyContext`, `PyContextVar` e `PyContextToken`. Por exemplo:

```
// no 3.7.0:
PyContext *PyContext_New(void);

// no 3.7.1+:
PyObject *PyContext_New(void);
```

Veja bpo-34762 para mais detalhes.

Esta seção detalha a API C pública para o módulo `contextvars`.

type **PyContext**

A estrutura C usada para representar um objeto `contextvars.Context`.

type **PyContextVar**

A estrutura C usada para representar um objeto `contextvars.ContextVar`.

type **PyContextToken**

A estrutura C usada para representar um objeto `contextvars.Token`

PyTypeObject **PyContext_Type**

O objeto de tipo que representa o tipo de *contexto*.

PyTypeObject **PyContextVar_Type**

O objeto de tipo que representa o tipo de *variável de contexto*.

PyTypeObject **PyContextToken_Type**

O objeto de tipo que representa o tipo de *token de variável de contexto*.

Macros de verificação de tipo:

int **PyContext_CheckExact**(*PyObject* *o)

Retorna verdadeiro se *o* for do tipo `PyContext_Type`. *o* não deve ser `NULL`. Esta função sempre tem sucesso.

int **PyContextVar_CheckExact**(*PyObject* *o)

Retorna verdadeiro se *o* for do tipo `PyContextVar_Type`. *o* não deve ser `NULL`. Esta função sempre tem sucesso.

int **PyContextToken_CheckExact**(*PyObject* *o)

Retorna verdadeiro se *o* for do tipo `PyContextToken_Type`. *o* não deve ser `NULL`. Esta função sempre tem sucesso.

Funções de gerenciamento de objetos de contexto:

PyObject ***PyContext_New**(void)

Retorna valor: Nova referência. Cria um novo objeto de contexto vazio. Retorna `NULL` se um erro ocorreu.

PyObject ***PyContext_Copy**(*PyObject* *ctx)

Retorna valor: Nova referência. Cria uma cópia rasa do objeto de contexto *ctx* passado. Retorna `NULL` se um erro ocorreu.

PyObject ***PyContext_CopyCurrent**(void)

Retorna valor: Nova referência. Cria uma cópia rasa do contexto da thread atual. Retorna `NULL` se um erro ocorreu.

int **PyContext_Enter**(*PyObject* *ctx)

Defina *ctx* como o contexto atual para o thread atual. Retorna `0` em caso de sucesso e `-1` em caso de erro.

int **PyContext_Exit**(*PyObject* *ctx)

Desativa o contexto *ctx* e restaura o contexto anterior como o contexto atual para a thread atual. Retorna `0` em caso de sucesso e `-1` em caso de erro.

Funções de variável de contexto:

PyObject ***PyContextVar_New**(const char *name, *PyObject* *def)

Retorna valor: Nova referência. Cria um novo objeto `ContextVar`. O parâmetro *name* é usado para fins de introspecção e depuração. O parâmetro *def* especifica um valor padrão para a variável de contexto, ou `NULL` para nenhum padrão. Se ocorrer um erro, esta função retorna `NULL`.

int **PyContextVar_Get**(*PyObject* *var, *PyObject* *default_value, *PyObject* **value)

Obtém o valor de uma variável de contexto. Retorna `-1` se um erro ocorreu durante a pesquisa, e `0` se nenhum erro ocorreu, se um valor foi encontrado ou não.

Se a variável de contexto foi encontrada, *value* será um ponteiro para ela. Se a variável de contexto *não* foi encontrada, *value* apontará para:

- *default_value*, se não for `NULL`;
- o valor padrão de *var*, se não for `NULL`;
- `NULL`

Exceto para `NULL`, a função retorna uma nova referência.

PyObject ***PyContextVar_Set**(*PyObject* *var, *PyObject* *value)

Retorna valor: Nova referência. Define o valor de *var* como *value* no contexto atual. Retorna um novo objeto token para esta alteração, ou `NULL` se um erro ocorreu.

int **PyContextVar_Reset**(*PyObject* *var, *PyObject* *token)

Redefine o estado da variável de contexto *var* para o estado que anterior a `PyContextVar_Set()` que retornou o *token* foi chamado. Esta função retorna `0` em caso de sucesso e `-1` em caso de erro.

8.6.13 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.

type **PyDateTime_Date**

Esta é uma instância de *PyObject* representando o objeto data do Python.

type **PyDateTime_DateTime**

Esta é uma instância de *PyObject* representando o objeto datetime do Python.

type **PyDateTime_Time**

Esta é uma instância de *PyObject* representando o objeto hora do Python.

type **PyDateTime_Delta**

Esta é uma instância de *PyObject* representando a diferença entre dois valores de datetime.

PyTypeObject **PyDateTime_DateType**

Esta instância de *PyTypeObject* representa o tipo data do Python; é o mesmo objeto que `datetime.date` na camada de Python.

PyTypeObject **PyDateTime_DateTimeType**

Esta instância de *PyTypeObject* representa o tipo datetime do Python; é o mesmo objeto que `datetime.datetime` na camada de Python.

PyTypeObject **PyDateTime_TimeType**

Esta instância de *PyTypeObject* representa o tipo hora do Python; é o mesmo objeto que `datetime.time` na camada de Python.

PyTypeObject **PyDateTime_DeltaType**

Esta instância de *PyTypeObject* representa o tipo Python para a diferença entre dois valores de datetime; é o mesmo objeto que `datetime.timedelta` na camada de Python.

PyTypeObject **PyDateTime_TZInfoType**

Esta instância de *PyTypeObject* representa o tipo fuso horário do Python; é o mesmo objeto que `datetime.tzinfo` na camada de Python.

Macro para acesso ao singleton UTC:

PyObject ***PyDateTime_TimeZone_UTC**

Retorna um singleton do fuso horário representando o UTC, o mesmo objeto que `datetime.timezone.utc`.

Adicionado na versão 3.7.

Macros de verificação de tipo:

int **PyDate_Check**(*PyObject* *ob)

Retorna verdadeiro se *ob* for do tipo *PyDateTime_DateType* ou um subtipo de `PyDateTime_DateType`. *ob* não deve ser `NULL`. Esta função sempre tem sucesso.

int **PyDate_CheckExact**(*PyObject* *ob)

Retorna verdadeiro se *ob* for do tipo *PyDateTime_DateType*. *ob* não deve ser `NULL`. Esta função sempre tem sucesso.

int **PyDateTime_Check**(*PyObject* *ob)

Retorna verdadeiro se *ob* é do tipo *PyDateTime_DateTimeType* ou um subtipo de `PyDateTime_DateTimeType`. *ob* não deve ser `NULL`. Esta função sempre tem sucesso.

int **PyDateTime_CheckExact**(*PyObject* *ob)

Retorna verdadeiro se *ob* for do tipo *PyDateTime_DateTimeType*. *ob* não deve ser `NULL`. Esta função sempre tem sucesso.

int **PyTime_Check**(*PyObject* *ob)

Retorna verdadeiro se *ob* é do tipo *PyDateTime_TimeType* ou um subtipo de `PyDateTime_TimeType`. *ob* não deve ser `NULL`. Esta função sempre tem sucesso.

int **PyTime_CheckExact**(*PyObject* *ob)

Retorna verdadeiro se *ob* for do tipo *PyDateTime_TimeType*. *ob* não deve ser `NULL`. Esta função sempre tem sucesso.

int **PyDelta_Check**(*PyObject* *ob)

Retorna verdadeiro se *ob* for do tipo *PyDateTime_DeltaType* ou um subtipo de `PyDateTime_DeltaType`. *ob* não pode ser `NULL`. Esta função sempre tem sucesso.

int **PyDelta_CheckExact**(*PyObject* *ob)

Retorna verdadeiro se *ob* for do tipo *PyDateTime_DeltaType*. *ob* não deve ser `NULL`. Esta função sempre tem sucesso.

int **PyTZInfo_Check**(*PyObject* *ob)

Retorna verdadeiro se *ob* for do tipo *PyDateTime_TZInfoType* ou um subtipo de `PyDateTime_TZInfoType`. *ob* não pode ser `NULL`. Esta função sempre tem sucesso.

int **PyTZInfo_CheckExact**(*PyObject* *ob)

Retorna verdadeiro se *ob* for do tipo *PyDateTime_TZInfoType*. *ob* não deve ser `NULL`. Esta função sempre tem sucesso.

Macros para criar objetos:

PyObject ***PyDate_FromDate**(int year, int month, int day)

Retorna valor: Nova referência. Retorna um objeto `datetime.date` com o ano, mês e dia especificados.

PyObject ***PyDateTime_FromDateAndTime**(int year, int month, int day, int hour, int minute, int second, int usecond)

Retorna valor: Nova referência. Retorna um objeto `datetime.datetime` com o ano, mês, dia, hora, minuto, segundo, microssegundo especificados.

PyObject ***PyDateTime_FromDateAndTimeAndFold**(int year, int month, int day, int hour, int minute, int second, int usecond, int fold)

Retorna valor: Nova referência. Retorna um objeto `datetime.datetime` com o ano, mês, dia, hora, minuto, segundo, microssegundo e a dobra especificados.

Adicionado na versão 3.6.

PyObject ***PyTime_FromTime**(int hour, int minute, int second, int usecond)

Retorna valor: Nova referência. Retorna um objeto `datetime.time` com a hora, minuto, segundo e microssegundo especificados.

PyObject ***PyTime_FromTimeAndFold**(int hour, int minute, int second, int usecond, int fold)

Retorna valor: Nova referência. Retorna um objeto `datetime.time` com a hora, minuto, segundo, microssegundo e a dobra especificados.

Adicionado na versão 3.6.

PyObject ***PyDelta_FromDSU**(int days, int seconds, int useconds)

Retorna valor: Nova referência. Retorna um objeto `datetime.timedelta` representando o número especificado de dias, segundos e microssegundos. A normalização é realizada para que o número resultante de microssegundos e segundos esteja nos intervalos documentados para objetos de `datetime.timedelta`.

PyObject ***PyTimeZone_FromOffset**(*PyObject* *offset)

Retorna valor: Nova referência. Retorna um objeto `datetime.timezone` com um deslocamento fixo sem nome representado pelo argumento *offset*.

Adicionado na versão 3.7.

PyObject ***PyTimeZone_FromOffsetAndName**(*PyObject* *offset, *PyObject* *name)

Retorna valor: Nova referência. Retorna um objeto `datetime.timezone` com um deslocamento fixo representado pelo argumento *offset* e com tzname *name*.

Adicionado na versão 3.7.

Macros para extrair campos de objetos *date*. O argumento deve ser uma instância de *PyDateTime_Date*, inclusive subclasses (como *PyDateTime_DateTime*). O argumento não deve ser `NULL`, e o tipo não é verificado:

int **PyDateTime_GET_YEAR**(*PyDateTime_Date* *o)

Retorna o ano, como um inteiro positivo.

int **PyDateTime_GET_MONTH**(*PyDateTime_Date* *o)

Retorna o mês, como um inteiro de 1 a 12.

int **PyDateTime_GET_DAY**(*PyDateTime_Date* *o)

Retorna o dia, como um inteiro de 1 a 31.

Macros para extrair campos de objetos *datetime*. O argumento deve ser uma instância de *PyDateTime_DateTime*, inclusive subclasses. O argumento não deve ser `NULL`, e o tipo não é verificado:

int **PyDateTime_DATE_GET_HOUR**(*PyDateTime_DateTime* *o)

Retorna a hora, como um inteiro de 0 a 23.

int **PyDateTime_DATE_GET_MINUTE**(*PyDateTime_DateTime* *o)

Retorna o minuto, como um inteiro de 0 a 59.

int **PyDateTime_DATE_GET_SECOND**(*PyDateTime_DateTime* *o)

Retorna o segundo, como um inteiro de 0 a 59.

int **PyDateTime_DATE_GET_MICROSECOND**(*PyDateTime_DateTime* *o)

Retorna o microssegundo, como um inteiro de 0 a 999999.

int **PyDateTime_DATE_GET_FOLD**(*PyDateTime_DateTime* *o)

Retorna a dobra, como um inteiro de 0 a 1.

Adicionado na versão 3.6.

PyObject ***PyDateTime_DATE_GET_TZINFO**(*PyDateTime_DateTime* *o)

Retorna o tzinfo (que pode ser `None`).

Adicionado na versão 3.10.

Macros para extrair campos de objetos *time*. O argumento deve ser uma instância de *PyDateTime_Time*, inclusive subclasses. O argumento não deve ser `NULL`, e o tipo não é verificado:

int **PyDateTime_TIME_GET_HOUR**(*PyDateTime_Time* *o)

Retorna a hora, como um inteiro de 0 a 23.

int **PyDateTime_TIME_GET_MINUTE**(*PyDateTime_Time* *o)

Retorna o minuto, como um inteiro de 0 a 59.

int **PyDateTime_TIME_GET_SECOND**(*PyDateTime_Time* *o)

Retorna o segundo, como um inteiro de 0 a 59.

int **PyDateTime_TIME_GET_MICROSECOND**(*PyDateTime_Time* *o)

Retorna o microssegundo, como um inteiro de 0 a 999999.

int **PyDateTime_TIME_GET_FOLD**(*PyDateTime_Time* *o)

Retorna a dobra, como um inteiro de 0 a 1.

Adicionado na versão 3.6.

PyObject ***PyDateTime_TIME_GET_TZINFO**(*PyDateTime_Time* *o)

Retorna o tzinfo (que pode ser `None`).

Adicionado na versão 3.10.

Macros para extrair campos de objetos *time delta*. O argumento deve ser uma instância de *PyDateTime_Delta*, inclusive subclasses. O argumento não deve ser `NULL`, e o tipo não é verificado:

int **PyDateTime_DELTA_GET_DAYS**(*PyDateTime_Delta* *o)

Retorna o número de dias, como um inteiro de -999999999 a 999999999.

Adicionado na versão 3.3.

int **PyDateTime_DELTA_GET_SECONDS**(*PyDateTime_Delta* *o)

Retorna o número de segundos, como um inteiro de 0 a 86399.

Adicionado na versão 3.3.

int **PyDateTime_DELTA_GET_MICROSECONDS**(*PyDateTime_Delta* *o)

Retorna o número de microssegundos, como um inteiro de 0 a 999999.

Adicionado na versão 3.3.

Macros para a conveniência de módulos implementando a API de DB:

PyObject ***PyDateTime_FromTimestamp**(*PyObject* *args)

Retorna valor: Nova referência. Cria e retorna um novo objeto `datetime.datetime` dado uma tupla de argumentos que possa ser passada ao método `datetime.datetime.fromtimestamp()`.

PyObject ***PyDate_FromTimestamp**(*PyObject* *args)

Retorna valor: Nova referência. Cria e retorna um novo objeto `datetime.date` dado uma tupla de argumentos que possa ser passada ao método `datetime.date.fromtimestamp()`.

8.6.14 Objetos de indicação de tipos

São fornecidos vários tipos embutidos para sugestão de tipo. Atualmente, dois tipos existem – GenericAlias e Union. Apenas `GenericAlias` está exposto ao C.

PyObject ***Py_GenericAlias**(*PyObject* *origin, *PyObject* *args)

Parte da ABI Estável desde a versão 3.9. Cria um objeto GenericAlias. Equivalente a chamar a classe Python `types.GenericAlias`. Os argumentos *origin* e *args* definem os atributos `__origin__` e `__args__` de `GenericAlias` respectivamente. *origin* deve ser um *PyTypeObject**, e *args* pode ser um *PyTupleObject** ou qualquer `PyObject*`. Se *args* passado não for uma tupla, uma tupla de 1 elemento é construída automaticamente e `__args__` é definido como `(args,)`. A verificação mínima é feita para os argumentos, então a função terá sucesso mesmo se *origin* não for um tipo. O atributo `__parameters__` de `GenericAlias` é construído lentamente a partir de `__args__`. Em caso de falha, uma exceção é levantada e `NULL` é retornado.

Aqui está um exemplo de como tornar um tipo de extensão genérico:

```
...
static PyMethodDef my_obj_methods[] = {
    // Outros métodos.
    ...
    {"__class_getitem__", Py_GenericAlias, METH_O|METH_CLASS, "Veja PEP 585"}
    ...
}
```

Ver também

O método de modelo de dados `__class_getitem__()`.

Adicionado na versão 3.9.

PyTypeObject **Py_GenericAliasType**

Parte da ABI Estável *desde a versão 3.9.* O tipo C do objeto retornado por `Py_GenericAlias()`. Equivalente a `types.GenericAlias` no Python.

Adicionado na versão 3.9.

CAPÍTULO 9

Inicialização, finalização e threads

Veja *Configuração de inicialização do Python* para detalhes sobre como configurar o interpretador antes da inicialização.

9.1 Antes da inicialização do Python

Em uma aplicação que incorpora Python, a função `Py_Initialize()` deve ser chamada antes de usar qualquer outra função da API Python/C; com exceção de algumas funções e as *variáveis globais de configuração*.

As seguintes funções podem ser seguramente chamadas antes da inicialização do Python.

- Funções que inicializam o interpretador:
  - `Py_Initialize()`
  - `Py_InitializeEx()`
  - `Py_InitializeFromConfig()`
  - `Py_BytesMain()`
  - `Py_Main()`
  - as funções de pré-inicialização de tempo de execução cobertas em *Configuração de Inicialização do Python*
- Funções de configuração
  - `PyImport_AppendInittab()`
  - `PyImport_ExtendInittab()`
  - `PyInitFrozenExtensions()`
  - `PyMem_SetAllocator()`
  - `PyMem_SetupDebugHooks()`
  - `PyObject_SetArenaAllocator()`
  - `Py_SetProgramName()`
  - `Py_SetPythonHome()`
  - `PySys_ResetWarnOptions()`

  - as funções de configuração cobertas em *Configuração de Inicialização do Python*

- Funções informativas:
  - *Py_IsInitialized()*
  - *PyMem_GetAllocator()*
  - *PyObject_GetArenaAllocator()*
  - *Py_GetBuildInfo()*
  - *Py_GetCompiler()*
  - *Py_GetCopyright()*
  - *Py_GetPlatform()*
  - *Py_GetVersion()*
  - *Py_IsInitialized()*
- Utilitários:
  - *Py_DecodeLocale()*
  - o relatório de status é funções utilitárias cobertas em *Configuração de Inicialização do Python*
- Alocadores de memória:
  - *PyMem_RawMalloc()*
  - *PyMem_RawRealloc()*
  - *PyMem_RawCalloc()*
  - *PyMem_RawFree()*
- Sincronização:
  - *PyMutex_Lock()*
  - *PyMutex_Unlock()*

Nota

Apesar de sua aparente semelhança com algumas das funções listadas acima, as seguintes funções **não devem ser chamadas** antes que o interpretador tenha sido inicializado: *Py_EncodeLocale()*, *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()*, *Py_GetProgramFullPath()*, *Py_GetPythonHome()*, *Py_GetProgramName()*, *PyEval_InitThreads()*, e *Py_RunMain()*.

9.2 Variáveis de configuração global

Python tem variáveis para a configuração global a fim de controlar diferentes características e opções. Por padrão, estes sinalizadores são controlados por opções de linha de comando.

Quando um sinalizador é definido por uma opção, o valor do sinalizador é o número de vezes que a opção foi definida. Por exemplo,"-b" define *Py_BytesWarningFlag* para 1 e `-bb` define *Py_BytesWarningFlag* para 2.

int **Py_BytesWarningFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.bytes_warning* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Emite um aviso ao comparar `bytes` ou `bytearray` com `str` ou `bytes` com `int`. Emite um erro se for maior ou igual a `2`.

Definida pela opção `-b`.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_DebugFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.parser_debug* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Ativa a saída de depuração do analisador sintático (somente para especialistas, dependendo das opções de compilação).

Definida pela a opção `-d` e a variável de ambiente `PYTHONDEBUG`.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_DontWriteBytecodeFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.write_bytecode* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Se definida como diferente de zero, o Python não tentará escrever arquivos `.pyc` na importação de módulos fonte.

Definida pela opção `-B` e pela variável de ambiente `PYTHONDONTWRITEBYTECODE`.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_FrozenFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.pathconfig_warnings* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Suprime mensagens de erro ao calcular o caminho de pesquisa do módulo em *Py_GetPath()*.

Sinalizador privado usado pelos programas `_freeze_module` e `frozenmain`.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_HashRandomizationFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração de *PyConfig.hash_seed* e *PyConfig.use_hash_seed* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Definida como `1` se a variável de ambiente `PYTHONHASHSEED` estiver definida como uma string não vazia.

Se o sinalizador for diferente de zero, lê a variável de ambiente `PYTHONHASHSEED` para inicializar a semente de hash secreta.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_IgnoreEnvironmentFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.use_environment* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Ignora todas as variáveis de ambiente `PYTHON*`, por exemplo `PYTHONPATH` e `PYTHONHOME`, que podem estar definidas.

Definida pelas opções `-E` e `-I`.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_InspectFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.inspect* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Quando um script é passado como primeiro argumento ou a opção `-c` é usada, entre no modo interativo após executar o script ou o comando, mesmo quando `sys.stdin` não parece ser um terminal.

Definida pela opção `-i` e pela variável de ambiente `PYTHONINSPECT`.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_InteractiveFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.interactive* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Definida pela opção `-i`.

Descontinuado desde a versão 3.12.

int **Py_IsolatedFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.isolated* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Executa o Python no modo isolado. No modo isolado, `sys.path` não contém nem o diretório do script nem o diretório de pacotes de sites do usuário.

Definida pela opção `-I`.

Adicionado na versão 3.4.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_LegacyWindowsFSEncodingFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyPreConfig.legacy_windows_fs_encoding* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Se o sinalizador for diferente de zero, use a codificação `mbcs` com o tratador de erros `replace`, em vez da codificação UTF-8 com o tratador de erros `surrogatepass`, para a codificação do sistema de arquivos e *tratador de erros e codificação do sistema de arquivos*.

Definida como `1` se a variável de ambiente `PYTHONLEGACYWINDOWSFSENCODING` estiver definida como uma string não vazia.

Veja **PEP 529** para mais detalhes.

Disponibilidade: Windows.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_LegacyWindowsStdioFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.legacy_windows_stdio* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Se o sinalizador for diferente de zero, usa `io.FileIO` em vez de `io._WindowsConsoleIO` para fluxos padrão `sys`.

Definida como `1` se a variável de ambiente `PYTHONLEGACYWINDOWSSTDIO` estiver definida como uma string não vazia.

Veja a **PEP 528** para mais detalhes.

Disponibilidade: Windows.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_NoSiteFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.site_import* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Desabilita a importação do módulo `site` e as manipulações dependentes do site de `sys.path` que isso acarreta. Também desabilita essas manipulações se `site` for explicitamente importado mais tarde (chame `site.main()` se você quiser que eles sejam acionados).

Definida pela opção `-S`.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_NoUserSiteDirectory**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.user_site_directory* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Não adiciona o `diretório site-packages de usuário` a `sys.path`.

Definida pelas opções `-s` e `-I`, e pela variável de ambiente `PYTHONNOUSERSITE`.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_OptimizeFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.optimization_level* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Definida pela opção `-O` e pela variável de ambiente `PYTHONOPTIMIZE`.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_QuietFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.quiet* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Não exibe as mensagens de direito autoral e de versão nem mesmo no modo interativo.

Definida pela opção `-q`.

Adicionado na versão 3.2.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_UnbufferedStdioFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.buffered_stdio* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Força os fluxos stdout e stderr a não serem armazenados em buffer.

Definida pela opção `-u` e pela variável de ambiente `PYTHONUNBUFFERED`.

Deprecated since version 3.12, will be removed in version 3.14.

int **Py_VerboseFlag**

Esta API é mantida para compatibilidade com versões anteriores: a configuração *PyConfig.verbose* deve ser usada em seu lugar, consulte *Configuração de inicialização do Python*.

Exibe uma mensagem cada vez que um módulo é inicializado, mostrando o local (nome do arquivo ou módulo embutido) de onde ele é carregado. Se maior ou igual a `2`, exibe uma mensagem para cada arquivo que é verificado durante a busca por um módulo. Também fornece informações sobre a limpeza do módulo na saída.

Definida pela a opção `-v` e a variável de ambiente `PYTHONVERBOSE`.

Deprecated since version 3.12, will be removed in version 3.14.

9.3 Inicializando e encerrando o interpretador

void **Py_Initialize**()

Parte da ABI Estável. Inicializa o interpretador Python. Em uma aplicação que incorpora o Python, isto deve ser chamado antes do uso de qualquer outra função do Python/C API; veja *Antes da Inicialização do Python* para algumas exceções.

Isso inicializa a tabela de módulos carregados (`sys.modules`) e cria os módulos fundamentais `builtins`, `__main__` e `sys`. Também inicializa o caminho de pesquisa de módulos (`sys.path`). Isso não define `sys.argv`; use a API da *Configuração de inicialização do Python* para isso. Isso é um no-op quando chamado pela segunda vez (sem chamar *Py_FinalizeEx()* primeiro). Não há valor de retorno; é um erro fatal se a inicialização falhar.

Usa *Py_InitializeFromConfig()* para personalizar a *Configuração de inicialização do Python*.

> **Nota**
>
> No Windows, altera o modo do console de `O_TEXT` para `O_BINARY`, o que também afetará usos não Python do console usando o Runtime C.

void **Py_InitializeEx**(int initsigs)

Parte da ABI Estável. Esta função funciona como *Py_Initialize()* se *initsigs* for `1`. Se *initsigs* for `0`, ela pula o registro de inicialização de manipuladores de sinal, o que pode ser útil quando o CPython é incorporado como parte de uma aplicação maior.

Usa *Py_InitializeFromConfig()* para personalizar a *Configuração de inicialização do Python*.

PyStatus **Py_InitializeFromConfig**(const *PyConfig* *config)

Inicializa o Python a partir da configuração *config*, conforme descrito em *Initialization with PyConfig*.

Consulte a seção *Configuração de Inicialização do Python* para obter detalhes sobre como pré-inicializar o interpretador, preencher a estrutura de configuração do tempo de execução e consultar a estrutura de status retornada.

int **Py_IsInitialized**()

Parte da ABI Estável. Retorna true (diferente de zero) quando o interpretador Python foi inicializado, false (zero) se não. Após *Py_FinalizeEx()* ser chamado, isso retorna false até que *Py_Initialize()* seja chamado novamente.

int **Py_IsFinalizing**()

Parte da ABI Estável *desde a versão 3.13.* Return true (non-zero) if the main Python interpreter is *shutting down*. Return false (zero) otherwise.

Adicionado na versão 3.13.

int **Py_FinalizeEx**()

Parte da ABI Estável *desde a versão 3.6.* Undo all initializations made by *Py_Initialize()* and subsequent use of Python/C API functions, and destroy all sub-interpreters (see *Py_NewInterpreter()* below) that were created and not yet destroyed since the last call to *Py_Initialize()*. Ideally, this frees all memory allocated by the Python interpreter. This is a no-op when called for a second time (without calling *Py_Initialize()* again first).

Since this is the reverse of *Py_Initialize()*, it should be called in the same thread with the same interpreter active. That means the main thread and the main interpreter. This should never be called while *Py_RunMain()* is running.

Normally the return value is `0`. If there were errors during finalization (flushing buffered data), `-1` is returned.

This function is provided for a number of reasons. An embedding application might want to restart Python without having to restart the application itself. An application that has loaded the Python interpreter from a dynamically loadable library (or DLL) might want to free all memory allocated by Python before unloading the DLL. During a hunt for memory leaks in an application a developer might want to free all memory allocated by Python before exiting from the application.

Bugs and caveats: The destruction of modules and objects in modules is done in random order; this may cause destructors (`__del__()` methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once; this can happen if an application calls *Py_Initialize()* and *Py_FinalizeEx()* more than once.

Levanta um evento de auditoria `cpython._PySys_ClearAuditHooks` sem argumentos.

Adicionado na versão 3.6.

void **Py_Finalize**()

Parte da ABI Estável. This is a backwards-compatible version of *Py_FinalizeEx()* that disregards the return value.

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

Parte da ABI Estável *desde a versão 3.8.* Similar to *Py_Main()* but *argv* is an array of bytes strings, allowing the calling application to delegate the text decoding step to the CPython runtime.

Adicionado na versão 3.8.

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

Parte da ABI Estável. The main program for the standard interpreter, encapsulating a full initialization/finalization cycle, as well as additional behaviour to implement reading configurations settings from the environment and command line, and then executing `__main__` in accordance with using-on-cmdline.

This is made available for programs which wish to support the full CPython command line interface, rather than just embedding a Python runtime in a larger application.

The *argc* and *argv* parameters are similar to those which are passed to a C program's `main()` function, except that the *argv* entries are first converted to `wchar_t` using *Py_DecodeLocale()*. It is also important to note that the argument list entries may be modified to point to strings other than those passed in (however, the contents of the strings pointed to by the argument list are not modified).

The return value will be `0` if the interpreter exits normally (i.e., without an exception), `1` if the interpreter exits due to an exception, or `2` if the argument 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. If `Py_InspectFlag` is set, execution will drop into the interactive Python prompt, at which point a second otherwise unhandled `SystemExit` will still exit the process, while any other means of exiting will set the return value as described above.

In terms of the CPython runtime configuration APIs documented in the *runtime configuration* section (and without accounting for error handling), `Py_Main` is approximately equivalent to:

```
PyConfig config;
PyConfig_InitPythonConfig(&config);
PyConfig_SetArgv(&config, argc, argv);
Py_InitializeFromConfig(&config);
PyConfig_Clear(&config);

Py_RunMain();
```

In normal usage, an embedding application will call this function *instead* of calling *Py_Initialize()*, *Py_InitializeEx()* or *Py_InitializeFromConfig()* directly, and all settings will be applied as described elsewhere in this documentation. If this function is instead called *after* a preceding runtime initialization API call, then exactly which environmental and command line configuration settings will be updated is version dependent (as it depends on which settings correctly support being modified after they have already been set once when the runtime was first initialized).

int **Py_RunMain**(void)

Executes the main module in a fully configured CPython runtime.

Executes the command (*PyConfig.run_command*), the script (*PyConfig.run_filename*) or the module (*PyConfig.run_module*) specified on the command line or in the configuration. If none of these values are set, runs the interactive Python prompt (REPL) using the `__main__` module's global namespace.

If *PyConfig.inspect* is not set (the default), the return value will be `0` if the interpreter exits normally (that is, without raising an exception), or `1` if the interpreter exits due to an exception. If an otherwise unhandled `SystemExit` is raised, the function will immediately exit the process instead of returning `1`.

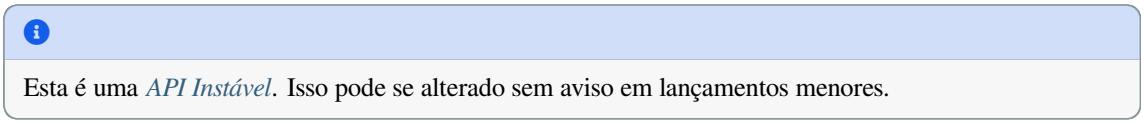
If *PyConfig.inspect* is set (such as when the `-i` option is used), rather than returning when the interpreter exits, execution will instead resume in an interactive Python prompt (REPL) using the `__main__` module's global namespace. If the interpreter exited with an exception, it is immediately raised in the REPL session.

The function return value is then determined by the way the *REPL session* terminates: returning `0` if the session terminates without raising an unhandled exception, exiting immediately for an unhandled `SystemExit`, and returning `1` for any other unhandled exception.

This function always finalizes the Python interpreter regardless of whether it returns a value or immediately exits the process due to an unhandled `SystemExit` exception.

See *Python Configuration* for an example of a customized Python that always runs in isolated mode using `Py_RunMain()`.

int **PyUnstable_AtExit**(*PyInterpreterState* *interp, void (*func)(void*), void *data)



Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Register an `atexit` callback for the target interpreter *interp*. This is similar to `Py_AtExit()`, but takes an explicit interpreter and data pointer for the callback.

The *GIL* must be held for *interp*.

Adicionado na versão 3.13.

9.4 Process-wide parameters

void **Py_SetProgramName**(const wchar_t *name)

Parte da ABI Estável. This API is kept for backward compatibility: setting `PyConfig.program_name` should be used instead, see *Python Initialization Configuration*.

Esta função deve ser chamada antes de `Py_Initialize()` ser chamada pela primeira vez, caso seja solicitada. Ela diz ao interpretador o valor do argumento `argv[0]` para a função `main()` do programa (convertido em caracteres amplos). Isto é utilizado por `Py_GetPath()` e algumas outras funções abaixo para encontrar as bibliotecas de tempo de execução relativas ao executável do interpretador. O valor padrão é `'python'`. O argumento deve apontar para um caractere string amplo terminado em zero no armazenamento estático, cujo conteúdo não mudará durante a execução do programa. Nenhum código no interpretador Python mudará o conteúdo deste armazenamento.

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_t*` string.

Descontinuado desde a versão 3.11.

wchar_t ***Py_GetProgramName**()

Parte da ABI Estável. Return the program name set with `PyConfig.program_name`, or the default. The returned string points into static storage; the caller should not modify its value.

This function should not be called before `Py_Initialize()`, otherwise it returns `NULL`.

Alterado na versão 3.10: It now returns `NULL` if called before `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Get `sys.executable` instead.

wchar_t ***Py_GetPrefix**()

Parte da ABI Estável. Return the *prefix* for installed platform-independent files. This is derived through a number of complicated rules from the program name set with `PyConfig.program_name` 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.base_prefix`. It is only useful on Unix. See also the next function.

This function should not be called before `Py_Initialize()`, otherwise it returns `NULL`.

Alterado na versão 3.10: It now returns `NULL` if called before `Py_Initialize()`.

Deprecated since version 3.13, will be removed in version 3.15: Get `sys.base_prefix` instead, or `sys.prefix` if virtual environments need to be handled.

wchar_t ***Py_GetExecPrefix**()

Parte da ABI Estável. Return the *exec-prefix* for installed platform-*dependent* files. This is derived through a number of complicated rules from the program name set with *PyConfig.program_name* 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.base_exec_prefix`. It is only useful on Unix.

Background: The exec-prefix differs from the prefix when platform dependent files (such as executables and shared libraries) are installed in a different directory tree. In a typical installation, platform dependent files may be installed in the `/usr/local/plat` subtree while platform independent may be installed in `/usr/local`.

Generally speaking, a platform is a combination of hardware and software families, e.g. Sparc machines running the Solaris 2.x operating system are considered the same platform, but Intel machines running Solaris 2.x are another platform, and Intel machines running Linux are yet another platform. Different major revisions of the same operating system generally also form different platforms. Non-Unix operating systems are a different story; the installation strategies on those systems are so different that the prefix and exec-prefix are meaningless, and set to the empty string. Note that compiled Python bytecode files are platform independent (but not independent from the Python version by which they were compiled!).

System administrators will know how to configure the **mount** or **automount** programs to share `/usr/local` between platforms while having `/usr/local/plat` be a different filesystem for each platform.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

Alterado na versão 3.10: It now returns `NULL` if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15: Get `sys.base_exec_prefix` instead, or `sys.exec_prefix` if virtual environments need to be handled.

wchar_t ***Py_GetProgramFullPath**()

Parte da ABI Estável. 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 *PyConfig.program_name*). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.executable`.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

Alterado na versão 3.10: It now returns `NULL` if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15: Get `sys.executable` instead.

wchar_t ***Py_GetPath**()

Parte da ABI Estável. Return the default module search path; this is computed from the program name (set by *PyConfig.program_name*) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is `':'` on Unix and macOS, `';'` on Windows. The returned string points into static storage; the caller should not modify its value. The list `sys.path` is initialized with this value on interpreter startup; it can be (and usually is) modified later to change the search path for loading modules.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

Alterado na versão 3.10: It now returns `NULL` if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15: Get `sys.path` instead.

const char ***Py_GetVersion**()

Parte da ABI Estável. Retorna a verão deste interpretador Python. Esta é uma string que se parece com

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

The first word (up to the first space character) is the current Python version; the first characters are the major and minor version separated by a period. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.version`.

