
The Python/C API

Release 3.9.18

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

fevereiro 06, 2024

**Python Software Foundation
Email: docs@python.org**

1	Introdução	3
1.1	Padrões de codificação	4
1.2	Incluir arquivos	4
1.3	Macros úteis	5
1.4	Objetos, tipos e contagens de referências	6
1.4.1	Contagens de referências	7
1.4.2	Tipos	10
1.5	Exceções	10
1.6	Incorporando Python	12
1.7	Compilações de depuração	13
2	Interface binária de aplicativo estável	15
3	A camada de Mais Alto Nível	17
4	Contagem de Referências	23
5	Manipulando Exceções	25
5.1	Impressão e limpeza	26
5.2	Lançando exceções	26
5.3	Emitindo advertências	29
5.4	Consultando o indicador de erro	30
5.5	Tratamento de sinal	31
5.6	Classes de exceção	32
5.7	Objeto Exceção	32
5.8	Objetos de exceção Unicode	33
5.9	Controle de recursão	34
5.10	Exceções Padrão	35
5.11	Categorias de aviso padrão	36
6	Utilitários	39
6.1	Utilitários do Sistema Operacional	39
6.2	System Functions	42
6.3	Process Control	43
6.4	Importando módulos	44
6.5	Suporte a <i>marshalling</i> de dados	48
6.6	Análise de argumentos e construção de valores	49

6.6.1	Análise de argumentos	49
6.6.2	Construindo valores	55
6.7	Conversão e formação de strings	57
6.8	Reflexão	58
6.9	Registro de codec e funções de suporte	59
6.9.1	API de pesquisa de codec	60
6.9.2	API de registro de tratamentos de erros de decodificação Unicode	60
7	Camada de Objetos Abstratos	63
7.1	Protocolo de objeto	63
7.2	Protocolo de chamada	67
7.2.1	O protocolo <i>tp_call</i>	67
7.2.2	O protocolo <i>vectorcall</i>	67
7.2.3	API de chamada de objetos	69
7.2.4	API de suporte a chamadas	72
7.3	Protocolo de número	72
7.4	Protocolo de sequência	75
7.5	Protocolo de mapeamento	76
7.6	Protocolo Iterador	78
7.7	Protocolo de Buffer	78
7.7.1	Estrutura de Buffer	79
7.7.2	Tipos de solicitação do buffer	81
7.7.3	Vetores Complexos	83
7.7.4	Funções relacionadas ao Buffer	84
7.8	Protocolo de Buffer Antigo	85
8	Camada de Objetos Concretos	87
8.1	Objetos Fundamentais	87
8.1.1	Objetos tipo	87
8.1.2	O Objeto <code>None</code>	91
8.2	Objetos Numéricos	91
8.2.1	Objetos Inteiros	91
8.2.2	Objetos Booleanos	94
8.2.3	Objetos de ponto flutuante	95
8.2.4	Objetos de números complexos	96
8.3	Objetos Sequência	97
8.3.1	Objetos Bytes	97
8.3.2	Objetos Byte Array	99
8.3.3	Objetos Unicode e Codecs	100
8.3.4	Objeto tupla	119
8.3.5	Objetos sequência de estrutura	121
8.3.6	Objeto List	122
8.4	Coleções	123
8.4.1	Objetos dicionários	123
8.4.2	Objeto Set	126
8.5	Objetos Função	128
8.5.1	Objetos Função	128
8.5.2	Objetos de Método de Instância	129
8.5.3	Objetos método	129
8.5.4	Objeto célula	130
8.5.5	Objetos código	130
8.6	Outros Objetos	131
8.6.1	Objetos arquivos	131
8.6.2	Objeto Module	132

8.6.3	Objetos Iteradores	139
8.6.4	Objetos Descritores	139
8.6.5	Objetos Slice	140
8.6.6	Objeto Ellipsis	141
8.6.7	Objetos MemoryView	141
8.6.8	Objetos de referência fraca	142
8.6.9	Capsules	143
8.6.10	Objetos Geradores	144
8.6.11	Objetos corrotina	145
8.6.12	Objetos de variáveis de contexto	145
8.6.13	Objetos DateTime	147
8.6.14	Objetos de indicação de tipos	150
9	Inicialização, Finalização e Threads	151
9.1	Antes da Inicialização do Python	151
9.2	Variáveis de configuração global	152
9.3	Inicializando e encerrando o interpretador	154
9.4	Process-wide parameters	155
9.5	Thread State and the Global Interpreter Lock	158
9.5.1	Releasing the GIL from extension code	158
9.5.2	Non-Python created threads	159
9.5.3	Cuidados com o uso de fork()	160
9.5.4	High-level API	160
9.5.5	Low-level API	162
9.6	Sub-interpreter support	165
9.6.1	Bugs and caveats	166
9.7	Notificações assíncronas	166
9.8	Profiling and Tracing	167
9.9	Advanced Debugger Support	168
9.10	Thread Local Storage Support	169
9.10.1	Thread Specific Storage (TSS) API	169
9.10.2	Thread Local Storage (TLS) API	170
10	Configuração de Inicialização do Python	171
10.1	PyWideStringList	172
10.2	PyStatus	173
10.3	PyPreConfig	174
10.4	Preinitialization with PyPreConfig	175
10.5	PyConfig	176
10.6	Initialization with PyConfig	181
10.7	Isolated Configuration	182
10.8	Configuração do Python	182
10.9	Path Configuration	183
10.10	Py_RunMain()	185
10.11	Py_GetArgcArgv()	185
10.12	Multi-Phase Initialization Private Provisional API	185
11	Gerenciamento de Memória	187
11.1	Visão Geral	187
11.2	Raw Memory Interface	188
11.3	Interface da Memória	189
11.4	Alocadores de objeto	190
11.5	Alocadores de memória padrão	191
11.6	Alocadores de memória	191

11.7	The pymalloc allocator	193
11.7.1	Customize pymalloc Arena Allocator	194
11.8	tracemalloc C API	194
11.9	Exemplos	194
12	Suporte a implementação de Objetos	197
12.1	Alocando Objetos na Pilha	197
12.2	Estruturas Comuns de Objetos	198
12.2.1	Base object types and macros	198
12.2.2	Implementing functions and methods	199
12.2.3	Accessing attributes of extension types	202
12.3	Objetos tipo	203
12.3.1	Referências rápidas	204
12.3.2	PyTypeObject Definition	209
12.3.3	PyObject Slots	210
12.3.4	PyVarObject Slots	211
12.3.5	PyTypeObject Slots	211
12.3.6	Heap Types	228
12.4	Number Object Structures	228
12.5	Mapping Object Structures	231
12.6	Sequence Object Structures	231
12.7	Buffer Object Structures	232
12.8	Async Object Structures	233
12.9	Slot Type typedefs	234
12.10	Exemplos	235
12.11	Suporte a Coleta Cíclica de Lixo	237
13	API e Versionamento ABI	241
A	Glossário	243
B	Sobre esses documentos	257
B.1	Contribuidores da Documentação Python	257
C	História e Licença	259
C.1	História do software	259
C.2	Termos e condições para acessar ou usar Python	260
C.2.1	ACORDO DE LICENCIAMENTO DA PSF PARA PYTHON 3.9.18	260
C.2.2	ACORDO DE LICENCIAMENTO DA BEOPEN.COM PARA PYTHON 2.0	261
C.2.3	CONTRATO DE LICENÇA DA CNRI PARA O PYTHON 1.6.1	262
C.2.4	ACORDO DE LICENÇA DA CWI PARA PYTHON 0.9.0 A 1.2	263
C.2.5	LICENÇA BSD DE ZERO CLÁUSULA PARA CÓDIGO NA DOCUMENTAÇÃO DO PYTHON 3.9.18	263
C.3	Licenças e Reconhecimentos para Software Incorporado	264
C.3.1	Mersenne Twister	264
C.3.2	Soquetes	265
C.3.3	Serviços de soquete assíncrono	265
C.3.4	Gerenciamento de cookies	266
C.3.5	Rastreamento de execução	266
C.3.6	Funções UUencode e UUdecode	267
C.3.7	Chamadas de procedimento remoto XML	267
C.3.8	test_epoll	268
C.3.9	kqueue de seleção	268
C.3.10	SipHash24	269
C.3.11	strtod e dtoa	270

C.3.12	OpenSSL	270
C.3.13	expat	272
C.3.14	libffi	273
C.3.15	zlib	274
C.3.16	cfuhash	274
C.3.17	libmpdec	275
C.3.18	Conjunto de testes C14N do W3C	275
D	Direitos autorais	277
	Índice	279

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.

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 Incluir arquivos

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.

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.

Novo na versão 3.7.

Py_ABS(x)

Retorna o valor absoluto de `x`.

Novo na versão 3.3.

Py_MIN(x, y)

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

Novo na versão 3.3.

Py_MAX(x, y)

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

Novo na versão 3.3.

Py_STRINGIFY(x)

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

Novo na versão 3.4.

Py_MEMBER_SIZE(type, member)

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

Novo na versão 3.6.

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_GETENV(s)

Como `getenv(s)`, mas retorna `NULL` se `-E` foi passada na linha de comando (isto é, se `Py_IgnoreEnvironmentFlag` estiver definida).

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; }.`

Novo na versão 3.4.

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.

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 sqlite_row_methods[] = {
    {"keys", (PyCFunction)sqlite_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

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

As contagens de referências são sempre manipuladas explicitamente. A maneira normal é usar a macro `Py_INCREF()` para incrementar a contagem de referências de um objeto em um, e `Py_DECREF()` para diminuí-la em um. A macro `Py_DECREF()` é consideravelmente mais complexa que a “`incrcf`”, uma vez que deve verificar se a contagem de referências torna-se zero e então fazer com que o desalocador do objeto seja chamado. O desalocador é um ponteiro de função contido na estrutura de tipo do objeto. O desalocador específico do tipo se encarrega de diminuir as contagens de referências para outros objetos contidos no objeto se este for um tipo de objeto composto, como uma lista, bem como realizar qualquer finalização adicional necessária. Não há chance de que a contagem de referências possa estourar; pelo menos tantos bits são usados para manter a contagem de referência quanto há locais de memória distintos na memória virtual (assumindo `sizeof(Py_ssize_t) >= sizeof(void*)`). Portanto, o incremento da contagem de referências é uma operação simples.

Não é necessário incrementar a contagem de referências de um objeto para cada variável local que contém um ponteiro para um objeto. Em teoria, a contagem de referências do objeto aumenta em um quando a variável é feita para apontar para ele e diminui em um quando a variável sai do escopo. No entanto, esses dois se cancelam, portanto, no final, a contagem de referências não mudou. A única razão real para usar a contagem de referências é evitar que o objeto seja desalocado enquanto nossa variável estiver apontando para ele. Se sabemos que existe pelo menos uma outra referência ao objeto que vive pelo menos tanto quanto nossa variável, não há necessidade de incrementar a contagem de referências temporariamente. Uma situação importante em que isso ocorre é em objetos que são passados como argumentos para funções C em um módulo de extensão que são chamados de Python; o mecanismo de chamada garante manter uma referência a todos os argumentos durante a chamada.

No entanto, uma armadilha comum é extrair um objeto de uma lista e mantê-lo por um tempo, sem incrementar sua contagem de referências. Alguma outra operação poderia remover o objeto da lista, diminuindo sua contagem de referências e possivelmente desalocando-o. O perigo real é que operações aparentemente inocentes podem invocar código Python arbitrário que poderia fazer isso; existe um caminho de código que permite que o controle flua de volta para o usuário a partir de um `Py_DECREF()`, então quase qualquer operação é potencialmente perigosa.

Uma abordagem segura é sempre usar as operações genéricas (funções cujo nome começa com `PyObject_`, `PyNumber_`, `PySequence_` ou `PyMapping_`). Essas operações sempre incrementam a contagem de referências do objeto que retornam. Isso deixa o chamador com a responsabilidade de chamar `Py_DECREF()` quando terminar com o resultado; isso logo se torna uma segunda natureza.

Detalhes da contagem de referências

O comportamento da contagem de referências de funções na API Python/C é melhor explicado em termos de *propriedade de referências*. A propriedade pertence às referências, nunca aos objetos (os objetos não são possuídos: eles são sempre compartilhados). “Possuir uma referência” significa ser responsável por chamar `Py_DECREF` nela quando a referência não for mais necessária. A propriedade também pode ser transferida, o que significa que o código que recebe a propriedade da referência torna-se responsável por eventualmente efetuar um `decrcf` nela chamando `Py_DECREF()` ou `Py_XDECREF()` quando não é mais necessário — ou passando essa responsabilidade (geralmente para o responsável pela chamada). Quando uma função passa a propriedade de uma referência para seu chamador, diz-se que o chamador recebe uma *nova* referência. Quando nenhuma propriedade é transferida, diz-se que o chamador *toma emprestado* a referência. Nada precisa ser feito para uma referência emprestada.

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 assume 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");
```

É muito mais comum usar `PyObject_SetItem()` e amigos com itens cujas referências você está apenas pegando emprestado, como argumentos que foram passados para a função que você está escrevendo. Nesse caso, o comportamento deles em relação às contagens de referência é muito mais são, já que você não precisa incrementar uma contagem de referências para que possa dar uma referência (“mande-a ser roubada”). Por exemplo, esta função define todos os itens de uma lista (na verdade, qualquer sequência mutável) para um determinado item:

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

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

(continua na próxima página)

(continuação da página anterior)

```

    }
    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
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)) {

```

(continua na próxima página)

(continuação da página anterior)

```

        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.

`Py_ssize_t`

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

1.5 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.

Um exemplo simples de detecção de exceções e transmiti-las é mostrado no exemplo `sum_sequence()` acima. Acontece que este exemplo não precisa limpar nenhuma referência de propriedade quando detecta um erro. A função de exemplo a seguir mostra alguma limpeza de erro. Primeiro, para lembrar por que você gosta de Python, mostramos o código Python equivalente:

```
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;
    }
    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)
```

(continua na próxima página)

```

    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()` traria 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, ela deve ser definida explicitamente com uma chamada com `PySys_SetArgvEx(argc, argv, updatepath)` após a chamada de `Py_Initialize()`.

Na maioria dos sistemas (em particular, no Unix e no Windows, embora os detalhes sejam ligeiramente diferentes), `Py_Initialize()` calcula o caminho de pesquisa do módulo com base em sua melhor estimativa para a localização do executável do interpretador Python padrão, assumindo 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`.

A aplicação de incorporação pode orientar a pesquisa chamando `Py_SetProgramName(file)` antes de chamar `Py_Initialize()`. Observe que `PYTHONHOME` ainda substitui isso e `PYTHONPATH` ainda é inserido na frente do caminho padrão. Uma aplicação que requer controle total deve fornecer sua própria implementação de `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()` e `Py_GetProgramFullPath()` (todas definidas em `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 loop do interpretador principal. Apenas as compilações usadas com mais frequência serão descritas no restante desta seção.

Compilar o interpretador com a macro `Py_DEBUG` definida produz o que geralmente se entende por “uma compilação de depuração” do Python. `Py_DEBUG` é habilitada na compilação Unix adicionando `--with-pydebug` ao comando `./configure`. Também está implícito na presença da macro não específica do Python `_DEBUG`. Quando `Py_DEBUG` está habilitado na compilação do Unix, a otimização do compilador é desabilitada.

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

- Verificações extras são adicionadas ao alocador de objeto.
- Verificações extras são adicionadas ao analisador e ao compilador.
- Downcasts de tipos amplos para tipos restritos são verificados quanto à perda de informações.
- Uma série de asserções são adicionadas ao dicionário e implementações definidas. Além disso, o objeto definido adquire um método `test_c_api()`.
- As verificações de integridade dos argumentos de entrada são adicionadas à criação de quadros.
- O armazenamento para ints é inicializado com um padrão inválido conhecido para capturar a referência a dígitos não inicializados.
- O rastreamento de baixo nível e a verificação de exceções extras são adicionados à máquina virtual de tempo de execução.
- Verificações extras são adicionadas à implementação da arena de memória.
- Depuração extra é adicionada ao módulo de thread.

Pode haver verificações adicionais não mencionadas aqui.

Definir `Py_TRACE_REFS` habilita o rastreamento de referência. 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.) Implicado por `Py_DEBUG`.

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

Interface binária de aplicativo estável

Tradicionalmente, a API C do Python mudará com cada versão. A maioria das mudanças será compatível com a origem, normalmente, apenas adicionando API, ao invés de alterar a API existente ou remover API (embora algumas interfaces sejam removidas depois de primeiro se tornarem obsoletas).

Infelizmente, a compatibilidade da API não se estende à compatibilidade binária (o ABI). O motivo é principalmente a evolução das definições de estrutura, onde a adição de um novo campo, ou a alteração do tipo de um campo, pode não quebrar a API, mas pode quebrar o ABI. Como consequência, os módulos de extensão precisam ser recompilados para cada versão do Python (embora exista uma exceção no Unix quando nenhuma das interfaces afetadas é usada). Além disso, no Windows, os módulos de extensão se conectam com um `pythonXY.dll` específico e precisam ser recompilados para vincular com um novo.

Desde o Python 3.2, um subconjunto da API foi declarado para garantir um ABI estável. Os módulos de extensão que desejam usar esta API (chamada “API limitada”) precisam definir `PY_LIMITED_API`. Uma série de detalhes do interpretador ficam escondidos do módulo de extensão; em troca, um módulo é construído que funciona em qualquer versão 3.x ($x \geq 2$) sem recompilação.

Em alguns casos, a ABI estável deve ser estendida com novas funções. Os módulos de extensão que desejam usar essas novas APIs precisam definir `PY_LIMITED_API` para o valor `PY_VERSION_HEX` (veja [API e Versionamento ABI](#)) da versão mínima do Python que eles querem suportar (por exemplo, “0x03030000” para Python 3.3). Esses módulos funcionarão em todas as versões subsequentes do Python, mas não carregarão (por causa dos símbolos que faltam) nos lançamentos mais antigos.

A partir do Python 3.2, o conjunto de funções disponíveis para a API limitada está documentado em [PEP 384](#). Na documentação da API C, os elementos da API que não fazem parte da API limitada são marcados como “Não faz parte da API limitada”.

A camada de Mais Alto Nível

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

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

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

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

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

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

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

Similar to `Py_Main()` but *argv* is an array of bytes strings.

Novo na versão 3.8.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Can be set to point to a function with the prototype `int func(void)`. The function will be called when

Python's interpreter prompt is about to become idle and wait for user input from the terminal. The return value is ignored. Overriding this hook can be used to integrate the interpreter's prompt with other event loops, as done in the `Modules/_tkinter.c` in the Python source code.

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

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

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

Alterado na versão 3.4: The result must be allocated by `PyMem_RawMalloc()` or `PyMem_RawRealloc()`, instead of being allocated by `PyMem_Malloc()` or `PyMem_Realloc()`.

`struct _node* PyParser_SimpleParseString (const char *str, int start)`

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

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

`struct _node* PyParser_SimpleParseStringFlags (const char *str, int start, int flags)`

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

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

`struct _node* PyParser_SimpleParseStringFlagsFilename (const char *str, const char *filename, int start, int flags)`

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

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

`struct _node* PyParser_SimpleParseFile (FILE *fp, const char *filename, int start)`

This is a simplified interface to `PyParser_SimpleParseFileFlags()` below, leaving *flags* set to 0.

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

`struct _node* PyParser_SimpleParseFileFlags (FILE *fp, const char *filename, int start, int flags)`

Similar to `PyParser_SimpleParseStringFlagsFilename()`, but the Python source code is read from *fp* instead of an in-memory string.

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

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

Return value: New reference. This is a simplified interface to `PyRun_StringFlags()` below, leaving *flags* set to NULL.

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

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

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

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

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

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

*PyObject** **PyRun_FileExFlags** (FILE *fp, const char *filename, int start, *PyObject* *globals, *PyObject* *locals, int closeit, *PyCompilerFlags* *flags)
Return value: New reference. Similar to *PyRun_StringFlags()*, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file, it is decoded from the filesystem encoding (*sys.getfilesystemencoding()*). If *closeit* is true, the file is closed before *PyRun_FileExFlags()* returns.

*PyObject** **Py_CompileString** (const char *str, const char *filename, int start)
Return value: New reference. This is a simplified interface to *Py_CompileStringFlags()* below, leaving *flags* set to NULL.

*PyObject** **Py_CompileStringFlags** (const char *str, const char *filename, int start, *PyCompilerFlags* *flags)
Return value: New reference. This is a simplified interface to *Py_CompileStringExFlags()* below, with *optimize* set to -1.

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

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

Novo na versão 3.4.

*PyObject** **Py_CompileStringExFlags** (const char *str, const char *filename, int start, *PyCompilerFlags* *flags, int optimize)
Return value: New reference. Like *Py_CompileStringObject()*, but *filename* is a byte string decoded from the filesystem encoding (*os.fsdecode()*).

Novo na versão 3.2.

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

*PyObject** **PyEval_EvalCodeEx** (*PyObject* *co, *PyObject* *globals, *PyObject* *locals, *PyObject* *const *args, int argcount, *PyObject* *const *kws, int kwcount, *PyObject* *const *defs, int defcount, *PyObject* *kwdefs, *PyObject* *closure)
Return value: New reference. Evaluate a precompiled code object, given a particular environment for its evaluation. This environment consists of a dictionary of global variables, a mapping object of local variables, arrays of

arguments, keywords and defaults, a dictionary of default values for *keyword-only* arguments and a closure tuple of cells.

PyFrameObject

The C structure of the objects used to describe frame objects. The fields of this type are subject to change at any time.

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

Return value: *New reference.* Evaluate an execution frame. This is a simplified interface to *PyEval_EvalFrameEx()*, for backward compatibility.

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

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

Alterado na versão 3.4: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

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

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

int **Py_eval_input**

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

int **Py_file_input**

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

int **Py_single_input**

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

struct **PyCompilerFlags**

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

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

int **cf_flags**

Compiler flags.

int **cf_feature_version**

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

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

Alterado na versão 3.8: Added *cf_feature_version* field.

int **CO_FUTURE_DIVISION**

This bit can be set in *flags* to cause division operator */* to be interpreted as “true division” according to [PEP 238](#).

Contagem de Referências

As macros nesta seção são usadas para gerenciar contagens de referências de objetos Python.

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

Aumenta a contagem de referências para o objeto *o*. O objeto não deve ser NULL; se você não tem certeza de que não é NULL, use *Py_XINCREF* ().

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

Aumenta a contagem de referências para o objeto *o*. O objeto pode ser NULL, caso em que a macro não tem efeito.

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

Diminui a contagem de referências para o objeto *o*. O objeto não deve ser NULL; se você não tem certeza de que não é NULL, use *Py_XDECREF* (). Se a contagem de referências chegar a zero, a função de desalocação do tipo de objeto (que não deve ser NULL) é chamada.

Aviso: A função de desalocação pode fazer com que o código Python arbitrário seja invocado (por exemplo, quando uma instância de classe com um método `__del__()` é desalocada). Embora as exceções em tal código não sejam propagadas, o código executado tem acesso livre a todas as variáveis globais do Python. Isso significa que qualquer objeto que é alcançável de uma variável global deve estar em um estado consistente antes de *Py_DECREF* () ser invocado. Por exemplo, o código para excluir um objeto de uma lista deve copiar uma referência ao objeto excluído em uma variável temporária, atualizar a estrutura de dados da lista e então chamar *Py_DECREF* () para a variável temporária.

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

Diminui a contagem de referências para o objeto *o*. O objeto pode ser NULL, caso em que a macro não tem efeito; caso contrário, o efeito é o mesmo de *Py_DECREF* (), e o mesmo aviso se aplica.

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

Diminui a contagem de referências para o objeto *o*. O objeto pode ser NULL, caso em que a macro não tem efeito; caso contrário, o efeito é o mesmo de *Py_DECREF* (), exceto que o argumento também é definido como NULL. O aviso para *Py_DECREF* () não se aplica em relação ao objeto passado porque a macro usa cuidadosamente uma variável temporária e define o argumento como NULL antes de diminuir sua contagem de referências.

É uma boa ideia usar essa macro sempre que diminuir a contagem de referências de um objeto que pode ser percorrido durante a coleta de lixo.

As seguintes funções são para incorporação dinâmica de Python em tempo de execução: `Py_IncRef(PyObject *o)`, `Py_DecRef(PyObject *o)`. Elas são simplesmente versões de função exportadas de `Py_XINCREF()` e `Py_XDECREF()`, respectivamente.

As seguintes funções ou macros são apenas para uso dentro do núcleo do interpretador: `_Py_Dealloc()`, `_Py_ForgetReference()`, `_Py_NewReference()`, bem como a variável global `_Py_RefTotal`.

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, or `-1` se retornarem um número inteiro (exceção: as funções `PyArg_*()` retornam `1` em caso de sucesso e `0` em caso de 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** ()

Limpe o indicador de erro. Se o indicador de erro não estiver definido, não haverá efeito.

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

Print a standard traceback to `sys.stderr` and clear the error indicator. **Unless** the error is a `SystemExit`, in that case 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 variables `sys.last_type`, `sys.last_value` and `sys.last_traceback` will be set to the type, value and traceback of the printed exception, respectively.

void **PyErr_Print** ()

Alias para “`PyErr_PrintEx(1)`”.

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

Chame `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.

Uma exceção deve ser definida ao chamar essa função.

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*)

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

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

Essa função é semelhante à `PyErr_SetString()` mas permite especificar um objeto Python arbitrário para o valor da exceção.

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

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

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

Return value: Always `NULL`. Igual a `PyErr_Format()`, mas usando o argumento *va_list* em vez de um número variável de argumentos.

Novo na versão 3.5.

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

Isso é uma abreviação para `PyErr_SetObject(type, Py_None)`.

int PyErr_BadArgument ()

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

PyObject* PyErr_NoMemory ()

Return value: Always *NULL*. 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)

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

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

Return value: Always *NULL*. 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)

Return value: Always *NULL*. Similar to `PyErr_SetFromErrnoWithFilenameObject()`, but takes a second filename object, for raising errors when a function that takes two filenames fails.

Novo na versão 3.4.

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

Return value: Always *NULL*. Similar to `PyErr_SetFromErrnoWithFilenameObject()`, but the filename is given as a C string. *filename* is decoded from the filesystem encoding (`os.fsdecode()`).

PyObject* PyErr_SetFromWindowsError (int ierr)

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

Disponibilidade: Windows.

PyObject* PyErr_SetExcFromWindowsError (PyObject *type, int ierr)

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

Disponibilidade: Windows.

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

Return value: Always *NULL*. Similar to `PyErr_SetFromWindowsErrorWithFilenameObject()`, mas o nome do arquivo é dados como uma String C. O nome do arquivo é decodificado a partir do sistema de arquivos (`os.fsdecode()`).

Disponibilidade: Windows.

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

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

Disponibilidade: Windows.

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

Return value: Always NULL. Similar à `PyErr_SetExcFromWindowsErrWithFilenameObject()`, mas aceita um segundo caminho do objeto.

Disponibilidade: Windows.

Novo na versão 3.4.

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

Return value: Always NULL. 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)

Return value: Always NULL. 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.

Novo na versão 3.3.

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

Return value: Always NULL. Muito parecido com `PyErr_SetImportError()` mas a função permite especificar uma subclasse de `ImportError` para levantar uma exceção.

Novo 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`.

Novo na versão 3.4.

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

Like `PyErr_SyntaxLocationObject()`, but *filename* is a byte string decoded from the filesystem encoding (`os.fsdecode()`).

Novo na versão 3.2.

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

Like `PyErr_SyntaxLocationEx()`, but the *col_offset* parameter is omitted.

void **PyErr_BadInternalCall** ()

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

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)

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.

Novo na versão 3.4.

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

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

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

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

Novo na versão 3.2.

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

Function similar to `PyErr_WarnFormat()`, but *category* is `ResourceWarning` and it passes *source* to `warnings.WarningMessage()`.

Novo na versão 3.6.

5.4 Consultando o indicador de erro

*PyObject** **PyErr_Occurred**()

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

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*)

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

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

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

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

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

Nota: Esta função, normalmente, é usada apenas pelo código que precisa capturar exceções ou pelo código que precisa salvar e restaurar temporariamente o indicador de erro. 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*)

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

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

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

Under certain circumstances, the values returned by `PyErr_Fetch()` below can be “unnormalized”, meaning

that `*exc` is a class object but `*val` is not an instance of the same class. This function can be used to instantiate the class in that case. If the values are already normalized, nothing happens. The delayed normalization is implemented to improve performance.

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);
}
```