See also the *Py_Version* constant.

const char ***Py_GetPlatform**()

Parte da ABI Estável. Return the platform identifier for the current platform. On Unix, this is formed from the "official" name of the operating system, converted to lower case, followed by the major revision number; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is `'sunos5'`. On macOS, it is `'darwin'`. On Windows, it is `'win'`. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.platform`.

const char ***Py_GetCopyright**()

Parte da ABI Estável. Retorna a string oficial de direitos autoriais para a versão atual do Python, por exemplo

```
'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**()

Parte da ABI Estável. Retorna uma indicação do compilador usado para construir a atual versão do Python, em colchetes, por exemplo:

```
"[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**()

Parte da ABI Estável. Retorna informação sobre o número de sequência e a data e hora da construção da instância atual do interpretador Python, por exemplo

```
"#67, Aug  1 1997, 22:34:28"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

void **PySys_SetArgvEx**(int argc, wchar_t **argv, int updatepath)

Parte da ABI Estável. This API is kept for backward compatibility: setting *PyConfig.argv*, *PyConfig.parse_argv* and *PyConfig.safe_path* should be used instead, see *Python Initialization Configuration*.

Set `sys.argv` based on *argc* and *argv*. These parameters are similar to those passed to the program's `main()` function with the difference that the first entry should refer to the script file to be executed rather than the executable hosting the Python interpreter. If there isn't a script that will be run, the first entry in *argv* can be an empty string. If this function fails to initialize `sys.argv`, a fatal condition is signalled using *Py_FatalError()*.

Se *updatepath* é zero, isto é tudo o que a função faz. Se *updatepath* não é zero, a função também modifica `sys.path` de acordo com o seguinte algoritmo:

- If the name of an existing script is passed in `argv[0]`, the absolute path of the directory where the script is located is prepended to `sys.path`.
- Otherwise (that is, if *argc* is `0` or `argv[0]` doesn't point to an existing file name), an empty string is prepended to `sys.path`, which is the same as prepending the current working directory (`"."`).

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_t*` string.

See also *PyConfig.orig_argv* and *PyConfig.argv* members of the *Python Initialization Configuration*.

> **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 3.1.3, you can achieve the same effect by manually popping the first `sys.path` element after having called *PySys_SetArgv()*, for example using:
>
> ```
> PyRun_SimpleString("import sys; sys.path.pop(0)\n");
> ```

Adicionado na versão 3.1.3.

Descontinuado desde a versão 3.11.

void **PySys_SetArgv**(int argc, wchar_t **argv)

Parte da ABI Estável. This API is kept for backward compatibility: setting *PyConfig.argv* and *PyConfig.parse_argv* should be used instead, see *Python Initialization Configuration*.

This function works like *PySys_SetArgvEx()* with *updatepath* set to `1` unless the **python** interpreter was started with the `-I`.

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_t*` string.

See also *PyConfig.orig_argv* and *PyConfig.argv* members of the *Python Initialization Configuration*.

Alterado na versão 3.4: The *updatepath* value depends on `-I`.

Descontinuado desde a versão 3.11.

void **Py_SetPythonHome**(const wchar_t *home)

Parte da ABI Estável. This API is kept for backward compatibility: setting *PyConfig.home* should be used instead, see *Python Initialization Configuration*.

Set the default "home" directory, that is, the location of the standard Python libraries. See `PYTHONHOME` for the meaning of the argument string.

The argument should point to a zero-terminated character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_t*` string.

Descontinuado desde a versão 3.11.

wchar_t ***Py_GetPythonHome**()

Parte da ABI Estável. Return the default "home", that is, the value set by *PyConfig.home*, or the value of the `PYTHONHOME` environment variable if it is set.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

Alterado na versão 3.10: It now returns `NULL` if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15: Get *PyConfig.home* or `PYTHONHOME` environment variable instead.

9.5 Thread State and the Global Interpreter Lock

The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the *global interpreter lock* or *GIL*, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

Therefore, the rule exists that only the thread that has acquired the *GIL* may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads (see

`sys.setswitchinterval()`). The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.

The Python interpreter keeps some thread-specific bookkeeping information inside a data structure called *PyThreadState*. There's also one global variable pointing to the current *PyThreadState*: it can be retrieved using *PyThreadState_Get()*.

9.5.1 Releasing the GIL from extension code

A maioria dos códigos de extensão que manipulam a *GIL* tem a seguinte estrutura:

```
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
```

A macro *Py_BEGIN_ALLOW_THREADS* abre um novo bloco e declara uma variável local oculta; a macro *Py_END_ALLOW_THREADS* fecha o bloco.

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.

9.5.2 Non-Python created threads

When threads are created using the dedicated Python APIs (such as the `threading` module), a thread state is automatically associated to them and the code showed above is therefore correct. However, when threads are created from C (for example by a third-party library with its own thread management), they don't hold the GIL, nor is there a thread state structure for them.

If you need to call Python code from these threads (often this will be part of a callback API provided by the aforementioned third-party library), you must first register these threads with the interpreter by creating a thread state data structure, then acquiring the GIL, and finally storing their thread state pointer, before you can start using the Python/C API. When you are done, you should reset the thread state pointer, release the GIL, and finally free the thread state data structure.

The *PyGILState_Ensure()* and *PyGILState_Release()* functions do all of the above automatically. The typical idiom for calling into Python from a C thread is:

```
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Release the thread. No Python API allowed beyond this point. */
PyGILState_Release(gstate);
```

Note that the `PyGILState_*` functions assume there is only one global interpreter (created automatically by *Py_Initialize()*). Python supports the creation of additional interpreters (using *Py_NewInterpreter()*), but mixing multiple interpreters and the `PyGILState_*` API is unsupported.

9.5.3 Cuidados com o uso de fork()

Another important thing to note about threads is their behaviour in the face of the C `fork()` call. On most systems with `fork()`, after a process forks only the thread that issued the fork will exist. This has a concrete impact both on how locks must be handled and on all stored state in CPython's runtime.

The fact that only the "current" thread remains means any locks held by other threads will never be released. Python solves this for `os.fork()` by acquiring the locks it uses internally before the fork, and releasing them afterwards. In addition, it resets any lock-objects in the child. When extending or embedding Python, there is no way to inform Python of additional (non-Python) locks that need to be acquired before or reset after a fork. OS facilities such as `pthread_atfork()` would need to be used to accomplish the same thing. Additionally, when extending or embedding Python, calling `fork()` directly rather than through `os.fork()` (and returning to or calling into Python) may result in a deadlock by one of Python's internal locks being held by a thread that is defunct after the fork. *PyOS_AfterFork_Child()* tries to reset the necessary locks, but is not always able to.

The fact that all other threads go away also means that CPython's runtime state there must be cleaned up properly, which `os.fork()` does. This means finalizing all other *PyThreadState* objects belonging to the current interpreter and all other *PyInterpreterState* objects. Due to this and the special nature of the *"main" interpreter*, `fork()` should only be called in that interpreter's "main" thread, where the CPython global runtime was originally initialized. The only exception is if `exec()` will be called immediately after.

9.5.4 High-level API

Estes são os tipos e as funções mais comumente usados na escrita de um código de extensão em C, ou ao incorporar o interpretador Python:

type **PyInterpreterState**

Parte da API Limitada *(como uma estrutura opaca).* This data structure represents the state shared by a number of cooperating threads. Threads belonging to the same interpreter share their module administration and a few other internal items. There are no public members in this structure.

Threads belonging to different interpreters initially share nothing, except process state like available memory, open file descriptors and such. The global interpreter lock is also shared by all threads, regardless of to which interpreter they belong.

type **PyThreadState**

Parte da API Limitada *(como uma estrutura opaca).* This data structure represents the state of a single thread. The only public data member is:

PyInterpreterState ***interp**

This thread's interpreter state.

void **PyEval_InitThreads**()

Parte da ABI Estável. Função descontinuada que não faz nada.

In Python 3.6 and older, this function created the GIL if it didn't exist.

Alterado na versão 3.9: The function now does nothing.

Alterado na versão 3.7: Esta função agora é chamada por *Py_Initialize()*, então não há mais necessidade de você chamá-la.

Alterado na versão 3.2: Esta função não pode mais ser chamada antes de *Py_Initialize()*.

Descontinuado desde a versão 3.9.

PyThreadState ***PyEval_SaveThread**()

Parte da ABI Estável. Release the global interpreter lock (if it has been created) and reset the thread state to NULL, returning the previous thread state (which is not NULL). If the lock has been created, the current thread must have acquired it.

void **PyEval_RestoreThread**(*PyThreadState* *tstate)

Parte da ABI Estável. Acquire the global interpreter lock (if it has been created) and set the thread state to *tstate*, which must not be NULL. If the lock has been created, the current thread must not have acquired it, otherwise deadlock ensues.

> **Nota**
>
> Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use *Py_IsFinalizing()* or sys.is_finalizing() to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

PyThreadState ***PyThreadState_Get**()

Parte da ABI Estável. 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).

See also *PyThreadState_GetUnchecked()*.

PyThreadState ***PyThreadState_GetUnchecked**()

Similar to *PyThreadState_Get()*, but don't kill the process with a fatal error if it is NULL. The caller is responsible to check if the result is NULL.

Adicionado na versão 3.13: In Python 3.5 to 3.12, the function was private and known as _PyThreadState_UncheckedGet().

PyThreadState ***PyThreadState_Swap**(*PyThreadState* *tstate)

Parte da ABI Estável. Swap the current thread state with the thread state given by the argument *tstate*, which may be NULL. The global interpreter lock must be held and is not released.

The following functions use thread-local storage, and are not compatible with sub-interpreters:

PyGILState_STATE **PyGILState_Ensure**()

Parte da ABI Estável. Certifique-se de que a thread atual esteja pronta para chamar a API Python C, independentemente do estado atual do Python ou da trava global do interpretador (GIL). Isso pode ser chamado quantas vezes desejar por uma thread, desde que cada chamada corresponda a uma chamada para *PyGILState_Release()*. Em geral, outras APIs relacionadas a threads podem ser usadas entre chamadas *PyGILState_Ensure()* e *PyGILState_Release()* desde que o estado da thread seja restaurado ao seu estado anterior antes de Release(). Por exemplo, o uso normal das macros *Py_BEGIN_ALLOW_THREADS* e *Py_END_ALLOW_THREADS* é aceitável.

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.

> **Nota**
>
> Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use *Py_IsFinalizing()* or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

void **PyGILState_Release**(PyGILState_STATE)

Parte da ABI Estável. Release any resources previously acquired. After this call, Python's state will be the same as it was prior to the corresponding *PyGILState_Ensure()* call (but generally this state will be unknown to the caller, hence the use of the GILState API).

Every call to *PyGILState_Ensure()* must be matched by a call to *PyGILState_Release()* on the same thread.

PyThreadState ***PyGILState_GetThisThreadState**()

Parte da ABI Estável. Get the current thread state for this thread. May return `NULL` if no GILState API has been used on the current thread. Note that the main thread always has such a thread-state, even if no auto-thread-state call has been made on the main thread. This is mainly a helper/diagnostic function.

int **PyGILState_Check**()

Return `1` if the current thread is holding the GIL and `0` otherwise. This function can be called from any thread at any time. Only if it has had its Python thread state initialized and currently is holding the GIL will it return `1`. This is mainly a helper/diagnostic function. It can be useful for example in callback contexts or memory allocation functions when knowing that the GIL is locked can allow the caller to perform sensitive actions or otherwise behave differently.

Adicionado na versão 3.4.

The following macros are normally used without a trailing semicolon; look for example usage in the Python source distribution.

Py_BEGIN_ALLOW_THREADS

Parte da ABI Estável. Esta macro se expande para `{ PyThreadState *_save; _save = PyEval_SaveThread();`. Observe que ele contém uma chave de abertura; ele deve ser combinado com a seguinte macro *Py_END_ALLOW_THREADS*. Veja acima para uma discussão mais aprofundada desta macro.

Py_END_ALLOW_THREADS

Parte da ABI Estável. Esta macro se expande para `PyEval_RestoreThread(_save); }`. Observe que ele contém uma chave de fechamento; ele deve ser combinado com uma macro *Py_BEGIN_ALLOW_THREADS* anterior. Veja acima para uma discussão mais aprofundada desta macro.

Py_BLOCK_THREADS

Parte da ABI Estável. Esta macro se expande para `PyEval_RestoreThread(_save);`: é equivalente a *Py_END_ALLOW_THREADS* sem a chave de fechamento.

Py_UNBLOCK_THREADS

Parte da ABI Estável. Esta macro se expande para `_save = PyEval_SaveThread();`: é equivalente a *Py_BEGIN_ALLOW_THREADS* sem a chave de abertura e declaração de variável.

9.5.5 Low-level API

All of the following functions must be called after *Py_Initialize()*.

Alterado na versão 3.7: *Py_Initialize()* now initializes the *GIL*.

PyInterpreterState ***PyInterpreterState_New**()

Parte da ABI Estável. 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.

Levanta um evento de auditoria `cpython.PyInterpreterState_New` sem argumentos.

void **PyInterpreterState_Clear**(*PyInterpreterState* *interp)

Parte da ABI Estável. Reset all information in an interpreter state object. The global interpreter lock must be held.

Levanta um evento de auditoria `cpython.PyInterpreterState_Clear` sem argumentos.

void **PyInterpreterState_Delete**(*PyInterpreterState* *interp)

Parte da ABI Estável. 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)

Parte da ABI Estável. 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)

Parte da ABI Estável. Reset all information in a thread state object. The global interpreter lock must be held.

Alterado na versão 3.9: This function now calls the `PyThreadState.on_delete` callback. Previously, that happened in *PyThreadState_Delete()*.

Alterado na versão 3.13: The `PyThreadState.on_delete` callback was removed.

void **PyThreadState_Delete**(*PyThreadState* *tstate)

Parte da ABI Estável. Destroy a thread state object. The global interpreter lock need not be held. The thread state must have been reset with a previous call to *PyThreadState_Clear()*.

void **PyThreadState_DeleteCurrent**(void)

Destroy the current thread state and release the global interpreter lock. Like *PyThreadState_Delete()*, the global interpreter lock must be held. The thread state must have been reset with a previous call to *PyThreadState_Clear()*.

PyFrameObject ***PyThreadState_GetFrame**(*PyThreadState* *tstate)

Parte da ABI Estável *desde a versão 3.10.* Get the current frame of the Python thread state *tstate*.

Return a *strong reference*. Return `NULL` if no frame is currently executing.

See also *PyEval_GetFrame()*.

tstate must not be `NULL`.

Adicionado na versão 3.9.

uint64_t **PyThreadState_GetID**(*PyThreadState* *tstate)

Parte da ABI Estável *desde a versão 3.10.* Get the unique thread state identifier of the Python thread state *tstate*.

tstate must not be `NULL`.

Adicionado na versão 3.9.

PyInterpreterState ***PyThreadState_GetInterpreter**(*PyThreadState* *tstate)

Parte da ABI Estável *desde a versão 3.10.* Get the interpreter of the Python thread state *tstate*.

tstate must not be `NULL`.

Adicionado na versão 3.9.

void **PyThreadState_EnterTracing**(*PyThreadState* *tstate)

Suspend tracing and profiling in the Python thread state *tstate*.

Resume them using the `PyThreadState_LeaveTracing()` function.

Adicionado na versão 3.11.

void **PyThreadState_LeaveTracing**(*PyThreadState* *tstate)

Resume tracing and profiling in the Python thread state *tstate* suspended by the `PyThreadState_EnterTracing()` function.

See also `PyEval_SetTrace()` and `PyEval_SetProfile()` functions.

Adicionado na versão 3.11.

PyInterpreterState ***PyInterpreterState_Get**(void)

Parte da ABI Estável *desde a versão 3.9.* Get the current interpreter.

Issue a fatal error if there no current Python thread state or no current interpreter. It cannot return NULL.

The caller must hold the GIL.

Adicionado na versão 3.9.

int64_t **PyInterpreterState_GetID**(*PyInterpreterState* *interp)

Parte da ABI Estável *desde a versão 3.7.* Return the interpreter's unique ID. If there was any error in doing so then `-1` is returned and an error is set.

The caller must hold the GIL.

Adicionado na versão 3.7.

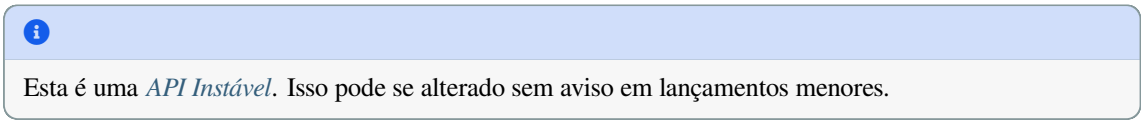
PyObject ***PyInterpreterState_GetDict**(*PyInterpreterState* *interp)

Parte da ABI Estável *desde a versão 3.8.* Return a dictionary in which interpreter-specific data may be stored. If this function returns `NULL` then no exception has been raised and the caller should assume no interpreter--specific dict is available.

This is not a replacement for `PyModule_GetState()`, which extensions should use to store interpreter--specific state information.

Adicionado na versão 3.8.

PyObject ***PyUnstable_InterpreterState_GetMainModule**(*PyInterpreterState* *interp)



Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Return a *strong reference* to the `__main__` *module object* for the given interpreter.

The caller must hold the GIL.

Adicionado na versão 3.13.

typedef *PyObject* *(***_PyFrameEvalFunction**)(*PyThreadState* *tstate, *_PyInterpreterFrame* *frame, int throwflag)

Type of a frame evaluation function.

The *throwflag* parameter is used by the `throw()` method of generators: if non-zero, handle the current exception.

Alterado na versão 3.9: The function now takes a *tstate* parameter.

Alterado na versão 3.11: The *frame* parameter changed from `PyFrameObject*` to `_PyInterpreterFrame*`.

_PyFrameEvalFunction **_PyInterpreterState_GetEvalFrameFunc**(*PyInterpreterState* *interp)

Get the frame evaluation function.

See the **PEP 523** "Adding a frame evaluation API to CPython".

Adicionado na versão 3.9.

void **_PyInterpreterState_SetEvalFrameFunc**(*PyInterpreterState* *interp, *_PyFrameEvalFunction* eval_frame)

Set the frame evaluation function.

See the **PEP 523** "Adding a frame evaluation API to CPython".

Adicionado na versão 3.9.

PyObject ***PyThreadState_GetDict**()

Retorna valor: Referência emprestada. *Parte da* ABI Estável. Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no current thread state is available. If this function returns `NULL`, no exception has been raised and the caller should assume no current thread state is available.

int **PyThreadState_SetAsyncExc**(unsigned long id, *PyObject* *exc)

Parte da ABI Estável. 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.

Alterado na versão 3.7: The type of the *id* parameter changed from `long` to `unsigned long`.

void **PyEval_AcquireThread**(*PyThreadState* *tstate)

Parte da ABI Estável. Acquire the global interpreter lock and set the current thread state to *tstate*, which must not be `NULL`. The lock must have been created earlier. If this thread already has the lock, deadlock ensues.

> **Nota**
>
> Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use *`Py_IsFinalizing()`* or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

Alterado na versão 3.8: Updated to be consistent with *`PyEval_RestoreThread()`*, *`Py_END_ALLOW_THREADS()`*, and *`PyGILState_Ensure()`*, and terminate the current thread if called while the interpreter is finalizing.

`PyEval_RestoreThread()` is a higher-level function which is always available (even when threads have not been initialized).

void **PyEval_ReleaseThread**(*PyThreadState* *tstate)

Parte da ABI Estável. Reset the current thread state to `NULL` and release the global interpreter lock. The lock must have been created earlier and must be held by the current thread. The *tstate* argument, which must not be `NULL`, is only used to check that it represents the current thread state — if it isn't, a fatal error is reported.

`PyEval_SaveThread()` is a higher-level function which is always available (even when threads have not been initialized).

9.6 Sub-interpreter support

While in most uses, you will only embed a single Python interpreter, there are cases where you need to create several independent interpreters in the same process and perhaps even in the same thread. Sub-interpreters allow you to do that.

The "main" interpreter is the first one created when the runtime initializes. It is usually the only Python interpreter in a process. Unlike sub-interpreters, the main interpreter has unique process-global responsibilities like signal handling. It is also responsible for execution during runtime initialization and is usually the active interpreter during runtime finalization. The *PyInterpreterState_Main()* function returns a pointer to its state.

You can switch between sub-interpreters using the *PyThreadState_Swap()* function. You can create and destroy them using the following functions:

type **PyInterpreterConfig**

Structure containing most parameters to configure a sub-interpreter. Its values are used only in *Py_NewInterpreterFromConfig()* and never modified by the runtime.

Adicionado na versão 3.12.

Campos de estrutura:

int **use_main_obmalloc**

If this is `0` then the sub-interpreter will use its own "object" allocator state. Otherwise it will use (share) the main interpreter's.

If this is `0` then *check_multi_interp_extensions* must be `1` (non-zero). If this is `1` then *gil* must not be *PyInterpreterConfig_OWN_GIL*.

int **allow_fork**

If this is `0` then the runtime will not support forking the process in any thread where the sub-interpreter is currently active. Otherwise fork is unrestricted.

Note that the `subprocess` module still works when fork is disallowed.

int **allow_exec**

If this is `0` then the runtime will not support replacing the current process via exec (e.g. `os.execv()`) in any thread where the sub-interpreter is currently active. Otherwise exec is unrestricted.

Note that the `subprocess` module still works when exec is disallowed.

int **allow_threads**

If this is `0` then the sub-interpreter's `threading` module won't create threads. Otherwise threads are allowed.

int **allow_daemon_threads**

If this is `0` then the sub-interpreter's `threading` module won't create daemon threads. Otherwise daemon threads are allowed (as long as *allow_threads* is non-zero).

int **check_multi_interp_extensions**

If this is `0` then all extension modules may be imported, including legacy (single-phase init) modules, in any thread where the sub-interpreter is currently active. Otherwise only multi-phase init extension modules (see **PEP 489**) may be imported. (Also see *Py_mod_multiple_interpreters*.)

This must be `1` (non-zero) if *use_main_obmalloc* is `0`.

int **gil**

This determines the operation of the GIL for the sub-interpreter. It may be one of the following:

PyInterpreterConfig_DEFAULT_GIL

Use the default selection (*PyInterpreterConfig_SHARED_GIL*).

PyInterpreterConfig_SHARED_GIL

Use (share) the main interpreter's GIL.

PyInterpreterConfig_OWN_GIL

Use the sub-interpreter's own GIL.

If this is *PyInterpreterConfig_OWN_GIL* then *PyInterpreterConfig.use_main_obmalloc* must be `0`.

PyStatus **Py_NewInterpreterFromConfig**(*PyThreadState* **tstate_p, const *PyInterpreterConfig* *config)

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 given *config* controls the options with which the interpreter is initialized.

Upon success, *tstate_p* will be set 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, *tstate_p* is set to `NULL`; 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. Likewise a current thread state must be set on entry. On success, the returned thread state will be set as current. If the sub-interpreter is created with its own GIL then the GIL of the calling interpreter will be released. When the function returns, the new interpreter's GIL will be held by the current thread and the previously interpreter's GIL will remain released here.

Adicionado na versão 3.12.

Sub-interpreters are most effective when isolated from each other, with certain functionality restricted:

```
PyInterpreterConfig config = {
    .use_main_obmalloc = 0,
    .allow_fork = 0,
    .allow_exec = 0,
    .allow_threads = 1,
    .allow_daemon_threads = 0,
    .check_multi_interp_extensions = 1,
    .gil = PyInterpreterConfig_OWN_GIL,
};
PyThreadState *tstate = NULL;
PyStatus status = Py_NewInterpreterFromConfig(&tstate, &config);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}
```

Note that the config is used only briefly and does not get modified. During initialization the config's values are converted into various *PyInterpreterState* values. A read-only copy of the config may be stored internally on the *PyInterpreterState*.

Extension modules are shared between (sub-)interpreters as follows:

- For modules using multi-phase initialization, e.g. *PyModule_FromDefAndSpec()*, a separate module object is created and initialized for each interpreter. Only C-level static and global variables are shared between these module objects.
- For modules using single-phase initialization, e.g. *PyModule_Create()*, the first time a particular extension is imported, it is initialized normally, and a (shallow) copy of its module's dictionary is squirreled away. When the same extension is imported by another (sub-)interpreter, a new module is initialized and filled with the contents of this copy; the extension's `init` function is not called. Objects in the module's dictionary thus end up shared across (sub-)interpreters, which might cause unwanted behavior (see *Bugs and caveats* below).

  Note that this is different from what happens when an extension is imported after the interpreter has been completely re-initialized by calling *Py_FinalizeEx()* and *Py_Initialize()*; in that case, the extension's `initmodule` function *is* called again. As with multi-phase initialization, this means that only C-level static and global variables are shared between these modules.

PyThreadState ***Py_NewInterpreter**(void)

Parte da ABI Estável. Create a new sub-interpreter. This is essentially just a wrapper around `Py_NewInterpreterFromConfig()` with a config that preserves the existing behavior. The result is an unisolated sub-interpreter that shares the main interpreter's GIL, allows fork/exec, allows daemon threads, and allows single-phase init modules.

void **Py_EndInterpreter**(*PyThreadState* *tstate)

Parte da ABI Estável. 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 used by the target interpreter must be held before calling this function. No GIL is held when it returns.

`Py_FinalizeEx()` will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

9.6.1 A Per-Interpreter GIL

Using `Py_NewInterpreterFromConfig()` you can create a sub-interpreter that is completely isolated from other interpreters, including having its own GIL. The most important benefit of this isolation is that such an interpreter can execute Python code without being blocked by other interpreters or blocking any others. Thus a single Python process can truly take advantage of multiple CPU cores when running Python code. The isolation also encourages a different approach to concurrency than that of just using threads. (See **PEP 554**.)

Using an isolated interpreter requires vigilance in preserving that isolation. That especially means not sharing any objects or mutable state without guarantees about thread-safety. Even objects that are otherwise immutable (e.g. `None`, `(1, 5)`) can't normally be shared because of the refcount. One simple but less-efficient approach around this is to use a global lock around all use of some state (or object). Alternately, effectively immutable objects (like integers or strings) can be made safe in spite of their refcounts by making them *immortal*. In fact, this has been done for the builtin singletons, small integers, and a number of other builtin objects.

Se você preservar o isolamento, terá acesso à computação multi-core adequada, sem as complicações que acompanham o uso de threads livres. A falha em preservar o isolamento traz a exposição a todas as consequências de threads livres, incluindo corridas e travamentos difíceis de depurar.

Aside from that, one of the main challenges of using multiple isolated interpreters is how to communicate between them safely (not break isolation) and efficiently. The runtime and stdlib do not provide any standard approach to this yet. A future stdlib module would help mitigate the effort of preserving isolation and expose effective tools for communicating (and sharing) data between interpreters.

Adicionado na versão 3.12.

9.6.2 Bugs and caveats

Because sub-interpreters (and the main interpreter) are part of the same process, the insulation between them isn't perfect — for example, using low-level file operations like `os.close()` they can (accidentally or maliciously) affect each other's open files. Because of the way extensions are shared between (sub-)interpreters, some extensions may not work properly; this is especially likely when using single-phase initialization or (static) global variables. It is possible to insert objects created in one sub-interpreter into a namespace of another (sub-)interpreter; this should be avoided if possible.

Special care should be taken to avoid sharing user-defined functions, methods, instances or classes between sub-interpreters, since import operations executed by such objects may affect the wrong (sub-)interpreter's dictionary of loaded modules. It is equally important to avoid sharing objects from which the above are reachable.

Also note that combining this functionality with `PyGILState_*` APIs is delicate, because these APIs assume a bijection between Python thread states and OS-level threads, an assumption broken by the presence of sub-interpreters. It is highly recommended that you don't switch sub-interpreters between a pair of matching `PyGILState_Ensure()` and `PyGILState_Release()` calls. Furthermore, extensions (such as `ctypes`) using these APIs to allow calling of Python code from non-Python created threads will probably be broken when using sub-interpreters.

9.7 Notificações assíncronas

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)

Parte da ABI Estável. Schedule a function to be called from the main interpreter thread. On success, `0` is returned and *func* is queued for being called in the main thread. On failure, `-1` is returned without setting any exception.

When successfully queued, *func* will be *eventually* called from the main interpreter thread with the argument *arg*. It will be called asynchronously with respect to normally running Python code, but with both these conditions met:

- on a *bytecode* boundary;
- with the main thread holding the *global interpreter lock* (*func* can therefore use the full C API).

func must return `0` on success, or `-1` on failure with an exception set. *func* won't be interrupted to perform another asynchronous notification recursively, but it can still be interrupted to switch threads if the global interpreter lock is released.

This function doesn't need a current thread state to run, and it doesn't need the global interpreter lock.

To call this function in a subinterpreter, the caller must hold the GIL. Otherwise, the function *func* can be scheduled to be called from the wrong interpreter.

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*.

Adicionado na versão 3.1.

Alterado na versão 3.9: If this function is called in a subinterpreter, the function *func* is now scheduled to be called from the subinterpreter, rather than being called from the main interpreter. Each subinterpreter now has its own list of scheduled calls.

9.8 Profiling and Tracing

The Python interpreter provides some low-level support for attaching profiling and execution tracing facilities. These are used for profiling, debugging, and coverage analysis tools.

This C interface allows the profiling or tracing code to avoid the overhead of calling through Python-level callable objects, making a direct C function call instead. The essential attributes of the facility have not changed; the interface allows trace functions to be installed per-thread, and the basic events reported to the trace function are the same as had been reported to the Python-level trace functions in previous versions.

typedef int (***Py_tracefunc**)(*PyObject* *obj, *PyFrameObject* *frame, int what, *PyObject* *arg)

The type of the trace function registered using `PyEval_SetProfile()` and `PyEval_SetTrace()`. The first parameter is the object passed to the registration function as *obj*, *frame* is the frame object to which the event pertains, *what* is one of the constants `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION`, `PyTrace_C_RETURN`, or `PyTrace_OPCODE`, and *arg* depends on the value of *what*:

Value of *what*	Meaning of *arg*
PyTrace_CALL	Always *Py_None*.
PyTrace_EXCEPTION	Exception information as returned by `sys.exc_info()`.
PyTrace_LINE	Always *Py_None*.
PyTrace_RETURN	Value being returned to the caller, or `NULL` if caused by an exception.
PyTrace_C_CALL	Function object being called.
PyTrace_C_EXCEPTION	Function object being called.
PyTrace_C_RETURN	Function object being called.
PyTrace_OPCODE	Always *Py_None*.

int **PyTrace_CALL**

The value of the *what* parameter to a *Py_tracefunc* function when a new call to a function or method is being reported, or a new entry into a generator. Note that the creation of the iterator for a generator function is not reported as there is no control transfer to the Python bytecode in the corresponding frame.

int **PyTrace_EXCEPTION**

The value of the *what* parameter to a *Py_tracefunc* function when an exception has been raised. The callback function is called with this value for *what* when after any bytecode is processed after which the exception becomes set within the frame being executed. The effect of this is that as exception propagation causes the Python stack to unwind, the callback is called upon return to each frame as the exception propagates. Only trace functions receives these events; they are not needed by the profiler.

int **PyTrace_LINE**

The value passed as the *what* parameter to a *Py_tracefunc* function (but not a profiling function) when a line-number event is being reported. It may be disabled for a frame by setting `f_trace_lines` to *0* on that frame.

int **PyTrace_RETURN**

The value for the *what* parameter to *Py_tracefunc* functions when a call is about to return.

int **PyTrace_C_CALL**

The value for the *what* parameter to *Py_tracefunc* functions when a C function is about to be called.

int **PyTrace_C_EXCEPTION**

The value for the *what* parameter to *Py_tracefunc* functions when a C function has raised an exception.

int **PyTrace_C_RETURN**

The value for the *what* parameter to *Py_tracefunc* functions when a C function has returned.

int **PyTrace_OPCODE**

The value for the *what* parameter to *Py_tracefunc* functions (but not profiling functions) when a new opcode is about to be executed. This event is not emitted by default: it must be explicitly requested by setting `f_trace_opcodes` to *1* on the frame.

void **PyEval_SetProfile**(*Py_tracefunc* func, *PyObject* *obj)

Set the profiler function to *func*. The *obj* parameter is passed to the function as its first parameter, and may be any Python object, or `NULL`. If the profile function needs to maintain state, using a different value for *obj* for each thread provides a convenient and thread-safe place to store it. The profile function is called for all monitored events except *PyTrace_LINE* *PyTrace_OPCODE* and *PyTrace_EXCEPTION*.

See also the `sys.setprofile()` function.

The caller must hold the *GIL*.

void **PyEval_SetProfileAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

Like *PyEval_SetProfile()* but sets the profile function in all running threads belonging to the current interpreter instead of the setting it only on the current thread.

The caller must hold the *GIL*.

As *PyEval_SetProfile()*, this function ignores any exceptions raised while setting the profile functions in all threads.

Adicionado na versão 3.12.

void **PyEval_SetTrace**(*Py_tracefunc* func, *PyObject* *obj)

Set the tracing function to *func*. This is similar to *PyEval_SetProfile()*, except the tracing function does receive line-number events and per-opcode events, but does not receive any event related to C function objects being called. Any trace function registered using *PyEval_SetTrace()* will not receive *PyTrace_C_CALL*, *PyTrace_C_EXCEPTION* or *PyTrace_C_RETURN* as a value for the *what* parameter.

See also the `sys.settrace()` function.

The caller must hold the *GIL*.

void **PyEval_SetTraceAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

Like *PyEval_SetTrace()* but sets the tracing function in all running threads belonging to the current interpreter instead of the setting it only on the current thread.

The caller must hold the *GIL*.

As *PyEval_SetTrace()*, this function ignores any exceptions raised while setting the trace functions in all threads.

Adicionado na versão 3.12.

9.9 Reference tracing

Adicionado na versão 3.13.

typedef int (***PyRefTracer**)(*PyObject**, int event, void *data)

The type of the trace function registered using *PyRefTracer_SetTracer()*. The first parameter is a Python object that has been just created (when **event** is set to *PyRefTracer_CREATE*) or about to be destroyed (when **event** is set to *PyRefTracer_DESTROY*). The **data** argument is the opaque pointer that was provided when *PyRefTracer_SetTracer()* was called.

Adicionado na versão 3.13.

int **PyRefTracer_CREATE**

The value for the *event* parameter to *PyRefTracer* functions when a Python object has been created.

int **PyRefTracer_DESTROY**

The value for the *event* parameter to *PyRefTracer* functions when a Python object has been destroyed.

int **PyRefTracer_SetTracer**(*PyRefTracer* tracer, void *data)

Register a reference tracer function. The function will be called when a new Python has been created or when an object is going to be destroyed. If **data** is provided it must be an opaque pointer that will be provided when the tracer function is called. Return `0` on success. Set an exception and return `-1` on error.

Not that tracer functions **must not** create Python objects inside or otherwise the call will be re-entrant. The tracer also **must not** clear any existing exception or set an exception. The GIL will be held every time the tracer function is called.

The GIL must be held when calling this function.

Adicionado na versão 3.13.

PyRefTracer **PyRefTracer_GetTracer**(void **data)

Get the registered reference tracer function and the value of the opaque data pointer that was registered when *PyRefTracer_SetTracer()* was called. If no tracer was registered this function will return NULL and will set the **data** pointer to NULL.

The GIL must be held when calling this function.

Adicionado na versão 3.13.

9.10 Advanced Debugger Support

These functions are only intended to be used by advanced debugging tools.

PyInterpreterState ***PyInterpreterState_Head**()

Return the interpreter state object at the head of the list of all such objects.

PyInterpreterState ***PyInterpreterState_Main**()

Return the main interpreter state object.

PyInterpreterState ***PyInterpreterState_Next**(*PyInterpreterState* *interp)

Return the next interpreter state object after *interp* from the list of all such objects.

PyThreadState ***PyInterpreterState_ThreadHead**(*PyInterpreterState* *interp)

Return the pointer to the first `PyThreadState` object in the list of threads associated with the interpreter *interp*.

PyThreadState ***PyThreadState_Next**(*PyThreadState* *tstate)

Return the next thread state object after *tstate* from the list of all such objects belonging to the same `PyInterpreterState` object.

9.11 Thread Local Storage Support

The Python interpreter provides low-level support for thread-local storage (TLS) which wraps the underlying native TLS implementation to support the Python-level thread local storage API (`threading.local`). The CPython C level APIs are similar to those offered by pthreads and Windows: use a thread key and functions to associate a `void*` value per thread.

The GIL does *not* need to be held when calling these functions; they supply their own locking.

Note that `Python.h` does not include the declaration of the TLS APIs, you need to include `pythread.h` to use thread-local storage.

Nota

None of these API functions handle memory management on behalf of the `void*` values. You need to allocate and deallocate them yourself. If the `void*` values happen to be `PyObject*`, these functions don't do refcount operations on them either.

9.11.1 Thread Specific Storage (TSS) API

TSS API is introduced to supersede the use of the existing TLS API within the CPython interpreter. This API uses a new type `Py_tss_t` instead of `int` to represent thread keys.

Adicionado na versão 3.7.

Ver também

"A New C-API for Thread-Local Storage in CPython" (**PEP 539**)

type **Py_tss_t**

This data structure represents the state of a thread key, the definition of which may depend on the underlying TLS implementation, and it has an internal field representing the key's initialization state. There are no public members in this structure.

Quando *Py_LIMITED_API* não é definido, a alocação estática deste tipo por `Py_tss_NEEDS_INIT` é permitida.

Py_tss_NEEDS_INIT

This macro expands to the initializer for *Py_tss_t* variables. Note that this macro won't be defined with *Py_LIMITED_API*.

Alocação dinâmica

Dynamic allocation of the *Py_tss_t*, required in extension modules built with *Py_LIMITED_API*, where static allocation of this type is not possible due to its implementation being opaque at build time.

Py_tss_t ***PyThread_tss_alloc**()

Parte da ABI Estável *desde a versão 3.7.* Retorna um valor que é o mesmo estado de um valor inicializado com *Py_tss_NEEDS_INIT*, ou `NULL` no caso de falha de alocação dinâmica.

void **PyThread_tss_free**(*Py_tss_t* *key)

Parte da ABI Estável *desde a versão 3.7.* Free the given *key* allocated by *PyThread_tss_alloc()*, after first calling *PyThread_tss_delete()* to ensure any associated thread locals have been unassigned. This is a no-op if the *key* argument is `NULL`.

> **Nota**
>
> A freed key becomes a dangling pointer. You should reset the key to `NULL`.

Métodos

The parameter *key* of these functions must not be `NULL`. Moreover, the behaviors of *PyThread_tss_set()* and *PyThread_tss_get()* are undefined if the given *Py_tss_t* has not been initialized by *PyThread_tss_create()*.

int **PyThread_tss_is_created**(*Py_tss_t* *key)

Parte da ABI Estável *desde a versão 3.7.* Return a non-zero value if the given *Py_tss_t* has been initialized by *PyThread_tss_create()*.

int **PyThread_tss_create**(*Py_tss_t* *key)

Parte da ABI Estável *desde a versão 3.7.* Retorna um valor zero na inicialização bem-sucedida de uma chave TSS. O comportamento é indefinido se o valor apontado pelo argumento *key* não for inicializado por *Py_tss_NEEDS_INIT*. Essa função pode ser chamada repetidamente na mesma tecla – chamá-la em uma tecla já inicializada não funciona e retorna imediatamente com sucesso.

void **PyThread_tss_delete**(*Py_tss_t* *key)

Parte da ABI Estável *desde a versão 3.7.* Destroy a TSS key to forget the values associated with the key across all threads, and change the key's initialization state to uninitialized. A destroyed key is able to be initialized again by *PyThread_tss_create()*. This function can be called repeatedly on the same key – calling it on an already destroyed key is a no-op.

int **PyThread_tss_set**(*Py_tss_t* *key, void *value)

Parte da ABI Estável *desde a versão 3.7.* Return a zero value to indicate successfully associating a `void*` value with a TSS key in the current thread. Each thread has a distinct mapping of the key to a `void*` value.

void ***PyThread_tss_get**(*Py_tss_t* *key)

Parte da ABI Estável *desde a versão 3.7.* Return the `void*` value associated with a TSS key in the current thread. This returns `NULL` if no value is associated with the key in the current thread.

9.11.2 Thread Local Storage (TLS) API

Descontinuado desde a versão 3.7: This API is superseded by *Thread Specific Storage (TSS) API*.

> **Nota**
>
> This version of the API does not support platforms where the native TLS key is defined in a way that cannot be safely cast to `int`. On such platforms, *PyThread_create_key()* will return immediately with a failure status, and the other TLS functions will all be no-ops on such platforms.

Due to the compatibility problem noted above, this version of the API should not be used in new code.

int **PyThread_create_key**()

Parte da ABI Estável.

void **PyThread_delete_key**(int key)

Parte da ABI Estável.

int **PyThread_set_key_value**(int key, void *value)

Parte da ABI Estável.

void ***PyThread_get_key_value**(int key)

Parte da ABI Estável.

void **PyThread_delete_key_value**(int key)

Parte da ABI Estável.

void **PyThread_ReInitTLS**()

Parte da ABI Estável.

9.12 Synchronization Primitives

The C-API provides a basic mutual exclusion lock.

type **PyMutex**

A mutual exclusion lock. The `PyMutex` should be initialized to zero to represent the unlocked state. For example:

```
PyMutex mutex = {0};
```

Instances of `PyMutex` should not be copied or moved. Both the contents and address of a `PyMutex` are meaningful, and it must remain at a fixed, writable location in memory.

> **Nota**
>
> A `PyMutex` currently occupies one byte, but the size should be considered unstable. The size may change in future Python releases without a deprecation period.

Adicionado na versão 3.13.

void **PyMutex_Lock**(*PyMutex* *m)

Lock mutex *m*. If another thread has already locked it, the calling thread will block until the mutex is unlocked. While blocked, the thread will temporarily release the *GIL* if it is held.

Adicionado na versão 3.13.

void **PyMutex_Unlock**(*PyMutex* *m)

Unlock mutex *m*. The mutex must be locked — otherwise, the function will issue a fatal error.

Adicionado na versão 3.13.

9.12.1 Python Critical Section API

The critical section API provides a deadlock avoidance layer on top of per-object locks for *free-threaded* CPython. They are intended to replace reliance on the *global interpreter lock*, and are no-ops in versions of Python with the global interpreter lock.

Critical sections avoid deadlocks by implicitly suspending active critical sections and releasing the locks during calls to `PyEval_SaveThread()`. When `PyEval_RestoreThread()` is called, the most recent critical section is resumed, and its locks reacquired. This means the critical section API provides weaker guarantees than traditional locks – they are useful because their behavior is similar to the *GIL*.

The functions and structs used by the macros are exposed for cases where C macros are not available. They should only be used as in the given macro expansions. Note that the sizes and contents of the structures may change in future Python versions.

Nota

Operations that need to lock two objects at once must use `Py_BEGIN_CRITICAL_SECTION2`. You *cannot* use nested critical sections to lock more than one object at once, because the inner critical section may suspend the outer critical sections. This API does not provide a way to lock more than two objects at once.

Exemplo de uso:

```
static PyObject *
set_field(MyObject *self, PyObject *value)
{
   Py_BEGIN_CRITICAL_SECTION(self);
   Py_SETREF(self->field, Py_XNewRef(value));
   Py_END_CRITICAL_SECTION();
   Py_RETURN_NONE;
}
```

In the above example, `Py_SETREF` calls `Py_DECREF`, which can call arbitrary code through an object's deallocation function. The critical section API avoids potential deadlocks due to reentrancy and lock ordering by allowing the runtime to temporarily suspend the critical section if the code triggered by the finalizer blocks and calls `PyEval_SaveThread()`.

Py_BEGIN_CRITICAL_SECTION(op)

Acquires the per-object lock for the object *op* and begins a critical section.

In the free-threaded build, this macro expands to:

```
{
    PyCriticalSection _py_cs;
    PyCriticalSection_Begin(&_py_cs, (PyObject*)(op))
```

In the default build, this macro expands to `{`.

Adicionado na versão 3.13.

Py_END_CRITICAL_SECTION()

Ends the critical section and releases the per-object lock.

In the free-threaded build, this macro expands to:

```
    PyCriticalSection_End(&_py_cs);
}
```

In the default build, this macro expands to `}`.

Adicionado na versão 3.13.

Py_BEGIN_CRITICAL_SECTION2(a, b)

Acquires the per-objects locks for the objects *a* and *b* and begins a critical section. The locks are acquired in a consistent order (lowest address first) to avoid lock ordering deadlocks.

In the free-threaded build, this macro expands to:

```
{
    PyCriticalSection2 _py_cs2;
    PyCriticalSection2_Begin(&_py_cs2, (PyObject*)(a), (PyObject*)(b))
```

In the default build, this macro expands to `{`.

Adicionado na versão 3.13.

Py_END_CRITICAL_SECTION2()

Ends the critical section and releases the per-object locks.

In the free-threaded build, this macro expands to:

```
    PyCriticalSection2_End(&_py_cs2);
}
```

In the default build, this macro expands to `}`.

Adicionado na versão 3.13.

CAPÍTULO 10

Configuração de Inicialização do Python

Adicionado na versão 3.8.

Python pode ser inicializado com `Py_InitializeFromConfig()` e a estrutura `PyConfig`. Pode ser pré-inicializado com `Py_PreInitialize()` e a estrutura `PyPreConfig`.

Existem dois tipos de configuração:

- A *Python Configuration* pode ser usada para construir um Python personalizado que se comporta como um Python comum. Por exemplo, variáveis de ambiente e argumento de linha de comando são usados para configurar Python.
- A *Configuração isolada* pode ser usada para incorporar Python em uma aplicação. Isso isola Python de um sistema. Por exemplo, variáveis de ambiente são ignoradas, a variável local LC_CTYPE fica inalterada e nenhum manipulador de sinal é registrado.

A função `Py_RunMain()` pode ser usada para escrever um programa Python personalizado.

Veja também *Inicialização, Finalização e Threads*.

Ver também

PEP 587 "Configuração da inicialização do Python".

10.1 Exemplo

Exemplo de Python personalizado sendo executado sempre em um modo isolado:

```
int main(int argc, char **argv)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config.isolated = 1;

    /* Decode command line arguments.
       Implicitly preinitialize Python (in isolated mode). */
```

(continua na próxima página)

(continuação da página anterior)

```
    status = PyConfig_SetBytesArgv(&config, argc, argv);
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);

    return Py_RunMain();

exception:
    PyConfig_Clear(&config);
    if (PyStatus_IsExit(status)) {
        return status.exitcode;
    }
    /* Display the error message and exit the process with
       non-zero exit code */
    Py_ExitStatusException(status);
}
```

10.2 PyWideStringList

type **PyWideStringList**

Lista de strings `wchar_t*`.

Se *length* é diferente de zero, *items* deve ser diferente de `NULL` e todas as strings devem ser diferentes de `NULL`.

Métodos:

PyStatus **PyWideStringList_Append**(*PyWideStringList* *list, const wchar_t *item)

Anexa *item* a *list*.

Python deve ser inicializado previamente antes de chamar essa função.

PyStatus **PyWideStringList_Insert**(*PyWideStringList* *list, *Py_ssize_t* index, const wchar_t *item)

Insere *item* na *list* na posição *index*.

Se *index* for maior ou igual ao comprimento da *list*, anexa o *item* a *list*.

index deve ser maior que ou igual a `0`.

Python deve ser inicializado previamente antes de chamar essa função.

Campos de estrutura:

Py_ssize_t **length**

Comprimento da lista.

wchar_t ***items**

Itens da lista.

10.3 PyStatus

type **PyStatus**

Estrutura para armazenar o status de uma função de inicialização: sucesso, erro ou saída.

Para um erro, ela pode armazenar o nome da função C que criou o erro.

Campos de estrutura:

int **exitcode**

Código de saída. Argumento passado para `exit()`.

const char ***err_msg**

Mensagem de erro.

const char ***func**

Nome da função que criou um erro. Pode ser `NULL`.

Funções para criar um status:

PyStatus **PyStatus_Ok**(void)

Sucesso.

PyStatus **PyStatus_Error**(const char *err_msg)

Erro de inicialização com uma mensagem.

err_msg não deve ser `NULL`.

PyStatus **PyStatus_NoMemory**(void)

Falha de alocação de memória (sem memória).

PyStatus **PyStatus_Exit**(int exitcode)

Sai do Python com o código de saída especificado.

Funções para manipular um status:

int **PyStatus_Exception**(*PyStatus* status)

O status é um erro ou uma saída? Se verdadeiro, a exceção deve ser tratada; chamando *Py_ExitStatusException()*, por exemplo.

int **PyStatus_IsError**(*PyStatus* status)

O resultado é um erro?

int **PyStatus_IsExit**(*PyStatus* status)

O resultado é uma saída?

void **Py_ExitStatusException**(*PyStatus* status)

Chama `exit(exitcode)` se *status* for uma saída. Exibe a mensagem de erro e sai com um código de saída diferente de zero se *status* for um erro. Deve ser chamado apenas se `PyStatus_Exception(status)` for diferente de zero.

Nota

Internamente, Python usa macros que definem `PyStatus.func`, enquanto funções para criar um status definem `func` para `NULL`.

Exemplo:

```
PyStatus alloc(void **ptr, size_t size)
{
    *ptr = PyMem_RawMalloc(size);
    if (*ptr == NULL) {
        return PyStatus_NoMemory();
    }
```

(continua na próxima página)

(continuação da página anterior)

```
    return PyStatus_Ok();
}

int main(int argc, char **argv)
{
    void *ptr;
    PyStatus status = alloc(&ptr, 16);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
    PyMem_Free(ptr);
    return 0;
}
```

10.4 PyPreConfig

type **PyPreConfig**

Estrutura usada para pré-inicializar o Python.

A função para inicializar uma pré-configuração:

void **PyPreConfig_InitPythonConfig**(*PyPreConfig* *preconfig)

Inicializa a pré-configuração com *Configuração do Python*.

void **PyPreConfig_InitIsolatedConfig**(*PyPreConfig* *preconfig)

Inicializa a pré-configuração com *Configuração isolada*.

Campos de estrutura:

int **allocator**

Nome de alocadores de memória em Python:

- `PYMEM_ALLOCATOR_NOT_SET` (`0`): não altera os alocadores de memória (usa o padrão).
- `PYMEM_ALLOCATOR_DEFAULT` (`1`): *alocadores de memória padrão*.
- `PYMEM_ALLOCATOR_DEBUG` (`2`): *alocadores de memória padrão* com *ganchos de depuração*.
- `PYMEM_ALLOCATOR_MALLOC` (`3`): usa `malloc()` da biblioteca C.
- `PYMEM_ALLOCATOR_MALLOC_DEBUG` (`4`): força uso de `malloc()` com *ganchos de depuração*.
- `PYMEM_ALLOCATOR_PYMALLOC` (`5`): *alocador de memória do Python pymalloc*.
- `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` (`6`): *alocador de memória deo Python pymalloc* com *ganchos de depuração*.
- `PYMEM_ALLOCATOR_MIMALLOC` (`6`): usa `mimalloc`, um substituto rápido do malloc.
- `PYMEM_ALLOCATOR_MIMALLOC_DEBUG` (`7`): use `mimalloc`, um substituto rápido do malloc com *ganchos de depuração*.

`PYMEM_ALLOCATOR_PYMALLOC` e `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` não são suportados se o Python estiver `configurado usando --without-pymalloc`.

`PYMEM_ALLOCATOR_MIMALLOC` e `PYMEM_ALLOCATOR_MIMALLOC_DEBUG` não são suportados se o Python estiver `configurado usando --without-mimalloc` ou se suporte atômico subjacente não estiver disponível.

Veja *Gerenciamento de memória*.

Padrão: `PYMEM_ALLOCATOR_NOT_SET`.

int **configure_locale**

Define a localidade LC_CTYPE para a localidade preferida do usuário.

Se igual a `0`, define os membros *coerce_c_locale* e *coerce_c_locale_warn* para `0`.

Veja a *codificação da localidade*.

Padrão: `1` na configuração do Python, `0` na configuração isolada.

int **coerce_c_locale**

Se igual a `2`, força a localidade para C.

Se for igual a `1`, lê a localidade LC_CTYPE para decidir se deve ser forçado.

Veja a *codificação da localidade*.

Padrão: `-1` na configuração do Python, `0` na configuração isolada.

int **coerce_c_locale_warn**

Se diferente de zero, emite um aviso se a localidade C for forçada.

Padrão: `-1` na configuração do Python, `0` na configuração isolada.

int **dev_mode**

Modo de desenvolvimento do Python: veja *PyConfig.dev_mode*.

Padrão: `-1` no modo do Python, `0` no modo isolado.

int **isolated**

Modo isolado: veja *PyConfig.isolated*.

Padrão: `0` no modo do Python, `1` no modo isolado.

int **legacy_windows_fs_encoding**

Se não zero:

- Define *PyPreConfig.utf8_mode* para `0`,
- Define *PyConfig.filesystem_encoding* para `"mbcs"`,
- Define *PyConfig.filesystem_errors* para `"replace"`.

Initialized from the `PYTHONLEGACYWINDOWSFSENCODING` environment variable value.

Disponível apenas no Windows. A macro `#ifdef MS_WINDOWS` pode ser usada para código específico do Windows.

Padrão: `0`.

int **parse_argv**

Se diferente de zero, *Py_PreInitializeFromArgs()* e *Py_PreInitializeFromBytesArgs()* analisam seu argumento `argv` da mesma forma que o Python regular analisa argumentos de linha de comando: veja Argumentos de linha de comando.

Padrão: `1` na configuração do Python, `0` na configuração isolada.

int **use_environment**

Usar variáveis de ambiente? Veja *PyConfig.use_environment*.

Padrão: `1` na configuração do Python e `0` na configuração isolada.

int **utf8_mode**

Se não zero, habilita o modo UTF-8 do Python.

Define para `0` ou `1` pela opção de linha de comando `-X utf8` e a variável de ambiente `PYTHONUTF8`.

Também define como `1` se a localidade `LC_CTYPE` for `C` ou `POSIX`.

Padrão: `-1` na configuração do Python e `0` na configuração isolada.

10.5 Pré-inicializar Python com PyPreConfig

A pré-inicialização do Python:

- Define os alocadores de memória Python (*PyPreConfig.allocator*)
- Configura a localidade LC_CTYPE (*codificação da localidade*)
- Define o Modo UTF-8 do Python (*PyPreConfig.utf8_mode*)

A pré-configuração atual (tipo `PyPreConfig`) é armazenada em `_PyRuntime.preconfig`.

Funções para pré-inicializar Python:

PyStatus **Py_PreInitialize**(const *PyPreConfig* *preconfig)

Pré-inicializa o Python a partir da pré-configuração *preconfig*.

preconfig não pode ser `NULL`..

PyStatus **Py_PreInitializeFromBytesArgs**(const *PyPreConfig* *preconfig, int argc, char *const *argv)

Pré-inicializa o Python a partir da pré-configuração *preconfig*.

Analisa argumentos de linha de comando *argv* (strings de bytes) se *parse_argv* de *preconfig* for diferente de zero.

preconfig não pode ser `NULL`..

PyStatus **Py_PreInitializeFromArgs**(const *PyPreConfig* *preconfig, int argc, wchar_t *const *argv)

Pré-inicializa o Python a partir da pré-configuração *preconfig*.

Analisa argumentos de linha de comando *argv* (strings largas) se *parse_argv* de *preconfig* for diferente de zero.

preconfig não pode ser `NULL`..

O chamador é responsável por manipular exceções (erro ou saída) usando *PyStatus_Exception()* e *Py_ExitStatusException()*.

Para *configuração do Python* (*PyPreConfig_InitPythonConfig()*), se o Python for inicializado com argumentos de linha de comando, os argumentos de linha de comando também devem ser passados para pré-inicializar o Python, pois eles têm um efeito na pré-configuração como codificações. Por exemplo, a opção de linha de comando `-X utf8` habilita o Modo UTF-8 do Python.

`PyMem_SetAllocator()` pode ser chamado depois de *Py_PreInitialize()* e antes de *Py_InitializeFromConfig()* para instalar um alocador de memória personalizado. Ele pode ser chamado antes de *Py_PreInitialize()* se *PyPreConfig.allocator* estiver definido como `PYMEM_ALLOCATOR_NOT_SET`.

Funções de alocação de memória do Python como *PyMem_RawMalloc()* não devem ser usadas antes da pré-inicialização do Python, enquanto chamar diretamente `malloc()` e `free()` é sempre seguro. *Py_DecodeLocale()* não deve ser chamado antes da pré-inicialização do Python.

Exemplo usando a pré-inicialização para habilitar o modo UTF-8 do Python.

```
PyStatus status;
PyPreConfig preconfig;
PyPreConfig_InitPythonConfig(&preconfig);

preconfig.utf8_mode = 1;

status = Py_PreInitialize(&preconfig);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}

/* at this point, Python speaks UTF-8 */
```

(continua na próxima página)

(continuação da página anterior)

```
Py_Initialize();
/* ... use Python API here ... */
Py_Finalize();
```

10.6 PyConfig

type **PyConfig**

Estrutura contendo a maioria dos parâmetros para configurar o Python.

Ao terminar, a função *PyConfig_Clear()* deve ser usada para liberar a memória de configuração.

Structure methods:

void **PyConfig_InitPythonConfig**(*PyConfig* *config)

Initialize configuration with the *Python Configuration*.

void **PyConfig_InitIsolatedConfig**(*PyConfig* *config)

Initialize configuration with the *Isolated Configuration*.

PyStatus **PyConfig_SetString**(*PyConfig* *config, wchar_t *const *config_str, const wchar_t *str)

Copy the wide character string *str* into `*config_str`.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesString**(*PyConfig* *config, wchar_t *const *config_str, const char *str)

Decode *str* using *Py_DecodeLocale()* and set the result into `*config_str`.

Preinitialize Python if needed.

PyStatus **PyConfig_SetArgv**(*PyConfig* *config, int argc, wchar_t *const *argv)

Set command line arguments (*argv* member of *config*) from the *argv* list of wide character strings.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesArgv**(*PyConfig* *config, int argc, char *const *argv)

Set command line arguments (*argv* member of *config*) from the *argv* list of bytes strings. Decode bytes using *Py_DecodeLocale()*.

Preinitialize Python if needed.

PyStatus **PyConfig_SetWideStringList**(*PyConfig* *config, *PyWideStringList* *list, *Py_ssize_t* length, wchar_t **items)

Set the list of wide strings *list* to *length* and *items*.

Preinitialize Python if needed.

PyStatus **PyConfig_Read**(*PyConfig* *config)

Read all Python configuration.

Fields which are already initialized are left unchanged.

Fields for *path configuration* are no longer calculated or modified when calling this function, as of Python 3.11.

The *PyConfig_Read()* function only parses *PyConfig.argv* arguments once: *PyConfig.parse_argv* is set to `2` after arguments are parsed. Since Python arguments are stripped from *PyConfig.argv*, parsing arguments twice would parse the application options as Python options.

Preinitialize Python if needed.

Alterado na versão 3.10: The *PyConfig.argv* arguments are now only parsed once, *PyConfig.parse_argv* is set to `2` after arguments are parsed, and arguments are only parsed if *PyConfig.parse_argv* equals `1`.

Alterado na versão 3.11: *PyConfig_Read()* no longer calculates all paths, and so fields listed under *Python Path Configuration* may no longer be updated until *Py_InitializeFromConfig()* is called.

void **PyConfig_Clear**(*PyConfig* *config)

Release configuration memory.

Most PyConfig methods *preinitialize Python* if needed. In that case, the Python preinitialization configuration (*PyPreConfig*) in based on the *PyConfig*. If configuration fields which are in common with *PyPreConfig* are tuned, they must be set before calling a *PyConfig* method:

- *PyConfig.dev_mode*
- *PyConfig.isolated*
- *PyConfig.parse_argv*
- *PyConfig.use_environment*

Moreover, if *PyConfig_SetArgv()* or *PyConfig_SetBytesArgv()* is used, this method must be called before other methods, since the preinitialization configuration depends on command line arguments (if *parse_argv* is non-zero).

The caller of these methods is responsible to handle exceptions (error or exit) using PyStatus_Exception() and Py_ExitStatusException().

Campos de estrutura:

PyWideStringList **argv**

Set sys.argv command line arguments based on *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.

Set *parse_argv* to 1 to parse *argv* the same way the regular Python parses Python command line arguments and then to strip Python arguments from *argv*.

If *argv* is empty, an empty string is added to ensure that sys.argv always exists and is never empty.

Padrão: NULL.

See also the *orig_argv* member.

int **safe_path**

If equals to zero, Py_RunMain() prepends a potentially unsafe path to sys.path at startup:

- If *argv[0]* is equal to L"-m" (python -m module), prepend the current working directory.
- If running a script (python script.py), prepend the script's directory. If it's a symbolic link, resolve symbolic links.
- Otherwise (python -c code and python), prepend an empty string, which means the current working directory.

Set to 1 by the -P command line option and the PYTHONSAFEPATH environment variable.

Default: 0 in Python config, 1 in isolated config.

Adicionado na versão 3.11.

wchar_t ***base_exec_prefix**

sys.base_exec_prefix.

Padrão: NULL.

Part of the *Python Path Configuration* output.

See also *PyConfig.exec_prefix*.

wchar_t ***base_executable**

Python base executable: `sys._base_executable`.

Set by the `__PYVENV_LAUNCHER__` environment variable.

Set from *PyConfig.executable* if `NULL`.

Padrão: `NULL`.

Part of the *Python Path Configuration* output.

See also *PyConfig.executable*.

wchar_t ***base_prefix**

`sys.base_prefix`.

Padrão: `NULL`.

Part of the *Python Path Configuration* output.

See also *PyConfig.prefix*.

int **buffered_stdio**

If equals to `0` and *configure_c_stdio* is non-zero, disable buffering on the C streams stdout and stderr.

Set to `0` by the `-u` command line option and the `PYTHONUNBUFFERED` environment variable.

stdin is always opened in buffered mode.

Padrão: `1`.

int **bytes_warning**

If equals to `1`, issue a warning when comparing `bytes` or `bytearray` with `str`, or comparing `bytes` with `int`.

If equal or greater to `2`, raise a `BytesWarning` exception in these cases.

Incremented by the `-b` command line option.

Padrão: `0`.

int **warn_default_encoding**

If non-zero, emit a `EncodingWarning` warning when `io.TextIOWrapper` uses its default encoding. See io-encoding-warning for details.

Padrão: `0`.

Adicionado na versão 3.10.

int **code_debug_ranges**

If equals to `0`, disables the inclusion of the end line and column mappings in code objects. Also disables traceback printing carets to specific error locations.

Set to `0` by the `PYTHONNODEBUGRANGES` environment variable and by the `-X no_debug_ranges` command line option.

Padrão: `1`.

Adicionado na versão 3.11.

wchar_t ***check_hash_pycs_mode**

Control the validation behavior of hash-based `.pyc` files: value of the `--check-hash-based-pycs` command line option.

Valores válidos:

- `L"always"`: Hash the source file for invalidation regardless of value of the 'check_source' flag.
- `L"never"`: Assume that hash-based pycs always are valid.

- `L"default"`: The 'check_source' flag in hash-based pycs determines invalidation.

Default: `L"default"`.

See also **PEP 552** "Deterministic pycs".

int **configure_c_stdio**

If non-zero, configure C standard streams:

- On Windows, set the binary mode (`O_BINARY`) on stdin, stdout and stderr.
- If *buffered_stdio* equals zero, disable buffering of stdin, stdout and stderr streams.
- If *interactive* is non-zero, enable stream buffering on stdin and stdout (only stdout on Windows).

Padrão: `1` na configuração do Python, `0` na configuração isolada.

int **dev_mode**

If non-zero, enable the Python Development Mode.

Set to `1` by the `-X dev` option and the `PYTHONDEVMODE` environment variable.

Padrão: `-1` no modo do Python, `0` no modo isolado.

int **dump_refs**

Dump Python references?

If non-zero, dump all objects which are still alive at exit.

Set to `1` by the `PYTHONDUMPREFS` environment variable.

Needs a special build of Python with the `Py_TRACE_REFS` macro defined: see the `configure --with-trace-refs option`.

Padrão: `0`.

wchar_t ***exec_prefix**

The site-specific directory prefix where the platform-dependent Python files are installed: `sys.exec_prefix`.

Padrão: `NULL`.

Part of the *Python Path Configuration* output.

See also *PyConfig.base_exec_prefix*.

wchar_t ***executable**

The absolute path of the executable binary for the Python interpreter: `sys.executable`.

Padrão: `NULL`.

Part of the *Python Path Configuration* output.

See also *PyConfig.base_executable*.

int **faulthandler**

Enable faulthandler?

If non-zero, call `faulthandler.enable()` at startup.

Set to `1` by `-X faulthandler` and the `PYTHONFAULTHANDLER` environment variable.

Padrão: `-1` no modo do Python, `0` no modo isolado.

wchar_t ***filesystem_encoding**

Filesystem encoding: `sys.getfilesystemencoding()`.

On macOS, Android and VxWorks: use `"utf-8"` by default.

On Windows: use `"utf-8"` by default, or `"mbcs"` if *legacy_windows_fs_encoding* of *PyPreConfig* is non-zero.

Default encoding on other platforms:

- `"utf-8"` if *PyPreConfig.utf8_mode* is non-zero.
- `"ascii"` if Python detects that `nl_langinfo(CODESET)` announces the ASCII encoding, whereas the `mbstowcs()` function decodes from a different encoding (usually Latin1).
- `"utf-8"` if `nl_langinfo(CODESET)` returns an empty string.
- Otherwise, use the *locale encoding*: `nl_langinfo(CODESET)` result.

At Python startup, the encoding name is normalized to the Python codec name. For example, `"ANSI_X3.4-1968"` is replaced with `"ascii"`.

See also the *filesystem_errors* member.

wchar_t ***filesystem_errors**

Filesystem error handler: `sys.getfilesystemencodeerrors()`.

On Windows: use `"surrogatepass"` by default, or `"replace"` if *legacy_windows_fs_encoding* of *PyPreConfig* is non-zero.

On other platforms: use `"surrogateescape"` by default.

Supported error handlers:

- `"strict"`
- `"surrogateescape"`
- `"surrogatepass"` (only supported with the UTF-8 encoding)

See also the *filesystem_encoding* member.

unsigned long **hash_seed**

int **use_hash_seed**

Randomized hash function seed.

If *use_hash_seed* is zero, a seed is chosen randomly at Python startup, and *hash_seed* is ignored.

Set by the `PYTHONHASHSEED` environment variable.

Default *use_hash_seed* value: `-1` in Python mode, `0` in isolated mode.

wchar_t ***home**

Set the default Python "home" directory, that is, the location of the standard Python libraries (see `PYTHONHOME`).

Set by the `PYTHONHOME` environment variable.

Padrão: `NULL`.

Part of the *Python Path Configuration* input.

int **import_time**

If non-zero, profile import time.

Set the `1` by the `-X importtime` option and the `PYTHONPROFILEIMPORTTIME` environment variable.

Padrão: `0`.

int **inspect**

Enter interactive mode after executing a script or a command.

If greater than `0`, enable inspect: when a script is passed as first argument or the -c option is used, enter interactive mode after executing the script or the command, even when `sys.stdin` does not appear to be a terminal.

Incremented by the `-i` command line option. Set to `1` if the `PYTHONINSPECT` environment variable is non-empty.

Padrão: `0`.

int **install_signal_handlers**

Install Python signal handlers?

Default: `1` in Python mode, `0` in isolated mode.

int **interactive**

If greater than `0`, enable the interactive mode (REPL).

Incremented by the `-i` command line option.

Padrão: `0`.

int **int_max_str_digits**

Configures the integer string conversion length limitation. An initial value of `-1` means the value will be taken from the command line or environment or otherwise default to 4300 (`sys.int_info.default_max_str_digits`). A value of `0` disables the limitation. Values greater than zero but less than 640 (`sys.int_info.str_digits_check_threshold`) are unsupported and will produce an error.

Configured by the `-X int_max_str_digits` command line flag or the `PYTHONINTMAXSTRDIGITS` environment variable.

Default: `-1` in Python mode. 4300 (`sys.int_info.default_max_str_digits`) in isolated mode.

Adicionado na versão 3.12.

int **cpu_count**

If the value of *cpu_count* is not `-1` then it will override the return values of `os.cpu_count()`, `os.process_cpu_count()`, and `multiprocessing.cpu_count()`.

Configured by the `-X cpu_count=`*n*`|default` command line flag or the `PYTHON_CPU_COUNT` environment variable.

Default: `-1`.

Adicionado na versão 3.13.

int **isolated**

If greater than `0`, enable isolated mode:

- Set *safe_path* to `1`: don't prepend a potentially unsafe path to `sys.path` at Python startup, such as the current directory, the script's directory or an empty string.
- Set *use_environment* to `0`: ignore `PYTHON` environment variables.
- Set *user_site_directory* to `0`: don't add the user site directory to `sys.path`.
- Python REPL doesn't import `readline` nor enable default readline configuration on interactive prompts.

Set to `1` by the `-I` command line option.

Padrão: `0` no modo do Python, `1` no modo isolado.

See also the *Isolated Configuration* and *PyPreConfig.isolated*.

int **legacy_windows_stdio**

If non-zero, use `io.FileIO` instead of `io._WindowsConsoleIO` for `sys.stdin`, `sys.stdout` and `sys.stderr`.

Definida como `1` se a variável de ambiente `PYTHONLEGACYWINDOWSSTDIO` estiver definida como uma string não vazia.

Disponível apenas no Windows. A macro `#ifdef MS_WINDOWS` pode ser usada para código específico do Windows.

Padrão: `0`.

See also the **PEP 528** (Change Windows console encoding to UTF-8).

int **malloc_stats**

If non-zero, dump statistics on *Python pymalloc memory allocator* at exit.

Set to `1` by the `PYTHONMALLOCSTATS` environment variable.

The option is ignored if Python is `configured using the --without-pymalloc option`.

Padrão: `0`.

wchar_t ***platlibdir**

Platform library directory name: `sys.platlibdir`.

Set by the `PYTHONPLATLIBDIR` environment variable.

Default: value of the `PLATLIBDIR` macro which is set by the `configure --with-platlibdir option` (default: `"lib"`, or `"DLLs"` on Windows).

Part of the *Python Path Configuration* input.

Adicionado na versão 3.9.

Alterado na versão 3.11: This macro is now used on Windows to locate the standard library extension modules, typically under `DLLs`. However, for compatibility, note that this value is ignored for any non-standard layouts, including in-tree builds and virtual environments.

wchar_t ***pythonpath_env**

Module search paths (`sys.path`) as a string separated by `DELIM` (`os.pathsep`).

Set by the `PYTHONPATH` environment variable.

Padrão: `NULL`.

Part of the *Python Path Configuration* input.

PyWideStringList **module_search_paths**

int **module_search_paths_set**

Module search paths: `sys.path`.

If *module_search_paths_set* is equal to `0`, *Py_InitializeFromConfig()* will replace *module_search_paths* and sets *module_search_paths_set* to `1`.

Default: empty list (`module_search_paths`) and `0` (`module_search_paths_set`).

Part of the *Python Path Configuration* output.

int **optimization_level**

Compilation optimization level:

- `0`: Peephole optimizer, set `__debug__` to `True`.
- `1`: Level 0, remove assertions, set `__debug__` to `False`.
- `2`: Level 1, strip docstrings.

Incremented by the `-O` command line option. Set to the `PYTHONOPTIMIZE` environment variable value.

Padrão: `0`.

PyWideStringList **orig_argv**

The list of the original command line arguments passed to the Python executable: `sys.orig_argv`.

If *orig_argv* list is empty and *argv* is not a list only containing an empty string, *PyConfig_Read()* copies *argv* into *orig_argv* before modifying *argv* (if *parse_argv* is non-zero).

See also the *argv* member and the *Py_GetArgcArgv()* function.

Padrão: lista vazia.

Adicionado na versão 3.10.

int **parse_argv**

Parse command line arguments?

If equals to `1`, parse *argv* the same way the regular Python parses command line arguments, and strip Python arguments from *argv*.

The *PyConfig_Read()* function only parses *PyConfig.argv* arguments once: *PyConfig.parse_argv* is set to `2` after arguments are parsed. Since Python arguments are stripped from *PyConfig.argv*, parsing arguments twice would parse the application options as Python options.

Default: `1` in Python mode, `0` in isolated mode.

Alterado na versão 3.10: The *PyConfig.argv* arguments are now only parsed if *PyConfig.parse_argv* equals to `1`.

int **parser_debug**

Parser debug mode. If greater than `0`, turn on parser debugging output (for expert only, depending on compilation options).

Incremented by the `-d` command line option. Set to the `PYTHONDEBUG` environment variable value.

Needs a debug build of Python (the `Py_DEBUG` macro must be defined).

Padrão: `0`.

int **pathconfig_warnings**

If non-zero, calculation of path configuration is allowed to log warnings into `stderr`. If equals to `0`, suppress these warnings.

Default: `1` in Python mode, `0` in isolated mode.

Part of the *Python Path Configuration* input.

Alterado na versão 3.11: Now also applies on Windows.

wchar_t ***prefix**

The site-specific directory prefix where the platform independent Python files are installed: `sys.prefix`.

Padrão: `NULL`.

Part of the *Python Path Configuration* output.

See also *PyConfig.base_prefix*.

wchar_t ***program_name**

Program name used to initialize *executable* and in early error messages during Python initialization.

- On macOS, use `PYTHONEXECUTABLE` environment variable if set.
- If the `WITH_NEXT_FRAMEWORK` macro is defined, use `__PYVENV_LAUNCHER__` environment variable if set.
- Use `argv[0]` of *argv* if available and non-empty.
- Otherwise, use `L"python"` on Windows, or `L"python3"` on other platforms.

Padrão: `NULL`.

Part of the *Python Path Configuration* input.

wchar_t ***pycache_prefix**

Directory where cached `.pyc` files are written: `sys.pycache_prefix`.

Set by the `-X pycache_prefix=PATH` command line option and the `PYTHONPYCACHEPREFIX` environment variable. The command-line option takes precedence.

If `NULL`, `sys.pycache_prefix` is set to `None`.

Padrão: `NULL`.

int **quiet**

Quiet mode. If greater than `0`, don't display the copyright and version at Python startup in interactive mode.

Incremented by the `-q` command line option.

Padrão: `0`.

wchar_t ***run_command**

Value of the `-c` command line option.

Used by *Py_RunMain()*.

Padrão: `NULL`.

wchar_t ***run_filename**

Filename passed on the command line: trailing command line argument without `-c` or `-m`. It is used by the *Py_RunMain()* function.

For example, it is set to `script.py` by the `python3 script.py arg` command line.

See also the *PyConfig.skip_source_first_line* option.

Padrão: `NULL`.

wchar_t ***run_module**

Value of the `-m` command line option.

Used by *Py_RunMain()*.

Padrão: `NULL`.

wchar_t ***run_presite**

`package.module` path to module that should be imported before `site.py` is run.

Set by the `-X presite=package.module` command-line option and the `PYTHON_PRESITE` environment variable. The command-line option takes precedence.

Needs a debug build of Python (the `Py_DEBUG` macro must be defined).

Padrão: `NULL`.

int **show_ref_count**

Show total reference count at exit (excluding *immortal* objects)?

Set to `1` by `-X showrefcount` command line option.

Needs a debug build of Python (the `Py_REF_DEBUG` macro must be defined).

Padrão: `0`.

int **site_import**

Import the `site` module at startup?

If equal to zero, disable the import of the module site and the site-dependent manipulations of `sys.path` that it entails.

Also disable these manipulations if the `site` module is explicitly imported later (call `site.main()` if you want them to be triggered).

Set to `0` by the `-S` command line option.

`sys.flags.no_site` is set to the inverted value of *site_import*.

Padrão: `1`.

int **skip_source_first_line**

If non-zero, skip the first line of the *PyConfig.run_filename* source.

It allows the usage of non-Unix forms of `#!cmd`. This is intended for a DOS specific hack only.

Set to `1` by the `-x` command line option.

Padrão: `0`.

wchar_t ***stdio_encoding**

wchar_t ***stdio_errors**

Encoding and encoding errors of `sys.stdin`, `sys.stdout` and `sys.stderr` (but `sys.stderr` always uses `"backslashreplace"` error handler).

Use the `PYTHONIOENCODING` environment variable if it is non-empty.

Codificação padrão:

- `"UTF-8"` if *PyPreConfig.utf8_mode* is non-zero.
- Otherwise, use the *locale encoding*.

Tratador de erros padrão:

- On Windows: use `"surrogateescape"`.
- `"surrogateescape"` if *PyPreConfig.utf8_mode* is non-zero, or if the LC_CTYPE locale is "C" or "POSIX".
- `"strict"` otherwise.

See also *PyConfig.legacy_windows_stdio*.

int **tracemalloc**

Enable tracemalloc?

If non-zero, call `tracemalloc.start()` at startup.

Set by `-X tracemalloc=N` command line option and by the `PYTHONTRACEMALLOC` environment variable.

Padrão: `-1` no modo do Python, `0` no modo isolado.

int **perf_profiling**

Enable compatibility mode with the perf profiler?

If non-zero, initialize the perf trampoline. See perf_profiling for more information.

Set by `-X perf` command-line option and by the `PYTHON_PERF_JIT_SUPPORT` environment variable for perf support with stack pointers and `-X perf_jit` command-line option and by the `PYTHON_PERF_JIT_SUPPORT` environment variable for perf support with DWARF JIT information.

Default: `-1`.

Adicionado na versão 3.12.

int **use_environment**

Use environment variables?

If equals to zero, ignore the environment variables.

Set to `0` by the `-E` environment variable.

Padrão: `1` na configuração do Python e `0` na configuração isolada.

int **user_site_directory**

If non-zero, add the user site directory to `sys.path`.

Set to `0` by the `-s` and `-I` command line options.

Set to `0` by the `PYTHONNOUSERSITE` environment variable.

Default: `1` in Python mode, `0` in isolated mode.

int **verbose**

Verbose mode. If greater than `0`, print a message each time a module is imported, showing the place (filename or built-in module) from which it is loaded.

If greater than or equal to `2`, print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Incremented by the `-v` command line option.

Set by the `PYTHONVERBOSE` environment variable value.

Padrão: `0`.

PyWideStringList **warnoptions**

Options of the `warnings` module to build warnings filters, lowest to highest priority: `sys.warnoptions`.

The `warnings` module adds `sys.warnoptions` in the reverse order: the last *PyConfig.warnoptions* item becomes the first item of `warnings.filters` which is checked first (highest priority).

The `-W` command line options adds its value to *warnoptions*, it can be used multiple times.

The `PYTHONWARNINGS` environment variable can also be used to add warning options. Multiple options can be specified, separated by commas (`,`).

Padrão: lista vazia.

int **write_bytecode**

If equal to `0`, Python won't try to write `.pyc` files on the import of source modules.

Set to `0` by the `-B` command line option and the `PYTHONDONTWRITEBYTECODE` environment variable.

`sys.dont_write_bytecode` is initialized to the inverted value of *write_bytecode*.

Padrão: `1`.

PyWideStringList **xoptions**

Values of the `-X` command line options: `sys._xoptions`.

Padrão: lista vazia.

If *parse_argv* is non-zero, *argv* arguments are parsed the same way the regular Python parses command line arguments, and Python arguments are stripped from *argv*.

The *xoptions* options are parsed to set other options: see the `-X` command line option.

Alterado na versão 3.9: The `show_alloc_count` field has been removed.

10.7 Initialization with PyConfig

Initializing the interpreter from a populated configuration struct is handled by calling *Py_InitializeFromConfig()*.

O chamador é responsável por manipular exceções (erro ou saída) usando *PyStatus_Exception()* e *Py_ExitStatusException()*.

If *PyImport_FrozenModules()*, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* are used, they must be set or called after Python preinitialization and before the Python initialization. If Python is initialized multiple times, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* must be called before each Python initialization.

The current configuration (`PyConfig` type) is stored in `PyInterpreterState.config`.

Example setting the program name:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name. Implicitly preinitialize Python. */
    status = PyConfig_SetString(&config, &config.program_name,
                                L"/path/to/my_program");
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);
    return;

exception:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}
```

More complete example modifying the default configuration, read the configuration, and then override some parameters. Note that since 3.11, many parameters are not calculated until initialization, and so values cannot be read from the configuration structure. Any values set before initialize is called will be left unchanged by initialization:

```
PyStatus init_python(const char *program_name)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name before reading the configuration
       (decode byte string from the locale encoding).

       Implicitly preinitialize Python. */
    status = PyConfig_SetBytesString(&config, &config.program_name,
                                     program_name);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Read all configuration at once */
    status = PyConfig_Read(&config);
    if (PyStatus_Exception(status)) {
```

(continua na próxima página)

(continuação da página anterior)

```
        goto done;
    }

    /* Specify sys.path explicitly */
    /* If you want to modify the default set of paths, finish
       initialization first and then use PySys_GetObject("path") */
    config.module_search_paths_set = 1;
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/stdlib");
    if (PyStatus_Exception(status)) {
        goto done;
    }
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/more/modules");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Override executable computed by PyConfig_Read() */
    status = PyConfig_SetString(&config, &config.executable,
                                L"/path/to/my_executable");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    status = Py_InitializeFromConfig(&config);

done:
    PyConfig_Clear(&config);
    return status;
}
```

10.8 Isolated Configuration

PyPreConfig_InitIsolatedConfig() and *PyConfig_InitIsolatedConfig()* functions create a configuration to isolate Python from the system. For example, to embed Python into an application.