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

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

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.

Novo na versão 3.3.

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

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

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.

Novo na versão 3.3.

5.5 Tratamento de sinal

int **PyErr_CheckSignals** ()

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

void **PyErr_SetInterrupt** ()

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

If `SIGINT` isn't handled by Python (it was set to `signal.SIG_DFL` or `signal.SIG_IGN`), this function does nothing.

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*)

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

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

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

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

Novo na versão 3.2.

5.7 Objeto Exceção

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

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

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

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

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

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

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

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*)

Return value: *New reference.* Return the cause (either an exception instance, or None, set by `raise ... from ...`) associated with the exception as a new reference, as accessible from Python through `__cause__`.

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

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

`__suppress_context__` para essa função é definido `True`, implicitamente.

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

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

Obsoleto desde a versão 3.3: 3.11

`Py_UNICODE` is deprecated since Python 3.3. Please migrate to `PyObject_CallFunction (PyExc_UnicodeEncodeError, "sOnns", ...)`.

***PyObject** PyUnicodeTranslateError_Create** (const *Py_UNICODE* **object*, *Py_ssize_t* *length*, *Py_ssize_t* *start*, *Py_ssize_t* *end*, const char **reason*)
Return value: New reference. Create a `UnicodeTranslateError` object with the attributes *object*, *length*, *start*, *end* and *reason*. *reason* is a UTF-8 encoded string.

Obsoleto desde a versão 3.3: 3.11

`Py_UNICODE` is deprecated since Python 3.3. Please migrate to `PyObject_CallFunction (PyExc_UnicodeTranslateError, "Onns", ...)`.

***PyObject** PyUnicodeDecodeError_GetEncoding** (*PyObject* **exc*)
***PyObject** PyUnicodeEncodeError_GetEncoding** (*PyObject* **exc*)
Return value: New reference. Retorna o atributo * *encoding** dado no objeto da exceção.

***PyObject** PyUnicodeDecodeError_GetObject** (*PyObject* **exc*)
***PyObject** PyUnicodeEncodeError_GetObject** (*PyObject* **exc*)
***PyObject** PyUnicodeTranslateError_GetObject** (*PyObject* **exc*)
Return value: New reference. 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*)
 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*)
 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*)
 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)
```

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)
```

Return value: New reference. Retorna o atributo *reason* dado no objeto da exceção.

```
int PyUnicodeDecodeError_SetReason (PyObject *exc, const char *reason)
int PyUnicodeEncodeError_SetReason (PyObject *exc, const char *reason)
int PyUnicodeTranslateError_SetReason (PyObject *exc, const char *reason)
```

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

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)
```

Marca um ponto em que a chamada recursiva em nível C está prestes a ser executada.

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

The function then checks if the recursion limit is reached. If this is the case, a `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)
```