This configuration ignores global configuration variables, environment variables, command line arguments (*PyConfig.argv* is not parsed) and user site directory. The C standard streams (ex: `stdout`) and the LC_CTYPE locale are left unchanged. Signal handlers are not installed.

Configuration files are still used with this configuration to determine paths that are unspecified. Ensure *PyConfig.home* is specified to avoid computing the default path configuration.

10.9 Configuração do Python

PyPreConfig_InitPythonConfig() and *PyConfig_InitPythonConfig()* functions create a configuration to build a customized Python which behaves as the regular Python.

Environments variables and command line arguments are used to configure Python, whereas global configuration variables are ignored.

This function enables C locale coercion (**PEP 538**) and Python UTF-8 Mode (**PEP 540**) depending on the LC_CTYPE locale, `PYTHONUTF8` and `PYTHONCOERCECLOCALE` environment variables.

10.10 Python Path Configuration

PyConfig contains multiple fields for the path configuration:

- Path configuration inputs:
  - *PyConfig.home*
  - *PyConfig.platlibdir*
  - *PyConfig.pathconfig_warnings*
  - *PyConfig.program_name*
  - *PyConfig.pythonpath_env*
  - current working directory: to get absolute paths
  - PATH environment variable to get the program full path (from *PyConfig.program_name*)
  - __PYVENV_LAUNCHER__ environment variable
  - (Windows only) Application paths in the registry under "SoftwarePythonPythonCoreX.YPythonPath" of HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE (where X.Y is the Python version).
- Path configuration output fields:
  - *PyConfig.base_exec_prefix*
  - *PyConfig.base_executable*
  - *PyConfig.base_prefix*
  - *PyConfig.exec_prefix*
  - *PyConfig.executable*
  - *PyConfig.module_search_paths_set*, *PyConfig.module_search_paths*
  - *PyConfig.prefix*

If at least one "output field" is not set, Python calculates the path configuration to fill unset fields. If *module_search_paths_set* is equal to 0, *module_search_paths* is overridden and *module_search_paths_set* is set to 1.

It is possible to completely ignore the function calculating the default path configuration by setting explicitly all path configuration output fields listed above. A string is considered as set even if it is non-empty. module_search_paths is considered as set if module_search_paths_set is set to 1. In this case, module_search_paths will be used without modification.

Set *pathconfig_warnings* to 0 to suppress warnings when calculating the path configuration (Unix only, Windows does not log any warning).

If *base_prefix* or *base_exec_prefix* fields are not set, they inherit their value from *prefix* and *exec_prefix* respectively.

Py_RunMain() and *Py_Main()* modify sys.path:

- If *run_filename* is set and is a directory which contains a __main__.py script, prepend *run_filename* to sys.path.
- If *isolated* is zero:
  - If *run_module* is set, prepend the current directory to sys.path. Do nothing if the current directory cannot be read.
  - If *run_filename* is set, prepend the directory of the filename to sys.path.
  - Otherwise, prepend an empty string to sys.path.

If *site_import* is non-zero, `sys.path` can be modified by the `site` module. If *user_site_directory* is non-zero and the user's site-package directory exists, the `site` module appends the user's site-package directory to `sys.path`.

The following configuration files are used by the path configuration:

- `pyvenv.cfg`
- `._pth` file (ex: `python._pth`)
- `pybuilddir.txt` (Unix only)

If a `._pth` file is present:

- Set *isolated* to `1`.
- Set *use_environment* to `0`.
- Set *site_import* to `0`.
- Set *safe_path* to `1`.

The `__PYVENV_LAUNCHER__` environment variable is used to set *PyConfig.base_executable*.

10.11 Py_GetArgcArgv()

void **Py_GetArgcArgv**(int *argc, wchar_t ***argv)

Get the original command line arguments, before Python modified them.

See also *PyConfig.orig_argv* member.

10.12 Multi-Phase Initialization Private Provisional API

This section is a private provisional API introducing multi-phase initialization, the core feature of **PEP 432**:

- "Core" initialization phase, "bare minimum Python":
  - Builtin types;
  - Builtin exceptions;
  - Builtin and frozen modules;
  - The `sys` module is only partially initialized (ex: `sys.path` doesn't exist yet).
- "Main" initialization phase, Python is fully initialized:
  - Install and configure `importlib`;
  - Apply the *Path Configuration*;
  - Install signal handlers;
  - Finish `sys` module initialization (ex: create `sys.stdout` and `sys.path`);
  - Enable optional features like `faulthandler` and `tracemalloc`;
  - Import the `site` module;
  - etc.

Private provisional API:

- `PyConfig._init_main`: if set to `0`, *Py_InitializeFromConfig()* stops at the "Core" initialization phase.

PyStatus **_Py_InitializeMain**(void)

Move to the "Main" initialization phase, finish the Python initialization.

No module is imported during the "Core" phase and the `importlib` module is not configured: the *Path Configuration* is only applied during the "Main" phase. It may allow to customize Python in Python to override or tune the *Path Configuration*, maybe install a custom `sys.meta_path` importer or an import hook, etc.

It may become possible to calculate the *Path Configuration* in Python, after the Core phase and before the Main phase, which is one of the **PEP 432** motivation.

The "Core" phase is not properly defined: what should be and what should not be available at this phase is not specified yet. The API is marked as private and provisional: the API can be modified or even be removed anytime until a proper public API is designed.

Example running Python code between "Core" and "Main" initialization phases:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config._init_main = 0;

    /* ... customize 'config' configuration ... */

    status = Py_InitializeFromConfig(&config);
    PyConfig_Clear(&config);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }

    /* Use sys.stderr because sys.stdout is only created
       by _Py_InitializeMain() */
    int res = PyRun_SimpleString(
        "import sys; "
        "print('Run Python code before _Py_InitializeMain', "
               "file=sys.stderr)");
    if (res < 0) {
        exit(1);
    }

    /* ... put more configuration code here ... */

    status = _Py_InitializeMain();
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
}
```

CAPÍTULO 11

Gerenciamento de Memória

11.1 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 = PyBytes_FromString(buf);
free(buf); /* malloc'ed */
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the bytes object returned as a result.

In most situations, however, it is recommended to allocate memory from the Python heap specifically because the latter is under control of the Python memory manager. For example, this is required when the interpreter is extended with new object types written in C. Another reason for using the Python heap is the desire to *inform* the Python memory manager about the memory needs of the extension module. Even when the requested memory is used exclusively for internal, highly specific purposes, delegating all memory requests to the Python memory manager causes the interpreter to have a more accurate image of its memory footprint as a whole. Consequently, under certain circumstances, the Python memory manager may or may not trigger appropriate actions, like garbage collection, memory compaction or other preventive procedures. Note that by using the C library allocator as shown in the previous example, the allocated memory for the I/O buffer escapes completely the Python memory manager.

Ver também

The `PYTHONMALLOC` environment variable can be used to configure the memory allocators used by Python.

The `PYTHONMALLOCSTATS` environment variable can be used to print statistics of the *pymalloc memory allocator* every time a new pymalloc object arena is created, and on shutdown.

11.2 Allocator Domains

All allocating functions belong to one of three different "domains" (see also `PyMemAllocatorDomain`). These domains represent different allocation strategies and are optimized for different purposes. The specific details on how every domain allocates memory or what internal functions each domain calls is considered an implementation detail, but for debugging purposes a simplified table can be found at *here*. The APIs used to allocate and free a block of memory must be from the same domain. For example, `PyMem_Free()` must be used to free memory allocated using `PyMem_Malloc()`.

The three allocation domains are:

- Raw domain: intended for allocating memory for general-purpose memory buffers where the allocation *must* go to the system allocator or where the allocator can operate without the *GIL*. The memory is requested directly from the system. See *Raw Memory Interface*.
- "Mem" domain: intended for allocating memory for Python buffers and general-purpose memory buffers where the allocation must be performed with the *GIL* held. The memory is taken from the Python private heap. See *Memory Interface*.
- Object domain: intended for allocating memory for Python objects. The memory is taken from the Python private heap. See *Object allocators*.

Nota

The *free-threaded* build requires that only Python objects are allocated using the "object" domain and that all Python objects are allocated using that domain. This differs from the prior Python versions, where this was only a best practice and not a hard requirement.

For example, buffers (non-Python objects) should be allocated using `PyMem_Malloc()`, `PyMem_RawMalloc()`, or `malloc()`, but not `PyObject_Malloc()`.

See Memory Allocation APIs.

11.3 Raw Memory Interface

The following function sets are wrappers to the system allocator. These functions are thread-safe, the *GIL* does not need to be held.

The *default raw memory allocator* uses the following functions: `malloc()`, `calloc()`, `realloc()` and `free()`; call `malloc(1)` (or `calloc(1, 1)`) when requesting zero bytes.

Adicionado na versão 3.4.

void *__PyMem_RawMalloc__(size_t n)

Parte da ABI Estável *desde a versão 3.13.* Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawMalloc(1)` had been called instead. The memory will not have been initialized in any way.

void *__PyMem_RawCalloc__(size_t nelem, size_t elsize)

Parte da ABI Estável *desde a versão 3.13.* Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawCalloc(1, 1)` had been called instead.

Adicionado na versão 3.5.

void *__PyMem_RawRealloc__(void *p, size_t n)

Parte da ABI Estável *desde a versão 3.13.* Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyMem_RawMalloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to *`PyMem_RawMalloc()`*, *`PyMem_RawRealloc()`* or *`PyMem_RawCalloc()`*.

If the request fails, *`PyMem_RawRealloc()`* returns `NULL` and *p* remains a valid pointer to the previous memory area.

void __PyMem_RawFree__(void *p)

Parte da ABI Estável *desde a versão 3.13.* Frees the memory block pointed to by *p*, which must have been returned by a previous call to *`PyMem_RawMalloc()`*, *`PyMem_RawRealloc()`* or *`PyMem_RawCalloc()`*. Otherwise, or if `PyMem_RawFree(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

11.4 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.

The *default memory allocator* uses the *pymalloc memory allocator*.

Aviso

The *GIL* must be held when using these functions.

Alterado na versão 3.6: The default allocator is now pymalloc instead of system `malloc()`.

void *__PyMem_Malloc__(size_t n)

Parte da ABI Estável. Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void *__PyMem_Calloc__(size_t nelem, size_t elsize)

Parte da ABI Estável *desde a versão 3.7.* Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_Calloc(1, 1)` had been called instead.

Adicionado na versão 3.5.

void ***PyMem_Realloc**(void *p, size_t n)

Parte da ABI Estável. Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyMem_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to *PyMem_Malloc()*, *PyMem_Realloc()* or *PyMem_Calloc()*.

If the request fails, *PyMem_Realloc()* returns `NULL` and *p* remains a valid pointer to the previous memory area.

void **PyMem_Free**(void *p)

Parte da ABI Estável. Frees the memory block pointed to by *p*, which must have been returned by a previous call to *PyMem_Malloc()*, *PyMem_Realloc()* or *PyMem_Calloc()*. Otherwise, or if `PyMem_Free(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

The following type-oriented macros are provided for convenience. Note that *TYPE* refers to any C type.

PyMem_New(TYPE, n)

Same as *PyMem_Malloc()*, but allocates `(n * sizeof(TYPE))` bytes of memory. Returns a pointer cast to `TYPE*`. The memory will not have been initialized in any way.

PyMem_Resize(p, TYPE, n)

Same as *PyMem_Realloc()*, but the memory block is resized to `(n * sizeof(TYPE))` bytes. Returns a pointer cast to `TYPE*`. On return, *p* will be a pointer to the new memory area, or `NULL` in the event of failure.

This is a C preprocessor macro; *p* is always reassigned. Save the original value of *p* to avoid losing memory when handling errors.

void **PyMem_Del**(void *p)

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(size)`
- `PyMem_NEW(type, size)`
- `PyMem_REALLOC(ptr, size)`
- `PyMem_RESIZE(ptr, type, size)`
- `PyMem_FREE(ptr)`
- `PyMem_DEL(ptr)`

11.5 Alocadores de objeto

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.

Nota

There is no guarantee that the memory returned by these allocators can be successfully cast to a Python object when intercepting the allocating functions in this domain by the methods described in the *Customize Memory Allocators* section.

The *default object allocator* uses the *pymalloc memory allocator*.

Aviso

The *GIL* must be held when using these functions.

void *`PyObject_Malloc`(size_t n)

Parte da ABI Estável. Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void *`PyObject_Calloc`(size_t nelem, size_t elsize)

Parte da ABI Estável *desde a versão 3.7.* Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Calloc(1, 1)` had been called instead.

Adicionado na versão 3.5.

void *`PyObject_Realloc`(void *p, size_t n)

Parte da ABI Estável. 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)

Parte da ABI Estável. Frees the memory block pointed to by *p*, which must have been returned by a previous call to *`PyObject_Malloc()`*, *`PyObject_Realloc()`* or *`PyObject_Calloc()`*. Otherwise, or if `PyObject_Free(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

11.6 Alocadores de memória padrão

Alocadores de memória padrão:

Configuração	Nome	PyMem_RawMalloc	PyMem_Malloc	PyObject_Malloc
Release build	`"pymalloc"`	`malloc`	`pymalloc`	`pymalloc`
Debug build	`"pymalloc_debug`	`malloc` + debug	`pymalloc` + debug	`pymalloc` + debug
Release build, without pymalloc	`"malloc"`	`malloc`	`malloc`	`malloc`
Debug build, without pymalloc	`"malloc_debug"`	`malloc` + debug	`malloc` + debug	`malloc` + debug

Legend:

- Name: value for `PYTHONMALLOC` environment variable.
- `malloc`: system allocators from the standard C library, C functions: `malloc()`, `calloc()`, `realloc()` and `free()`.
- `pymalloc`: *pymalloc memory allocator*.
- `mimalloc`: *mimalloc memory allocator*. The pymalloc allocator will be used if mimalloc support isn't available.
- "+ debug": with *debug hooks on the Python memory allocators*.
- "Debug build": Python build in debug mode.

11.7 Alocadores de memória

Adicionado na versão 3.4.

type **PyMemAllocatorEx**

Structure used to describe a memory block allocator. The structure has the following fields:

Campo	Significado
`void *ctx`	user context passed as first argument
`void* malloc(void *ctx, size_t size)`	allocate a memory block
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	allocate a memory block initialized with zeros
`void* realloc(void *ctx, void *ptr, size_t new_size)`	allocate or resize a memory block
`void free(void *ctx, void *ptr)`	free a memory block

Alterado na versão 3.5: The `PyMemAllocator` structure was renamed to *`PyMemAllocatorEx`* and a new `calloc` field was added.

type **PyMemAllocatorDomain**

Enum used to identify an allocator domain. Domains:

PYMEM_DOMAIN_RAW

Funções:

- *`PyMem_RawMalloc()`*
- *`PyMem_RawRealloc()`*
- *`PyMem_RawCalloc()`*
- *`PyMem_RawFree()`*

PYMEM_DOMAIN_MEM

Funções:

- *PyMem_Malloc()*,
- *PyMem_Realloc()*
- *PyMem_Calloc()*
- *PyMem_Free()*

PYMEM_DOMAIN_OBJ

Funções:

- *PyObject_Malloc()*
- *PyObject_Realloc()*
- *PyObject_Calloc()*
- *PyObject_Free()*

void **PyMem_GetAllocator**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

Get the memory block allocator of the specified domain.

void **PyMem_SetAllocator**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

Set the memory block allocator of the specified domain.

The new allocator must return a distinct non-`NULL` pointer when requesting zero bytes.

For the *PYMEM_DOMAIN_RAW* domain, the allocator must be thread-safe: the *GIL* is not held when the allocator is called.

For the remaining domains, the allocator must also be thread-safe: the allocator may be called in different interpreters that do not share a `GIL`.

If the new allocator is not a hook (does not call the previous allocator), the *PyMem_SetupDebugHooks()* function must be called to reinstall the debug hooks on top on the new allocator.

See also *PyPreConfig.allocator* and *Preinitialize Python with PyPreConfig*.

Aviso

PyMem_SetAllocator() does have the following contract:

- It can be called after *Py_PreInitialize()* and before *Py_InitializeFromConfig()* to install a custom memory allocator. There are no restrictions over the installed allocator other than the ones imposed by the domain (for instance, the Raw Domain allows the allocator to be called without the GIL held). See *the section on allocator domains* for more information.
- If called after Python has finish initializing (after *Py_InitializeFromConfig()* has been called) the allocator **must** wrap the existing allocator. Substituting the current allocator for some other arbitrary one is **not supported**.

Alterado na versão 3.12: All allocators must be thread-safe.

void **PyMem_SetupDebugHooks**(void)

Setup *debug hooks in the Python memory allocators* to detect memory errors.

11.8 Debug hooks on the Python memory allocators

When Python is built in debug mode, the *PyMem_SetupDebugHooks()* function is called at the *Python preinitialization* to setup debug hooks on Python memory allocators to detect memory errors.

The PYTHONMALLOC environment variable can be used to install debug hooks on a Python compiled in release mode (ex: PYTHONMALLOC=debug).

The *PyMem_SetupDebugHooks()* function can be used to set debug hooks after calling *PyMem_SetAllocator()*.

These debug hooks fill dynamically allocated memory blocks with special, recognizable bit patterns. Newly allocated memory is filled with the byte 0xCD (PYMEM_CLEANBYTE), freed memory is filled with the byte 0xDD (PYMEM_DEADBYTE). Memory blocks are surrounded by "forbidden bytes" filled with the byte 0xFD (PYMEM_FORBIDDENBYTE). Strings of these bytes are unlikely to be valid addresses, floats, or ASCII strings.

Checagens em Tempo de Execução:

- Detect API violations. For example, detect if *PyObject_Free()* is called on a memory block allocated by *PyMem_Malloc()*.
- Detect write before the start of the buffer (buffer underflow).
- Detect write after the end of the buffer (buffer overflow).
- Check that the *GIL* is held when allocator functions of *PYMEM_DOMAIN_OBJ* (ex: *PyObject_Malloc()*) and *PYMEM_DOMAIN_MEM* (ex: *PyMem_Malloc()*) domains are called.

On error, the debug hooks use the tracemalloc module to get the traceback where a memory block was allocated. The traceback is only displayed if tracemalloc is tracing Python memory allocations and the memory block was traced.

Let *S* = sizeof(size_t). 2*S bytes are added at each end of each block of *N* bytes requested. The memory layout is like so, where p represents the address returned by a malloc-like or realloc-like function (p[i:j] means the slice of bytes from *(p+i) inclusive up to *(p+j) exclusive; note that the treatment of negative indices differs from a Python slice):

p[-2*S:-S]
Number of bytes originally asked for. This is a size_t, big-endian (easier to read in a memory dump).

p[-S]
API identifier (ASCII character):

- 'r' for *PYMEM_DOMAIN_RAW*.
- 'm' for *PYMEM_DOMAIN_MEM*.
- 'o' for *PYMEM_DOMAIN_OBJ*.

p[-S+1:0]
Copies of PYMEM_FORBIDDENBYTE. Used to catch under- writes and reads.

p[0:N]
The requested memory, filled with copies of PYMEM_CLEANBYTE, used to catch reference to uninitialized memory. When a realloc-like function is called requesting a larger memory block, the new excess bytes are also filled with PYMEM_CLEANBYTE. When a free-like function is called, these are overwritten with PYMEM_DEADBYTE, to catch reference to freed memory. When a realloc- like function is called requesting a smaller memory block, the excess old bytes are also filled with PYMEM_DEADBYTE.

p[N:N+S]
Copies of PYMEM_FORBIDDENBYTE. Used to catch over- writes and reads.

p[N+S:N+2*S]
Only used if the PYMEM_DEBUG_SERIALNO macro is defined (not defined by default).

A serial number, incremented by 1 on each call to a malloc-like or realloc-like function. Big-endian size_t. If "bad memory" is detected later, the serial number gives an excellent way to set a breakpoint on the next run, to capture the instant at which this block was passed out. The static function bumpserialno() in obmalloc.c is the only place the serial number is incremented, and exists so you can set such a breakpoint easily.

A realloc-like or free-like function first checks that the PYMEM_FORBIDDENBYTE bytes at each end are intact. If they've been altered, diagnostic output is written to stderr, and the program is aborted via Py_FatalError(). The other main failure mode is provoking a memory error when a program reads up one of the special bit patterns and

tries to use it as an address. If you get in a debugger then and look at the object, you're likely to see that it's entirely filled with PYMEM_DEADBYTE (meaning freed memory is getting used) or PYMEM_CLEANBYTE (meaning uninitialized memory is getting used).

Alterado na versão 3.6: The *PyMem_SetupDebugHooks()* function now also works on Python compiled in release mode. On error, the debug hooks now use `tracemalloc` to get the traceback where a memory block was allocated. The debug hooks now also check if the GIL is held when functions of *PYMEM_DOMAIN_OBJ* and *PYMEM_DOMAIN_MEM* domains are called.

Alterado na versão 3.8: Byte patterns `0xCB` (`PYMEM_CLEANBYTE`), `0xDB` (`PYMEM_DEADBYTE`) and `0xFB` (`PYMEM_FORBIDDENBYTE`) have been replaced with `0xCD`, `0xDD` and `0xFD` to use the same values than Windows CRT debug `malloc()` and `free()`.

11.9 The pymalloc allocator

Python has a *pymalloc* allocator optimized for small objects (smaller or equal to 512 bytes) with a short lifetime. It uses memory mappings called "arenas" with a fixed size of either 256 KiB on 32-bit platforms or 1 MiB on 64-bit platforms. It falls back to *PyMem_RawMalloc()* and *PyMem_RawRealloc()* for allocations larger than 512 bytes.

pymalloc is the *default allocator* of the *PYMEM_DOMAIN_MEM* (ex: *PyMem_Malloc()*) and *PYMEM_DOMAIN_OBJ* (ex: *PyObject_Malloc()*) domains.

The arena allocator uses the following functions:

- `VirtualAlloc()` and `VirtualFree()` on Windows,
- `mmap()` and `munmap()` if available,
- `malloc()` e `free()` do contrário.

This allocator is disabled if Python is configured with the `--without-pymalloc` option. It can also be disabled at runtime using the `PYTHONMALLOC` environment variable (ex: `PYTHONMALLOC=malloc`).

11.9.1 Customize pymalloc Arena Allocator

Adicionado na versão 3.4.

type **PyObjectArenaAllocator**

Structure used to describe an arena allocator. The structure has three fields:

Campo	Significado
`void *ctx`	user context passed as first argument
`void* alloc(void *ctx, size_t size)`	allocate an arena of size bytes
`void free(void *ctx, void *ptr, size_t size)`	free an arena

void **PyObject_GetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

Get the arena allocator.

void **PyObject_SetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

Set the arena allocator.

11.10 The mimalloc allocator

Adicionado na versão 3.13.

Python supports the mimalloc allocator when the underlying platform support is available. mimalloc "is a general purpose allocator with excellent performance characteristics. Initially developed by Daan Leijen for the runtime systems of the Koka and Lean languages."

11.11 tracemalloc C API

Adicionado na versão 3.7.

int **PyTraceMalloc_Track**(unsigned int domain, uintptr_t ptr, size_t size)

Track an allocated memory block in the `tracemalloc` module.

Return `0` on success, return `-1` on error (failed to allocate memory to store the trace). Return `-2` if tracemalloc is disabled.

If memory block is already tracked, update the existing trace.

int **PyTraceMalloc_Untrack**(unsigned int domain, uintptr_t ptr)

Untrack an allocated memory block in the `tracemalloc` module. Do nothing if the block was not tracked.

Return `-2` if tracemalloc is disabled, otherwise return `0`.

11.12 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 = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

The same code using the type-oriented function set:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

Note that in the two examples above, the buffer is always manipulated via functions belonging to the same set. Indeed, it is required to use the same memory API family for a given memory block, so that the risk of mixing different allocators is reduced to a minimum. The following code sequence contains two errors, one of which is labeled as *fatal* because it mixes two different allocators operating on different heaps.

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);  /* Wrong -- should be PyMem_Free() */
free(buf2);       /* Right -- allocated via malloc() */
free(buf1);       /* Fatal -- should be PyMem_Del()  */
```

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with *PyObject_New*, *PyObject_NewVar* and *PyObject_Del()*.

These will be explained in the next chapter on defining and implementing new object types in C.

CAPÍTULO 12

Suporte a implementação de Objetos

Este capítulo descreve as funções, tipos e macros usados ao definir novos tipos de objeto.

12.1 Alocando Objetos na Pilha

PyObject ***_PyObject_New**(*PyTypeObject* *type)

Retorna valor: Nova referência.

PyVarObject ***_PyObject_NewVar**(*PyTypeObject* *type, *Py_ssize_t* size)

Retorna valor: Nova referência.

PyObject ***PyObject_Init**(*PyObject* *op, *PyTypeObject* *type)

Retorna valor: Referência emprestada. Parte da ABI Estável. Inicializa um objeto *op* recém-alocado com seu tipo e referência inicial. Retorna o objeto inicializado. Outros campos do objeto não são afetados.

PyVarObject ***PyObject_InitVar**(*PyVarObject* *op, *PyTypeObject* *type, *Py_ssize_t* size)

Retorna valor: Referência emprestada. Parte da ABI Estável. Isto faz tudo que o `PyObject_Init()` faz e também inicializa a informação de comprimento para um objeto de tamanho variável.

PyObject_New(TYPE, typeobj)

Aloca um novo objeto Python usando o tipo de estrutura do C *TYPE* e o objeto Python do tipo *typeobj* (`PyTypeObject*`). Campos não definidos pelo cabeçalho do objeto Python não são inicializados. O chamador será dono da apenas a referência ao objeto (isto é, sua contagem de referências será uma). O tamanho da alocação de memória é determinado do campo `tp_basicsize` do objeto tipo.

Note que esta função não é adequada se *typeobj* tiver `Py_TPFLAGS_HAVE_GC` definido. Para tais objetos, use `PyObject_GC_New()` em vez disso.

PyObject_NewVar(TYPE, typeobj, size)

Aloca um novo objeto Python usando o tipo de estrutura do C *TYPE* e o objeto Python do tipo *typeobj* (`PyTypeObject*`). Campos não definidos pelo cabeçalho do objeto Python não são inicializados. A memória alocada permite a estrutura *TYPE* e os campos *size* (`Py_ssize_t`) do tamanho dado pelo campo `tp_itemsize` do tipo *typeobj*. Isto é útil para implementar objetos como tuplas, as quais são capazes de determinar seu tamanho no tempo da construção. Incorporando o vetor de campos dentro da mesma alocação diminuindo o numero de alocações, melhorando a eficiência do gerenciamento de memória.

Note que esta função não é adequada se *typeobj* tiver `Py_TPFLAGS_HAVE_GC` definido. Para tais objetos, use `PyObject_GC_NewVar()` em vez disso.

void **PyObject_Del**(void *op)

Libera memória alocada a um objeto usando *PyObject_New* ou *PyObject_NewVar*. Isto é normalmente chamado pelo manipulador de *tp_dealloc* especificado no tipo do objeto. Os campos do objeto não devem ser acessados após esta chamada como a memória não é mais um objeto Python válido.

PyObject **_Py_NoneStruct**

Objeto o qual é visível no Python como `None`. Isto só deve ser acessado usando a macro *Py_None*, o qual avalia como um ponteiro para este objeto.

Ver também

PyModule_Create()
Para alocar e criar módulos de extensão.

12.2 Estruturas comuns de objetos

Existe um grande número de estruturas usadas para a definição de tipos objeto para o Python. Esta seção descreve essas estruturas e como são usadas.

12.2.1 Base object types and macros

Todos os objetos Python por fim compartilham um pequeno número de campos no começo da representação o objeto na memória. Esses são representados pelos tipos *PyObject* e *PyVarObject*, que são definidos, por sua vez, pelas expansões de alguns macros também, utilizados, direta ou indiretamente, na definição de todos outros objetos Python. Macros adicionais podem ser encontrados em *contagem de referências*.

type **PyObject**

Parte da API Limitada. *(Somente alguns membros são parte da ABI estável.)* All object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In a normal "release" build, it contains only the object's reference count and a pointer to the corresponding type object. Nothing is actually declared to be a *PyObject*, but every pointer to a Python object can be cast to a *PyObject**. Access to the members must be done by using the macros *Py_REFCNT* and *Py_TYPE*.

type **PyVarObject**

Parte da API Limitada. *(Somente alguns membros são parte da ABI estável.)* Esta é uma extensão de *PyObject* que adiciona o campo *ob_size*. Isso é usado apenas para objetos que têm alguma noção de *comprimento*. Esse tipo não costuma aparecer na API Python/C. O acesso aos membros deve ser feito através das macros *Py_REFCNT*, *Py_TYPE*, e *Py_SIZE*.

PyObject_HEAD

Este é um macro usado ao declarar novos tipos que representam objetos sem comprimento variável. O macro PyObject_HEAD se expande para:

```
PyObject ob_base;
```

Veja documentação de *PyObject* acima.

PyObject_VAR_HEAD

This is a macro used when declaring new types which represent objects with a length that varies from instance to instance. The PyObject_VAR_HEAD macro expands to:

```
PyVarObject ob_base;
```

Veja documentação de *PyVarObject* acima.

PyTypeObject **PyBaseObject_Type**

Parte da ABI Estável. The base class of all other objects, the same as `object` in Python.

int **Py_Is**(*PyObject* *x, *PyObject* *y)

Parte da ABI Estável *desde a versão 3.10.* Testa se o objeto *x* é o objeto *y*, o mesmo que `x is y` em Python.

Adicionado na versão 3.10.

int **Py_IsNone**(*PyObject* *x)

Parte da ABI Estável *desde a versão 3.10.* Test if an object is the `None` singleton, the same as `x is None` in Python.

Adicionado na versão 3.10.

int **Py_IsTrue**(*PyObject* *x)

Parte da ABI Estável *desde a versão 3.10.* Test if an object is the `True` singleton, the same as `x is True` in Python.

Adicionado na versão 3.10.

int **Py_IsFalse**(*PyObject* *x)

Parte da ABI Estável *desde a versão 3.10.* Test if an object is the `False` singleton, the same as `x is False` in Python.

Adicionado na versão 3.10.

PyTypeObject ***Py_TYPE**(*PyObject* *o)

Retorna valor: Referência emprestada. Get the type of the Python object *o*.

Return a *borrowed reference*.

Use the *Py_SET_TYPE()* function to set an object type.

Alterado na versão 3.11: *Py_TYPE()* is changed to an inline static function. The parameter type is no longer `const PyObject*`.

int **Py_IS_TYPE**(*PyObject* *o, *PyTypeObject* *type)

Return non-zero if the object *o* type is *type*. Return zero otherwise. Equivalent to: `Py_TYPE(o) == type`.

Adicionado na versão 3.9.

void **Py_SET_TYPE**(*PyObject* *o, *PyTypeObject* *type)

Set the object *o* type to *type*.

Adicionado na versão 3.9.

Py_ssize_t **Py_SIZE**(*PyVarObject* *o)

Get the size of the Python object *o*.

Use the *Py_SET_SIZE()* function to set an object size.

Alterado na versão 3.11: *Py_SIZE()* is changed to an inline static function. The parameter type is no longer `const PyVarObject*`.

void **Py_SET_SIZE**(*PyVarObject* *o, *Py_ssize_t* size)

Set the object *o* size to *size*.

Adicionado na versão 3.9.

PyObject_HEAD_INIT(type)

This is a macro which expands to initialization values for a new *PyObject* type. This macro expands to:

```
_PyObject_EXTRA_INIT
1, type,
```

PyVarObject_HEAD_INIT(type, size)

This is a macro which expands to initialization values for a new *PyVarObject* type, including the *ob_size* field. This macro expands to:

```
_PyObject_EXTRA_INIT
1, type, size,
```

12.2.2 Implementing functions and methods

type **PyCFunction**

Parte da ABI Estável. 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.

A assinatura da função é:

```
PyObject *PyCFunction(PyObject *self,
                      PyObject *args);
```

type **PyCFunctionWithKeywords**

Parte da ABI Estável. Type of the functions used to implement Python callables in C with signature *METH_VARARGS | METH_KEYWORDS*. The function signature is:

```
PyObject *PyCFunctionWithKeywords(PyObject *self,
                                  PyObject *args,
                                  PyObject *kwargs);
```

type **PyCFunctionFast**

Parte da ABI Estável *desde a versão 3.13.* Type of the functions used to implement Python callables in C with signature *METH_FASTCALL*. The function signature is:

```
PyObject *PyCFunctionFast(PyObject *self,
                          PyObject *const *args,
                          Py_ssize_t nargs);
```

type **PyCFunctionFastWithKeywords**

Parte da ABI Estável *desde a versão 3.13.* Type of the functions used to implement Python callables in C with signature *METH_FASTCALL | METH_KEYWORDS*. The function signature is:

```
PyObject *PyCFunctionFastWithKeywords(PyObject *self,
                                      PyObject *const *args,
                                      Py_ssize_t nargs,
                                      PyObject *kwnames);
```

type **PyCMethod**

Type of the functions used to implement Python callables in C with signature *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*. The function signature is:

```
PyObject *PyCMethod(PyObject *self,
                    PyTypeObject *defining_class,
                    PyObject *const *args,
                    Py_ssize_t nargs,
                    PyObject *kwnames)
```

Adicionado na versão 3.9.

type **PyMethodDef**

Parte da ABI Estável *(incluindo todos os membros).* Structure used to describe a method of an extension type. This structure has four fields:

const char ***ml_name**

Nome do método.

PyCFunction **ml_meth**

Pointer to the C implementation.

int **ml_flags**

Flags bits indicating how the call should be constructed.

const char ***ml_doc**

Points to the contents of the docstring.

The *ml_meth* is a C function pointer. The functions may be of different types, but they always return *PyObject**. If the function is not of the *PyCFunction*, the compiler will require a cast in the method table. Even though *PyCFunction* defines the first parameter as *PyObject**, it is common that the method implementation uses the specific C type of the *self* object.

The *ml_flags* field is a bitfield which can include the following flags. The individual flags indicate either a calling convention or a binding convention.

There are these calling conventions:

METH_VARARGS

This is the typical calling convention, where the methods have the type *PyCFunction*. The function expects two *PyObject** values. The first one is the *self* object for methods; for module functions, it is the module object. The second parameter (often called *args*) is a tuple object representing all arguments. This parameter is typically processed using *PyArg_ParseTuple()* or *PyArg_UnpackTuple()*.

METH_KEYWORDS

Can only be used in certain combinations with other flags: *METH_VARARGS | METH_KEYWORDS*, *METH_FASTCALL | METH_KEYWORDS* and *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*.

METH_VARARGS* | *METH_KEYWORDS

Methods with these flags must be of type *PyCFunctionWithKeywords*. The function expects three parameters: *self*, *args*, *kwargs* where *kwargs* is a dictionary of all the keyword arguments or possibly `NULL` if there are no keyword arguments. The parameters are typically processed using *PyArg_ParseTupleAndKeywords()*.

METH_FASTCALL

Fast calling convention supporting only positional arguments. The methods have the type *PyCFunctionFast*. The first parameter is *self*, the second parameter is a C array of *PyObject** values indicating the arguments and the third parameter is the number of arguments (the length of the array).

Adicionado na versão 3.7.

Alterado na versão 3.10: `METH_FASTCALL` is now part of the *stable ABI*.

METH_FASTCALL* | *METH_KEYWORDS

Extension of *METH_FASTCALL* supporting also keyword arguments, with methods of type *PyCFunctionFastWithKeywords*. Keyword arguments are passed the same way as in the *vectorcall protocol*: there is an additional fourth *PyObject** parameter which is a tuple representing the names of the keyword arguments (which are guaranteed to be strings) or possibly `NULL` if there are no keywords. The values of the keyword arguments are stored in the *args* array, after the positional arguments.

Adicionado na versão 3.7.

METH_METHOD

Can only be used in the combination with other flags: *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*.

METH_METHOD* | *METH_FASTCALL* | *METH_KEYWORDS

Extension of *METH_FASTCALL | METH_KEYWORDS* supporting the *defining class*, that is, the class that contains the method in question. The defining class might be a superclass of `Py_TYPE(self)`.

The method needs to be of type *PyCMethod*, the same as for `METH_FASTCALL | METH_KEYWORDS` with `defining_class` argument added after `self`.

Adicionado na versão 3.9.