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)
```

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*)

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

Continuação na próxima página

Tabela 1 – continuação da página anterior

Nome C	Nome Python	Notas
PyExc_SyntaxError	SyntaxError	
PyExc_SystemError	SystemError	
PyExc_SystemExit	SystemExit	
PyExc_TabError	TabError	
PyExc_TimeoutError	TimeoutError	
PyExc_TypeError	TypeError	
PyExc_UnboundLocalError	UnboundLocalError	
PyExc_UnicodeDecodeError	UnicodeDecodeError	
PyExc_UnicodeEncodeError	UnicodeEncodeError	
PyExc_UnicodeError	UnicodeError	
PyExc_UnicodeTranslateError	UnicodeTranslateError	
PyExc_ValueError	ValueError	
PyExc_ZeroDivisionError	ZeroDivisionError	

Novo 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 and PyExc_TimeoutError were introduced following [PEP 3151](#).

Novo na versão 3.5: PyExc_StopAsyncIteration and PyExc_RecursionError.

Novo 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:

¹ Esta é uma classe base para outras exceções padrão.

² Define apenas no Windows; proteja o código que usa isso testando se a macro do pré-processador `MS_WINDOWS` está definida.

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	

Novo na versão 3.2: PyExc_ResourceWarning.

Notas:

³ Esta é uma classe base para outras categorias de aviso padrão.

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 análise 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)

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

Novo na versão 3.6.

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

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

void **PyOS_BeforeFork** ()

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

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

Novo na versão 3.7.

void **PyOS_AfterFork_Parent** ()

Function to update some internal state after a process fork. This should be called from the parent process after

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

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

Novo na versão 3.7.

void **PyOS_AfterFork_Child()**

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

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

Novo na versão 3.7.

Ver também:

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

void **PyOS_AfterFork()**

Função para atualizar algum estado interno após uma bifurcação 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.

Obsoleto desde a versão 3.7: This function is superseded by `PyOS_AfterFork_Child()`.

int **PyOS_CheckStack()**

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

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

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

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

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

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

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

Encoding, highest priority to lowest priority:

- UTF-8 on macOS, Android, and VxWorks;
- UTF-8 on Windows if `Py_LegacyWindowsFSEncodingFlag` is zero;
- UTF-8 if the Python UTF-8 mode is enabled;

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

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

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

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

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

Ver também:

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

Novo na versão 3.5.

Alterado na versão 3.7: The function now uses the UTF-8 encoding in the UTF-8 mode.

Alterado na versão 3.8: The function now uses the UTF-8 encoding on Windows if `Py_LegacyWindowsFSEncodingFlag` is zero;

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

Encode a wide character string to the locale encoding with the surrogateescape error handler: surrogate characters in the range U+DC80..U+DCFF are converted to bytes 0x80..0xFF.

Encoding, highest priority to lowest priority:

- UTF-8 on macOS, Android, and VxWorks;
- UTF-8 on Windows if `Py_LegacyWindowsFSEncodingFlag` is zero;
- UTF-8 if the Python UTF-8 mode is enabled;
- ASCII if the LC_CTYPE locale is "C", `nl_langinfo(CODESET)` returns the ASCII encoding (or an alias), and `mbstowcs()` and `wcstombs()` functions uses the ISO-8859-1 encoding.
- the current locale encoding.

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

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

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

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

Ver também:

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

Novo na versão 3.5.

Alterado na versão 3.7: The function now uses the UTF-8 encoding in the UTF-8 mode.

Alterado na versão 3.8: The function now uses the UTF-8 encoding on Windows if `Py_LegacyWindowsFSEncodingFlag` 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)

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

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

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

void **PySys_ResetWarnOptions** ()

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

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

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

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

Append *unicode* to `sys.warnoptions`.

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

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

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

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

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

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

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

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

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

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

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

Novo na versão 3.2.

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

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

Novo na versão 3.2.

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

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

Novo na versão 3.2.

PyObject *PySys_GetXOptions ()

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

Novo na versão 3.2.

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

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

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

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

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

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

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

Levanta um evento de auditoria `sys.addaudithook` com nenhum argumento.

Novo na versão 3.8.

6.3 Process Control

void Py_FatalError (const char *message)

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

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*)

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*)())

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

6.4 Importando módulos

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

Return value: *New reference.* Esta é uma interface simplificada para *PyImport_ImportModuleEx()* abaixo, deixando os argumentos *globals* e *locals* definidos como NULL e *level* definido como 0. Quando o argumento *name* contém um caractere de ponto (quando especifica um submódulo de um pacote), o argumento *fromlist* é definido para a lista ['*'] de modo que o valor de retorno é o módulo nomeado em vez do pacote de nível superior que o contém como faria caso contrário, seja o caso. (Infelizmente, isso tem um efeito colateral adicional quando *name* de fato especifica um subpacote em vez de um submódulo: os submódulos especificados na variável `__all__` do pacote são carregados.) Retorna uma nova referência ao módulo importado, ou NULL com uma exceção definida em caso de falha. Uma importação com falha de um módulo não deixa o módulo em `sys.modules`.

Esta função sempre usa importações absolutas.

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

Return value: *New reference.* Esta função é um alias descontinuado de *PyImport_ImportModule()*.

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

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

Return value: *New reference.* 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*)

Return value: *New reference.* 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.

Novo na versão 3.3.

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

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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_AddModuleObject** (*PyObject* *name)

Return value: Borrowed reference. Retorna o objeto módulo correspondente a um nome de módulo. O argumento *name* pode ter a forma `package.module`. Primeiro verifica o dicionário de módulos se houver algum, caso contrário, cria um novo e insere-o no dicionário de módulos. Retorna NULL com uma exceção definida em caso de falha.

Nota: Esta função não carrega ou importa o módulo; se o módulo não foi carregado, você receberá um objeto de módulo vazio. Use `PyImport_ImportModule()` ou uma de suas variações para importar um módulo. Estruturas de pacotes implícitos por um nome pontilhado para a *name* não são criados se não estiverem presentes.

Novo na versão 3.3.

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

Return value: Borrowed reference. Semelhante para `PyImport_AddModuleObject()`, mas o nome é uma string codificada em UTF-8 em vez de um objeto Unicode.

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

Return value: New reference. Dado um nome de módulo (possivelmente na forma `package.module`) e um objeto código lido de um arquivo de bytecode Python ou obtido da função embutida `compile()`, carrega o módulo. Retorna uma nova referência ao objeto do módulo, ou NULL com uma exceção definida se ocorrer um erro. *name* é removido de `sys.modules` em casos de erro, mesmo se *name* já estivesse em `sys.modules` na entrada para `PyImport_ExecCodeModule()`. Deixar módulos incompletamente inicializados em `sys.modules` é perigoso, pois as importações de tais módulos não têm como saber se o objeto módulo é um estado desconhecido (e provavelmente danificado em relação às intenções do autor do módulo).

O `__spec__` e o `__loader__` do módulo serão definidos, se não estiverem, com os valores apropriados. O carregador do spec será definido para o `__loader__` do módulo (se definido) e para uma instância da classe `SourceFileLoader` em caso contrário.

O atributo `__file__` do módulo será definido para o `co_filename` do objeto código. Se aplicável, `__cached__` também será definido.

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()`.

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

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

Veja também *PyImport_ExecCodeModuleWithPathnames()*.

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

Return value: New reference. Como *PyImport_ExecCodeModuleEx()*, mas o atributo `__cached__` do objeto módulo é definido como *cpathname* se não for NULL. Das três funções, esta é a preferida para usar.

Novo na versão 3.3.

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

Return value: New reference. 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.

Novo na versão 3.2.

Alterado na versão 3.3: Usa `imp.source_from_cache()` no cálculo do caminho de origem se apenas o caminho do bytecode for fornecido.

long **PyImport_GetMagicNumber** ()

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** ()

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.

Novo na versão 3.2.

*PyObject** **PyImport_GetModuleDict** ()

Return value: Borrowed reference. 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)

Return value: New reference. 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.

Novo na versão 3.7.

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

Return value: New reference. Retorna um objeto localizador para o item *path* de `sys.path/pkg.__path__`, possivelmente obtendo-o do dicionário `sys.path_importer_cache`. Se ainda não foi armazenado em cache, atravessa `sys.path_hooks` até que um gancho seja encontrado que possa lidar com o item de caminho. Retorna `None` se nenhum gancho puder; isso diz ao nosso chamador que o *localizador baseado no caminho* não conseguiu encontrar um localizador para este item de caminho. Armazena o resultado em `sys.path_importer_cache`. Retorna uma nova referência ao objeto localizador.

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

Return value: New reference. 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.)

Novo na versão 3.3.

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

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

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;
};
```

const struct **_frozen*** **PyImport_FrozenModules**

Este ponteiro é inicializado para apontar para um vetor de registros de `struct _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))

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**

Estrutura que descreve uma única entrada na lista de módulos embutidos. Cada uma dessas estruturas fornece o nome e a função de inicialização para um módulo embutido ao interpretador. O nome é uma string codificada em ASCII. Os programas que embutem Python podem usar um vetor dessas estruturas em conjunto com `PyImport_ExtendInittab()` para fornecer módulos embutidos adicionais. A estrutura é definida em `Include/import.h` como:

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

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

Adiciona uma coleção de módulos à tabela de módulos embutidos. O vetor *newtab* deve terminar com uma entrada sentinela que contém NULL para o campo *name*; a falha em fornecer o valor sentinela pode resultar em uma falha de memória. Retorna 0 em caso de sucesso ou -1 se memória insuficiente puder ser alocada para estender a tabela interna. Em caso de falha, nenhum módulo é adicionado à tabela interna. Deve ser chamado antes de `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 possui suporte a duas versões do formato de dados: a versão 0 é a versão histórica, a versão 1 compartilha sequências de caracteres 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.

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

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

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

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

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

Return value: New reference. 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*)

Return value: New reference. 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*)

Return value: New reference. 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 ele 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*)

Return value: New reference. 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 funções e métodos das suas extensões. 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 and buffers

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.

Em geral, quando um formato define um ponteiro para um buffer, o buffer é gerenciado pelo objeto Python correspondente e o buffer compartilha o tempo de vida desse objeto. Você não terá que liberar nenhuma memória por conta própria. As únicas exceções são `es`, `es#`, `et` e `et#`.

No entanto, quando uma estrutura de `Py_buffer` é preenchida, o buffer subjacente é bloqueado para que o chamador possa posteriormente usar o buffer mesmo dentro de um bloco `Py_BEGIN_ALLOW_THREADS` sem o risco de dados mutáveis serem redimensionados ou destruídos. Como resultado, **você precisa chamar `PyBuffer_Release()`** depois de ter concluído o processamento dos dados (ou em algum caso de interrupção precoce).

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

Alguns formatos requerem um *objeto byte ou similar* somente leitura e definem um ponteiro em vez de uma estrutura de buffer. Eles trabalham verificando se o campo `PyBufferProcs.bf_releasebuffer` do objeto é NULL, o que não permite objetos mutáveis, como `bytearray`.

Nota: Para todas as variantes # de formatos (`s#`, `y#` etc.), o tipo do argumento comprimento (`int` ou `Py_ssize_t`) é controlado definindo a macro `PY_SSIZE_T_CLEAN` antes de incluir `Python.h`. Se a macro foi definida, o comprimento é um `Py_ssize_t` em vez de um `int`. Este comportamento irá mudar em uma versão futura do Python para suportar apenas `Py_ssize_t` e descartar suporte a `int`. É melhor sempre definir `PY_SSIZE_T_CLEAN`.

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 em 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 ***, **int** ou `Py_ssize_t`] Como `s*`, exceto que não aceita objetos mutáveis. 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 ***, **int** ou `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; 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 ***, **int** ou `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**) [**const Py_UNICODE ***] Converte um objeto Python Unicode para um ponteiro C para um buffer de caracteres Unicode terminado em NUL. Você deve passar o endereço de uma variável ponteiro `Py_UNICODE`, que será preenchida com um ponteiro para um buffer Unicode existente. Por favor, note que o comprimento de um caractere `Py_UNICODE` depende das opções de compilação (está entre 16 ou 32 bits). A string Python não deve conter pontos de código nulos incorporados, 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.

Deprecated since version 3.3, will be removed in version 3.12: Parte do estilo antigo `Py_UNICODE` API; por favor migre o uso para `PyUnicode_AsWideCharString()`.

u# (**str**) [**const Py_UNICODE ***, **int** ou `Py_ssize_t`] Esta variante de `u` armazena em duas variáveis C, a primeira um ponteiro para um buffer Unicode de dados, a segunda para seu comprimento. Esta variante permite ponteiros para nulos.

Deprecated since version 3.3, will be removed in version 3.12: Parte do estilo antigo `Py_UNICODE` API; por favor migre o uso para `PyUnicode_AsWideCharString()`.

Z (**str** ou `None`) [**const Py_UNICODE ***] Como `u`, mas o objeto Python também pode ser `None`, nesse caso o ponteiro `Py_UNICODE` é definido como `NULL`.

Deprecated since version 3.3, will be removed in version 3.12: Parte do estilo antigo `Py_UNICODE` API; por favor migre o uso para `PyUnicode_AsWideCharString()`.

z# (str ou None) [const Py_UNICODE *, int ou Py_ssize_t] Como `u#`, mas o objeto Python também pode ser `None`, nesse caso o ponteiro `Py_UNICODE` é definido como `NULL`

Deprecated since version 3.3, will be removed in version 3.12: Parte do estilo antigo `Py_UNICODE` API; por favor migre o uso para `PyUnicode_AsWideCharString()`.

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 a `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 de cadeia de bytes são passados sem os recodificar. Em vez disso, a implementação assume que o objeto de cadeia de bytes usa a codificação passada como parâmetro.

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

Números

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

B (int) [unsigned char] Converte um inteiro Python para um pequeno inteiro (tiny int) 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. A contagem de referências do objeto 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, anything] 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.

Alterado na versão 3.1: 109 `Py_CLEANUP_SUPPORTED` foi adicionado.

p (bool) [int] Testa o valor transmitido para a verdade (um booleano *predicado*) 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.

Novo 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.