METH_NOARGS

Methods without parameters don't need to check whether arguments are given if they are listed with the *METH_NOARGS* flag. They need to be of type *PyCFunction*. The first parameter is typically named *self* and will hold a reference to the module or object instance. In all cases the second parameter will be `NULL`.

The function must have 2 parameters. Since the second parameter is unused, *Py_UNUSED* can be used to prevent a compiler warning.

METH_O

Methods with a single object argument can be listed with the *METH_O* flag, instead of invoking *PyArg_ParseTuple()* with a `"O"` argument. They have the type *PyCFunction*, with the *self* parameter, and a *PyObject** parameter representing the single argument.

These two constants are not used to indicate the calling convention but the binding when use with methods of classes. These may not be used for functions defined for modules. At most one of these flags may be set for any given method.

METH_CLASS

The method will be passed the type object as the first parameter rather than an instance of the type. This is used to create *class methods*, similar to what is created when using the `classmethod()` built-in function.

METH_STATIC

The method will be passed `NULL` as the first parameter rather than an instance of the type. This is used to create *static methods*, similar to what is created when using the `staticmethod()` built-in function.

One other constant controls whether a method is loaded in place of another definition with the same method name.

METH_COEXIST

The method will be loaded in place of existing definitions. Without *METH_COEXIST*, the default is to skip repeated definitions. Since slot wrappers are loaded before the method table, the existence of a *sq_contains* slot, for example, would generate a wrapped method named `__contains__()` and preclude the loading of a corresponding PyCFunction with the same name. With the flag defined, the PyCFunction will be loaded in place of the wrapper object and will co-exist with the slot. This is helpful because calls to PyCFunctions are optimized more than wrapper object calls.

PyObject ***PyCMethod_New**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module, *PyTypeObject* *cls)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.9.* Turn *ml* into a Python *callable* object. The caller must ensure that *ml* outlives the *callable*. Typically, *ml* is defined as a static variable.

The *self* parameter will be passed as the *self* argument to the C function in `ml->ml_meth` when invoked. *self* can be `NULL`.

The *callable* object's `__module__` attribute can be set from the given *module* argument. *module* should be a Python string, which will be used as name of the module the function is defined in. If unavailable, it can be set to `None` or `NULL`.

Ver também

`function.__module__`

The *cls* parameter will be passed as the *defining_class* argument to the C function. Must be set if *METH_METHOD* is set on `ml->ml_flags`.

Adicionado na versão 3.9.

PyObject ***PyCFunction_NewEx**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module)

Retorna valor: Nova referência. Parte da ABI Estável. Equivalente a `PyCMethod_New(ml, self, module, NULL)`.

PyObject ***PyCFunction_New**(*PyMethodDef* *ml, *PyObject* *self)

Retorna valor: Nova referência. Parte da ABI Estável *desde a versão 3.4.* Equivalente a `PyCMethod_New(ml, self, NULL, NULL)`.

12.2.3 Accessing attributes of extension types

type **PyMemberDef**

Parte da ABI Estável *(incluindo todos os membros).* Structure which describes an attribute of a type which corresponds to a C struct member. When defining a class, put a NULL-terminated array of these structures in the *tp_members* slot.

Its fields are, in order:

const char ***name**

Name of the member. A NULL value marks the end of a `PyMemberDef[]` array.

The string should be static, no copy is made of it.

int **type**

The type of the member in the C struct. See *Member types* for the possible values.

Py_ssize_t **offset**

The offset in bytes that the member is located on the type's object struct.

int **flags**

Zero or more of the *Member flags*, combined using bitwise OR.

const char ***doc**

The docstring, or NULL. The string should be static, no copy is made of it. Typically, it is defined using *PyDoc_STR*.

By default (when *flags* is `0`), members allow both read and write access. Use the *Py_READONLY* flag for read-only access. Certain types, like *Py_T_STRING*, imply *Py_READONLY*. Only *Py_T_OBJECT_EX* (and legacy *T_OBJECT*) members can be deleted.

For heap-allocated types (created using *PyType_FromSpec()* or similar), `PyMemberDef` may contain a definition for the special member `"__vectorcalloffset__"`, corresponding to *tp_vectorcall_offset* in type objects. These must be defined with `Py_T_PYSSIZET` and `Py_READONLY`, for example:

```
static PyMemberDef spam_type_members[] = {
    {"__vectorcalloffset__", Py_T_PYSSIZET,
     offsetof(Spam_object, vectorcall), Py_READONLY},
    {NULL}  /* Sentinel */
};
```

(You may need to `#include <stddef.h>` for `offsetof()`.)

The legacy offsets *tp_dictoffset* and *tp_weaklistoffset* can be defined similarly using `"__dictoffset__"` and `"__weaklistoffset__"` members, but extensions are strongly encouraged to use *Py_TPFLAGS_MANAGED_DICT* and *Py_TPFLAGS_MANAGED_WEAKREF* instead.

Alterado na versão 3.12: `PyMemberDef` is always available. Previously, it required including `"structmember.h"`.

PyObject ***PyMember_GetOne**(const char *obj_addr, struct *PyMemberDef* *m)

Parte da ABI Estável. Get an attribute belonging to the object at address *obj_addr*. The attribute is described by `PyMemberDef` *m*. Returns `NULL` on error.

Alterado na versão 3.12: `PyMember_GetOne` is always available. Previously, it required including `"structmember.h"`.

int **PyMember_SetOne**(char *obj_addr, struct *PyMemberDef* *m, *PyObject* *o)

Parte da ABI Estável. Set an attribute belonging to the object at address *obj_addr* to object *o*. The attribute to set is described by `PyMemberDef` *m*. Returns `0` if successful and a negative value on failure.

Alterado na versão 3.12: `PyMember_SetOne` is always available. Previously, it required including `"structmember.h"`.

Member flags

The following flags can be used with *`PyMemberDef.flags`*:

`Py_READONLY`

Not writable.

`Py_AUDIT_READ`

Emit an `object.__getattr__` audit event before reading.

`Py_RELATIVE_OFFSET`

Indicates that the *`offset`* of this `PyMemberDef` entry indicates an offset from the subclass-specific data, rather than from `PyObject`.

Can only be used as part of *`Py_tp_members` `slot`* when creating a class using negative *`basicsize`*. It is mandatory in that case.

This flag is only used in *`PyType_Slot`*. When setting *`tp_members`* during class creation, Python clears it and sets *`PyMemberDef.offset`* to the offset from the `PyObject` struct.

Alterado na versão 3.10: The `RESTRICTED`, `READ_RESTRICTED` and `WRITE_RESTRICTED` macros available with `#include "structmember.h"` are deprecated. `READ_RESTRICTED` and `RESTRICTED` are equivalent to *`Py_AUDIT_READ`*; `WRITE_RESTRICTED` does nothing.

Alterado na versão 3.12: The `READONLY` macro was renamed to *`Py_READONLY`*. The `PY_AUDIT_READ` macro was renamed with the `Py_` prefix. The new names are now always available. Previously, these required `#include "structmember.h"`. The header is still available and it provides the old names.

Member types

`PyMemberDef.type` can be one of the following macros corresponding to various C types. When the member is accessed in Python, it will be converted to the equivalent Python type. When it is set from Python, it will be converted back to the C type. If that is not possible, an exception such as `TypeError` or `ValueError` is raised.

Unless marked (D), attributes defined this way cannot be deleted using e.g. `del` or `delattr()`.

Macro name	Tipo em C	Tipo em Python
Py_T_BYTE	`char`	`int`
Py_T_SHORT	`short`	`int`
Py_T_INT	`int`	`int`
Py_T_LONG	`long`	`int`
Py_T_LONGLONG	`long long`	`int`
Py_T_UBYTE	`unsigned char`	`int`
Py_T_UINT	`unsigned int`	`int`
Py_T_USHORT	`unsigned short`	`int`
Py_T_ULONG	`unsigned long`	`int`
Py_T_ULONGLONG	`unsigned long long`	`int`
Py_T_PYSSIZET	*Py_ssize_t*	`int`
Py_T_FLOAT	`float`	`float`
Py_T_DOUBLE	`double`	`float`
Py_T_BOOL	`char` (escrito como 0 ou 1)	`bool`
Py_T_STRING	`const char*` (*)	`str` (RO)
Py_T_STRING_INPLACE	`const char[]` (*)	`str` (RO)
Py_T_CHAR	`char` (0-127)	`str` (**)
Py_T_OBJECT_EX	*PyObject**	`object` (D)

(*): Zero-terminated, UTF8-encoded C string. With `Py_T_STRING` the C representation is a pointer; with `Py_T_STRING_INPLACE` the string is stored directly in the structure.

(**): String of length 1. Only ASCII is accepted.

(RO): implica *Py_READONLY*.

(D): pode ser deletado, neste caso o ponteiro é definido para NULL. Ler um ponteiro NULL levanta uma exceção AttributeError.

Adicionado na versão 3.12: In previous versions, the macros were only available with `#include "structmember.h"` and were named without the Py_ prefix (e.g. as T_INT). The header is still available and contains the old names, along with the following deprecated types:

T_OBJECT

Like Py_T_OBJECT_EX, but NULL is converted to None. This results in surprising behavior in Python: deleting the attribute effectively sets it to None.

T_NONE

Sempre None. Deve ser usado com *Py_READONLY*.

Defining Getters and Setters

type **PyGetSetDef**

Parte da ABI Estável *(incluindo todos os membros)*. Structure to define property-like access for a type. See also description of the *PyTypeObject.tp_getset* slot.

const char ***name**

attribute name

getter **get**

C function to get the attribute.

setter **set**

Optional C function to set or delete the attribute. If NULL, the attribute is read-only.

const char ***doc**

optional docstring

void ***closure**

Optional user data pointer, providing additional data for getter and setter.

typedef *PyObject* *(***getter**)(*PyObject**, void*)

Parte da ABI Estável. The get function takes one *PyObject** parameter (the instance) and a user data pointer (the associated closure):

It should return a new reference on success or NULL with a set exception on failure.

typedef int (***setter**)(*PyObject**, *PyObject**, void*)

Parte da ABI Estável. set functions take two *PyObject** parameters (the instance and the value to be set) and a user data pointer (the associated closure):

In case the attribute should be deleted the second parameter is NULL. Should return 0 on success or -1 with a set exception on failure.

12.3 Type Object Structures

Perhaps one of the most important structures of the Python object system is the structure that defines a new type: the *PyTypeObject* structure. Type objects can be handled using any of the PyObject_* or PyType_* functions, but do not offer much that's interesting to most Python applications. These objects are fundamental to how objects behave, so they are very important to the interpreter itself and to any extension module that implements new types.

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.

Além da referência rápida a seguir, a seção *Exemplos* fornece uma visão geral do significado e uso de `PyTypeObject`.

12.3.1 Referências rápidas

"slots tp"

Slot de PyTypeObject[Página 280, 1]	*Tipo*	métodos/atributos especiais	Info[Página 280, 2] C	T	D	I
<R> `tp_name`	const char *	__name__	X	X		
`tp_basicsize`	`Py_ssize_t`		X	X		X
`tp_itemsize`	`Py_ssize_t`			X		X
`tp_dealloc`	`destructor`		X	X		X
`tp_vectorcall_offset`	`Py_ssize_t`			X		X
(`tp_getattr`)	`getattrfunc`	__getattribute__, __getattr__				G
(`tp_setattr`)	`setattrfunc`	__setattr__, __delattr__				G
`tp_as_async`	`PyAsyncMethods` *	*sub-slots*				%
`tp_repr`	`reprfunc`	__repr__	X	X		X
`tp_as_number`	`PyNumberMethods` *	*sub-slots*				%
`tp_as_sequence`	`PySequenceMethods` *	*sub-slots*				%
`tp_as_mapping`	`PyMappingMethods` *	*sub-slots*				%
`tp_hash`	`hashfunc`	__hash__	X			G
`tp_call`	`ternaryfunc`	__call__		X		X
`tp_str`	`reprfunc`	__str__	X			X
`tp_getattro`	`getattrofunc`	__getattribute__, __getattr__	X	X		G
`tp_setattro`	`setattrofunc`	__setattr__, __delattr__	X	X		G
`tp_as_buffer`	`PyBufferProcs` *					%
`tp_flags`	unsigned long		X	X		?
`tp_doc`	const char *	__doc__	X	X		
`tp_traverse`	`traverseproc`			X		G
`tp_clear`	`inquiry`			X		G
`tp_richcompare`	`richcmpfunc`	__lt__, __le__, __eq__, __ne__, __gt__, __ge__	X			G
(`tp_weaklistoffset`)	`Py_ssize_t`			X		?
`tp_iter`	`getiterfunc`	__iter__				X
`tp_iternext`	`iternextfunc`	__next__				X
`tp_methods`	`PyMethodDef` []		X	X		
`tp_members`	`PyMemberDef` []			X		
`tp_getset`	`PyGetSetDef` []		X	X		
`tp_base`	`PyTypeObject` *	__base__			X	
`tp_dict`	`PyObject` *	__dict__			?	
`tp_descr_get`	`descrgetfunc`	__get__				X
`tp_descr_set`	`descrsetfunc`	__set__, __delete__				X
(`tp_dictoffset`)	`Py_ssize_t`			X		?
`tp_init`	`initproc`	__init__	X	X		X
`tp_alloc`	`allocfunc`		X		?	?
`tp_new`	`newfunc`	__new__	X	X	?	?
`tp_free`	`freefunc`		X	X	?	?
`tp_is_gc`	`inquiry`			X		X
<`tp_bases`>	`PyObject` *	__bases__			~	
<`tp_mro`>	`PyObject` *	__mro__			~	
[`tp_cache`]	`PyObject` *					
[`tp_subclasses`]	void *	__subclasses__				
[`tp_weaklist`]	`PyObject` *					
(`tp_del`)	`destructor`					
[`tp_version_tag`]	unsigned int					
`tp_finalize`	`destructor`	__del__				X

continua na próxima página

Tabela 1 – continuação da página anterior

Slot de PyTypeObject[1]	*Tipo*	métodos/atributos especiais	Info[2] C T D I
tp_vectorcall	*vectorcallfunc*		
[*tp_watched*]	unsigned char		

sub-slots

Slot	*Tipo*	special methods
am_await	*unaryfunc*	__await__
am_aiter	*unaryfunc*	__aiter__
am_anext	*unaryfunc*	__anext__
am_send	*sendfunc*	
nb_add	*binaryfunc*	__add__ __radd__
nb_inplace_add	*binaryfunc*	__iadd__
nb_subtract	*binaryfunc*	__sub__ __rsub__
nb_inplace_subtract	*binaryfunc*	__isub__
nb_multiply	*binaryfunc*	__mul__ __rmul__
nb_inplace_multiply	*binaryfunc*	__imul__
nb_remainder	*binaryfunc*	__mod__ __rmod__
nb_inplace_remainder	*binaryfunc*	__imod__
nb_divmod	*binaryfunc*	__divmod__ __rdivmod__
nb_power	*ternaryfunc*	__pow__ __rpow__
nb_inplace_power	*ternaryfunc*	__ipow__
nb_negative	*unaryfunc*	__neg__
nb_positive	*unaryfunc*	__pos__
nb_absolute	*unaryfunc*	__abs__
nb_bool	*inquiry*	__bool__
nb_invert	*unaryfunc*	__invert__
nb_lshift	*binaryfunc*	__lshift__ __rlshift__
nb_inplace_lshift	*binaryfunc*	__ilshift__
nb_rshift	*binaryfunc*	__rshift__ __rrshift__
nb_inplace_rshift	*binaryfunc*	__irshift__
nb_and	*binaryfunc*	__and__ __rand__

continua na próxima página

[1] (): A slot name in parentheses indicates it is (effectively) deprecated.
<>: Names in angle brackets should be initially set to `NULL` and treated as read-only.
[]: Names in square brackets are for internal use only.
<R> (as a prefix) means the field is required (must be non-`NULL`).

[2] Columns:
"O": set on *PyBaseObject_Type*
"T": set on *PyType_Type*
"D": default (if slot is set to `NULL`)

```
X - PyType_Ready sets this value if it is NULL
~ - PyType_Ready always sets this value (it should be NULL)
? - PyType_Ready may set this value depending on other slots

Also see the inheritance column ("I").
```

"I": inheritance

```
X - type slot is inherited via *PyType_Ready* if defined with a *NULL* value
% - the slots of the sub-struct are inherited individually
G - inherited, but only in combination with other slots; see the slot's description
? - it's complicated; see the slot's description
```

Note that some slots are effectively inherited through the normal attribute lookup chain.

Tabela 2 – continuação da página anterior

Slot	*Tipo*	special methods
nb_inplace_and	*binaryfunc*	__iand__
nb_xor	*binaryfunc*	__xor__ __rxor__
nb_inplace_xor	*binaryfunc*	__ixor__
nb_or	*binaryfunc*	__or__ __ror__
nb_inplace_or	*binaryfunc*	__ior__
nb_int	*unaryfunc*	__int__
nb_reserved	void *	
nb_float	*unaryfunc*	__float__
nb_floor_divide	*binaryfunc*	__floordiv__
nb_inplace_floor_divide	*binaryfunc*	__ifloordiv__
nb_true_divide	*binaryfunc*	__truediv__
nb_inplace_true_divide	*binaryfunc*	__itruediv__
nb_index	*unaryfunc*	__index__
nb_matrix_multiply	*binaryfunc*	__matmul__ __rmatmul__
nb_inplace_matrix_multiply	*binaryfunc*	__imatmul__
mp_length	*lenfunc*	__len__
mp_subscript	*binaryfunc*	__getitem__
mp_ass_subscript	*objobjargproc*	__setitem__, __delitem__
sq_length	*lenfunc*	__len__
sq_concat	*binaryfunc*	__add__
sq_repeat	*ssizeargfunc*	__mul__
sq_item	*ssizeargfunc*	__getitem__
sq_ass_item	*ssizeobjargproc*	__setitem__ __delitem__
sq_contains	*objobjproc*	__contains__
sq_inplace_concat	*binaryfunc*	__iadd__
sq_inplace_repeat	*ssizeargfunc*	__imul__
bf_getbuffer	*getbufferproc()*	
bf_releasebuffer	*releasebufferproc()*	

slot typedefs

typedef	Parameter Types	Return Type
allocfunc	*PyTypeObject* * *Py_ssize_t*	*PyObject* *
destructor	*PyObject* *	void
freefunc	void *	void
traverseproc	*PyObject* * *visitproc* void *	int
newfunc	*PyTypeObject* * *PyObject* * *PyObject* *	*PyObject* *
initproc	*PyObject* * *PyObject* * *PyObject* *	int
reprfunc	*PyObject* *	*PyObject* *
getattrfunc	*PyObject* * const char *	*PyObject* *
setattrfunc	*PyObject* * const char * *PyObject* *	int
getattrofunc	*PyObject* * *PyObject* *	*PyObject* *
setattrofunc	*PyObject* * *PyObject* * *PyObject* *	int
descrgetfunc	*PyObject* * *PyObject* * *PyObject* *	*PyObject* *
descrsetfunc	*PyObject* * *PyObject* * *PyObject* *	int
hashfunc	*PyObject* *	Py_hash_t
richcmpfunc		*PyObject* *

See *Slot Type typedefs* below for more detail.

12.3.2 PyTypeObject Definition

The structure definition for *PyTypeObject* can be found in `Include/cpython/object.h`. For convenience of reference, this repeats the definition found there:

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;
```

(continua na próxima página)

(continuação da página anterior)

```
    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    // Strong reference on a heap type, borrowed reference on a static type
    struct _typeobject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
    freefunc tp_free; /* Low-level free-memory routine */
    inquiry tp_is_gc; /* For PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_cache;
    PyObject *tp_subclasses;
    PyObject *tp_weaklist;
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6 */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;

    /* bitset of which type-watchers care about this type */
    unsigned char tp_watched;
} PyTypeObject;
```

12.3.3 PyObject Slots

The type object structure extends the *PyVarObject* structure. The *ob_size* field is used for dynamic types (created by `type_new()`, usually called from a class statement). Note that *PyType_Type* (the metatype) initializes *tp_itemsize*, which means that its instances (i.e. type objects) *must* have the *ob_size* field.

Py_ssize_t *PyObject*.**ob_refcnt**

Parte da ABI Estável. This is the type object's reference count, initialized to `1` by the `PyObject_HEAD_INIT` macro. Note that for *statically allocated type objects*, the type's instances (objects whose *ob_type* points back to the type) do *not* count as references. But for *dynamically allocated type objects*, the instances *do* count as references.

Inheritance:

This field is not inherited by subtypes.

PyTypeObject **PyObject*.**ob_type**

Parte da ABI Estável. This is the type's type, in other words its metatype. It is initialized by the argument to the `PyObject_HEAD_INIT` macro, and its value should normally be `&PyType_Type`. However, for dynamically loadable extension modules that must be usable on Windows (at least), the compiler complains that this is not a valid initializer. Therefore, the convention is to pass `NULL` to the `PyObject_HEAD_INIT` macro and to

initialize this field explicitly at the start of the module's initialization function, before doing anything else. This is typically done like this:

```
Foo_Type.ob_type = &PyType_Type;
```

This should be done before any instances of the type are created. *PyType_Ready()* checks if *ob_type* is NULL, and if so, initializes it to the *ob_type* field of the base class. *PyType_Ready()* will not change this field if it is non-zero.

Inheritance:

This field is inherited by subtypes.

12.3.4 PyVarObject Slots

Py_ssize_t *PyVarObject*.**ob_size**

Parte da ABI Estável. 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 should be accessed using the *Py_SIZE()* and *Py_SET_SIZE()* macros.

Inheritance:

This field is not inherited by subtypes.

12.3.5 PyTypeObject Slots

Each slot has a section describing inheritance. If *PyType_Ready()* may set a value when the field is set to NULL then there will also be a "Default" section. (Note that many fields set on *PyBaseObject_Type* and *PyType_Type* effectively act as defaults.)

const char **PyTypeObject*.**tp_name**

Pointer to a NUL-terminated string containing the name of the type. For types that are accessible as module globals, the string should be the full module name, followed by a dot, followed by the type name; for built-in types, it should be just the type name. If the module is a submodule of a package, the full package name is part of the full module name. For example, a type named T defined in module M in subpackage Q in package P should have the *tp_name* initializer "P.Q.M.T".

For *dynamically allocated type objects*, this should just be the type name, and the module name explicitly stored in the type dict as the value for key '__module__'.

For *statically allocated type objects*, the *tp_name* field should contain a dot. Everything before the last dot is made accessible as the __module__ attribute, and everything after the last dot is made accessible as the __name__ attribute.

If no dot is present, the entire *tp_name* field is made accessible as the __name__ attribute, and the __module__ attribute is undefined (unless explicitly set in the dictionary, as explained above). This means your type will be impossible to pickle. Additionally, it will not be listed in module documentations created with pydoc.

This field must not be NULL. It is the only required field in *PyTypeObject()* (other than potentially *tp_itemsize*).

Inheritance:

This field is not inherited by subtypes.

Py_ssize_t *PyTypeObject*.**tp_basicsize**

Py_ssize_t *PyTypeObject*.**tp_itemsize**

These fields allow calculating the size in bytes of instances of the type.

There are two kinds of types: types with fixed-length instances have a zero tp_itemsize field, types with variable-length instances have a non-zero tp_itemsize field. For a type with fixed-length instances, all instances have the same size, given in tp_basicsize. (Exceptions to this rule can be made using *PyUnstable_Object_GC_NewWithExtraData()*.)

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.

Functions like *PyObject_NewVar()* will take the value of N as an argument, and store in the instance's *ob_size* field. Note that the *ob_size* field may later be used for other purposes. For example, int instances use the bits of *ob_size* in an implementation-defined way; the underlying storage and its size should be acessed using PyLong_Export().

Nota

The *ob_size* field should be accessed using the *Py_SIZE()* and *Py_SET_SIZE()* macros.

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 list type has fixed-length instances, yet those instances have a *ob_size* field. (As with int, avoid reading lists' ob_size directly. Call *PyList_Size()* instead.)

The tp_basicsize includes size needed for data of the type's *tp_base*, plus any extra data needed by each instance.

The correct way to set tp_basicsize is to use the sizeof operator on the struct used to declare the instance layout. This struct must include the struct used to declare the base type. In other words, tp_basicsize must be greater than or equal to the base's tp_basicsize.

Since every type is a subtype of object, this struct must include *PyObject* or *PyVarObject* (depending on whether *ob_size* should be included). These are usually defined by the macro *PyObject_HEAD* or *PyObject_VAR_HEAD*, respectively.

The basic size does not include the GC header size, as that header is not part of *PyObject_HEAD*.

For cases where struct used to declare the base type is unknown, see *PyType_Spec.basicsize* and *PyType_FromMetaclass()*.

Notes about alignment:

- tp_basicsize must be a multiple of _Alignof(PyObject). When using sizeof on a struct that includes *PyObject_HEAD*, as recommended, the compiler ensures this. When not using a C struct, or when using compiler extensions like __attribute__((packed)), it is up to you.
- If the variable items require a particular alignment, tp_basicsize and tp_itemsize must each be a multiple of that alignment. For example, if a type's variable part stores a double, it is your responsibility that both fields are a multiple of _Alignof(double).

Inheritance:

These fields are inherited separately by subtypes. (That is, if the field is set to zero, *PyType_Ready()* will copy the value from the base type, indicating that the instances do not need additional storage.)

If the base type has a non-zero *tp_itemsize*, it is generally not safe to set *tp_itemsize* to a different non-zero value in a subtype (though this depends on the implementation of the base type).

destructor *PyTypeObject*.**tp_dealloc**

A pointer to the instance destructor function. This function must be defined unless the type guarantees that its instances will never be deallocated (as is the case for the singletons None and Ellipsis). The function signature is:

```
void tp_dealloc(PyObject *self);
```

The destructor function is called by the *Py_DECREF()* and *Py_XDECREF()* macros when the new reference count is zero. At this point, the instance is still in existence, but there are no references to it. The destructor function should free all references which the instance owns, free all memory buffers owned by the instance (using the freeing function corresponding to the allocation function used to allocate the buffer), and call the type's *tp_free* function. If the type is not subtypable (doesn't have the *Py_TPFLAGS_BASETYPE* flag bit set), it is permissible to call the object deallocator directly instead of via *tp_free*. The object deallocator should be the one used to allocate the instance; this is normally *PyObject_Del()* if the instance was allocated

using *PyObject_New* or *PyObject_NewVar*, or *PyObject_GC_Del()* if the instance was allocated using *PyObject_GC_New* or *PyObject_GC_NewVar*.

If the type supports garbage collection (has the *Py_TPFLAGS_HAVE_GC* flag bit set), the destructor should call *PyObject_GC_UnTrack()* before clearing any member fields.

```
static void foo_dealloc(foo_object *self) {
    PyObject_GC_UnTrack(self);
    Py_CLEAR(self->ref);
    Py_TYPE(self)->tp_free((PyObject *)self);
}
```

Finally, if the type is heap allocated (*Py_TPFLAGS_HEAPTYPE*), the deallocator should release the owned reference to its type object (via *Py_DECREF()*) after calling the type deallocator. In order to avoid dangling pointers, the recommended way to achieve this is:

```
static void foo_dealloc(foo_object *self) {
    PyTypeObject *tp = Py_TYPE(self);
    // free references and buffers here
    tp->tp_free(self);
    Py_DECREF(tp);
}
```

Aviso

In a garbage collected Python, `tp_dealloc` may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a garbage collection on any thread). This is not a problem for Python API calls, since the thread on which `tp_dealloc` is called will own the Global Interpreter Lock (GIL). However, if the object being destroyed in turn destroys objects from some other C or C++ library, care should be taken to ensure that destroying those objects on the thread which called `tp_dealloc` will not violate any assumptions of the library.

Inheritance:

This field is inherited by subtypes.

Py_ssize_t *PyTypeObject*.**tp_vectorcall_offset**

An optional offset to a per-instance function that implements calling the object using the *vectorcall protocol*, a more efficient alternative of the simpler *tp_call*.

This field is only used if the flag *Py_TPFLAGS_HAVE_VECTORCALL* is set. If so, this must be a positive integer containing the offset in the instance of a *vectorcallfunc* pointer.

The *vectorcallfunc* pointer may be `NULL`, in which case the instance behaves as if *Py_TPFLAGS_HAVE_VECTORCALL* was not set: calling the instance falls back to *tp_call*.

Any class that sets `Py_TPFLAGS_HAVE_VECTORCALL` must also set *tp_call* and make sure its behaviour is consistent with the *vectorcallfunc* function. This can be done by setting *tp_call* to *PyVectorcall_Call()*.

Alterado na versão 3.8: Before version 3.8, this slot was named `tp_print`. In Python 2.x, it was used for printing to a file. In Python 3.0 to 3.7, it was unused.

Alterado na versão 3.12: Before version 3.12, it was not recommended for *mutable heap types* to implement the vectorcall protocol. When a user sets `__call__` in Python code, only *tp_call* is updated, likely making it inconsistent with the vectorcall function. Since 3.12, setting `__call__` will disable vectorcall optimization by clearing the *Py_TPFLAGS_HAVE_VECTORCALL* flag.

Inheritance:

This field is always inherited. However, the *Py_TPFLAGS_HAVE_VECTORCALL* flag is not always inherited. If it's not set, then the subclass won't use *vectorcall*, except when *PyVectorcall_Call()* is explicitly called.

getattrfunc *PyTypeObject*.**tp_getattr**

An optional pointer to the get-attribute-string function.

This field is deprecated. When it is defined, it should point to a function that acts the same as the *tp_getattro* function, but taking a C string instead of a Python string object to give the attribute name.

Inheritance:

Group: *tp_getattr*, *tp_getattro*

This field is inherited by subtypes together with *tp_getattro*: a subtype inherits both *tp_getattr* and *tp_getattro* from its base type when the subtype's *tp_getattr* and *tp_getattro* are both `NULL`.

setattrfunc *PyTypeObject*.**tp_setattr**

An optional pointer to the function for setting and deleting attributes.

This field is deprecated. When it is defined, it should point to a function that acts the same as the *tp_setattro* function, but taking a C string instead of a Python string object to give the attribute name.

Inheritance:

Group: *tp_setattr*, *tp_setattro*

This field is inherited by subtypes together with *tp_setattro*: a subtype inherits both *tp_setattr* and *tp_setattro* from its base type when the subtype's *tp_setattr* and *tp_setattro* are both `NULL`.

PyAsyncMethods **PyTypeObject*.**tp_as_async**

Pointer to an additional structure that contains fields relevant only to objects which implement *awaitable* and *asynchronous iterator* protocols at the C-level. See *Async Object Structures* for details.

Adicionado na versão 3.5: Formerly known as `tp_compare` and `tp_reserved`.

Inheritance:

The *tp_as_async* field is not inherited, but the contained fields are inherited individually.

reprfunc *PyTypeObject*.**tp_repr**

An optional pointer to a function that implements the built-in function `repr()`.

The signature is the same as for *PyObject_Repr()*:

```
PyObject *tp_repr(PyObject *self);
```

The function must return a string or a Unicode object. Ideally, this function should return a string that, when passed to `eval()`, given a suitable environment, returns an object with the same value. If this is not feasible, it should return a string starting with `'<'` and ending with `'>'` from which both the type and the value of the object can be deduced.

Inheritance:

This field is inherited by subtypes.

Padrão:

When this field is not set, a string of the form `<%s object at %p>` is returned, where `%s` is replaced by the type name, and `%p` by the object's memory address.

PyNumberMethods **PyTypeObject*.**tp_as_number**

Pointer to an additional structure that contains fields relevant only to objects which implement the number protocol. These fields are documented in *Number Object Structures*.

Inheritance:

The *tp_as_number* field is not inherited, but the contained fields are inherited individually.

PySequenceMethods **PyTypeObject.tp_as_sequence*

Pointer to an additional structure that contains fields relevant only to objects which implement the sequence protocol. These fields are documented in *Sequence Object Structures*.

Inheritance:

The *tp_as_sequence* field is not inherited, but the contained fields are inherited individually.

PyMappingMethods **PyTypeObject.tp_as_mapping*

Pointer to an additional structure that contains fields relevant only to objects which implement the mapping protocol. These fields are documented in *Mapping Object Structures*.

Inheritance:

The *tp_as_mapping* field is not inherited, but the contained fields are inherited individually.

hashfunc *PyTypeObject*.**tp_hash**

An optional pointer to a function that implements the built-in function `hash()`.

The signature is the same as for *PyObject_Hash()*:

```
Py_hash_t tp_hash(PyObject *);
```

The value `-1` should not be returned as a normal return value; when an error occurs during the computation of the hash value, the function should set an exception and return `-1`.

When this field is not set (*and* *tp_richcompare* is not set), an attempt to take the hash of the object raises `TypeError`. This is the same as setting it to *PyObject_HashNotImplemented()*.

This field can be set explicitly to *PyObject_HashNotImplemented()* to block inheritance of the hash method from a parent type. This is interpreted as the equivalent of `__hash__ = None` at the Python level, causing `isinstance(o, collections.Hashable)` to correctly return `False`. Note that the converse is also true - setting `__hash__ = None` on a class at the Python level will result in the `tp_hash` slot being set to *PyObject_HashNotImplemented()*.

Inheritance:

Group: *tp_hash*, *tp_richcompare*

This field is inherited by subtypes together with *tp_richcompare*: a subtype inherits both of *tp_richcompare* and *tp_hash*, when the subtype's *tp_richcompare* and *tp_hash* are both `NULL`.

Padrão:

PyBaseObject_Type uses *PyObject_GenericHash()*.

ternaryfunc *PyTypeObject*.**tp_call**

An optional pointer to a function that implements calling the object. This should be `NULL` if the object is not callable. The signature is the same as for *PyObject_Call()*:

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

Inheritance:

This field is inherited by subtypes.

reprfunc *PyTypeObject*.**tp_str**

An optional pointer to a function that implements the built-in operation `str()`. (Note that `str` is a type now, and `str()` calls the constructor for that type. This constructor calls *PyObject_Str()* to do the actual work, and *PyObject_Str()* will call this handler.)

The signature is the same as for *PyObject_Str()*:

```
PyObject *tp_str(PyObject *self);
```

The function must return a string or a Unicode object. It should be a "friendly" string representation of the object, as this is the representation that will be used, among other things, by the `print()` function.

Inheritance:

This field is inherited by subtypes.

Padrão:

When this field is not set, *PyObject_Repr()* is called to return a string representation.

getattrofunc *PyTypeObject*.**tp_getattro**

An optional pointer to the get-attribute function.

The signature is the same as for *PyObject_GetAttr()*:

```
PyObject *tp_getattro(PyObject *self, PyObject *attr);
```

It is usually convenient to set this field to *PyObject_GenericGetAttr()*, which implements the normal way of looking for object attributes.

Inheritance:

Group: *tp_getattr*, *tp_getattro*

This field is inherited by subtypes together with *tp_getattr*: a subtype inherits both *tp_getattr* and *tp_getattro* from its base type when the subtype's *tp_getattr* and *tp_getattro* are both `NULL`.

Padrão:

PyBaseObject_Type uses *PyObject_GenericGetAttr()*.

setattrofunc *PyTypeObject*.**tp_setattro**

An optional pointer to the function for setting and deleting attributes.

The signature is the same as for *PyObject_SetAttr()*:

```
int tp_setattro(PyObject *self, PyObject *attr, PyObject *value);
```

In addition, setting *value* to `NULL` to delete an attribute must be supported. It is usually convenient to set this field to *PyObject_GenericSetAttr()*, which implements the normal way of setting object attributes.

Inheritance:

Group: *tp_setattr*, *tp_setattro*

This field is inherited by subtypes together with *tp_setattr*: a subtype inherits both *tp_setattr* and *tp_setattro* from its base type when the subtype's *tp_setattr* and *tp_setattro* are both `NULL`.

Padrão:

PyBaseObject_Type uses *PyObject_GenericSetAttr()*.

PyBufferProcs **PyTypeObject*.**tp_as_buffer**

Pointer to an additional structure that contains fields relevant only to objects which implement the buffer interface. These fields are documented in *Buffer Object Structures*.

Inheritance:

The *tp_as_buffer* field is not inherited, but the contained fields are inherited individually.

unsigned long *PyTypeObject*.**tp_flags**

This field is a bit mask of various flags. Some flags indicate variant semantics for certain situations; others are used to indicate that certain fields in the type object (or in the extension structures referenced via *tp_as_number*, *tp_as_sequence*, *tp_as_mapping*, and *tp_as_buffer*) that were historically not always present are valid; if such a flag bit is clear, the type fields it guards must not be accessed and must be considered to have a zero or `NULL` value instead.

Inheritance:

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type's value of the flag bit is copied into the subtype together with a pointer to the extension structure. The *Py_TPFLAGS_HAVE_GC* flag bit is inherited together with the *tp_traverse* and *tp_clear* fields, i.e. if the *Py_TPFLAGS_HAVE_GC* flag bit is clear in the subtype and the *tp_traverse* and *tp_clear* fields in the subtype exist and have NULL values. .. XXX are most flag bits *really* inherited individually?

Padrão:

PyBaseObject_Type uses Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE.

Bit Masks:

The following bit masks are currently defined; these can be ORed together using the | operator to form the value of the *tp_flags* field. The macro *PyType_HasFeature()* takes a type and a flags value, *tp* and *f*, and checks whether tp->tp_flags & f is non-zero.

Py_TPFLAGS_HEAPTYPE

This bit is set when the type object itself is allocated on the heap, for example, types created dynamically using *PyType_FromSpec()*. In this case, the *ob_type* field of its instances is considered a reference to the type, and the type object is INCREF'ed when a new instance is created, and DECREF'ed when an instance is destroyed (this does not apply to instances of subtypes; only the type referenced by the instance's ob_type gets INCREF'ed or DECREF'ed). Heap types should also *support garbage collection* as they can form a reference cycle with their own module object.

Inheritance:

???

Py_TPFLAGS_BASETYPE

This bit is set when the type can be used as the base type of another type. If this bit is clear, the type cannot be subtyped (similar to a "final" class in Java).

Inheritance:

???

Py_TPFLAGS_READY

This bit is set when the type object has been fully initialized by *PyType_Ready()*.

Inheritance:

???

Py_TPFLAGS_READYING

This bit is set while *PyType_Ready()* is in the process of initializing the type object.

Inheritance:

???

Py_TPFLAGS_HAVE_GC

This bit is set when the object supports garbage collection. If this bit is set, instances must be created using *PyObject_GC_New* and destroyed using *PyObject_GC_Del()*. More information in section *Suporte a Coleta Cíclica de Lixo*. This bit also implies that the GC-related fields *tp_traverse* and *tp_clear* are present in the type object.

Inheritance:

Group: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

The *Py_TPFLAGS_HAVE_GC* flag bit is inherited together with the *tp_traverse* and *tp_clear* fields, i.e. if the *Py_TPFLAGS_HAVE_GC* flag bit is clear in the subtype and the *tp_traverse* and *tp_clear* fields in the subtype exist and have NULL values.

Py_TPFLAGS_DEFAULT

This is a bitmask of all the bits that pertain to the existence of certain fields in the type object and its extension structures. Currently, it includes the following bits: `Py_TPFLAGS_HAVE_STACKLESS_EXTENSION`.

Inheritance:

???

Py_TPFLAGS_METHOD_DESCRIPTOR

This bit indicates that objects behave like unbound methods.

If this flag is set for `type(meth)`, then:

- `meth.__get__(obj, cls)(*args, **kwds)` (with `obj` not None) must be equivalent to `meth(obj, *args, **kwds)`.
- `meth.__get__(None, cls)(*args, **kwds)` must be equivalent to `meth(*args, **kwds)`.

This flag enables an optimization for typical method calls like `obj.meth()`: it avoids creating a temporary "bound method" object for `obj.meth`.

Adicionado na versão 3.8.

Inheritance:

This flag is never inherited by types without the *Py_TPFLAGS_IMMUTABLETYPE* flag set. For extension types, it is inherited whenever *tp_descr_get* is inherited.

Py_TPFLAGS_MANAGED_DICT

This bit indicates that instances of the class have a *~object.__dict__* attribute, and that the space for the dictionary is managed by the VM.

If this flag is set, *Py_TPFLAGS_HAVE_GC* should also be set.

The type traverse function must call *PyObject_VisitManagedDict()* and its clear function must call *PyObject_ClearManagedDict()*.

Adicionado na versão 3.12.

Inheritance:

This flag is inherited unless the *tp_dictoffset* field is set in a superclass.

Py_TPFLAGS_MANAGED_WEAKREF

This bit indicates that instances of the class should be weakly referenceable.

Adicionado na versão 3.12.

Inheritance:

This flag is inherited unless the *tp_weaklistoffset* field is set in a superclass.

Py_TPFLAGS_ITEMS_AT_END

Only usable with variable-size types, i.e. ones with non-zero *tp_itemsize*.

Indicates that the variable-sized portion of an instance of this type is at the end of the instance's memory area, at an offset of `Py_TYPE(obj)->tp_basicsize` (which may be different in each subclass).

When setting this flag, be sure that all superclasses either use this memory layout, or are not variable-sized. Python does not check this.

Adicionado na versão 3.12.

Inheritance:

This flag is inherited.

Py_TPFLAGS_LONG_SUBCLASS

Py_TPFLAGS_LIST_SUBCLASS

Py_TPFLAGS_TUPLE_SUBCLASS

Py_TPFLAGS_BYTES_SUBCLASS

Py_TPFLAGS_UNICODE_SUBCLASS

Py_TPFLAGS_DICT_SUBCLASS

Py_TPFLAGS_BASE_EXC_SUBCLASS

Py_TPFLAGS_TYPE_SUBCLASS

These flags are used by functions such as *PyLong_Check()* to quickly determine if a type is a subclass of a built-in type; such specific checks are faster than a generic check, like *PyObject_IsInstance()*. Custom types that inherit from built-ins should have their *tp_flags* set appropriately, or the code that interacts with such types will behave differently depending on what kind of check is used.

Py_TPFLAGS_HAVE_FINALIZE

This bit is set when the *tp_finalize* slot is present in the type structure.

Adicionado na versão 3.4.

Descontinuado desde a versão 3.8: This flag isn't necessary anymore, as the interpreter assumes the *tp_finalize* slot is always present in the type structure.

Py_TPFLAGS_HAVE_VECTORCALL

This bit is set when the class implements the *vectorcall protocol*. See *tp_vectorcall_offset* for details.

Inheritance:

This bit is inherited if *tp_call* is also inherited.

Adicionado na versão 3.9.

Alterado na versão 3.12: This flag is now removed from a class when the class's `__call__()` method is reassigned.

This flag can now be inherited by mutable classes.

Py_TPFLAGS_IMMUTABLETYPE

This bit is set for type objects that are immutable: type attributes cannot be set nor deleted.

PyType_Ready() automatically applies this flag to *static types*.

Inheritance:

This flag is not inherited.

Adicionado na versão 3.10.

Py_TPFLAGS_DISALLOW_INSTANTIATION

Disallow creating instances of the type: set *tp_new* to NULL and don't create the `__new__` key in the type dictionary.

The flag must be set before creating the type, not after. For example, it must be set before *PyType_Ready()* is called on the type.

The flag is set automatically on *static types* if *tp_base* is NULL or `&PyBaseObject_Type` and *tp_new* is NULL.

Inheritance:

This flag is not inherited. However, subclasses will not be instantiable unless they provide a non-NULL *tp_new* (which is only possible via the C API).

> **Nota**
>
> To disallow instantiating a class directly but allow instantiating its subclasses (e.g. for an *abstract base class*), do not use this flag. Instead, make `tp_new` only succeed for subclasses.

Adicionado na versão 3.10.

Py_TPFLAGS_MAPPING

This bit indicates that instances of the class may match mapping patterns when used as the subject of a `match` block. It is automatically set when registering or subclassing `collections.abc.Mapping`, and unset when registering `collections.abc.Sequence`.

> **Nota**
>
> `Py_TPFLAGS_MAPPING` and `Py_TPFLAGS_SEQUENCE` are mutually exclusive; it is an error to enable both flags simultaneously.

Inheritance:

This flag is inherited by types that do not already set `Py_TPFLAGS_SEQUENCE`.

> **Ver também**
>
> **PEP 634** – Structural Pattern Matching: Specification

Adicionado na versão 3.10.

Py_TPFLAGS_SEQUENCE

This bit indicates that instances of the class may match sequence patterns when used as the subject of a `match` block. It is automatically set when registering or subclassing `collections.abc.Sequence`, and unset when registering `collections.abc.Mapping`.

> **Nota**
>
> `Py_TPFLAGS_MAPPING` and `Py_TPFLAGS_SEQUENCE` are mutually exclusive; it is an error to enable both flags simultaneously.

Inheritance:

This flag is inherited by types that do not already set `Py_TPFLAGS_MAPPING`.

> **Ver também**
>
> **PEP 634** – Structural Pattern Matching: Specification

Adicionado na versão 3.10.

Py_TPFLAGS_VALID_VERSION_TAG

Internal. Do not set or unset this flag. To indicate that a class has changed call `PyType_Modified()`

> **Aviso**
>
> This flag is present in header files, but is not be used. It will be removed in a future version of CPython

const char *PyTypeObject.**tp_doc**

An optional pointer to a NUL-terminated C string giving the docstring for this type object. This is exposed as the `__doc__` attribute on the type and instances of the type.

Inheritance:

This field is *not* inherited by subtypes.

traverseproc PyTypeObject.**tp_traverse**

An optional pointer to a traversal function for the garbage collector. This is only used if the *Py_TPFLAGS_HAVE_GC* flag bit is set. The signature is:

```
int tp_traverse(PyObject *self, visitproc visit, void *arg);
```

More information about Python's garbage collection scheme can be found in section *Suporte a Coleta Cíclica de Lixo*.

The *tp_traverse* pointer is used by the garbage collector to detect reference cycles. A typical implementation of a *tp_traverse* function simply calls *Py_VISIT()* on each of the instance's members that are Python objects that the instance owns. For example, this is function `local_traverse()` from the `_thread` extension module:

```
static int
local_traverse(localobject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

Note that *Py_VISIT()* is called only on those members that can participate in reference cycles. Although there is also a `self->key` member, it can only be `NULL` or a Python string and therefore cannot be part of a reference cycle.

On the other hand, even if you know a member can never be part of a cycle, as a debugging aid you may want to visit it anyway just so the `gc` module's `get_referents()` function will include it.

Heap types (*Py_TPFLAGS_HEAPTYPE*) must visit their type with:

```
Py_VISIT(Py_TYPE(self));
```

It is only needed since Python 3.9. To support Python 3.8 and older, this line must be conditional:

```
#if PY_VERSION_HEX >= 0x03090000
    Py_VISIT(Py_TYPE(self));
#endif
```

If the *Py_TPFLAGS_MANAGED_DICT* bit is set in the *tp_flags* field, the traverse function must call *PyObject_VisitManagedDict()* like this:

```
PyObject_VisitManagedDict((PyObject*)self, visit, arg);
```

Aviso

When implementing *tp_traverse*, only the members that the instance *owns* (by having *strong references* to them) must be visited. For instance, if an object supports weak references via the *tp_weaklist* slot, the pointer supporting the linked list (what *tp_weaklist* points to) must **not** be visited as the instance does not directly own the weak references to itself (the weakreference list is there to support the weak reference

machinery, but the instance has no strong reference to the elements inside it, as they are allowed to be removed even if the instance is still alive).

Note that *Py_VISIT()* requires the *visit* and *arg* parameters to `local_traverse()` to have these specific names; don't name them just anything.

Instances of *heap-allocated types* hold a reference to their type. Their traversal function must therefore either visit *Py_TYPE(self)*, or delegate this responsibility by calling `tp_traverse` of another heap-allocated type (such as a heap-allocated superclass). If they do not, the type object may not be garbage-collected.

Alterado na versão 3.9: Heap-allocated types are expected to visit `Py_TYPE(self)` in `tp_traverse`. In earlier versions of Python, due to bug 40217, doing this may lead to crashes in subclasses.

Inheritance:

Group: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

This field is inherited by subtypes together with *tp_clear* and the *Py_TPFLAGS_HAVE_GC* flag bit: the flag bit, *tp_traverse*, and *tp_clear* are all inherited from the base type if they are all zero in the subtype.

inquiry *PyTypeObject*.**tp_clear**

An optional pointer to a clear function for the garbage collector. This is only used if the *Py_TPFLAGS_HAVE_GC* flag bit is set. The signature is:

```
int tp_clear(PyObject *);
```

The *tp_clear* member function is used to break reference cycles in cyclic garbage detected by the garbage collector. Taken together, all *tp_clear* functions in the system must combine to break all reference cycles. This is subtle, and if in any doubt supply a *tp_clear* function. For example, the tuple type does not implement a *tp_clear* function, because it's possible to prove that no reference cycle can be composed entirely of tuples. Therefore the *tp_clear* functions of other types must be sufficient to break any cycle containing a tuple. This isn't immediately obvious, and there's rarely a good reason to avoid implementing *tp_clear*.

Implementations of *tp_clear* should drop the instance's references to those of its members that may be Python objects, and set its pointers to those members to `NULL`, as in the following example:

```
static int
local_clear(localobject *self)
{
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

The *Py_CLEAR()* macro should be used, because clearing references is delicate: the reference to the contained object must not be released (via *Py_DECREF()*) until after the pointer to the contained object is set to `NULL`. This is because releasing the reference may cause the contained object to become trash, triggering a chain of reclamation activity that may include invoking arbitrary Python code (due to finalizers, or weakref callbacks, associated with the contained object). If it's possible for such code to reference *self* again, it's important that the pointer to the contained object be `NULL` at that time, so that *self* knows the contained object can no longer be used. The *Py_CLEAR()* macro performs the operations in a safe order.

If the *Py_TPFLAGS_MANAGED_DICT* bit is set in the *tp_flags* field, the traverse function must call *PyObject_ClearManagedDict()* like this:

```
PyObject_ClearManagedDict((PyObject*)self);
```

Note that *tp_clear* is not *always* called before an instance is deallocated. For example, when reference counting is enough to determine that an object is no longer used, the cyclic garbage collector is not involved

and *tp_dealloc* is called directly.

Because the goal of *tp_clear* functions is to break reference cycles, it's not necessary to clear contained objects like Python strings or Python integers, which can't participate in reference cycles. On the other hand, it may be convenient to clear all contained Python objects, and write the type's *tp_dealloc* function to invoke *tp_clear*.

More information about Python's garbage collection scheme can be found in section *Suporte a Coleta Cíclica de Lixo*.

Inheritance:

Group: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

This field is inherited by subtypes together with *tp_traverse* and the *Py_TPFLAGS_HAVE_GC* flag bit: the flag bit, *tp_traverse*, and *tp_clear* are all inherited from the base type if they are all zero in the subtype.

richcmpfunc *PyTypeObject*.**tp_richcompare**

An optional pointer to the rich comparison function, whose signature is:

```
PyObject *tp_richcompare(PyObject *self, PyObject *other, int op);
```

The first parameter is guaranteed to be an instance of the type that is defined by *PyTypeObject*.

The function should return the result of the comparison (usually `Py_True` or `Py_False`). If the comparison is undefined, it must return `Py_NotImplemented`, if another error occurred it must return `NULL` and set an exception condition.

The following constants are defined to be used as the third argument for *tp_richcompare* and for *PyObject_RichCompare()*:

Constante	Comparação
Py_LT	<
Py_LE	<=
Py_EQ	==
Py_NE	!=
Py_GT	>
Py_GE	>=

The following macro is defined to ease writing rich comparison functions:

Py_RETURN_RICHCOMPARE(VAL_A, VAL_B, op)

Return `Py_True` or `Py_False` from the function, depending on the result of a comparison. VAL_A and VAL_B must be orderable by C comparison operators (for example, they may be C ints or floats). The third argument specifies the requested operation, as for *PyObject_RichCompare()*.

The returned value is a new *strong reference*.

On error, sets an exception and returns `NULL` from the function.

Adicionado na versão 3.7.

Inheritance:

Group: *tp_hash*, *tp_richcompare*

This field is inherited by subtypes together with *tp_hash*: a subtype inherits *tp_richcompare* and *tp_hash* when the subtype's *tp_richcompare* and *tp_hash* are both NULL.

Padrão:

PyBaseObject_Type provides a *tp_richcompare* implementation, which may be inherited. However, if only *tp_hash* is defined, not even the inherited function is used and instances of the type will not be able to participate in any comparisons.

Py_ssize_t *PyTypeObject*.**tp_weaklistoffset**

While this field is still supported, *Py_TPFLAGS_MANAGED_WEAKREF* should be used instead, if at all possible.

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.

It is an error to set both the *Py_TPFLAGS_MANAGED_WEAKREF* bit and *tp_weaklistoffset*.

Inheritance:

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype uses a different weak reference list head than the base type. Since the list head is always found via *tp_weaklistoffset*, this should not be a problem.

Padrão:

If the *Py_TPFLAGS_MANAGED_WEAKREF* bit is set in the *tp_flags* field, then *tp_weaklistoffset* will be set to a negative value, to indicate that it is unsafe to use this field.

getiterfunc *PyTypeObject*.**tp_iter**

An optional pointer to a function that returns an *iterator* for the object. Its presence normally signals that the instances of this type are *iterable* (although sequences may be iterable without this function).

This function has the same signature as *PyObject_GetIter()*:

```
PyObject *tp_iter(PyObject *self);
```

Inheritance:

This field is inherited by subtypes.

iternextfunc *PyTypeObject*.**tp_iternext**

An optional pointer to a function that returns the next item in an *iterator*. The signature is:

```
PyObject *tp_iternext(PyObject *self);
```

When the iterator is exhausted, it must return NULL; a StopIteration exception may or may not be set. When another error occurs, it must return NULL too. Its presence signals that the instances of this type are iterators.

Iterator types should also define the *tp_iter* function, and that function should return the iterator instance itself (not a new iterator instance).

This function has the same signature as *PyIter_Next()*.

Inheritance:

This field is inherited by subtypes.

struct *PyMethodDef* **PyTypeObject.tp_methods*

An optional pointer to a static NULL-terminated array of *PyMethodDef* structures, declaring regular methods of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a method descriptor.

Inheritance:

This field is not inherited by subtypes (methods are inherited through a different mechanism).

struct *PyMemberDef* **PyTypeObject.tp_members*

An optional pointer to a static NULL-terminated array of *PyMemberDef* structures, declaring regular data members (fields or slots) of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a member descriptor.

Inheritance:

This field is not inherited by subtypes (members are inherited through a different mechanism).

struct *PyGetSetDef* **PyTypeObject.tp_getset*

An optional pointer to a static NULL-terminated array of *PyGetSetDef* structures, declaring computed attributes of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a getset descriptor.

Inheritance:

This field is not inherited by subtypes (computed attributes are inherited through a different mechanism).

PyTypeObject **PyTypeObject.tp_base*

An optional pointer to a base type from which type properties are inherited. At this level, only single inheritance is supported; multiple inheritance require dynamically creating a type object by calling the metatype.

> **Nota**
>
> Slot initialization is subject to the rules of initializing globals. C99 requires the initializers to be "address constants". Function designators like *PyType_GenericNew()*, with implicit conversion to a pointer, are valid C99 address constants.
>
> However, the unary '&' operator applied to a non-static variable like *PyBaseObject_Type* is not required to produce an address constant. Compilers may support this (gcc does), MSVC does not. Both compilers are strictly standard conforming in this particular behavior.
>
> Consequently, *tp_base* should be set in the extension module's init function.

Inheritance:

This field is not inherited by subtypes (obviously).

Padrão:

This field defaults to `&PyBaseObject_Type` (which to Python programmers is known as the type `object`).

PyObject **PyTypeObject.tp_dict*

The type's dictionary is stored here by *PyType_Ready()*.

This field should normally be initialized to NULL before PyType_Ready is called; it may also be initialized to a dictionary containing initial attributes for the type. Once *PyType_Ready()* has initialized the type, extra attributes for the type may be added to this dictionary only if they don't correspond to overloaded operations (like `__add__()`). Once initialization for the type has finished, this field should be treated as read-only.

Some types may not store their dictionary in this slot. Use *PyType_GetDict()* to retrieve the dictionary for an arbitrary type.

Alterado na versão 3.12: Internals detail: For static builtin types, this is always NULL. Instead, the dict for such types is stored on PyInterpreterState. Use *PyType_GetDict()* to get the dict for an arbitrary type.

Inheritance:

This field is not inherited by subtypes (though the attributes defined in here are inherited through a different mechanism).

Padrão:

If this field is NULL, *PyType_Ready()* will assign a new dictionary to it.

> **Aviso**
>
> It is not safe to use *PyDict_SetItem()* on or otherwise modify *tp_dict* with the dictionary C-API.

descrgetfunc *PyTypeObject*.**tp_descr_get**

An optional pointer to a "descriptor get" function.

A assinatura da função é:

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *type);
```

Inheritance:

This field is inherited by subtypes.

descrsetfunc *PyTypeObject*.**tp_descr_set**

An optional pointer to a function for setting and deleting a descriptor's value.

A assinatura da função é:

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

The *value* argument is set to NULL to delete the value.

Inheritance:

This field is inherited by subtypes.

Py_ssize_t *PyTypeObject*.**tp_dictoffset**

While this field is still supported, *Py_TPFLAGS_MANAGED_DICT* should be used instead, if at all possible.

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.

The value specifies the offset of the dictionary from the start of the instance structure.

The *tp_dictoffset* should be regarded as write-only. To get the pointer to the dictionary call *PyObject_GenericGetDict()*. Calling *PyObject_GenericGetDict()* may need to allocate memory for the dictionary, so it is may be more efficient to call *PyObject_GetAttr()* when accessing an attribute on the object.

It is an error to set both the *Py_TPFLAGS_MANAGED_DICT* bit and *tp_dictoffset*.

Inheritance:

This field is inherited by subtypes. A subtype should not override this offset; doing so could be unsafe, if C code tries to access the dictionary at the previous offset. To properly support inheritance, use *Py_TPFLAGS_MANAGED_DICT*.

Padrão:

This slot has no default. For *static types*, if the field is `NULL` then no `__dict__` gets created for instances.

If the *Py_TPFLAGS_MANAGED_DICT* bit is set in the *tp_flags* field, then *tp_dictoffset* will be set to `-1`, to indicate that it is unsafe to use this field.

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.

A assinatura da função é:

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwds);
```

The self argument is the instance to be initialized; the *args* and *kwds* arguments represent positional and keyword arguments of the call to `__init__()`.

The *tp_init* function, if not `NULL`, is called when an instance is created normally by calling its type, after the type's *tp_new* function has returned an instance of the type. If the *tp_new* function returns an instance of some other type that is not a subtype of the original type, no *tp_init* function is called; if *tp_new* returns an instance of a subtype of the original type, the subtype's *tp_init* is called.

Returns `0` on success, `-1` and sets an exception on error.

Inheritance:

This field is inherited by subtypes.

Padrão:

For *static types* this field does not have a default.

allocfunc *PyTypeObject*.**tp_alloc**

An optional pointer to an instance allocation function.

A assinatura da função é:

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems);
```

Inheritance:

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement).

Padrão:

For dynamic subtypes, this field is always set to *PyType_GenericAlloc()*, to force a standard heap allocation strategy.

For static subtypes, *PyBaseObject_Type* uses *PyType_GenericAlloc()*. That is the recommended value for all statically defined types.

newfunc *PyTypeObject*.**tp_new**

An optional pointer to an instance creation function.

A assinatura da função é:

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds);
```

The *subtype* argument is the type of the object being created; the *args* and *kwds* arguments represent positional and keyword arguments of the call to the type. Note that *subtype* doesn't have to equal the type whose *tp_new* function is called; it may be a subtype of that type (but not an unrelated type).

The *tp_new* function should call `subtype->tp_alloc(subtype, nitems)` to allocate space for the object, and then do only as much further initialization as is absolutely necessary. Initialization that can safely be

ignored or repeated should be placed in the *tp_init* handler. A good rule of thumb is that for immutable types, all initialization should take place in *tp_new*, while for mutable types, most initialization should be deferred to *tp_init*.

Set the *Py_TPFLAGS_DISALLOW_INSTANTIATION* flag to disallow creating instances of the type in Python.

Inheritance:

This field is inherited by subtypes, except it is not inherited by *static types* whose *tp_base* is `NULL` or `&PyBaseObject_Type`.

Padrão:

For *static types* this field has no default. This means if the slot is defined as `NULL`, the type cannot be called to create new instances; presumably there is some other way to create instances, like a factory function.

freefunc *PyTypeObject*.**tp_free**

An optional pointer to an instance deallocation function. Its signature is:

```
void tp_free(void *self);
```

An initializer that is compatible with this signature is *PyObject_Free()*.

Inheritance:

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement)

Padrão:

In dynamic subtypes, this field is set to a deallocator suitable to match *PyType_GenericAlloc()* and the value of the *Py_TPFLAGS_HAVE_GC* flag bit.

For static subtypes, *PyBaseObject_Type* uses *PyObject_Del()*.

inquiry *PyTypeObject*.**tp_is_gc**

An optional pointer to a function called by the garbage collector.

The garbage collector needs to know whether a particular object is collectible or not. Normally, it is sufficient to look at the object's type's *tp_flags* field, and check the *Py_TPFLAGS_HAVE_GC* flag bit. But some types have a mixture of statically and dynamically allocated instances, and the statically allocated instances are not collectible. Such types should define this function; it should return `1` for a collectible instance, and `0` for a non-collectible instance. The signature is:

```
int tp_is_gc(PyObject *self);
```

(The only example of this are types themselves. The metatype, *PyType_Type*, defines this function to distinguish between statically and *dynamically allocated types*.)

Inheritance:

This field is inherited by subtypes.

Padrão:

This slot has no default. If this field is `NULL`, *Py_TPFLAGS_HAVE_GC* is used as the functional equivalent.

PyObject ***PyTypeObject*.**tp_bases**

Tuple of base types.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is *initialized*.

For dynamically created classes, the `Py_tp_bases` *slot* can be used instead of the *bases* argument of *PyType_FromSpecWithBases()*. The argument form is preferred.

> **Aviso**
>
> Multiple inheritance does not work well for statically defined types. If you set `tp_bases` to a tuple, Python will not raise an error, but some slots will only be inherited from the first base.

Inheritance:

This field is not inherited.

PyObject **PyTypeObject*.**tp_mro**

Tuple containing the expanded set of base types, starting with the type itself and ending with `object`, in Method Resolution Order.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is *initialized*.

Inheritance:

This field is not inherited; it is calculated fresh by *PyType_Ready()*.

PyObject **PyTypeObject*.**tp_cache**

Unused. Internal use only.

Inheritance:

This field is not inherited.

void **PyTypeObject*.**tp_subclasses**

A collection of subclasses. Internal use only. May be an invalid pointer.

To get a list of subclasses, call the Python method `__subclasses__()`.

Alterado na versão 3.12: For some types, this field does not hold a valid *PyObject**. The type was changed to `void*` to indicate this.

Inheritance:

This field is not inherited.

PyObject **PyTypeObject*.**tp_weaklist**

Weak reference list head, for weak references to this type object. Not inherited. Internal use only.

Alterado na versão 3.12: Internals detail: For the static builtin types this is always `NULL`, even if weakrefs are added. Instead, the weakrefs for each are stored on `PyInterpreterState`. Use the public C-API or the internal `_PyObject_GET_WEAKREFS_LISTPTR()` macro to avoid the distinction.

Inheritance:

This field is not inherited.

destructor *PyTypeObject*.**tp_del**

This field is deprecated. Use *tp_finalize* instead.

unsigned int *PyTypeObject*.**tp_version_tag**

Used to index into the method cache. Internal use only.

Inheritance:

This field is not inherited.

destructor *PyTypeObject*.**tp_finalize**

An optional pointer to an instance finalization function. Its signature is:

```
void tp_finalize(PyObject *self);
```

If *tp_finalize* is set, the interpreter calls it once when finalizing an instance. It is called either from the garbage collector (if the instance is part of an isolated reference cycle) or just before the object is deallocated. Either way, it is guaranteed to be called before attempting to break reference cycles, ensuring that it finds the object in a sane state.

tp_finalize should not mutate the current exception status; therefore, a recommended way to write a non-trivial finalizer is:

```
static void
local_finalize(PyObject *self)
{
    /* Save the current exception, if any. */
    PyObject *exc = PyErr_GetRaisedException();

    /* ... */

    /* Restore the saved exception. */
    PyErr_SetRaisedException(exc);
}
```

Inheritance:

This field is inherited by subtypes.

Adicionado na versão 3.4.

Alterado na versão 3.8: Before version 3.8 it was necessary to set the *Py_TPFLAGS_HAVE_FINALIZE* flags bit in order for this field to be used. This is no longer required.

Ver também

"Finalização segura de objetos" (**PEP 442**)

vectorcallfunc *PyTypeObject*.**tp_vectorcall**

Vectorcall function to use for calls of this type object. In other words, it is used to implement *vectorcall* for `type.__call__`. If `tp_vectorcall` is `NULL`, the default call implementation using `__new__()` and `__init__()` is used.

Inheritance:

This field is never inherited.

Adicionado na versão 3.9: (the field exists since 3.8 but it's only used since 3.9)

unsigned char *PyTypeObject*.**tp_watched**

Internal. Do not use.

Adicionado na versão 3.12.

12.3.6 Static Types

Traditionally, types defined in C code are *static*, that is, a static *PyTypeObject* structure is defined directly in code and initialized using *PyType_Ready()*.

This results in types that are limited relative to types defined in Python:

- Static types are limited to one base, i.e. they cannot use multiple inheritance.
- Static type objects (but not necessarily their instances) are immutable. It is not possible to add or modify the type object's attributes from Python.
- Static type objects are shared across *sub-interpreters*, so they should not include any subinterpreter-specific state.

Also, since *PyTypeObject* is only part of the *Limited API* as an opaque struct, any extension modules using static types must be compiled for a specific Python minor version.

12.3.7 Tipos no heap

An alternative to *static types* is *heap-allocated types*, or *heap types* for short, which correspond closely to classes created by Python's `class` statement. Heap types have the *Py_TPFLAGS_HEAPTYPE* flag set.

This is done by filling a *PyType_Spec* structure and calling *PyType_FromSpec()*, *PyType_FromSpecWithBases()*, *PyType_FromModuleAndSpec()*, or *PyType_FromMetaclass()*.

12.3.8 Number Object Structures

type **PyNumberMethods**

This structure holds pointers to the functions which an object uses to implement the number protocol. Each function is used by the function of similar name documented in the *Protocolo de número* section.

Here is the structure definition:

```
typedef struct {
     binaryfunc nb_add;
     binaryfunc nb_subtract;
     binaryfunc nb_multiply;
     binaryfunc nb_remainder;
     binaryfunc nb_divmod;
     ternaryfunc nb_power;
     unaryfunc nb_negative;
     unaryfunc nb_positive;
     unaryfunc nb_absolute;
     inquiry nb_bool;
     unaryfunc nb_invert;
     binaryfunc nb_lshift;
     binaryfunc nb_rshift;
     binaryfunc nb_and;
     binaryfunc nb_xor;
     binaryfunc nb_or;
     unaryfunc nb_int;
     void *nb_reserved;
     unaryfunc nb_float;

     binaryfunc nb_inplace_add;
     binaryfunc nb_inplace_subtract;
     binaryfunc nb_inplace_multiply;
     binaryfunc nb_inplace_remainder;
     ternaryfunc nb_inplace_power;
     binaryfunc nb_inplace_lshift;
     binaryfunc nb_inplace_rshift;
     binaryfunc nb_inplace_and;
     binaryfunc nb_inplace_xor;
     binaryfunc nb_inplace_or;

     binaryfunc nb_floor_divide;
     binaryfunc nb_true_divide;
     binaryfunc nb_inplace_floor_divide;
     binaryfunc nb_inplace_true_divide;

     unaryfunc nb_index;
```

(continua na próxima página)

(continuação da página anterior)

```
    binaryfunc nb_matrix_multiply;
    binaryfunc nb_inplace_matrix_multiply;
} PyNumberMethods;
```

Nota

Binary and ternary functions must check the type of all their operands, and implement the necessary conversions (at least one of the operands is an instance of the defined type). If the operation is not defined for the given operands, binary and ternary functions must return `Py_NotImplemented`, if another error occurred they must return `NULL` and set an exception.

Nota

The *nb_reserved* field should always be `NULL`. It was previously called `nb_long`, and was renamed in Python 3.0.1.

binaryfunc *PyNumberMethods*.**nb_add**

binaryfunc *PyNumberMethods*.**nb_subtract**

binaryfunc *PyNumberMethods*.**nb_multiply**

binaryfunc *PyNumberMethods*.**nb_remainder**

binaryfunc *PyNumberMethods*.**nb_divmod**

ternaryfunc *PyNumberMethods*.**nb_power**

unaryfunc *PyNumberMethods*.**nb_negative**

unaryfunc *PyNumberMethods*.**nb_positive**

unaryfunc *PyNumberMethods*.**nb_absolute**

inquiry *PyNumberMethods*.**nb_bool**

unaryfunc *PyNumberMethods*.**nb_invert**

binaryfunc *PyNumberMethods*.**nb_lshift**

binaryfunc *PyNumberMethods*.**nb_rshift**

binaryfunc *PyNumberMethods*.**nb_and**

binaryfunc *PyNumberMethods*.**nb_xor**

binaryfunc *PyNumberMethods*.**nb_or**

unaryfunc *PyNumberMethods*.**nb_int**

void **PyNumberMethods*.**nb_reserved**

unaryfunc *PyNumberMethods*.**nb_float**

binaryfunc *PyNumberMethods*.**nb_inplace_add**

binaryfunc *PyNumberMethods*.**nb_inplace_subtract**

binaryfunc *PyNumberMethods*.**nb_inplace_multiply**

binaryfunc *PyNumberMethods*.**nb_inplace_remainder**

ternaryfunc *PyNumberMethods*.**nb_inplace_power**

binaryfunc *PyNumberMethods*.**nb_inplace_lshift**

binaryfunc *PyNumberMethods*.**nb_inplace_rshift**

binaryfunc *PyNumberMethods*.**nb_inplace_and**

binaryfunc *PyNumberMethods*.**nb_inplace_xor**

binaryfunc *PyNumberMethods*.**nb_inplace_or**

binaryfunc *PyNumberMethods*.**nb_floor_divide**

binaryfunc *PyNumberMethods*.**nb_true_divide**

binaryfunc *PyNumberMethods*.**nb_inplace_floor_divide**

binaryfunc *PyNumberMethods*.**nb_inplace_true_divide**

unaryfunc *PyNumberMethods*.**nb_index**

binaryfunc *PyNumberMethods*.**nb_matrix_multiply**

binaryfunc *PyNumberMethods*.**nb_inplace_matrix_multiply**

12.3.9 Mapping Object Structures

type **PyMappingMethods**

This structure holds pointers to the functions which an object uses to implement the mapping protocol. It has three members:

lenfunc *PyMappingMethods*.**mp_length**

This function is used by *PyMapping_Size()* and *PyObject_Size()*, and has the same signature. This slot may be set to `NULL` if the object has no defined length.

binaryfunc *PyMappingMethods*.**mp_subscript**

This function is used by *PyObject_GetItem()* and *PySequence_GetSlice()*, and has the same signature as `PyObject_GetItem()`. This slot must be filled for the *PyMapping_Check()* function to return `1`, it can be `NULL` otherwise.

objobjargproc *PyMappingMethods*.**mp_ass_subscript**

This function is used by *PyObject_SetItem()*, *PyObject_DelItem()*, *PySequence_SetSlice()* and *PySequence_DelSlice()*. It has the same signature as `PyObject_SetItem()`, but *v* can also be set to `NULL` to delete an item. If this slot is `NULL`, the object does not support item assignment and deletion.

12.3.10 Sequence Object Structures

type **PySequenceMethods**

This structure holds pointers to the functions which an object uses to implement the sequence protocol.

lenfunc *PySequenceMethods*.**sq_length**

This function is used by *PySequence_Size()* and *PyObject_Size()*, and has the same signature. It is also used for handling negative indices via the *sq_item* and the *sq_ass_item* slots.

binaryfunc *PySequenceMethods*.**sq_concat**

This function is used by *PySequence_Concat()* and has the same signature. It is also used by the + operator, after trying the numeric addition via the *nb_add* slot.

ssizeargfunc *PySequenceMethods*.**sq_repeat**

This function is used by *PySequence_Repeat()* and has the same signature. It is also used by the `*` operator, after trying numeric multiplication via the *nb_multiply* slot.

ssizeargfunc *PySequenceMethods*.**sq_item**

This function is used by *PySequence_GetItem()* and has the same signature. It is also used by *PyObject_GetItem()*, after trying the subscription via the *mp_subscript* slot. This slot must be filled for the *PySequence_Check()* function to return `1`, it can be `NULL` otherwise.

Negative indexes are handled as follows: if the *sq_length* slot is filled, it is called and the sequence length is used to compute a positive index which is passed to *sq_item*. If `sq_length` is `NULL`, the index is passed as is to the function.

ssizeobjargproc *PySequenceMethods*.**sq_ass_item**

This function is used by *PySequence_SetItem()* and has the same signature. It is also used by *PyObject_SetItem()* and *PyObject_DelItem()*, after trying the item assignment and deletion via the *mp_ass_subscript* slot. This slot may be left to `NULL` if the object does not support item assignment and deletion.

objobjproc *PySequenceMethods*.**sq_contains**

This function may be used by *PySequence_Contains()* and has the same signature. This slot may be left to `NULL`, in this case `PySequence_Contains()` simply traverses the sequence until it finds a match.

binaryfunc *PySequenceMethods*.**sq_inplace_concat**

This function is used by *PySequence_InPlaceConcat()* and has the same signature. It should modify its first operand, and return it. This slot may be left to `NULL`, in this case `PySequence_InPlaceConcat()` will fall back to *PySequence_Concat()*. It is also used by the augmented assignment `+=`, after trying numeric in-place addition via the *nb_inplace_add* slot.

ssizeargfunc *PySequenceMethods*.**sq_inplace_repeat**

This function is used by *PySequence_InPlaceRepeat()* and has the same signature. It should modify its first operand, and return it. This slot may be left to `NULL`, in this case `PySequence_InPlaceRepeat()` will fall back to *PySequence_Repeat()*. It is also used by the augmented assignment `*=`, after trying numeric in-place multiplication via the *nb_inplace_multiply* slot.

12.3.11 Buffer Object Structures

type **PyBufferProcs**

This structure holds pointers to the functions required by the *Buffer protocol*. The protocol defines how an exporter object can expose its internal data to consumer objects.

getbufferproc *PyBufferProcs*.**bf_getbuffer**

The signature of this function is:

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

Handle a request to *exporter* to fill in *view* as specified by *flags*. Except for point (3), an implementation of this function MUST take these steps:

(1) Check if the request can be met. If not, raise `BufferError`, set `view->obj` to `NULL` and return `-1`.

(2) Fill in the requested fields.

(3) Increment an internal counter for the number of exports.

(4) Set `view->obj` to *exporter* and increment `view->obj`.

(5) Retorna `0`.

If *exporter* is part of a chain or tree of buffer providers, two main schemes can be used:

- Re-export: Each member of the tree acts as the exporting object and sets `view->obj` to a new reference to itself.

- Redirect: The buffer request is redirected to the root object of the tree. Here, `view->obj` will be a new reference to the root object.

The individual fields of *view* are described in section *Buffer structure*, the rules how an exporter must react to specific requests are in section *Buffer request types*.

All memory pointed to in the *Py_buffer* structure belongs to the exporter and must remain valid until there are no consumers left. *format*, *shape*, *strides*, *suboffsets* and *internal* are read-only for the consumer.

PyBuffer_FillInfo() provides an easy way of exposing a simple bytes buffer while dealing correctly with all request types.

PyObject_GetBuffer() is the interface for the consumer that wraps this function.

releasebufferproc *PyBufferProcs*.**bf_releasebuffer**

The signature of this function is:

```
void (PyObject *exporter, Py_buffer *view);
```

Handle a request to release the resources of the buffer. If no resources need to be released, *PyBufferProcs.bf_releasebuffer* may be `NULL`. Otherwise, a standard implementation of this function will take these optional steps:

(1) Decrement an internal counter for the number of exports.

(2) If the counter is `0`, free all memory associated with *view*.

The exporter MUST use the *internal* field to keep track of buffer-specific resources. This field is guaranteed to remain constant, while a consumer MAY pass a copy of the original buffer as the *view* argument.

This function MUST NOT decrement `view->obj`, since that is done automatically in *PyBuffer_Release()* (this scheme is useful for breaking reference cycles).

PyBuffer_Release() is the interface for the consumer that wraps this function.

12.3.12 Async Object Structures

Adicionado na versão 3.5.

type **PyAsyncMethods**

This structure holds pointers to the functions required to implement *awaitable* and *asynchronous iterator* objects.

Here is the structure definition:

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
    sendfunc am_send;
} PyAsyncMethods;
```

unaryfunc *PyAsyncMethods*.**am_await**

The signature of this function is:

```
PyObject *am_await(PyObject *self);
```

The returned object must be an *iterator*, i.e. *PyIter_Check()* must return `1` for it.