É possível passar inteiros “long” (inteiros em que o valor excede a constante da plataforma `LONG_MAX`) contudo nenhuma checagem de intervalo é propriamente feita — os bits mais significativos são silenciosamente truncados quando o campo de recebimento é muito pequeno para receber o valor (na verdade, a semântica é herdada de downcasts no C — seu raio de ação pode variar).

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).

“{TX-PL-LABEL}#x60;“ `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.

Novo 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 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, ...)

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 vars)

Idêntico a *PyArg_ParseTuple()*, exceto que aceita uma *va_list* ao invés de um número variável de argumentos.

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

Analisa os parâmetros de uma função que recebe ambos parâmetros posicionais e de palavra reservada em variáveis locais. O argumento *keywords* é um vetor terminado por NULL de nomes de parâmetros de palavra reservada. Nomes vazios denotam *positional-only parameters*. Retorna verdadeiro em caso de sucesso; em caso de falha, retorna falso e levanta a exceção apropriada.

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

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

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

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

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.

Novo na versão 3.2.

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

Função usada para desconstruir as listas de argumento de funções “old-style” — estas são funções que usam o método de análise de parâmetro METH_OLDARGS, que foi removido no Python 3. Isso não é recomendado para uso de análise de parâmetro em código novo, e a maior parte do código no interpretador padrão foi modificada para não usar mais isso para esse propósito. Ela continua um modo conveniente de decompor outras tuplas, contudo, e pode continuar a ser usada para esse propósito.

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

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

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

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

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

(continua na próxima página)

(continuação da página anterior)

```

    }
    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)
```

6.6.2 Construindo valores

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

Return value: New reference. Cria um novo valor baseado em uma string de formatação similar às 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 *, int ou 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 *, int ou 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 *, int ou 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 *, int ou 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 *, int ou *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 (exceto por sua contagem de referências, que é incrementada por um). Se o objeto passado é um ponteiro `NULL`, assume-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 incrementa a contagem de referências do objeto. 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 args)

Return value: New reference. 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, ...)

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)

Saída não superior a *size* bytes para *str* de acordo com o formato string *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* e *format != NULL*.

Se a plataforma não tiver *vsnprintf()* e o tamanho do buffer necessário para evitar o truncamento exceder *size* em mais de 512 bytes, o Python aborta com um *Py_FatalError()*.

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.

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

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; levantar essa exceção e retornar `-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`.

Novo na versão 3.1.

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

Converte um `double` *val* 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()`.

Novo 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 o caso.

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 o caso.

6.8 Reflexão

*PyObject** **PyEval_GetBuiltins** (void)

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

*PyObject** **PyEval_GetLocals** (void)

Return value: Borrowed reference. Retorna um dicionário das variáveis locais no quadro de execução atual ou `NULL` se nenhum quadro estiver sendo executado no momento.

*PyObject** **PyEval_GetGlobals** (void)

Return value: Borrowed reference. 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)

Return value: Borrowed reference. Retorna o quadro do estado atual da thread, que é `NULL` se nenhum quadro estiver em execução no momento.

Vea também `PyThreadState_GetFrame()`.

*PyFrameObject** **PyFrame_GetBack** (*PyFrameObject* **frame*)

Obtém o *frame* próximo ao quadro externo.

Retorna uma referência forte, ou `NULL` se *frame* não tiver quadro externo.

frame não deve ser NULL.

Novo na versão 3.9.

*PyCodeObject** **PyFrame_GetCode** (*PyFrameObject* **frame*)

Obtém o código de *frame*.

Retorna uma referência forte.

frame não deve ser NULL. O resultado (código do quadro) não pode ser NULL.

Novo na versão 3.9.

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

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

frame não deve ser NULL.

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

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

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

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

6.9 Registro de codec e funções de suporte

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

Registra uma nova função de busca de codec.

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

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

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*)

Return value: New reference. 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*)

Return value: New reference. 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.

6.9.1 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*)

Return value: *New reference.* Obtém uma função de codificação para o *encoding* dado.

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

Return value: *New reference.* Obtém uma função de decodificação para o *encoding* dado.

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

Return value: *New reference.* Obtém um objeto `IncrementalEncoder` para o *encoding* dado.

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

Return value: *New reference.* Obtém um objeto `IncrementalDecoder` para o *encoding* dado.

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

Return value: *New reference.* Obtém uma função de fábrica `StreamReader` para o *encoding* dado.

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

Return value: *New reference.* Obtém uma função de fábrica `StreamWriter` para o *encoding* dado.

6.9.2 API de registro de tratamentos de erros de decodificação Unicode

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

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

O retorno de chamada obtém um único argumento, uma instância de `UnicodeEncodeError`, `UnicodeDecodeError` ou `UnicodeTranslateError` que contém informações sobre a sequência 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*)

Return value: *New reference.* 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*)

Return value: *Always NULL.* Levanta *exc* como uma exceção.

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

Return value: *New reference.* Ignora o erro de unicode, ignorando a entrada que causou o erro.

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

Return value: *New reference.* Substitui o erro de unicode por ? ou U+FFFD.

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

Return value: *New reference.* Substitui o erro de unicode por caracteres da referência XML.

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

Return value: *New reference.* Substitui o erro de unicode com escapes de barra invertida (`\x`, `\u` e `\U`).

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

Return value: *New reference.* Substitui os erros de codificação unicode com escapes `\N{ . . . }`.

Novo na versão 3.5.

Camada de Objetos Abstratos

As funções neste capítulo interagem com os objetos do Python independentemente do tipo deles ou com classes amplas dos tipos de objetos (por exemplo, todos os tipos numéricos ou todos os tipos de sequência). Quando usado nos tipos de objetos pros quais eles não se aplicam eles criarão uma exceção no Python.

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

7.1 Protocolo de objeto

*PyObject** `Py_NotImplemented`

O singleton `NotImplemented`, usado para sinalizar que uma operação não foi implementada para a combinação de tipo fornecida.

`Py_RETURN_NOTIMPLEMENTED`

Trata corretamente o retorno de *Py_NotImplemented* de dentro de uma função C (ou seja, incrementa a contagem de referências de `NotImplemented` e retorna-a).

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

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

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

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

Note that exceptions which occur while calling `__getattr__()` and `__getattribute__()` methods will get suppressed. To get error reporting use *PyObject_GetAttr()* instead.

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

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

Note that exceptions which occur while calling `__getattr__()` and `__getattribute__()` methods and creating a temporary string object will get suppressed. To get error reporting use `PyObject_GetAttrString()` instead.

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

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

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

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

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

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

int **PyObject_SetAttr** (*PyObject* **o*, *PyObject* **attr_name*, *PyObject* **v*)

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

If *v* is NULL, the attribute is deleted. 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*)

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, but this feature is deprecated in favour of using `PyObject_DelAttrString()`.

int **PyObject_GenericSetAttr** (*PyObject* **o*, *PyObject* **name*, *PyObject* **value*)

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

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

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

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

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

*PyObject** **PyObject_GenericGetDict** (*PyObject* **o*, void **context*)

Return value: *New reference.* A generic implementation for the getter of a `__dict__` descriptor. It creates the dictionary if necessary.

Novo na versão 3.3.

int **PyObject_GenericSetDict** (*PyObject* **o*, *PyObject* **value*, void **context*)

A generic implementation for the setter of a `__dict__` descriptor. This implementation does not allow the dictionary to be deleted.

Novo na versão 3.3.

*PyObject** **PyObject_RichCompare** (*PyObject* **o1*, *PyObject* **o2*, int *opid*)

Return value: *New reference.* Compare the values of *o1* and *o2* using the operation specified by *opid*, which must be one of `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, or `Py_GE`, corresponding to `<`, `<=`, `==`, `!=`, `>`, or `>=`.

respectively. This is the equivalent of the Python expression `o1 op o2`, where `op` is the operator corresponding to *opid*. Returns the value of the comparison on success, or NULL on failure.

int PyObject_RichCompareBool (*PyObject* *o1, *PyObject* *o2, int opid)

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

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

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

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

Alterado na versão 3.4: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

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

Return value: *New reference.* 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)

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

Alterado na versão 3.4: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

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

Return value: *New reference.* 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)

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)

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*)

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*)

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*)

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

int PyObject_Not (*PyObject* **o*)

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

PyObject* PyObject_Type (*PyObject* **o*)

Return value: New reference. When *o* is non-NULL, returns a type object corresponding to the object type of object *o*. On failure, raises `SystemError` and returns NULL. This is equivalent to the Python expression `type(o)`. This function increments the reference count of the return value. There's really no reason to use this function instead of the `Py_TYPE()` function, which returns a pointer of type `PyTypeObject*`, except when the incremented reference count is needed.

int PyObject_TypeCheck (*PyObject* **o*, *PyTypeObject* **type*)

Return true if the object *o* is of type *type* or a subtype of *type*. Both parameters must be non-NULL.

Py_ssize_t PyObject_Size (*PyObject* **o*)

Py_ssize_t PyObject_Length (*PyObject* **o*)

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

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)`.

Novo na versão 3.4.

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

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

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

Map the object *key* to the value *v*. Raise an exception and return `-1` on failure; return 0 on success. This is the equivalent of the Python statement `o[key] = v`. This function *does not* steal a reference to *v*.

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

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

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

Return value: New reference. This is equivalent to the Python expression `dir(o)`, returning a (possibly empty)

list of strings appropriate for the object argument, or `NULL` if there was an error. If the argument is `NULL`, this is like the Python `dir()`, returning the names of the current locals; in this case, if no execution frame is active then `NULL` is returned but `PyErr_Occurred()` will return false.

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

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

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)` em 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.

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

7.2.2 O protocolo `vectorcall`

Novo 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 invocável suportar. Entretanto, isso não é uma regra rígida. Ademais, algumas 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 invocável precisa se comportar da mesma forma independente 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.

Uma classe não deve implementar chamadas de vetores 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 chamada de vetor.

Classes podem implementar o protocolo de chamada de vetor ativando a flag `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:

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

- *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-chave;** 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-chave, então *kwnames* deve então ser *NULL*.

PY_VECTORCALL_ARGUMENTS_OFFSET

Se essa flag é definida em um argumento de chamada de vetor *nargsf*, 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.

Para invocar um objeto que implementa vectorcall, utilize a função *call API* como qualquer outra invocável. `PyObject_Vectorcall()` será normalmente mais eficiente.

Nota: No CPython 3.8, a API vectorcall e funções relacionadas estavam disponíveis provisoriamente sob nomes com um sublinhado inicial: `_PyObject_Vectorcall`, `_Py_TPFLAGS_HAVE_VECTORCALL`, `_PyObject_VectorcallMethod`, `_PyVectorcall_Function`, `_PyObject_CallOneArg`, `_PyObject_CallMethodNoArgs`, `_PyObject_CallMethodOneArg`. Além disso, `PyObject_VectorcallDict` estava disponível como `_PyObject_FastCallDict`. Os nomes antigos ainda estão definidos como apelidos para os novos nomes sem o sublinhado.

Controle de recursão

Quando utilizando *tp_call*, invocadores não precisam se preocupar sobre *recursion*:CPython usar `Py_EnterRecursiveCall` e `:c:func:`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*)

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.

This function is not part of the *limited API*.

Novo 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`.

This function is not part of the *limited API*.

Novo na versão 3.8.

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

Chame 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 checa a flag *Py_TPFLAGS_HAVE_VECTORCALL* e não retorna para *tp_call*.

This function is not part of the *limited API*.

Novo na versão 3.8.

7.2.3 API de chamada de objetos

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

A tabela a seguir sumariza as funções disponíveis; por favor, veja a documentação individual para detalhes.

Função	chamável	args	kwargs
<i>PyObject_Call()</i>	<i>PyObject</i> *	tupla	dict/NULL
<i>PyObject_CallNoArgs()</i>	<i>PyObject</i> *	—	—
<i>PyObject_CallOneArg()</i>	<i>PyObject</i> *	1 objeto	—
<i>PyObject_CallObject()</i>	<i>PyObject</i> *	tupla/NULL	—
<i>PyObject_CallFunction()</i>	<i>PyObject</i> *	formato	—
<i>PyObject_CallMethod()</i>	obj + char*	formato	—
<i>PyObject_CallFunctionObjArgs()</i>	<i>PyObject</i> *	variádica	—
<i>PyObject_CallMethodObjArgs()</i>	obj + nome	variádica	—
<i>PyObject_CallMethodNoArgs()</i>	obj + nome	—	—
<i>PyObject_CallMethodOneArg()</i>	obj + nome	1 objeto	—
<i>PyObject_Vectorcall()</i>	<i>PyObject</i> *	vectorcall	vectorcall
<i>PyObject_VectorcallDict()</i>	<i>PyObject</i> *	vectorcall	dict/NULL
<i>PyObject_VectorcallMethod()</i>	arg + nome	vectorcall	vectorcall

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

Return value: *New reference*. Chame 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*.

Retorne o resultado da chamada em sucesso, ou levante uma exceção e retorne *NULL* em caso de falha.

Esse é o equivalente da expressão Python: `callable(*args, **kwargs)`.

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

Chame um objeto Python chamável de *callable* sem nenhum argumento. É o jeito mais eficiente de chamar um objeto Python sem nenhum argumento.

Retorne o resultado da chamada em sucesso, ou levante uma exceção e retorne *NULL* em caso de falha.

Novo na versão 3.9.

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

Chame um objeto Python chamável de *callable* com exatamente 1 argumento posicional *arg* e nenhum argumento nomeado.

Retorne o resultado da chamada em sucesso, ou levante uma exceção e retorne *NULL* em caso de falha.

This function is not part of the *limited API*.

Novo na versão 3.9.

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

Return value: New reference. Chame um objeto Python chamável de *callable* com argumentos dados pela tupla *args*. Se nenhum argumento é necessário, *args* pode ser *NULL*.

Retorne o resultado da chamada em sucesso, ou levante uma exceção e retorne *NULL* em caso de falha.

Este é o equivalente da expressão Python: `callable(*args)`.

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

Return value: New reference. Chame 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.

Retorne o resultado da chamada em sucesso, ou levante uma exceção e retorne *NULL* em caso de falha.

Este é o equivalente da expressão Python: `callable(*args)`.

Note that if you only pass *PyObject* *args, *PyObject_CallFunctionObjArgs()* is a faster alternative.

Alterado na versão 3.4: O tipo de *format* foi mudado de `char *`.

*PyObject** **PyObject_CallMethod** (*PyObject* *obj, const char *name, const char *format, ...)

Return value: New reference. Chame 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.

Retorne o resultado da chamada em sucesso, ou levante uma exceção e retorne *NULL* em caso de falha.

Este é o equivalente da expressão Python: `obj.name(arg1, arg2, ...)`.

Note that if you only pass *PyObject* *args, *PyObject_CallMethodObjArgs()* is a faster alternative.

Alterado na versão 3.4: Os tipos de *name* e *format* foram mudados de `char *`.

*PyObject** **PyObject_CallFunctionObjArgs** (*PyObject* *callable, ...)

Return value: New reference. Call a callable Python object *callable*, with a variable number of *PyObject* * arguments. The arguments are provided as a variable number of parameters followed by *NULL*.

Retorne o resultado da chamada em sucesso, ou levante uma exceção e retorne *NULL* em caso de falha.

Este é o equivalente da expressão Python: `callable(arg1, arg2, ...)`.

*PyObject** **PyObject_CallMethodObjArgs** (*PyObject* *obj, *PyObject* *name, ...)

Return value: New reference. Call a method of the Python object *obj*, where the name of the method is given as a Python string object in *name*. It is called with a variable number of *PyObject* * arguments. The arguments are provided as a variable number of parameters followed by *NULL*.

Retorne o resultado da chamada em sucesso, ou levante uma exceção e retorne *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*.

Retorne o resultado da chamada em sucesso, ou levante uma exceção e retorne *NULL* em caso de falha.

This function is not part of the *limited API*.

Novo 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*.

Retorne o resultado da chamada em sucesso, ou levante uma exceção e retorne *NULL* em caso de falha.

This function is not part of the *limited API*.

Novo na versão 3.9.

*PyObject** **PyObject_Vectorcall** (*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kw-names)

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*.

Retorne o resultado da chamada em sucesso, ou levante uma exceção e retorne *NULL* em caso de falha.

This function is not part of the *limited API*.

Novo 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.

This function is not part of the *limited API*.

Novo na versão 3.9.

*PyObject** **PyObject_VectorcallMethod** (*PyObject* *name, *PyObject* *const *args, size_t nargsf, *PyObject* *kw-names)

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 a feature *Py_TPFLAGS_METHOD_DESCRIPTOR*, isso irá chamar o objeto de método não vinculado com o vetor *args* inteiro como argumentos.

Retorne o resultado da chamada em sucesso, ou levante uma exceção e retorne *NULL* em caso de falha.

This function is not part of the *limited API*.

Novo na versão 3.9.

7.2.4 API de suporte a chamadas

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

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)

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)

Return value: *New reference.* 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)

Return value: *New reference.* 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)

Return value: *New reference.* 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)

Return value: *New reference.* 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`.

Novo na versão 3.5.

*PyObject** **PyNumber_FloorDivide** (*PyObject* *o1, *PyObject* *o2)

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

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

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

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

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

*PyObject** **PyNumber_Divmod** (*PyObject* *o1, *PyObject* *o2)

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

*PyObject** **PyNumber_Power** (*PyObject* *o1, *PyObject* *o2, *PyObject* *o3)

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

*PyObject** **PyNumber_Negative** (*PyObject* *o)

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

*PyObject** **PyNumber_Positive** (*PyObject* **o*)

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

*PyObject** **PyNumber_Absolute** (*PyObject* **o*)

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

*PyObject** **PyNumber_Invert** (*PyObject* **o*)

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

*PyObject** **PyNumber_Lshift** (*PyObject* **o1*, *PyObject* **o2*)

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

*PyObject** **PyNumber_Rshift** (*PyObject* **o1*, *PyObject* **o2*)

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

*PyObject** **PyNumber_And** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Returns the “bitwise and” of *o1* and *o2* on success and NULL on failure. This is the equivalent of the Python expression `o1 & o2`.

*PyObject** **PyNumber_Xor** (*PyObject* **o1*, *PyObject* **o2*)

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

*PyObject** **PyNumber_Or** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* Returns the “bitwise or” of *o1* and *o2* on success, or NULL on failure. This is the equivalent of the Python expression `o1 | o2`.

*PyObject** **PyNumber_InPlaceAdd** (*PyObject* **o1*, *PyObject* **o2*)

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

*PyObject** **PyNumber_InPlaceSubtract** (*PyObject* **o1*, *PyObject* **o2*)

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

*PyObject** **PyNumber_InPlaceMultiply** (*PyObject* **o1*, *PyObject* **o2*)

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

*PyObject** **PyNumber_InPlaceMatrixMultiply** (*PyObject* **o1*, *PyObject* **o2*)

Return value: *New reference.* 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`.

Novo na versão 3.5.

*PyObject** **PyNumber_InPlaceFloorDivide** (*PyObject* **o1*, *PyObject* **o2*)

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

*PyObject** **PyNumber_InPlaceTrueDivide** (*PyObject* **o1*, *PyObject* **o2*)

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

*PyObject** **PyNumber_InPlaceRemainder** (*PyObject* **o1*, *PyObject* **o2*)

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

*PyObject** **PyNumber_InPlacePower** (*PyObject* **o1*, *PyObject* **o2*, *PyObject* **o3*)

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

*PyObject** **PyNumber_InPlaceLshift** (*PyObject* **o1*, *PyObject* **o2*)

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

*PyObject** **PyNumber_InPlaceRshift** (*PyObject* **o1*, *PyObject* **o2*)

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

*PyObject** **PyNumber_InPlaceAnd** (*PyObject* **o1*, *PyObject* **o2*)

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

*PyObject** **PyNumber_InPlaceXor** (*PyObject* **o1*, *PyObject* **o2*)

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

*PyObject** **PyNumber_InPlaceOr** (*PyObject* **o1*, *PyObject* **o2*)

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

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

Return value: *New reference.* 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*)

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

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

Return value: *New reference.* Returns the *o* converted to a Python int on success or NULL with a `TypeError` exception raised on failure.

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

Return value: *New reference.* 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*)

Returns *o* converted to a `Py_ssize_t` value if *o* can be interpreted as an integer. If the call fails, an exception is raised and `-1` is returned.

If *o* can be converted to a Python int but the attempt to convert to a `Py_ssize_t` value would raise an `OverflowError`, then the *exc* argument is the type of exception that will be raised (usually `IndexError` or `OverflowError`). If *exc* is NULL, then the exception is cleared and the value is clipped to `PY_SSIZE_T_MIN` for a negative integer or `PY_SSIZE_T_MAX` for a positive integer.

int **PyIndex_Check** (*PyObject* **o*)

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

7.4 Protocolo de sequência

int PySequence_Check (*PyObject* *o)

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

Py_ssize_t **PySequence_Size** (*PyObject* *o)

Py_ssize_t **PySequence_Length** (*PyObject* *o)

Retorna o número de objetos em sequência *o* em caso de sucesso e `-1` em caso de falha. Isso é equivalente à expressão Python `len(o)`.

*PyObject** **PySequence_Concat** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. Retorna a concatenação de *o1* e *o2* em caso de sucesso, e `NULL` em caso de falha. Este é o equivalente da expressão Python `o1 + o2`.

*PyObject** **PySequence_Repeat** (*PyObject* *o, *Py_ssize_t* count)

Return value: New reference. Retorna o resultado da repetição do objeto sequência *o* *count* vezes ou `NULL` em caso de falha. Este é o equivalente da expressão Python `o * count`.

*PyObject** **PySequence_InPlaceConcat** (*PyObject* *o1, *PyObject* *o2)

Return value: New reference. Retorna a concatenação de *o1* e *o2* em caso de sucesso, e `NULL` em caso de falha. A operação é feita *no local* quando *o1* suportar. Este é o equivalente da expressão Python `o1 += o2`.

*PyObject** **PySequence_InPlaceRepeat** (*PyObject* *o, *Py_ssize_t* count)

Return value: New reference. Retorna o resultado da repetição do objeto sequência *o* *count* vezes ou `NULL` em caso de falha. A operação é feita *localmente* quando *o* suportar. Este é o equivalente da expressão Python `o *= count`.

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

Return value: New reference. Retorna o elemento *i* de *o* ou `NULL` em caso de falha. Este é o equivalente da expressão Python `o[i]`.

*PyObject** **PySequence_GetSlice** (*PyObject* *o, *Py_ssize_t* i1, *Py_ssize_t* i2)

Return value: New reference. Retorna a fatia do objeto sequência *o* entre *i1* e *i2*, ou `NULL` em caso de falha. Este é o equivalente da expressão Python `o[i1:i2]`.

int PySequence_SetItem (*PyObject* *o, *Py_ssize_t* i, *PyObject* *v)

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. Isso é equivalente à instrução Python `o[i]=v`. Esta função *não* rouba uma referência a *v*.