This slot may be set to `NULL` if an object is not an *awaitable*.

unaryfunc *PyAsyncMethods*.**am_aiter**

The signature of this function is:

```
PyObject *am_aiter(PyObject *self);
```

Must return an *asynchronous iterator* object. See `__anext__()` for details.

This slot may be set to `NULL` if an object does not implement asynchronous iteration protocol.

unaryfunc `PyAsyncMethods.`**`am_anext`**

The signature of this function is:

```
PyObject *am_anext(PyObject *self);
```

Must return an *awaitable* object. See `__anext__()` for details. This slot may be set to `NULL`.

sendfunc `PyAsyncMethods.`**`am_send`**

The signature of this function is:

```
PySendResult am_send(PyObject *self, PyObject *arg, PyObject **result);
```

See *PyIter_Send()* for details. This slot may be set to `NULL`.

Adicionado na versão 3.10.

12.3.13 Slot Type typedefs

typedef *PyObject* *(***allocfunc**)(*PyTypeObject* *cls, *Py_ssize_t* nitems)

Parte da ABI Estável. The purpose of this function is to separate memory allocation from memory initialization. It should return a pointer to a block of memory of adequate length for the instance, suitably aligned, and initialized to zeros, but with *ob_refcnt* set to `1` and *ob_type* set to the type argument. If the type's *tp_itemsize* is non-zero, the object's *ob_size* field should be initialized to *nitems* and the length of the allocated memory block should be `tp_basicsize + nitems*tp_itemsize`, rounded up to a multiple of `sizeof(void*)`; otherwise, *nitems* is not used and the length of the block should be *tp_basicsize*.

This function should not do any other instance initialization, not even to allocate additional memory; that should be done by *tp_new*.

typedef void (***destructor**)(*PyObject**)

Parte da ABI Estável.

typedef void (***freefunc**)(void*)

See *tp_free*.

typedef *PyObject* *(***newfunc**)(*PyTypeObject**, *PyObject**, *PyObject**)

Parte da ABI Estável. See *tp_new*.

typedef int (***initproc**)(*PyObject**, *PyObject**, *PyObject**)

Parte da ABI Estável. See *tp_init*.

typedef *PyObject* *(***reprfunc**)(*PyObject**)

Parte da ABI Estável. See *tp_repr*.

typedef *PyObject* *(***getattrfunc**)(*PyObject* *self, char *attr)

Parte da ABI Estável. Return the value of the named attribute for the object.

typedef int (***setattrfunc**)(*PyObject* *self, char *attr, *PyObject* *value)

Parte da ABI Estável. Set the value of the named attribute for the object. The value argument is set to `NULL` to delete the attribute.

typedef *PyObject* *(***getattrofunc**)(*PyObject* *self, *PyObject* *attr)

Parte da ABI Estável. Return the value of the named attribute for the object.

See *tp_getattro*.

typedef int (***setattrofunc**)(*PyObject* *self, *PyObject* *attr, *PyObject* *value)

Parte da ABI Estável. Set the value of the named attribute for the object. The value argument is set to `NULL` to delete the attribute.

See *tp_setattro*.

typedef *PyObject* *(***descrgetfunc**)(*PyObject**, *PyObject**, *PyObject**)

Parte da ABI Estável. See *tp_descr_get*.

typedef int (***descrsetfunc**)(*PyObject**, *PyObject**, *PyObject**)

Parte da ABI Estável. See *tp_descr_set*.

typedef *Py_hash_t* (***hashfunc**)(*PyObject**)

Parte da ABI Estável. See *tp_hash*.

typedef *PyObject* *(***richcmpfunc**)(*PyObject**, *PyObject**, int)

Parte da ABI Estável. See *tp_richcompare*.

typedef *PyObject* *(***getiterfunc**)(*PyObject**)

Parte da ABI Estável. See *tp_iter*.

typedef *PyObject* *(***iternextfunc**)(*PyObject**)

Parte da ABI Estável. See *tp_iternext*.

typedef *Py_ssize_t* (***lenfunc**)(*PyObject**)

Parte da ABI Estável.

typedef int (***getbufferproc**)(*PyObject**, *Py_buffer**, int)

Parte da ABI Estável *desde a versão 3.12.*

typedef void (***releasebufferproc**)(*PyObject**, *Py_buffer**)

Parte da ABI Estável *desde a versão 3.12.*

typedef *PyObject* *(***unaryfunc**)(*PyObject**)

Parte da ABI Estável.

typedef *PyObject* *(***binaryfunc**)(*PyObject**, *PyObject**)

Parte da ABI Estável.

typedef *PySendResult* (***sendfunc**)(*PyObject**, *PyObject**, *PyObject***)

See *am_send*.

typedef *PyObject* *(***ternaryfunc**)(*PyObject**, *PyObject**, *PyObject**)

Parte da ABI Estável.

typedef *PyObject* *(***ssizeargfunc**)(*PyObject**, *Py_ssize_t*)

Parte da ABI Estável.

typedef int (***ssizeobjargproc**)(*PyObject**, *Py_ssize_t*, *PyObject**)

Parte da ABI Estável.

typedef int (***objobjproc**)(*PyObject**, *PyObject**)

Parte da ABI Estável.

typedef int (***objobjargproc**)(*PyObject**, *PyObject**, *PyObject**)

Parte da ABI Estável.

12.3.14 Exemplos

The following are simple examples of Python type definitions. They include common usage you may encounter. Some demonstrate tricky corner cases. For more examples, practical info, and a tutorial, see defining-new-types and new-types-topics.

A basic *static type*:

```
typedef struct {
    PyObject_HEAD
    const char *data;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_new = myobj_new,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
};
```

You may also find older code (especially in the CPython code base) with a more verbose initializer:

```
static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "mymod.MyObject",               /* tp_name */
    sizeof(MyObject),               /* tp_basicsize */
    0,                              /* tp_itemsize */
    (destructor)myobj_dealloc,      /* tp_dealloc */
    0,                              /* tp_vectorcall_offset */
    0,                              /* tp_getattr */
    0,                              /* tp_setattr */
    0,                              /* tp_as_async */
    (reprfunc)myobj_repr,           /* tp_repr */
    0,                              /* tp_as_number */
    0,                              /* tp_as_sequence */
    0,                              /* tp_as_mapping */
    0,                              /* tp_hash */
    0,                              /* tp_call */
    0,                              /* tp_str */
    0,                              /* tp_getattro */
    0,                              /* tp_setattro */
    0,                              /* tp_as_buffer */
    0,                              /* tp_flags */
    PyDoc_STR("My objects"),        /* tp_doc */
    0,                              /* tp_traverse */
    0,                              /* tp_clear */
    0,                              /* tp_richcompare */
    0,                              /* tp_weaklistoffset */
    0,                              /* tp_iter */
    0,                              /* tp_iternext */
    0,                              /* tp_methods */
    0,                              /* tp_members */
    0,                              /* tp_getset */
    0,                              /* tp_base */
    0,                              /* tp_dict */
    0,                              /* tp_descr_get */
    0,                              /* tp_descr_set */
    0,                              /* tp_dictoffset */
    0,                              /* tp_init */
    0,                              /* tp_alloc */
    myobj_new,                      /* tp_new */
};
```

A type that supports weakrefs, instance dicts, and hashing:

```
typedef struct {
    PyObject_HEAD
    const char *data;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
         Py_TPFLAGS_HAVE_GC | Py_TPFLAGS_MANAGED_DICT |
         Py_TPFLAGS_MANAGED_WEAKREF,
    .tp_new = myobj_new,
    .tp_traverse = (traverseproc)myobj_traverse,
    .tp_clear = (inquiry)myobj_clear,
    .tp_alloc = PyType_GenericNew,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
    .tp_hash = (hashfunc)myobj_hash,
    .tp_richcompare = PyBaseObject_Type.tp_richcompare,
};
```

A str subclass that cannot be subclassed and cannot be called to create instances (e.g. uses a separate factory func) using *Py_TPFLAGS_DISALLOW_INSTANTIATION* flag:

```
typedef struct {
    PyUnicodeObject raw;
    char *extra;
} MyStr;

static PyTypeObject MyStr_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyStr",
    .tp_basicsize = sizeof(MyStr),
    .tp_base = NULL,  // set to &PyUnicode_Type in module init
    .tp_doc = PyDoc_STR("my custom str"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_DISALLOW_INSTANTIATION,
    .tp_repr = (reprfunc)myobj_repr,
};
```

The simplest *static type* with fixed-length instances:

```
typedef struct {
    PyObject_HEAD
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
};
```

The simplest *static type* with variable-length instances:

```
typedef struct {
    PyObject_VAR_HEAD
```

(continua na próxima página)

(continuação da página anterior)

```
    const char *data[1];
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject) - sizeof(char *),
    .tp_itemsize = sizeof(char *),
};
```

12.4 Suporte a Coleta Cíclica de Lixo

O suporte do Python para detectar e coletar o lixo, que envolve referencias circulares, requer suporte dos tipos de objetos que são "contêiners" para outros objetos que também podem ser contêiners. Tipos que não armazenam referências a outros tipos de objetos, ou que apenas armazenam referências a tipos atômicos (como números ou strings), não precisam fornecer nenhum suporte explicito para coleta de lixo.

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
: Objetos com esse tipo de sinalizador definido devem estar em conformidade com regras documentadas aqui. Por conveniência esses objetos serão referenciados como objetos de contêiner.

Construtores para tipos de contêiner devem obedecer a duas regras:

1. The memory for the object must be allocated using *PyObject_GC_New* or *PyObject_GC_NewVar*.
2. Uma vez que todos os campos que podem conter referências a outros containers foram inicializados, deve-se chamar *PyObject_GC_Track()*.

Da mesma forma, o desalocador para o objeto deve estar em conformidade com regras semelhantes:

1. Antes que os campos que fazer referência a outros containers sejam invalidados, *PyObject_GC_UnTrack()* deve ser chamado.
2. A memória destinada ao objeto deve ser desalocada usando *PyObject_GC_Del()*.

> **Aviso**
>
> If a type adds the Py_TPFLAGS_HAVE_GC, then it *must* implement at least a *tp_traverse* handler or explicitly use one from its subclass or subclasses.
>
> When calling *PyType_Ready()* or some of the APIs that indirectly call it like *PyType_FromSpecWithBases()* or *PyType_FromSpec()* the interpreter will automatically populate the *tp_flags*, *tp_traverse* and *tp_clear* fields if the type inherits from a class that implements the garbage collector protocol and the child class does *not* include the *Py_TPFLAGS_HAVE_GC* flag.

PyObject_GC_New(TYPE, typeobj)
: Analogous to *PyObject_New* but for container objects with the *Py_TPFLAGS_HAVE_GC* flag set.

PyObject_GC_NewVar(TYPE, typeobj, size)
: Analogous to *PyObject_NewVar* but for container objects with the *Py_TPFLAGS_HAVE_GC* flag set.

PyObject ***PyUnstable_Object_GC_NewWithExtraData**(*PyTypeObject* *type, size_t extra_size)

Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Analogous to `PyObject_GC_New` but allocates *extra_size* bytes at the end of the object (at offset `tp_basicsize`). The allocated memory is initialized to zeros, except for the `Python object header`.

The extra data will be deallocated with the object, but otherwise it is not managed by Python.

Aviso

The function is marked as unstable because the final mechanism for reserving extra data after an instance is not yet decided. For allocating a variable number of fields, prefer using `PyVarObject` and `tp_itemsize` instead.

Adicionado na versão 3.12.

PyObject_GC_Resize(TYPE, op, newsize)

Resize an object allocated by `PyObject_NewVar`. Returns the resized object of type `TYPE*` (refers to any C type) or `NULL` on failure.

op must be of type `PyVarObject*` and must not be tracked by the collector yet. *newsize* must be of type `Py_ssize_t`.

void **PyObject_GC_Track**(*PyObject* *op)

Parte da ABI Estável. Adds the object *op* to the set of container objects tracked by the collector. The collector can run at unexpected times so objects must be valid while being tracked. This should be called once all the fields followed by the `tp_traverse` handler become valid, usually near the end of the constructor.

int **PyObject_IS_GC**(*PyObject* *obj)

Returns non-zero if the object implements the garbage collector protocol, otherwise returns 0.

The object cannot be tracked by the garbage collector if this function returns 0.

int **PyObject_GC_IsTracked**(*PyObject* *op)

Parte da ABI Estável *desde a versão 3.9*. Returns 1 if the object type of *op* implements the GC protocol and *op* is being currently tracked by the garbage collector and 0 otherwise.

This is analogous to the Python function `gc.is_tracked()`.

Adicionado na versão 3.9.

int **PyObject_GC_IsFinalized**(*PyObject* *op)

Parte da ABI Estável *desde a versão 3.9*. Returns 1 if the object type of *op* implements the GC protocol and *op* has been already finalized by the garbage collector and 0 otherwise.

This is analogous to the Python function `gc.is_finalized()`.

Adicionado na versão 3.9.

void **PyObject_GC_Del**(void *op)

Parte da ABI Estável. Releases memory allocated to an object using `PyObject_GC_New` or `PyObject_GC_NewVar`.

void **PyObject_GC_UnTrack**(void *op)

Parte da ABI Estável. 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.

Alterado na versão 3.8: The `_PyObject_GC_TRACK()` and `_PyObject_GC_UNTRACK()` macros have been removed from the public C API.

The *tp_traverse* handler accepts a function parameter of this type:

typedef int (***visitproc**)(*PyObject* *object, void *arg)

Parte da ABI Estável. Type of the visitor function passed to the *tp_traverse* handler. The function should be called with an object to traverse as *object* and the third parameter to the *tp_traverse* handler as *arg*. The Python core uses several visitor functions to implement cyclic garbage detection; it's not expected that users will need to write their own visitor functions.

The *tp_traverse* handler must have the following type:

typedef int (***traverseproc**)(*PyObject* *self, *visitproc* visit, void *arg)

Parte da ABI Estável. Traversal function for a container object. Implementations must call the *visit* function for each object directly contained by *self*, with the parameters to *visit* being the contained object and the *arg* value passed to the handler. The *visit* function must not be called with a `NULL` object argument. If *visit* returns a non-zero value that value should be returned immediately.

To simplify writing *tp_traverse* handlers, a *Py_VISIT()* macro is provided. In order to use this macro, the *tp_traverse* implementation must name its arguments exactly *visit* and *arg*:

void **Py_VISIT**(*PyObject* *o)

If *o* is not `NULL`, call the *visit* callback, with arguments *o* and *arg*. If *visit* returns a non-zero value, then return it. Using this macro, *tp_traverse* handlers look like:

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

The *tp_clear* handler must be of the *inquiry* type, or `NULL` if the object is immutable.

typedef int (***inquiry**)(*PyObject* *self)

Parte da ABI Estável. Drop references that may have created reference cycles. Immutable objects do not have to define this method since they can never directly create reference cycles. Note that the object must still be valid after calling this method (don't just call *Py_DECREF()* on a reference). The collector will call this method if it detects that this object is involved in a reference cycle.

12.4.1 Controlando o estado do coletor de lixo

The C-API provides the following functions for controlling garbage collection runs.

Py_ssize_t **PyGC_Collect**(void)

Parte da ABI Estável. Perform a full garbage collection, if the garbage collector is enabled. (Note that `gc.collect()` runs it unconditionally.)

Returns the number of collected + unreachable objects which cannot be collected. If the garbage collector is disabled or already collecting, returns `0` immediately. Errors during garbage collection are passed to `sys.unraisablehook`. This function does not raise exceptions.

int **PyGC_Enable**(void)

Parte da ABI Estável *desde a versão 3.10.* Enable the garbage collector: similar to `gc.enable()`. Returns the previous state, 0 for disabled and 1 for enabled.

Adicionado na versão 3.10.

int **PyGC_Disable**(void)

Parte da ABI Estável *desde a versão 3.10.* Disable the garbage collector: similar to `gc.disable()`. Returns the previous state, 0 for disabled and 1 for enabled.

Adicionado na versão 3.10.

int **PyGC_IsEnabled**(void)

Parte da ABI Estável *desde a versão 3.10.* Query the state of the garbage collector: similar to `gc.isenabled()`. Returns the current state, 0 for disabled and 1 for enabled.

Adicionado na versão 3.10.

12.4.2 Querying Garbage Collector State

The C-API provides the following interface for querying information about the garbage collector.

void **PyUnstable_GC_VisitObjects**(*gcvisitobjects_t* callback, void *arg)

Esta é uma *API Instável*. Isso pode se alterado sem aviso em lançamentos menores.

Run supplied *callback* on all live GC-capable objects. *arg* is passed through to all invocations of *callback*.

Aviso

If new objects are (de)allocated by the callback it is undefined if they will be visited.

Garbage collection is disabled during operation. Explicitly running a collection in the callback may lead to undefined behaviour e.g. visiting the same objects multiple times or not at all.

Adicionado na versão 3.12.

typedef int (***gcvisitobjects_t**)(*PyObject* *object, void *arg)

Type of the visitor function to be passed to `PyUnstable_GC_VisitObjects()`. *arg* is the same as the *arg* passed to `PyUnstable_GC_VisitObjects`. Return `0` to continue iteration, return `1` to stop iteration. Other return values are reserved for now so behavior on returning anything else is undefined.

Adicionado na versão 3.12.

CAPÍTULO 13

API e Versionamento de ABI

O CPython expõe seu número de versão nas seguintes macros. Note que estes correspondem com o código da versão que está **construída**, não necessariamente a versão usada no **run time**.

Veja *Estabilidade da API C* para uma discussão da estabilidade da API e ABI através das versões.

PY_MAJOR_VERSION

O `3` em `3.4.1a2`.

PY_MINOR_VERSION

O `4` em `3.4.1a2`.

PY_MICRO_VERSION

O `1` em `3.4.1a2`.

PY_RELEASE_LEVEL

O `a` em `3.4.1a2`. Isto pode ser `0xA` para alfa, `0xB` para beta, `0xC` para o candidato a lançamento ou `0xF` para final.

PY_RELEASE_SERIAL

O `2` em `3.4.1a2`. Zero para os lançamentos finais.

PY_VERSION_HEX

O número da versão do Python codificado em um único inteiro.

As informações da versão subjacente podem ser achadas tratando-as como um número de 32 bits da seguinte maneira:

Bytes	Bits (big endian order)	Significado	Valor para `3.4.1a2`
1	1-8	`PY_MAJOR_VERSION`	`0x03`
2	9-16	`PY_MINOR_VERSION`	`0x04`
3	17-24	`PY_MICRO_VERSION`	`0x01`
4	25-28	`PY_RELEASE_LEVEL`	`0xA`
	29-32	`PY_RELEASE_SERIAL`	`0x2`

Assim `3.4.1a2` é a versão hex `0x030401a2` e `3.10.0` é a versãos hex `0x030a00f0`.

Use isso para comparações numéricas como, por exemplo, `#if PY_VERSION_HEX >= ...`.

Esta versão também está disponível através do símbolo *`Py_Version`*.

const unsigned long **Py_Version**

Parte da ABI Estável *desde a versão 3.11.* O número da versão do runtime do Python codificado em um único inteiro constante, com o mesmo formato da macro *PY_VERSION_HEX*. Ele contém a versão do Python usada em tempo de execução.

Adicionado na versão 3.11.

Todas as macros dadas estão definidas em Include/patchlevel.h.

CAPÍTULO 14

API C de monitoramento

Adicionada na versão 3.13.

Uma extensão pode precisar interagir com o sistema de monitoramento de eventos. É possível registrar funções de retorno e se inscrever para receber eventos através da API Python exposta em `sys.monitoring`.

CAPÍTULO 15

Gerando eventos de execução

As funções abaixo permitem que extensões dispararem eventos de monitoramento emulando a execução de código Python. Cada uma dessas funções recebe uma estrutura `PyMonitoringState` que contém informação concisa sobre o estado de ativação de eventos, bem como os argumentos dos eventos, que incluem um `PyObject*` representando o objeto de código, a posição da instrução no bytecode, e, em alguns casos, argumentos adicionais específicos para o evento (veja `sys.monitoring` para detalhes sobre as assinaturas das funções de retorno de diferentes eventos). O argumento `codelike` deve ser uma instância da classe `types.CodeType` ou de um tipo que a emule.

A VM desabilita o rastreamento quando dispara um evento, de forma que não há necessidade do código do usuário fazer isso.

Funções de monitoramento não devem ser chamadas com uma exceção definida, exceto as marcadas abaixo como funções que trabalham com a exceção atual.

type **PyMonitoringState**

Representação do estado de um tipo de evento. Alocada pelo usuário, enquanto o seu conteúdo é mantido pelas funções da API de monitoramento descritas abaixo.

Todas as funções abaixo retornam 0 para indicar sucesso e -1 (com uma exceção definida) para indicar erro.

Veja `sys.monitoring` para descrições dos eventos.

int **PyMonitoring_FirePyStartEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Dispara um evento `PY_START`.

int **PyMonitoring_FirePyResumeEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Dispara um evento `PY_RESUME`.

int **PyMonitoring_FirePyReturnEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Dispara um evento `PY_RETURN`.

int **PyMonitoring_FirePyYieldEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Dispara um evento `PY_YIELD`.

int **PyMonitoring_FireCallEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *callable, *PyObject* *arg0)

Dispara um evento `CALL`.

int **PyMonitoring_FireLineEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, int lineno)

Dispara um evento `LINE`.

int **PyMonitoring_FireJumpEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

Dispara um evento `JUMP`.

int **PyMonitoring_FireBranchEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

Dispara um evento `BRANCH`.

int **PyMonitoring_FireCReturnEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Dispara um evento `C_RETURN`.

int **PyMonitoring_FirePyThrowEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Dispara um evento `PY_THROW` com a exceção atual (conforme retornada por *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireRaiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Dispara um evento `RAISE` com a exceção atual (conforme retornada por *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireCRaiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Dispara um evento `C_RAISE` com a exceção atual (conforme retornada por *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireReraiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Dispara um evento `RERAISE` com a exceção atual (conforme retornada por *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireExceptionHandledEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Dispara um evento `EXCEPTION_HANDLED` com a exceção atual (conforme retornada por *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FirePyUnwindEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Dispara um evento `PY_UNWIND` com a exceção atual (conforme retornada por *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireStopIterationEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *value)

Dispara um evento `STOP_ITERATION`. Se `value` for uma instância de `StopIteration`, ele é usado. Caso contrário, uma nova instância de `StopIteration` é criada com `value` como o argumento.

15.1 Gerenciando o Estado de um Monitoramento

Estados de monitoramento podem ser gerenciados com a ajuda de escopos de monitoramento. Um escopo corresponderia tipicamente a um função python.

int **PyMonitoring_EnterScope**(*PyMonitoringState* *state_array, uint64_t *version, const uint8_t *event_types, *Py_ssize_t* length)

Insere um escopo monitorado. `event_types` é um vetor de IDs de eventos para eventos que podem ser disparados do escopo. Por exemplo, a ID de um evento `PY_START` é o valor `PY_MONITORING_EVENT_PY_START`, que é numericamente igual ao logaritmo de base 2 de `sys.monitoring.events.PY_START`. `state_array` é um vetor com uma entrada de estado de monitoramento para cada evento em `event_types`, é alocado pelo usuário, mas preenchido por `PyMonitoring_EnterScope()` com informações sobre o estado de ativação do evento. O tamanho de `event_types` (e, portanto, também de `state_array`) é fornecido em `length`.

O argumento `version` é um ponteiro para um valor que deve ser alocado pelo usuário junto com `state_array` e inicializado como 0, e então definido somente pelo próprio `PyMonitoring_EnterScope()`. Ele permite que esta função determine se os estados de eventos mudaram desde a chamada anterior, e retorne rapidamente se não mudaram.

Os escopos mencionados aqui são escopos lexicais: uma função, classe ou método. `PyMonitoring_EnterScope()` deve ser chamada sempre que o escopo lexical for inserido. Os escopos podem ser inseridos novamente, reutilizando o mesmo *state_array* e *version*, em situações como ao emular uma função Python recursiva. Quando a execução de um código semelhante é pausada, como ao emular um gerador, o escopo precisa ser encerrado e inserido novamente.

As macros para *event_types* são:

Macro	Evento
PY_MONITORING_EVENT_BRANCH	BRANCH
PY_MONITORING_EVENT_CALL	CALL
PY_MONITORING_EVENT_C_RAISE	C_RAISE
PY_MONITORING_EVENT_C_RETURN	C_RETURN
PY_MONITORING_EVENT_EXCEPTION_HANDLED	EXCEPTION_HANDLED
PY_MONITORING_EVENT_INSTRUCTION	INSTRUCTION
PY_MONITORING_EVENT_JUMP	JUMP
PY_MONITORING_EVENT_LINE	LINE
PY_MONITORING_EVENT_PY_RESUME	PY_RESUME
PY_MONITORING_EVENT_PY_RETURN	PY_RETURN
PY_MONITORING_EVENT_PY_START	PY_START
PY_MONITORING_EVENT_PY_THROW	PY_THROW
PY_MONITORING_EVENT_PY_UNWIND	PY_UNWIND
PY_MONITORING_EVENT_PY_YIELD	PY_YIELD
PY_MONITORING_EVENT_RAISE	RAISE
PY_MONITORING_EVENT_RERAISE	RERAISE
PY_MONITORING_EVENT_STOP_ITERATION	STOP_ITERATION

int **PyMonitoring_ExitScope**(void)

Sai do último escopo no qual se entrou com `PyMonitoring_EnterScope()`.

int **PY_MONITORING_IS_INSTRUMENTED_EVENT**(uint8_t ev)

Retorna verdadeiro se o evento correspondente ao ID do evento *ev* for um evento local.

Adicionado na versão 3.13.

Descontinuado desde a versão 3.13.2 (unreleased): Esta função está *suavemente descontinuada*.

APÊNDICE A

Glossário

>>>
: O prompt padrão do console *interativo* do Python. Normalmente visto em exemplos de código que podem ser executados interativamente no interpretador.

...
: Pode se referir a:

  - O prompt padrão do console *interativo* do Python ao inserir o código para um bloco de código recuado, quando dentro de um par de delimitadores correspondentes esquerdo e direito (parênteses, colchetes, chaves ou aspas triplas) ou após especificar um decorador.
  - A constante embutida `Ellipsis`.

classe base abstrata
: Classes bases abstratas complementam *tipagem pato*, fornecendo uma maneira de definir interfaces quando outras técnicas, como `hasattr()`, seriam desajeitadas ou sutilmente erradas (por exemplo, com métodos mágicos). ABCs introduzem subclasses virtuais, classes que não herdam de uma classe mas ainda são reconhecidas por `isinstance()` e `issubclass()`; veja a documentação do módulo `abc`. Python vem com muitas ABCs embutidas para estruturas de dados (no módulo `collections.abc`), números (no módulo `numbers`), fluxos (no módulo `io`), localizadores e carregadores de importação (no módulo `importlib.abc`). Você pode criar suas próprias ABCs com o módulo `abc`.

anotação
: Um rótulo associado a uma variável, um atributo de classe ou um parâmetro de função ou valor de retorno, usado por convenção como *dica de tipo*.

  Anotações de variáveis locais não podem ser acessadas em tempo de execução, mas anotações de variáveis globais, atributos de classe e funções são armazenadas no atributo especial `__annotations__` de módulos, classes e funções, respectivamente.

  Veja *anotação de variável*, *anotação de função*, **PEP 484** e **PEP 526**, que descrevem esta funcionalidade. Veja também annotations-howto para as melhores práticas sobre como trabalhar com anotações.

argumento
: Um valor passado para uma *função* (ou *método*) ao chamar a função. Existem dois tipos de argumento:

  - *argumento nomeado*: um argumento precedido por um identificador (por exemplo, `name=`) 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.

Veja também o termo *parâmetro* no glossário, a pergunta no FAQ sobre a diferença entre argumentos e parâmetros e **PEP 362**.

gerenciador de contexto assíncrono
Um objeto que controla o ambiente visto numa instrução `async with` por meio da definição dos métodos `__aenter__()` e `__aexit__()`. Introduzido pela **PEP 492**.

gerador assíncrono
Uma função que retorna um *iterador gerador assíncrono*. É parecida com uma função de corrotina definida com `async def` exceto pelo fato de conter instruções `yield` para produzir uma série de valores que podem ser usados em um laço `async for`.

Normalmente se refere a uma função geradora assíncrona, mas pode se referir a um *iterador gerador assíncrono* em alguns contextos. Em casos em que o significado não esteja claro, usar o termo completo evita a ambiguidade.

Uma função geradora assíncrona pode conter expressões `await` e também as instruções `async for` e `async with`.

iterador gerador assíncrono
Um objeto criado por uma função *geradora assíncrona*.

Este é um *iterador assíncrono* que, quando chamado usando o método `__anext__()`, retorna um objeto aguardável que executará o corpo da função geradora assíncrona até a próxima expressão `yield`.

Cada `yield` suspende temporariamente o processamento, lembrando o estado de execução (incluindo variáveis locais e instruções `try` pendentes). Quando o *iterador gerador assíncrono* é efetivamente retomado com outro aguardável retornado por `__anext__()`, ele inicia de onde parou. Veja **PEP 492** e **PEP 525**.

iterável assíncrono
Um objeto que pode ser usado em uma instrução `async for`. Deve retornar um *iterador assíncrono* do seu método `__aiter__()`. Introduzido por **PEP 492**.

iterador assíncrono
Um objeto que implementa os métodos `__aiter__()` e `__anext__()`. `__anext__()` deve retornar um objeto *aguardável*. `async for` resolve os aguardáveis retornados por um método `__anext__()` do iterador assíncrono até que ele levante uma exceção `StopAsyncIteration`. Introduzido pela **PEP 492**.

atributo
Um valor associado a um objeto que é geralmente referenciado pelo nome separado por um ponto. Por exemplo, se um objeto *o* tem um atributo *a* esse seria referenciado como *o.a*.

É possível dar a um objeto um atributo cujo nome não seja um identificador conforme definido por identifiers, por exemplo usando `setattr()`, se o objeto permitir. Tal atributo não será acessível usando uma expressão pontilhada e, em vez disso, precisaria ser recuperado com `getattr()`.

aguardável
Um objeto que pode ser usado em uma expressão `await`. Pode ser uma *corrotina* ou um objeto com um método `__await__()`. Veja também a **PEP 492**.

BDFL
Abreviação da expressão da língua inglesa "Benevolent Dictator for Life" (em português, "Ditador Benevolente Vitalício"), referindo-se a Guido van Rossum, criador do Python.

arquivo binário
Um *objeto arquivo* capaz de ler e gravar em *objetos bytes ou similar*. Exemplos de arquivos binários são arquivos abertos no modo binário (`'rb'`, `'wb'` ou `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, e instâncias de `io.BytesIO` e `gzip.GzipFile`.

Veja também *arquivo texto* para um objeto arquivo capaz de ler e gravar em objetos `str`.

referência emprestada
Na API C do Python, uma referência emprestada é uma referência a um objeto que não é dona da referência. Ela se torna um ponteiro solto se o objeto for destruído. Por exemplo, uma coleta de lixo pode remover a última *referência forte* para o objeto e assim destruí-lo.

Chamar `Py_INCREF()` na *referência emprestada* é recomendado para convertê-lo, internamente, em uma *referência forte*, exceto quando o objeto não pode ser destruído antes do último uso da referência emprestada. A função `Py_NewRef()` pode ser usada para criar uma nova *referência forte*.

objeto byte ou similar
Um objeto com suporte ao o *Protocolo de Buffer* e que pode exportar um buffer C *contíguo*. Isso inclui todos os objetos `bytes`, `bytearray` e `array.array`, além de muitos objetos `memoryview` comuns. Objetos byte ou similar podem ser usados para várias operações que funcionam com dados binários; isso inclui compactação, salvamento em um arquivo binário e envio por um soquete.

Algumas operações precisam que os dados binários sejam mutáveis. A documentação geralmente se refere a eles como "objetos byte ou similar para leitura e escrita". Exemplos de objetos de buffer mutável incluem `bytearray` e um `memoryview` de um `bytearray`. Outras operações exigem que os dados binários sejam armazenados em objetos imutáveis ("objetos byte ou similar para somente leitura"); exemplos disso incluem `bytes` e a `memoryview` de um objeto `bytes`.

bytecode
O código-fonte Python é compilado para bytecode, a representação interna de um programa em Python no interpretador CPython. O bytecode também é mantido em cache em arquivos `.pyc` e `.pyo`, de forma que executar um mesmo arquivo é mais rápido na segunda vez (a recompilação dos fontes para bytecode não é necessária). Esta "linguagem intermediária" é adequada para execução em uma *máquina virtual*, que executa o código de máquina correspondente para cada bytecode. Tenha em mente que não se espera que bytecodes sejam executados entre máquinas virtuais Python diferentes, nem que se mantenham estáveis entre versões de Python.

Uma lista de instruções bytecode pode ser encontrada na documentação para o módulo dis.

chamável
Um chamável é um objeto que pode ser chamado, possivelmente com um conjunto de argumentos (veja *argumento*), com a seguinte sintaxe:

```
chamavel(argumento1, argumento2, argumentoN)
```

Uma *função*, e por extensão um *método*, é um chamável. Uma instância de uma classe que implementa o método `__call__()` também é um chamável.

função de retorno
Também conhecida como callback, é uma função sub-rotina que é passada como um argumento a ser executado em algum ponto no futuro.

classe
Um modelo para criação de objetos definidos pelo usuário. Definições de classe normalmente contém definições de métodos que operam sobre instâncias da classe.

variável de classe
Uma variável definida em uma classe e destinada a ser modificada apenas no nível da classe (ou seja, não em uma instância da classe).

variável de clausura
Uma *variável livre* referenciada de um *escopo aninhado* que é definida em um escopo externo em vez de ser resolvida em tempo de execução a partir dos espaços de nomes embutido ou globais. Pode ser explicitamente definida com a palavra reservada `nonlocal` para permitir acesso de gravação, ou implicitamente definida se a variável estiver sendo somente lida.

Por exemplo, na função `interna` no código a seguir, tanto `x` quanto `print` são *variáveis livres*, mas somente `x` é uma *variável de clausura*:

```
def externa():
    x = 0
    def interna():
        nonlocal x
        x += 1
        print(x)
    return interna
```

Devido ao atributo `codeobject.co_freevars` (que, apesar do nome, inclui apenas os nomes das variáveis de clausura em vez de listar todas as variáveis livres referenciadas), o termo mais geral *variável livre* às vezes é usado mesmo quando o significado pretendido é se referir especificamente às variáveis de clausura.

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.

contexto
Este termo tem diferentes significados dependendo de onde e como ele é usado. Alguns significados comuns:

- O estado ou ambiente temporário estabelecido por um *gerenciador de contexto* por meio de uma instrução `with`.
- A coleção de ligações de chave-valor associadas a um objeto `contextvars.Context` específico e acessadas por meio de objetos `ContextVar`. Veja também *variável de contexto*.
- Um objeto `contextvars.Context`. Veja também *contexto atual*.

protocolo de gerenciamento de contexto
Os métodos `__enter__()` e `__exit__()` chamados pela instrução `with`. Veja **PEP 343**.

gerenciador de contexto
Um objeto que implementa o *protocolo de gerenciamento de contexto* e controla o ambiente visto em uma instrução `with`. Veja **PEP 343**.

variável de contexto
Uma variável cujo valor depende de qual contexto é o *contexto atual*. Os valores são acessados por meio de objetos `contextvars.ContextVar`. Variáveis de contexto são usadas principalmente para isolar o estado entre tarefas assíncronas simultâneas.

contíguo
Um buffer é considerado contíguo exatamente se for *contíguo C* ou *contíguo Fortran*. Os buffers de dimensão zero são contíguos C e Fortran. Em vetores unidimensionais, os itens devem ser dispostos na memória próximos um do outro, em ordem crescente de índices, começando do zero. Em vetores multidimensionais contíguos C, o último índice varia mais rapidamente ao visitar itens em ordem de endereço de memória. No entanto, nos vetores contíguos do Fortran, o primeiro índice varia mais rapidamente.

corrotina
Corrotinas são uma forma mais generalizada de sub-rotinas. Sub-rotinas tem a entrada iniciada em um ponto, e a saída em outro ponto. Corrotinas podem entrar, sair, e continuar em muitos pontos diferentes. Elas podem ser implementadas com a instrução `async def`. Veja também **PEP 492**.

função de corrotina
Uma função que retorna um objeto do tipo *corrotina*. Uma função de corrotina pode ser definida com a instrução `async def`, e pode conter as palavras chaves `await`, `async for`, e `async with`. Isso foi introduzido pela **PEP 492**.

CPython
A implementação canônica da linguagem de programação Python, como disponibilizada pelo python.org. O termo "CPython" é usado quando necessário distinguir esta implementação de outras como Jython ou IronPython.

contexto atual
O *contexto* (objeto `contextvars.Context`) que é usado atualmente pelos objetos `ContextVar` para acessar (obter ou definir) os valores de *variáveis de contexto*. Cada thread tem seu próprio contexto atual. Frameworks para executar tarefas assíncronas (veja `asyncio`) associam cada tarefa a um contexto que se torna o contexto atual sempre que a tarefa inicia ou retoma a execução.

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(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

O mesmo conceito existe para as classes, mas não é comumente utilizado. Veja a documentação de definições de função e definições de classe para obter mais informações sobre decoradores.

descritor
Qualquer objeto que define os métodos `__get__()`, `__set__()` ou `__delete__()`. Quando um atributo de classe é um descritor, seu comportamento de associação especial é acionado no acesso a um atributo. Normalmente, ao se utilizar *a.b* para se obter, definir ou excluir, um atributo dispara uma busca no objeto chamado *b* no dicionário de classe de *a*, mas se *b* for um descritor, o respectivo método descritor é chamado. Compreender descritores é a chave para um profundo entendimento de Python pois eles são a base de muitas funcionalidades incluindo funções, métodos, propriedades, métodos de classe, métodos estáticos e referências para superclasses.