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

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

Exclui o elemento *i* do objeto *o*. Retorna `-1` em caso de falha. Isso é equivalente à instrução Python `del o[i]`.

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

Atribui o objeto sequência *v* à fatia no objeto sequência *o* de *i1* a *i2*. Isso é equivalente à instrução Python `o[i1:i2] = v`.

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

Exclui a fatia no objeto sequência *o* de *i1* a *i2*. Retorna `-1` em caso de falha. Isso é equivalente à instrução Python `del o[i1:i2]`.

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

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

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

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

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

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

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

Return value: New reference. Return a list object with the same contents as the sequence or iterable *o*, or NULL on failure. The returned list is guaranteed to be new. This is equivalent to the Python expression `list(o)`.

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

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

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

Return value: New reference. Return the sequence or iterable *o* as an object usable by the other `PySequence_Fast*` family of functions. If the object is not a sequence or iterable, raises `TypeError` with *m* as the message text. Returns NULL on failure.

The `PySequence_Fast*` functions are thus named because they assume *o* is a `PyTupleObject` or a `PyListObject` and access the data fields of *o* directly.

As a CPython implementation detail, if *o* is already a sequence or list, it will be returned.

Py_ssize_t **PySequence_Fast_GET_SIZE** (*PyObject* **o*)

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

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

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

*PyObject*** **PySequence_Fast_ITEMS** (*PyObject* **o*)

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

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

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

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

7.5 Protocolo de mapeamento

Veja também `PyObject_GetItem()`, `PyObject_SetItem()` e `PyObject_DelItem()`.

int PyMapping_Check (*PyObject* **o*)

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

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

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

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*)

Return value: New reference. Retorna o elemento de *o* correspondente à string *key* ou `NULL` em caso de falha. Este é o equivalente da expressão Python `o[key]`. Veja também *PyObject_GetItem()*.

int **PyMapping_SetItemString** (*PyObject* **o*, const char **key*, *PyObject* **v*)

Mapeia a string *key* para o valor *v* no objeto *o*. Retorna `-1` em caso de falha. Este é o equivalente da instrução Python `o[key] = v`. Veja também *PyObject_SetItem()*. Esta função *não* rouba uma referência a *v*.

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

Remove o mapeamento para o objeto *key* do objeto *o*. Retorna `-1` em caso de falha. Isso é equivalente à instrução Python `del o[key]`. Este é um alias de *PyObject_DelItem()*.

int **PyMapping_DelItemString** (*PyObject* **o*, const char **key*)

Remove o mapeamento para a string *key* do objeto *o*. Retorna `-1` em caso de falha. Isso é equivalente à instrução Python `del o[key]`.

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

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.

Observe que as exceções que ocorrem ao chamar o método `__getitem__()` serão suprimidas. Para obter relatórios de erros, use *PyObject_GetItem()*.

int **PyMapping_HasKeyString** (*PyObject* **o*, const char **key*)

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.

Observe que as exceções que ocorrem ao chamar o método `__getitem__()` e criar um objeto string temporário serão suprimidas. Para obter relatórios de erros, use *PyMapping_GetItemString()*.

*PyObject** **PyMapping_Keys** (*PyObject* **o*)

Return value: New reference. 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*)

Return value: New reference. 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*)

Return value: New reference. 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 específicas para trabalhar com iteradores.

int **PyIter_Check** (*PyObject* *o)

Retorna verdadeiro se o objeto *o* tiver suporte ao protocolo iterador. Esta função sempre tem sucesso.

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

Return value: New reference. Retorna o próximo valor da iteração *o*. O objeto deve ser um iterador (cabe ao chamador verificar isso). Se não houver valores restantes, retorna NULL sem nenhuma exceção. Se ocorrer um erro ao recuperar o item, retornará NULL e passará 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) {
    /* propagate error */
}

while ((item = PyIter_Next(iterator))) {
    /* do something with item */
    ...
    /* release reference when done */
    Py_DECREF(item);
}

Py_DECREF(iterator);

if (PyErr_Occurred()) {
    /* propagate error */
}
else {
    /* continue doing useful work */
}
```

7.7 Protocolo de Buffer

Certos objetos disponíveis em 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 consumidor da interface de buffer é o método `write()` de objetos arquivo: qualquer objeto que pode exportar uma série de bytes através da interface de buffer pode ser gravado em um arquivo. Enquanto `write()` só precisa de acesso somente leitura aos conteúdos internos do objeto passado, outros métodos, tais como `readinto()` precisam de acesso de gravação ao conteúdo de seu argumento. A interface de buffer permite aos objetos permitir ou rejeitar seletivamente a exportação de buffers de leitura e escrita e de somente leitura.

Existem duas maneiras para um usuário da interface de buffer adquirir um buffer em um objeto alvo:

- chamada `PyObject_GetBuffer()` com os parâmetros certos;
- chamada `PyArg_ParseTuple()` (ou um dos seus irmãos) com um dos `y*`, `w*` ou `s*` *format codes*.

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()`.

Py_buffer

`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.

`void *obj`

Uma nova referência ao objeto exportador. A referência é possuída pelo consumidor e automaticamente decrementada e definida para NULL por `PyBuffer_Release()`. O campo é o equivalente ao valor de retorno de qualquer função 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 contínuas, este é o comprimento do bloco de memória subjacente. Para matrizes não contínuas, 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 assumir `itemsize == 1`.

const char *format

Uma string terminada por `NUL` no estilo de sintaxe de módulo `struct` descrevendo os conteúdos de um único item. Se isso é NULL, "B" (unsigned bytes) é assumido.

Este campo é controlado pelo sinalizador `PyBUF_FORMAT`.

int ndim

O número de dimensões que a memória representa como um vetor n-dimensional. Se é 0, `buf` aponta para um único item representando um escalar. Neste caso, `shape`, `strides` e `suboffsets` DEVEM ser NULL.

A macro `PyBUF_MAX_NDIM` limita o número máximo de dimensões a 64. Os exportadores DEVEM respeitar esse limite, os consumidores de buffers multidimensionais DEVEM ser capazes de lidar com dimensões `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 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.

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 *Py_buffer* são inequivocamente definidos pelo tipo de solicitação.

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 reportar falha. Caso contrário, o exportador pode fornecer um buffer de somente leitura ou gravável, mas a escolha DEVE ser consistente para todos os consumidores.

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 l'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 pode ser l'd para qualquer um dos sinalizadores, exceto *PyBUF_SIMPLE*. O último já implica o formato B (bytes não assinados).

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
<i>PyBUF_ND</i>	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
<code>PyBUF_FULL</code>	sim	sim	se necessário	U	0	sim
<code>PyBUF_FULL_RO</code>	sim	sim	se necessário	U	1 ou 0	sim
<code>PyBUF_RECORDS</code>	sim	sim	NULL	U	0	sim
<code>PyBUF_RECORDS_RO</code>	sim	sim	NULL	U	1 ou 0	sim
<code>PyBUF_STRIDED</code>	sim	sim	NULL	U	0	NULL
<code>PyBUF_STRIDED_RO</code>	sim	sim	NULL	U	1 ou 0	NULL
<code>PyBUF_CONTIG</code>	sim	NULL	NULL	C	0	NULL
<code>PyBUF_CONTIG_RO</code>	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):
    """Verify that the parameters represent a valid array within
    the bounds of the allocated memory:
        char *mem: start of the physical memory block
        memlen: length of the physical memory block
        offset: (char *)buf - mem
    """
    if offset % itemsize:
        return False
    if offset < 0 or offset+itemsize > memlen:
        return False
    if any(v % itemsize for v in strides):
        return False
```

(continua na próxima página)