Para obter mais informações sobre os métodos dos descritores, veja: descriptors ou o Guia de Descritores.

dicionário
Um vetor associativo em que chaves arbitrárias são mapeadas para valores. As chaves podem ser quaisquer objetos que possuam os métodos `__hash__()` e `__eq__()`. Isso é chamado de hash em Perl.

compreensão de dicionário
Uma maneira compacta de processar todos ou parte dos elementos de um iterável e retornar um dicionário com os resultados. `results = {n: n ** 2 for n in range(10)}` gera um dicionário contendo a chave `n` mapeada para o valor `n ** 2`. Veja comprehensions.

visão de dicionário
Os objetos retornados por `dict.keys()`, `dict.values()` e `dict.items()` são chamados de visões de dicionário. Eles fornecem uma visão dinâmica das entradas do dicionário, o que significa que quando o dicionário é alterado, a visão reflete essas alterações. Para forçar a visão de dicionário a se tornar uma lista completa use `list(dictview)`. Veja dict-views.

docstring
Abreviatura de "documentation string" (string de documentação). 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.

tipagem pato
Também conhecida como *duck-typing*, é 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 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 no Python presume 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
Uma parte da sintaxe que pode ser avaliada para algum valor. Em outras palavras, uma expressão é a acumulação de elementos de expressão como literais, nomes, atributos de acesso, operadores ou chamadas de funções, todos os quais retornam um valor. Em contraste com muitas outras linguagens, nem todas as construções de linguagem são expressões. Também existem *instruções*, as quais não podem ser usadas como expressões, como, por exemplo, `while`. Atribuições também são instruções, não expressões.

módulo de extensão
Um módulo escrito em C ou C++, usando a API C do Python para interagir tanto com código de usuário quanto do núcleo.

f-string
Literais string prefixadas com `'f'` ou `'F'` são conhecidas como "f-strings" que é uma abreviação de formatted string literals. Veja também **PEP 498**.

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 *objetos arquivo ou similares* ou *fluxos*.

Atualmente há três categorias de objetos arquivo: *arquivos binários* brutos, *arquivos binários* em buffer e *arquivos textos*. Suas interfaces estão definidas no módulo `io`. A forma canônica para criar um objeto arquivo é usando a função `open()`.

objeto arquivo ou similar
Um sinônimo do termo *objeto arquivo*.

tratador de erros e codificação do sistema de arquivos
Tratador de erros e codificação usado pelo Python para decodificar bytes do sistema operacional e codificar Unicode para o sistema operacional.

A codificação do sistema de arquivos deve garantir a decodificação bem-sucedida de todos os bytes abaixo de 128. Se a codificação do sistema de arquivos falhar em fornecer essa garantia, as funções da API podem levantar `UnicodeError`.

As funções `sys.getfilesystemencoding()` e `sys.getfilesystemencodeerrors()` podem ser usadas para obter o tratador de erros e codificação do sistema de arquivos.

O *tratador de erros e codificação do sistema de arquivos* são configurados na inicialização do Python pela função `PyConfig_Read()`: veja os membros `filesystem_encoding` e `filesystem_errors` do `PyConfig`.

Veja também *codificação da localidade*.

localizador
Um objeto que tenta encontrar o *carregador* para um módulo que está sendo importado.

Existem dois tipos de localizador: *localizadores de metacaminho* para uso com `sys.meta_path`, e *localizadores de entrada de caminho* para uso com `sys.path_hooks`.

Veja finders-and-loaders e `importlib` para muito mais detalhes.

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**.

threads livres
Um modelo de threads onde múltiplas threads podem simultaneamente executar bytecode Python no mesmo interpretador. Isso está em contraste com a *trava global do interpretador* que permite apenas uma thread por vez executar bytecode Python. Veja **PEP 703**.

variável livre
Formalmente, conforme definido no modelo de execução de linguagem, uma variável livre é qualquer variável usada em um espaço de nomes que não seja uma variável local naquele espaço de nomes. Veja *variável de clausura* para um exemplo. Pragmaticamente, devido ao nome do atributo `codeobject.co_freevars`, o termo também é usado algumas vezes como sinônimo de *variável de clausura*.

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*, *método* e a seção function.

anotação de função
Uma *anotação* de um parâmetro de função ou valor de retorno.

Anotações de função são comumente usados por *dicas de tipo*: por exemplo, essa função espera receber dois argumentos `int` e também é esperado que devolva um valor `int`:

```
def soma_dois_numeros(a: int, b: int) -> int:
   return a + b
```

A sintaxe de anotação de função é explicada na seção function.

Veja *anotação de variável* e **PEP 484**, que descrevem esta funcionalidade. Veja também annotations-howto para as melhores práticas sobre como trabalhar com anotações.

__future__
A instrução future, `from __future__ import <feature>`, direciona o compilador a compilar o módulo atual usando sintaxe ou semântica que será padrão em uma versão futura de Python. O módulo `__future__` documenta os possíveis valores de *feature*. Importando esse módulo e avaliando suas variáveis, você pode ver quando um novo recurso foi inicialmente adicionado à linguagem e quando será (ou se já é) o padrão:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

coleta de lixo
Também conhecido como *garbage collection*, é o processo de liberar a memória quando ela não é mais utilizada. Python executa a liberação da memória através da contagem de referências e um coletor de lixo cíclico que é capaz de detectar e interromper referências cíclicas. O coletor de lixo pode ser controlado usando o módulo `gc`.

gerador
Uma função que retorna um *iterador gerador*. É parecida com uma função normal, exceto pelo fato de conter expressões `yield` para produzir uma série de valores que podem ser usados em um laço "for" ou que podem ser obtidos um de cada vez com a função `next()`.

Normalmente refere-se a uma função geradora, mas pode referir-se a um *iterador gerador* em alguns contextos. Em alguns casos onde o significado desejado não está claro, usar o termo completo evita ambiguidade.

iterador gerador
Um objeto criado por uma função *geradora*.

Cada `yield` suspende temporariamente o processamento, memorizando o estado da execução (incluindo variáveis locais e instruções try pendentes). Quando o *iterador gerador* retorna, ele se recupera do último ponto onde estava (em contrapartida as funções que iniciam uma nova execução a cada vez que são invocadas).

expressão geradora
Uma *expressão* que retorna um *iterador*. Parece uma expressão normal, seguido de uma cláusula `for` definindo uma variável de laço, um intervalo, e uma cláusula `if` opcional. A expressão combinada gera valores para uma função encapsuladora:

```
>>> sum(i*i for i in range(10))         # soma dos quadrados 0, 1, 4, ... 81
285
```

função genérica
Uma função composta por várias funções implementando a mesma operação para diferentes tipos. Qual implementação deverá ser usada durante a execução é determinada pelo algoritmo de despacho.

Veja também a entrada *despacho único* no glossário, o decorador `functools.singledispatch()`, e a **PEP 443**.

tipo genérico
Um *tipo* que pode ser parametrizado; tipicamente uma classe contêiner tal como `list` ou `dict`. Usado para *dicas de tipo* e *anotações*.

Para mais detalhes, veja tipo apelido genérico, **PEP 483**, **PEP 484**, **PEP 585**, e o módulo `typing`.

GIL
Veja *trava global do interpretador*.

trava 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 embutidos 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 a GIL ao realizar tarefas computacionalmente muito intensas, como compactação ou cálculos de hash. Além disso, a GIL é sempre liberado nas operações de E/S.

A partir de Python 3.13, o GIL pode ser desabilitado usando a configuração de construção `--disable-gil`. Depois de construir Python com essa opção, o código deve ser executado com a opção `-X gil=0` ou a variável de ambiente `PYTHON_GIL=0` deve estar definida. Esse recurso provê um desempenho melhor para aplicações com múltiplas threads e torna mais fácil o uso eficiente de CPUs com múltiplos núcleos. Para mais detalhes, veja **PEP 703**.

pyc baseado em hash
Um arquivo de cache em bytecode que usa hash ao invés do tempo, no qual o arquivo de código-fonte foi modificado pela última vez, para determinar a sua validade. Veja pyc-invalidation.

hasheável
Um objeto é *hasheável* se tem um valor de hash que nunca muda durante seu ciclo de vida (precisa ter um método `__hash__()`) e pode ser comparado com outros objetos (precisa ter um método `__eq__()`). Objetos hasheáveis que são comparados como iguais devem ter o mesmo valor de hash.

A hasheabilidade faz com que um objeto possa ser usado como uma chave de dicionário e como um membro de conjunto, pois estas estruturas de dados utilizam os valores de hash internamente.

A maioria dos objetos embutidos imutáveis do Python são hasheáveis; containers mutáveis (tais como listas ou dicionários) não são; containers imutáveis (tais como tuplas e frozensets) são hasheáveis apenas se os seus

elementos são hasheáveis. Objetos que são instâncias de classes definidas pelo usuário são hasheáveis por padrão. Todos eles comparam de forma desigual (exceto entre si mesmos), e o seu valor hash é derivado a partir do seu `id()`.

IDLE
Um ambiente de desenvolvimento e aprendizado integrado para Python. idle é um editor básico e um ambiente interpretador que vem junto com a distribuição padrão do Python.

imortal
Objetos imortais são um detalhe da implementação do CPython introduzida na **PEP 683**.

Se um objeto é imortal, sua *contagem de referências* nunca é modificada e, portanto, nunca é desalocado enquanto o interpretador está em execução. Por exemplo, `True` e `None` são imortais no CPython.

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.

caminho de importação
Uma lista de localizações (ou *entradas de caminho*) que são buscadas pelo *localizador baseado no caminho* por módulos para importar. Durante a importação, esta lista de localizações usualmente vem a partir de `sys.path`, mas para subpacotes ela também pode vir do atributo `__path__` de pacotes-pai.

importação
O processo pelo qual o código Python em um módulo é disponibilizado para o código Python em outro módulo.

importador
Um objeto que localiza e carrega um módulo; Tanto um *localizador* e o objeto *carregador*.

interativo
Python tem um interpretador interativo, o que significa que você pode digitar instruções 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)`). Para saber mais sobre modo interativo, veja tut-interac.

interpretado
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*.

desligamento do interpretador
Quando solicitado para desligar, o interpretador Python entra em uma fase especial, onde ele gradualmente libera todos os recursos alocados, tais como módulos e várias estruturas internas críticas. Ele também faz diversas chamadas para o *coletor de lixo*. Isto pode disparar a execução de código em destrutores definidos pelo usuário ou função de retorno de referência fraca. Código executado durante a fase de desligamento pode encontrar diversas exceções, pois os recursos que ele depende podem não funcionar mais (exemplos comuns são os módulos de bibliotecas, ou os mecanismos de avisos).

A principal razão para o interpretador desligar, é que o módulo `__main__` ou o script sendo executado terminou sua execução.

iterável
Um objeto capaz de retornar seus membros um de cada vez. Exemplos de iteráveis incluem todos os tipos de sequência (tais como `list`, `str` e `tuple`) e alguns tipos de não-sequência, como o `dict`, *objetos arquivos*, além dos objetos de quaisquer classes que você definir com um método `__iter__()` ou `__getitem__()` que implementam a semântica de *sequência* .

Iteráveis podem ser usados em um laço `for` e em vários outros lugares em que uma sequência é necessária (`zip()`, `map()`, ...). Quando um objeto iterável é passado como argumento para a função embutida `iter()`, ela retorna um iterador para o objeto. Este iterador é adequado para se varrer todo o conjunto de valores. Ao

usar iteráveis, normalmente não é necessário chamar `iter()` ou lidar com os objetos iteradores em si. A instrução `for` faz isso automaticamente para você, criando uma variável temporária para armazenar o iterador durante a execução do laço. Veja também *iterador*, *sequência*, e *gerador*.

iterador
Um objeto que representa um fluxo de dados. Repetidas chamadas ao método `__next__()` de um iterador (ou passando o objeto para a função embutida `next()`) vão retornar itens sucessivos do fluxo. Quando não houver mais dados disponíveis uma exceção `StopIteration` será levantada. Neste ponto, o objeto iterador se esgotou e quaisquer chamadas subsequentes a seu método `__next__()` vão apenas levantar a exceção `StopIteration` novamente. Iteradores precisam ter um método `__iter__()` que retorne o objeto iterador em si, de forma que todo iterador também é iterável e pode ser usado na maioria dos lugares em que um iterável é requerido. Uma notável exceção é código que tenta realizar passagens em múltiplas iterações. Um objeto contêiner (como uma `list`) produz um novo iterador a cada vez que você passá-lo para a função `iter()` ou utilizá-lo em um laço `for`. Tentar isso com o mesmo iterador apenas iria retornar o mesmo objeto iterador esgotado já utilizado na iteração anterior, como se fosse um contêiner vazio.

Mais informações podem ser encontradas em typeiter.

O CPython não aplica consistentemente o requisito de que um iterador defina `__iter__()`. E também observe que o CPython com threads livres não garante a segurança do thread das operações do iterador.

função chave
Uma função chave ou função colação é um chamável que retorna um valor usado 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.

Uma porção de ferramentas no Python aceitam funções chave para controlar como os elementos são ordenados ou agrupados. Algumas delas incluem `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` e `itertools.groupby()`.

Há várias maneiras de se criar funções chave. Por exemplo, o método `str.lower()` pode servir como uma função chave para ordenações insensíveis à caixa. Alternativamente, uma função chave ad-hoc pode ser construída a partir de uma expressão `lambda`, como `lambda r: (r[0], r[2])`. Além disso, `operator.attrgetter()`, `operator.itemgetter()` e `operator.methodcaller()` são três construtores de função chave. Consulte o guia de Ordenação para ver exemplos de como criar e utilizar funções chave.

argumento nomeado
Veja *argumento*.

lambda
Uma função de linha anônima consistindo de uma única *expressão*, 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 muitas instruções `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 travas ou usando a abordagem EAFP.

analisador léxico
Nome formal para o *tokenizador*; veja *token*.

lista
Uma *sequência* 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 $O(1)$.

compreensão de lista
Uma maneira compacta de processar todos ou parte dos elementos de uma sequência e retornar os resultados em uma lista. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` gera uma lista de strings contendo números hexadecimais (0x..) no intervalo de 0 a 255. A cláusula `if` é opcional. Se omitida, todos os elementos no `range(256)` serão processados.

carregador
Um objeto que carrega um módulo. Ele deve definir os métodos `exec_module()` e `create_module()` para implementar a interface `Loader`. Um carregador é normalmente retornado por um *localizador*. Veja também:

- finders-and-loaders
- `importlib.abc.Loader`
- **PEP 302**

codificação da localidade
No Unix, é a codificação da localidade do LC_CTYPE, que pode ser definida com `locale.setlocale(locale.LC_CTYPE, new_locale)`.

No Windows, é a página de código ANSI (ex: `"cp1252"`).

No Android e no VxWorks, o Python usa `"utf-8"` como a codificação da localidade.

`locale.getencoding()` pode ser usado para obter a codificação da localidade.

Veja também *tratador de erros e codificação do sistema de arquivos*.

método mágico
Um sinônimo informal para um *método especial*.

mapeamento
Um objeto contêiner que tem suporte a pesquisas de chave arbitrária e implementa os métodos especificados nas `collections.abc.Mapping` ou `collections.abc.MutableMapping` classes base abstratas. Exemplos incluem `dict`, `collections.defaultdict`, `collections.OrderedDict` e `collections.Counter`.

localizador de metacaminho
Um *localizador* retornado por uma busca de `sys.meta_path`. Localizadores de metacaminho são relacionados a, mas diferentes de, *localizadores de entrada de caminho*.

Veja `importlib.abc.MetaPathFinder` para os métodos que localizadores de metacaminho implementam.

metaclasse
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.

Mais informações podem ser encontradas em metaclasses.

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 *escopo aninhado*.

ordem de resolução de métodos
Ordem de resolução de métodos é a ordem em que os membros de uma classe base são buscados durante a pesquisa. Veja python_2.3_mro para detalhes do algoritmo usado pelo interpretador do Python desde a versão 2.3.

módulo
Um objeto que serve como uma unidade organizacional de código Python. Os módulos têm um espaço de nomes contendo objetos Python arbitrários. Os módulos são carregados pelo Python através do processo de *importação*.

Veja também *pacote*.

spec de módulo
Um espaço de nomes que contém as informações relacionadas à importação usadas para carregar um módulo. Uma instância de `importlib.machinery.ModuleSpec`.

Veja também module-specs.

MRO
Veja *ordem de resolução de métodos*.

mutável
Objeto mutável é aquele que pode modificar seus valor mas manter seu `id()`. Veja também *imutável*.

tupla nomeada
O termo "tupla nomeada" é aplicado a qualquer tipo ou classe que herda de tupla e cujos elementos indexáveis também são acessíveis usando atributos nomeados. O tipo ou classe pode ter outras funcionalidades também.

Diversos tipos embutidos são tuplas nomeadas, incluindo os valores retornados por `time.localtime()` e `os.stat()`. Outro exemplo é `sys.float_info`:

```
>>> sys.float_info[1]                    # acesso indexado
1024
>>> sys.float_info.max_exp               # acesso a campo nomeado
1024
>>> isinstance(sys.float_info, tuple)    # tipo de tupla
True
```

Algumas tuplas nomeadas são tipos embutidos (tal como os exemplos acima). Alternativamente, uma tupla nomeada pode ser criada a partir de uma definição de classe regular, que herde de `tuple` e que defina campos nomeados. Tal classe pode ser escrita a mão, ou ela pode ser criada herdando `typing.NamedTuple` ou com uma função fábrica `collections.namedtuple()`. As duas últimas técnicas também adicionam alguns métodos extras, que podem não ser encontrados quando foi escrita manualmente, ou em tuplas nomeadas embutidas.

espaço de nomes
O lugar em que uma variável é armazenada. Espaços de nomes são implementados como dicionários. Existem os espaços de nomes local, global e nativo, bem como espaços de nomes aninhados em objetos (em métodos). Espaços de nomes suportam modularidade ao prevenir conflitos de nomes. Por exemplo, as funções `__builtin__.open()` e `os.open()` são diferenciadas por seus espaços de nomes. Espaços de nomes também auxiliam na legibilidade e na manutenibilidade ao torar mais claro quais módulos implementam uma função. Escrever `random.seed()` ou `itertools.izip()`, por exemplo, deixa claro que estas funções são implementadas pelos módulos `random` e `itertools` respectivamente.

pacote de espaço de nomes
Um *pacote* que serve apenas como contêiner para subpacotes. Pacotes de espaços de nomes podem não ter representação física, e especificamente não são como um *pacote regular* porque eles não tem um arquivo `__init__.py`.

Pacotes de espaço de nomes permitem que vários pacotes instaláveis individualmente tenham um pacote pai comum. Caso contrário, é recomendado usar um *pacote regular*.

Para mais informações, veja **PEP 420** e reference-namespace-package.

Veja também *módulo*.

escopo aninhado
A habilidade de referir-se a uma variável em uma definição de fechamento. Por exemplo, uma função definida dentro de outra pode referenciar variáveis da função externa. Perceba que escopos aninhados por padrão funcionam apenas por referência e não por atribuição. Variáveis locais podem ler e escrever no escopo mais interno. De forma similar, variáveis globais podem ler e escrever para o espaço de nomes global. O `nonlocal` permite escrita para escopos externos.

classe estilo novo
Antigo nome para o tipo de classes agora usado para todos os objetos de classes. Em versões anteriores do Python, apenas classes estilo podiam usar recursos novos e versáteis do Python, tais como `__slots__`, descritores, propriedades, `__getattribute__()`, métodos de classe, e métodos estáticos.

objeto
Qualquer dado que tenha estado (atributos ou valores) e comportamento definidos (métodos). Também a

última classe base de qualquer *classe estilo novo*.

escopo otimizado
Um escopo no qual os nomes das variáveis locais de destino são conhecidos de forma confiável pelo compilador quando o código é compilado, permitindo a otimização do acesso de leitura e gravação a esses nomes. Os espaços de nomes locais para funções, geradores, corrotinas, compreensões e expressões geradoras são otimizados desta forma. Nota: a maioria das otimizações de interpretador são aplicadas a todos os escopos, apenas aquelas que dependem de um conjunto conhecido de nomes de variáveis locais e não locais são restritas a escopos otimizados.

pacote
Um *módulo* Python é capaz de conter submódulos ou recursivamente, subpacotes. Tecnicamente, um pacote é um módulo Python com um atributo `__path__`.

Veja também *pacote regular* e *pacote de espaço de nomes*.

parâmetro
Uma entidade nomeada na definição de uma *função* (ou método) que específica um *argumento* (ou em alguns casos, argumentos) que a função pode receber. Existem cinco tipos de parâmetros:

- *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): ...
```

- *somente-posicional*: especifica um argumento que pode ser fornecido apenas por posição. Parâmetros somente-posicionais podem ser definidos incluindo o caractere `/` na lista de parâmetros da definição da função após eles, por exemplo *somentepos1* e *somentepos2* a seguir:

```
def func(somentepos1, somentepos2, /, posicional_ou_nomeado): ...
```

- *somente-nomeado*: especifica um argumento que pode ser passado para a função somente por nome. Parâmetros somente-nomeados podem ser definidos com um simples parâmetro var-posicional ou um `*` antes deles na lista de parâmetros na definição da função, por exemplo *somente_nom1* and *somente_nom2* a seguir:

```
def func(arg, *, somente_nom1, somente_nom2): ...
```

- *var-posicional*: especifica que 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 do parâmetro, 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 padrão para alguns argumentos opcionais.

Veja também o termo *argumento* no glossário, a pergunta do FAQ sobre a diferença entre argumentos e parâmetros, a classe `inspect.Parameter`, a seção function e a **PEP 362**.

entrada de caminho
Um local único no *caminho de importação* que o *localizador baseado no caminho* consulta para encontrar módulos a serem importados.

localizador de entrada de caminho
Um *localizador* retornado por um chamável em `sys.path_hooks` (ou seja, um *gancho de entrada de caminho*) que sabe como localizar os módulos *entrada de caminho*.

Veja `importlib.abc.PathEntryFinder` para os métodos que localizadores de entrada de caminho implementam.

gancho de entrada de caminho
Um chamável na lista `sys.path_hooks` que retorna um *localizador de entrada de caminho* caso saiba como localizar módulos em uma *entrada de caminho* específica.

localizador baseado no caminho
Um dos *localizadores de metacaminho* padrão que procura por um *caminho de importação* de módulos.

objeto caminho ou similar
Um objeto representando um caminho de sistema de arquivos. Um objeto caminho ou similar é ou um objeto `str` ou `bytes` representando um caminho, ou um objeto implementando o protocolo `os.PathLike`. Um objeto que suporta o protocolo `os.PathLike` pode ser convertido para um arquivo de caminho do sistema `str` ou `bytes`, através da chamada da função `os.fspath()`; `os.fsdecode()` e `os.fsencode()` podem ser usadas para garantir um `str` ou `bytes` como resultado, respectivamente. Introduzido na **PEP 519**.

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 têm 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**.

porção
Um conjunto de arquivos em um único diretório (possivelmente armazenado em um arquivo zip) que contribuem para um pacote de espaço de nomes, conforme definido em **PEP 420**.

argumento posicional
Veja *argumento*.

API provisória
Uma API provisória é uma API que foi deliberadamente excluída das bibliotecas padrões com compatibilidade retroativa garantida. Enquanto mudanças maiores para tais interfaces não são esperadas, contanto que elas sejam marcadas como provisórias, mudanças retroativas incompatíveis (até e incluindo a remoção da interface) podem ocorrer se consideradas necessárias pelos desenvolvedores principais. Tais mudanças não serão feitas gratuitamente – elas irão ocorrer apenas se sérias falhas fundamentais forem descobertas, que foram esquecidas anteriormente a inclusão da API.

Mesmo para APIs provisórias, mudanças retroativas incompatíveis são vistas como uma "solução em último caso" - cada tentativa ainda será feita para encontrar uma resolução retroativa compatível para quaisquer problemas encontrados.

Esse processo permite que a biblioteca padrão continue a evoluir com o passar do tempo, sem se prender em erros de design problemáticos por períodos de tempo prolongados. Veja **PEP 411** para mais detalhes.

pacote provisório
Veja *API provisória*.

Python 3000
Apelido para a linha de lançamento da versão do Python 3.x (cunhada há muito tempo, quando o lançamento da versão 3 era algo em um futuro muito distante.) Esse termo possui a seguinte abreviação: "Py3k".

Pythônico
Uma ideia ou um pedaço de código que segue de perto as formas de escritas mais comuns da linguagem Python, ao invés de implementar códigos usando conceitos comuns a outras linguagens. Por exemplo, um formato comum em Python é fazer um laço 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(comida)):
    print(comida[i])
```

Ao contrário do método mais limpo, Pythônico:

```
for parte in comida:
    print(parte)
```

nome qualificado
Um nome pontilhado (quando 2 termos são ligados por um ponto) que mostra o "path" do escopo global de um módulo para uma classe, função ou método definido num determinado módulo, conforme definido pela **PEP 3155**. Para funções e classes de nível superior, o nome qualificado é o mesmo que o nome do objeto:

```
>>> class C:
...     class D:
...         def metodo(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.metodo.__qualname__
'C.D.metodo'
```

Quando usado para se referir a módulos, o *nome totalmente qualificado* significa todo o caminho pontilhado para o módulo, incluindo quaisquer pacotes pai, por exemplo: `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

contagem de referências
O número de referências a um objeto. Quando a contagem de referências de um objeto cai para zero, ele é desalocado. Alguns objetos são *imortais* e têm contagens de referências que nunca são modificadas e, portanto, os objetos nunca são desalocados. A contagem de referências geralmente não é visível para o código Python, mas é um elemento-chave da implementação do *CPython*. Os programadores podem chamar a função `sys.getrefcount()` para retornar a contagem de referências para um objeto específico.

pacote regular
Um *pacote* tradicional, como um diretório contendo um arquivo `__init__.py`.

Veja também *pacote de espaço de nomes*.

REPL
Um acrônimo para "read–eval–print loop", outro nome para o console *interativo* do interpretador.

__slots__
Uma declaração dentro de uma classe que economiza memória pré-declarando espaço para atributos de instâncias, e eliminando dicionários de instâncias. Apesar de popular, a técnica é um tanto quanto complicada de acertar, e é melhor se for reservada para casos raros, onde existe uma grande quantidade de instâncias em uma aplicação onde a memória é crítica.

sequência
Um *iterável* com suporte para acesso eficiente a seus elementos através de índices inteiros via método especial `__getitem__()` e que define o método `__len__()` que devolve o tamanho da sequência. Alguns tipos de sequência embutidos são: `list`, `str`, `tuple`, e `bytes`. Note que `dict` também tem suporte para `__getitem__()` e `__len__()`, mas é considerado um mapeamento e não uma sequência porque a busca usa uma chave *hasheável* arbitrária em vez de inteiros.

A classe base abstrata `collections.abc.Sequence` define uma interface mais rica que vai além de apenas `__getitem__()` e `__len__()`, adicionando `count()`, `index()`, `__contains__()`, e `__reversed__()`. Tipos que implementam essa interface podem ser explicitamente registrados usando `register()`. Para mais documentação sobre métodos de sequências em geral, veja Operações comuns de sequências.

compreensão de conjunto
Uma maneira compacta de processar todos ou parte dos elementos em iterável e retornar um conjunto com os resultados. `results = {c for c in 'abracadabra' if c not in 'abc'}` gera um conjunto de strings `{'r', 'd'}`. Veja comprehensions.

despacho único
Uma forma de despacho de *função genérica* onde a implementação é escolhida com base no tipo de um único argumento.

fatia
Um objeto geralmente contendo uma parte de uma *sequência*. Uma fatia é criada usando a notação de subscrito `[]` pode conter também até dois pontos entre números, como em `variable_name[1:3:5]`. A notação de suporte (subscrito) utiliza objetos `slice` internamente.

suavemente descontinuado
Uma API suavemente descontinuada não deve ser usada em código novo, mas é seguro para código já existente usá-la. A API continua documentada e testada, mas não será aprimorada mais.

A descontinuação suave, diferentemente da descontinuação normal, não planeja remover a API e não emitirá avisos.

Veja PEP 387: Descontinuação suave.

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 *expressão* ou uma de várias construções com uma palavra reservada, tal como `if`, `while` ou `for`.

verificador de tipo estático
Uma ferramenta externa que lê o código Python e o analisa, procurando por problemas como tipos incorretos. Consulte também *dicas de tipo* e o módulo `typing`.

referência forte
Na API C do Python, uma referência forte é uma referência a um objeto que pertence ao código que contém a referência. A referência forte é obtida chamando *`Py_INCREF()`* quando a referência é criada e liberada com *`Py_DECREF()`* quando a referência é excluída.

A função *`Py_NewRef()`* pode ser usada para criar uma referência forte para um objeto. Normalmente, a função *`Py_DECREF()`* deve ser chamada na referência forte antes de sair do escopo da referência forte, para evitar o vazamento de uma referência.

Veja também *referência emprestada*.

codificador de texto
Uma string em Python é uma sequência de pontos de código Unicode (no intervalo `U+0000–U+10FFFF`). Para armazenar ou transferir uma string, ela precisa ser serializada como uma sequência de bytes.

A serialização de uma string em uma sequência de bytes é conhecida como "codificação" e a recriação da string a partir de uma sequência de bytes é conhecida como "decodificação".

Há uma variedade de diferentes serializações de texto codecs, que são coletivamente chamadas de "codificações de texto".

arquivo texto
Um *objeto arquivo* apto a ler e escrever objetos `str`. Geralmente, um arquivo texto, na verdade, acessa um fluxo de dados de bytes e captura o *codificador de texto* automaticamente. Exemplos de arquivos texto são: arquivos abertos em modo texto (`'r'` or `'w'`), `sys.stdin`, `sys.stdout`, e instâncias de `io.StringIO`.

Veja também *arquivo binário* para um objeto arquivo apto a ler e escrever *objetos byte ou similar*.

token
Uma pequena unidade de código-fonte, gerada pelo analisador léxico (também chamado de *tokenizador*). Nomes, números, strings, operadores, quebras de linha e similares são representados por tokens.

O módulo `tokenize` expõe o analisador léxico do Python. O módulo `token` contém informações sobre os vários tipos de tokens.

aspas triplas
Uma string que está definida com três ocorrências de aspas duplas (") ou apóstrofos ('). 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 escapadas 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.

tipo
O tipo de um objeto Python determina qual classe 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)`.

apelido de tipo
Um sinônimo para um tipo, criado através da atribuição do tipo para um identificador.

Apelidos de tipo são úteis para simplificar *dicas de tipo*. Por exemplo:

```
def remove_tons_de_cinza(
        cores: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

pode tornar-se mais legível desta forma:

```
Cor = tuple[int, int, int]

def remove_tons_de_cinza(cores: list[Cor]) -> list[Cor]:
    pass
```

Veja `typing` e **PEP 484**, a qual descreve esta funcionalidade.

dica de tipo
Uma *anotação* que especifica o tipo esperado para uma variável, um atributo de classe, ou um parâmetro de função ou um valor de retorno.

Dicas de tipo são opcionais e não são forçadas pelo Python, mas elas são úteis para *verificadores de tipo estático*. Eles também ajudam IDEs a completar e refatorar código.

Dicas de tipos de variáveis globais, atributos de classes, e funções, mas não de variáveis locais, podem ser acessadas usando `typing.get_type_hints()`.

Veja `typing` e **PEP 484**, a qual descreve esta funcionalidade.

novas linhas universais
Uma maneira de interpretar fluxos de textos, na qual todos estes são reconhecidos como caracteres de fim de linha: a convenção para fim de linha no Unix `'\n'`, a convenção no Windows `'\r\n'`, e a antiga convenção no Macintosh `'\r'`. Veja **PEP 278** e **PEP 3116**, bem como `bytes.splitlines()` para uso adicional.

anotação de variável
Uma *anotação* de uma variável ou um atributo de classe.

Ao fazer uma anotação de uma variável ou um atributo de classe, a atribuição é opcional:

```
class C:
    campo: 'anotação'
```

Anotações de variáveis são normalmente usadas para *dicas de tipo*: por exemplo, espera-se que esta variável receba valores do tipo `int`:

```
contagem: int = 0
```

A sintaxe de anotação de variável é explicada na seção annassign.

Veja *anotação de função*, **PEP 484** e **PEP 526**, que descrevem esta funcionalidade. Veja também annotations--howto para as melhores práticas sobre como trabalhar com anotações.

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.

Veja também `venv`.

máquina virtual
Um computador definido inteiramente em software. A máquina virtual de Python executa o *bytecode* emitido pelo compilador de bytecode.

Zen do 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 esta documentação

A documentação do Python é gerada a partir de fontes reStructuredText usando Sphinx, um gerador de documentação criado originalmente para Python e agora mantido como um projeto independente.

O desenvolvimento da documentação e de suas ferramentas é um esforço totalmente voluntário, como 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 Python e autor de boa parte do conteúdo;
- O projeto Docutils por criar reStructuredText e o pacote Docutils;
- Fredrik Lundh, pelo seu projeto de referência alternativa em Python, do qual Sphinx pegou muitas boas ideias.

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

Python foi criado no início dos anos 1990 por Guido van Rossum no Stichting Mathematisch Centrum (CWI, veja https://www.cwi.nl) na Holanda como sucessor de uma linguagem chamada ABC. Guido continua sendo o principal autor do Python, embora inclua muitas contribuições de outros.

Em 1995, Guido continuou seu trabalho em Python na Corporation for National Research Initiatives (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 de desenvolvimento do núcleo Python mudaram-se para BeOpen.com para formar a equipe BeOpen PythonLabs. Em outubro do mesmo ano, a equipe PythonLabs mudou-se para a Digital Creations, que se tornou Zope Corporation. Em 2001, a Python Software Foundation (PSF, veja https://www.python.org/psf/) foi formada, uma organização sem fins lucrativos criada especificamente para possuir Propriedade Intelectual relacionada ao Python. A Zope Corporation era um membro patrocinador da 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.

Versão	Derivada de	Ano	Proprietário	Compatível com a GPL? (1)
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	sim (2)
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

(1) Compatível com a 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 a GPL possibilitam combinar o Python com outro software lançado sob a GPL; os outros não.

(2) De acordo com Richard Stallman, 1.6.1 não é compatível com GPL, porque sua licença tem uma cláusula de escolha de lei. De acordo com a CNRI, no entanto, o advogado de Stallman disse ao advogado da CNRI que 1.6.1 "não é incompatível" com a GPL.

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

O software e a documentação do Python são licenciados sob a Python Software Foundation License Versão 2.

A partir do Python 3.8.6, exemplos, receitas e outros códigos na documentação são licenciados duplamente sob o Licença PSF versão 2 e a *Licença BSD de Zero Cláusula*.

Alguns softwares incorporados ao Python estão sob licenças diferentes. As licenças são listadas com o código abrangido por essa licença. Veja *Licenças e Reconhecimentos para Software Incorporado* para uma lista incompleta dessas licenças.

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 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.

4. PSF is making Python available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, 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.
```

(continua na próxima página)

(continuação da página anterior)

```
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, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 ACORDO DE LICENCIAMENTO DA BEOPEN.COM PARA PYTHON 2.0

ACORDO DE LICENCIAMENTO DA BEOPEN DE FONTE ABERTA DO 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
   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
   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.
```

(continua na próxima página)

(continuação da página anterior)

```
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.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 Licenças e Reconhecimentos para Software Incorporado

Esta seção é uma lista incompleta, mas crescente, de licenças e reconhecimentos para softwares de terceiros incorporados na distribuição do Python.

C.3.1 Mersenne Twister

A extensão C `_random` subjacente ao 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)
```

(continua na próxima página)

(continuação da página anterior)

```
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Soquetes

O módulo `socket` usa as funções `getaddrinfo()` e `getnameinfo()`, que são codificadas em arquivos de origem separados do Projeto WIDE, https://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.
```

(continua na próxima página)

(continuação da página anterior)

```
THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Serviços de soquete assíncrono

Os módulos `test.support.asynchat` e `test.support.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.3.4 Gerenciamento de cookies

O módulo `http.cookies` contém o seguinte aviso:

```
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
```

(continua na próxima página)

(continuação da página anterior)

```
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 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.3.6 Funções UUencode e UUdecode

O codec `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
```

(continua na próxima página)

(continuação da página anterior)

```
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 Chamadas de procedimento remoto XML

O módulo `xmlrpc.client` contém o seguinte aviso:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

O módulo `test.test_epoll` contém o seguinte aviso:

```
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:
```

(continua na próxima página)

(continuação da página anterior)

```
The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 kqueue de seleção

O módulo `select` contém o seguinte aviso para a interface do kqueue:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

O arquivo `Python/pyhash.c` contém a implementação de Marek Majkowski do algoritmo SipHash24 de Dan Bernstein. Contém a seguinte nota:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
```

(continua na próxima página)

(continuação da página anterior)

```
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod 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 https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. 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.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

Os módulos `hashlib`, `posix` e `ssl` 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: Para o lançamento do OpenSSL 3.0, e lançamentos posteriores derivados deste, se aplica a Apache License v2:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.
```

(continua na próxima página)

(continuação da página anterior)

```
   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
```

(continua na próxima página)

(continuação da página anterior)

```
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
```

(continua na próxima página)

(continuação da página anterior)

```
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

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.3.14 libffi

A extensão C `_ctypes` subjacente ao módulo `ctypes` é construída usando uma cópia incluída das fontes do libffi, a menos que a construção esteja configurada com `--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.3.15 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) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

A implementação da tabela de hash usada pelo `tracemalloc` é baseada no projeto cfuhash:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
```

(continua na próxima página)

(continuação da página anterior)

```
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

A extensão C `_decimal` subjacente ao módulo `decimal` é construída usando uma cópia incluída da biblioteca libmpdec, a menos que a construção esteja configurada com `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 Conjunto de testes C14N do W3C

O conjunto de testes C14N 2.0 no pacote `test` (`Lib/test/xmltestdata/c14n-20/`) foi recuperado do site do W3C em https://www.w3.org/TR/xml-c14n2-testcases/ e é distribuído sob a licença BSD de 3 cláusulas:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
```

(continua na próxima página)

(continuação da página anterior)

```
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

Licença MIT:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

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.3.20 asyncio

Partes do módulo `asyncio` são incorporadas do uvloop 0.16, que é distribuído sob a licença MIT:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.
```

(continua na próxima página)

(continuação da página anterior)

```
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.3.21 Global Unbounded Sequences (GUS)

O arquivo `Python/qsbr.c` é adaptado do esquema de recuperação de memória segura "Global Unbounded Sequences" do FreeBSD em subr_smr.c. O arquivo é distribuído sob a licença BSD de 2 cláusulas:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

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 unmodified, 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 "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 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.
```

APÊNDICE D

Direitos autorais

Veja: *História e Licença* para informações completas de licença e permissões.

Índice

Não alfabético

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z