(continuação da página anterior)

```

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+itemsiz <= 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++) {
        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)

Retorne 1 se *obj* suporta a 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)

Envia uma requisição para o *exporter* para preencher *view* como especificado por *flags*. Se o exportador não consegue prover um buffer do mesmo tipo, ele DEVE levantar `PyExc_BufferError`, definir *view->obj* para NULL e retornar -1.

Em caso de sucesso, preencha *view*, defina *view->obj* para uma nova referência para *exporter* e retorne 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* (Ver *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)

Lança o buffer *view* e decrementa a contagem de referências para `view->obj`. Esta função DEVE ser chamada quando o buffer não está mais sendo usado, senão podem ocorrer vazamentos de referência.

É um erro chamar essa função em um buffer que não foi obtido via `PyObject_GetBuffer()`.

Py_ssize_t **PyBuffer_SizeFromFormat** (const char *format)

Retorna o implícito *itemsize* de *format*. Se houver erro, levanta uma exceção e retorna -1.

Novo na versão 3.9.

int **PyBuffer_IsContiguous** (*Py_buffer* *view, char order)

Retorna 1 se a memória definida pela *view* é no estilo C (*order* é 'C') ou no estilo Fortran (*order* é 'F') *contiguous* ou qualquer outra (*order* é 'A'). Retorna 0 caso contrário. Essa função é sempre bem sucedida.

void* **PyBuffer_GetPointer** (*Py_buffer* *view, *Py_ssize_t* *indices)

Recebe a área de memória apontada pelos *indices* dentro da *view* dada. *indices* deve apontar para uma matriz de `view->ndim` índices.

int **PyBuffer_FromContiguous** (*Py_buffer* *view, void *buf, *Py_ssize_t* len, char fort)

Copia bytes *len* contíguos de *buf* para *view*. *fort* pode ser 'C' ou 'F' (para ordenação estilo C ou estilo Fortran). Retorna 0 caso haja sucesso e -1 caso haja erro.

int **PyBuffer_ToContiguous** (void *buf, *Py_buffer* *src, *Py_ssize_t* len, char order)

Copia bytes *len* 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*.

void **PyBuffer_FillContiguousStrides** (int ndims, *Py_ssize_t* *shape, *Py_ssize_t* *strides, int itemsize, char order)

Preenche a matriz *strides* com byte-strides de uma matriz *contiguous* (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)

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 apenas-leitura e `PyBUF_WRITABLE` esteja definido em *flags*.

Em caso de sucesso, defina `view->obj` para uma nova referência para *exporter* e retorne 0. Caso contrário, levante `PyExc_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.

7.8 Protocolo de Buffer Antigo

Obsoleto desde a versão 3.0.

Essas funções faziam parte da API do “protocolo de buffer antigo” no Python 2. No Python 3, esse protocolo não existe mais, mas as funções ainda estão expostas para facilitar a portabilidade do código 2.x. Eles atuam como um wrapper de compatibilidade em torno do *novο protocolo de buffer*, mas não oferecem controle sobre a vida útil dos recursos adquiridos quando um buffer é exportado.

Portanto, é recomendável que você chame `PyObject_GetBuffer()` (ou os *códigos de formatação* `y*` ou `w*` com o família de funções de `PyArg_ParseTuple()`) para obter uma visão de buffer sobre um objeto e `PyBuffer_Release()` quando a visão de buffer puder ser liberada.

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

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

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

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

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

Retorna 1 se *o* tiver suporte a interface de buffer legível de segmento único. Caso contrário, retorna 0. Esta função sempre tem sucesso.

Observe que esta função tenta obter e liberar um buffer, e as exceções que ocorrem ao chamar as funções correspondentes serão suprimidas. Para obter o relatório de erros, use `PyObject_GetBuffer()` em vez disso.

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

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

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

PyTypeObject

A estrutura C dos objetos usados para descrever tipos embutidos.

PyTypeObject **PyType_Type**

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** ()

Limpa o cache de pesquisa interno. Retorna a marcação de versão atual.

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

Retorna o membro *tp_flags* de *type*. Esta função deve ser usada principalmente com *Py_LIMITED_API*; os bits sinalizadores individuais têm garantia de estabilidade em todas as versões do Python, mas o acesso a *tp_flags* não faz parte da API limitada.

Novo na versão 3.2.

Alterado na versão 3.4: O tipo de retorno é agora um `unsigned long` em vez de um `long`.

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

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_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*)

Retorna verdadeiro se o objeto tipo incluir suporte para o detector de ciclo; isso testa o sinalizador de tipo *Py_TPFLAGS_HAVE_GC*.

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

Retorna verdadeiro se *a* for um subtipo de *b*.

Esta função só verifica pelos subtipos, o que significa que `__subclasscheck__()` não é chamado em *b*. Chame *PyObject_IsSubclass()* para fazer a mesma verificação que `issubclass()` faria.

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

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

Return value: *New reference*. 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*)

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.

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

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*.

Uma exceção é levantada se *type* não é um tipo heap.

Novo na versão 3.4.

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

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.

Novo na versão 3.9.

*void** **PyType_GetModuleState** (*PyTypeObject* *type)

Return the state of the module object associated with the given type. This is a shortcut for calling *PyModule_GetState()* on the result of *PyType_GetModule()*.

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.

Novo na versão 3.9.

Creating Heap-Allocated Types

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

*PyObject** **PyType_FromModuleAndSpec** (*PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

Return value: New reference. Creates and returns a heap type object from the *spec* (`Py_TPFLAGS_HEAPTYPE`).

If *bases* is a tuple, the created heap type contains all types contained in it as base types.

If *bases* is `NULL`, the *Py_tp_bases* slot is used instead. If that also is `NULL`, the *Py_tp_base* slot is used instead. If that also is `NULL`, the new type derives from `object`.

The *module* argument can be used to record the module in which the new class is defined. It must be a module object or `NULL`. If not `NULL`, the module is associated with the new type and can later be retrieved with *PyType_GetModule()*. The associated module is not inherited by subclasses; it must be specified for each class individually.

This function calls *PyType_Ready()* on the new type.

Novo na versão 3.9.

*PyObject** **PyType_FromSpecWithBases** (*PyType_Spec* *spec, *PyObject* *bases)

Return value: New reference. Equivalent to `PyType_FromModuleAndSpec(NULL, spec, bases)`.

Novo na versão 3.3.

*PyObject** **PyType_FromSpec** (*PyType_Spec* *spec)

Return value: New reference. Equivalent to `PyType_FromSpecWithBases(spec, NULL)`.

PyType_Spec

Structure defining a type's behavior.

`const char*` **PyType_Spec.name**

Name of the type, used to set *PyTypeObject.tp_name*.

`int` **PyType_Spec.basicsize**

`int` **PyType_Spec.itemsize**

Size of the instance in bytes, used to set *PyTypeObject.tp_basicsize* and *PyTypeObject.tp_itemsize*.

int **PyType_Spec.flags**

Type flags, used to set *PyTypeObject.tp_flags*.

If the `Py_TPFLAGS_HEAPTYPE` flag is not set, *PyType_FromSpecWithBases()* sets it automatically.

PyType_Slot ***PyType_Spec.slots**

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

PyType_Slot

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

int **PyType_Slot.slot**

A slot ID.

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

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

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

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

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

- *bf_getbuffer*
- *bf_releasebuffer*

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

Alterado na versão 3.9: Slots in *PyBufferProcs* may be set in the unlimited API.

void ***PyType_Slot.pfunc**

The desired value of the slot. In most cases, this is a pointer to a function.

May not be NULL.

8.1.2 O Objeto None

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

*PyObject** **Py_None**

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

Py_RETURN_NONE

Manipular devidamente o retorno *Py_None* de dentro de uma função C (ou seja, incrementar a contagem de referência de *None* e devolvê-la.)

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.

PyLongObject

Este subtipo de *PyObject* representa um objeto inteiro Python.

PyTypeObject **PyLong_Type**

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*)

Return value: New reference. 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*)

Return value: New reference. Retorna um novo objeto *PyLongObject* de um unsigned long C ou NULL em caso de falha.

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

Return value: New reference. Retorna um novo objeto *PyLongObject* de um *Py_ssize_t* C ou NULL em caso de falha.

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

Return value: New reference. Retorna um novo objeto *PyLongObject* de um *size_t* C ou NULL em caso de falha.

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

Return value: *New reference.* Retorna um novo objeto *PyLongObject* de um long long C ou NULL em caso de falha.

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

Return value: *New reference.* Retorna um novo objeto *PyLongObject* de um unsigned long long C ou NULL em caso de falha.

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

Return value: *New reference.* 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*)

Return value: *New reference.* Retorna um novo *PyLongObject* com base no valor da string em *str*, que é interpretado de acordo com a raiz em *base*. Se *pend* não for NULL, **pend* apontará para o primeiro caractere em *str* que segue a representação do número. Se *base* for 0, *str* é interpretado usando a definição de integers; neste caso, zeros à esquerda em um número decimal diferente de zero aumenta um *ValueError*. Se *base* não for 0, deve estar entre 2 e 36, inclusive. Espaços iniciais e sublinhados simples após um especificador de base e entre dígitos são ignorados. Se não houver dígitos, *ValueError* será levantada.

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

Return value: *New reference.* Converte uma sequência de dígitos Unicode para um valor inteiro Python.

Deprecated since version 3.3, will be removed in version 3.10: Parte do estilo antigo *Py_UNICODE* API; por favor, migre o uso para *PyLong_FromUnicodeObject* ().

*PyObject** **PyLong_FromUnicodeObject** (*PyObject* **u*, int *base*)

Return value: *New reference.* Converte uma sequência de dígitos Unicode na string *u* para um valor inteiro Python.

Novo na versão 3.3.

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

Return value: *New reference.* Cria um inteiro Python a partir do ponteiro *p*. O valor do ponteiro pode ser recuperado do valor resultante usando *PyLong_AsVoidPtr* ().

long **PyLong_AsLong** (*PyObject* **obj*)

Retorna uma representação de long C de *obj*. Se *obj* não for uma instância de *PyLongObject*, primeiro chama seu método *__index__* () ou *__int__* () (se presente) para convertê-lo em um *PyLongObject*.

Levanta *OverflowError* se o valor de *obj* estiver fora do intervalo de um long.

Retorna -1 no caso de erro. Use *PyErr_Occurred* () para desambiguar.

Alterado na versão 3.8: Usa *__index__* (), se disponível.

Obsoleto desde a versão 3.8: O uso de *__int__* () foi descontinuado.

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

Retorna uma representação de long C de *obj*. Se *obj* não for uma instância de *PyLongObject*, primeiro chama seu método *__index__* () ou *__int__* () (se presente) para convertê-lo em um *PyLongObject*.

Se o valor de *obj* for maior que *LONG_MAX* ou menor que *LONG_MIN*, define **overflow* para 1 ou -1, respectivamente, e retorna -1; caso contrário, define **overflow* para 0. Se qualquer outra exceção ocorrer, define **overflow* para 0 e retorne -1 como de costume.

Retorna -1 no caso de erro. Use *PyErr_Occurred* () para desambiguar.

Alterado na versão 3.8: Usa *__index__* (), se disponível.

Obsoleto desde a versão 3.8: O uso de *__int__* () foi descontinuado.

`long long PyLong_AsLongLong (PyObject *obj)`

Retorna uma representação de `long long C` de *obj*. Se *obj* não for uma instância de *PyLongObject*, primeiro chama seu método `__index__()` ou `__int__()` (se presente) para convertê-lo em um *PyLongObject*.

Levanta *OverflowError* se o valor de *obj* estiver fora do intervalo de um `long long`.

Retorna `-1` no caso de erro. Use *PyErr_Occurred()* para desambiguar.

Alterado na versão 3.8: Usa `__index__()`, se disponível.

Obsoleto desde a versão 3.8: O uso de `__int__()` foi descontinuado.

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

Retorna uma representação de `long long C` de *obj*. Se *obj* não for uma instância de *PyLongObject*, primeiro chama seu método `__index__()` ou `__int__()` (se presente) para convertê-lo em um *PyLongObject*.

Se o valor de *obj* for maior que `LLONG_MAX` ou menor que `LLONG_MIN`, define **overflow* para 1 ou `-1`, respectivamente, e retorna `-1`; caso contrário, define **overflow* para 0. Se qualquer outra exceção ocorrer, define **overflow* para 0 e retorne `-1` como de costume.

Retorna `-1` no caso de erro. Use *PyErr_Occurred()* para desambiguar.

Novo na versão 3.2.

Alterado na versão 3.8: Usa `__index__()`, se disponível.

Obsoleto desde a versão 3.8: O uso de `__int__()` foi descontinuado.

`Py_ssize_t PyLong_AsSsize_t (PyObject *pylong)`

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)`

Retorna uma representação de `unsigned long C` de *pylong*. *pylong* deve ser uma instância de *PyLongObject*.

Levanta *OverflowError* se o valor de *pylong* estiver fora do intervalo de um `unsigned long`.

Retorna `(unsigned long)-1` no caso de erro. Use *PyErr_Occurred()* para desambiguar.

`size_t PyLong_AsSize_t (PyObject *pylong)`

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)`

Retorna uma representação de `unsigned long long C` de *pylong*. *pylong* deve ser uma instância de *PyLongObject*.

Levanta *OverflowError* se o valor de *pylong* estiver fora do intervalo de um `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)`

Retorna uma representação de `unsigned long C` de *obj*. Se *obj* não for uma instância de *PyLongObject*, primeiro chama seu método `__index__()` ou `__int__()` (se presente) para convertê-lo em um *PyLongObject*.

Se o valor de *obj* estiver fora do intervalo para um unsigned long, retorna a redução desse módulo de valor `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.

Obsoleto desde a versão 3.8: O uso de `__int__()` foi descontinuado.

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

Retorna uma representação de unsigned long long C de *obj*. Se *obj* não for uma instância de `PyLongObject`, primeiro chama seu método `__index__()` ou `__int__()` (se presente) para convertê-lo em um `PyLongObject`.

Se o valor de *obj* estiver fora do intervalo para um unsigned long long, retorna a redução desse módulo de valor `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.

Obsoleto desde a versão 3.8: O uso de `__int__()` foi descontinuado.

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

Retorna uma representação de double C de *pylong*. *pylong* deve ser uma instância de `PyLongObject`.

Levanta `OverflowError` se o valor de *pylong* estiver fora do intervalo de um double.

Retorna -1.0 no caso de erro. Use `PyErr_Occurred()` para desambiguar.

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

Converte um inteiro Python *pylong* em um ponteiro void C. Se *pylong* não puder ser convertido, uma `OverflowError` será levantada. Isso só é garantido para produzir um ponteiro utilizável void para valores criados com `PyLong_FromVoidPtr()`.

Retorna NULL no caso de erro. Use `PyErr_Occurred()` para desambiguar.

8.2.2 Objetos Booleanos

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

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

Retorna verdadeiro se *o* for do tipo `PyBool_Type`. Esta função sempre tem sucesso.

*PyObject** **Py_False**

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

*PyObject** **Py_True**

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

Py_RETURN_FALSE

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

Py_RETURN_TRUE

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

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

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

8.2.3 Objetos de ponto flutuante

PyFloatObject

Este subtipo de *PyObject* representa um objeto de ponto flutuante do Python.

PyObject **PyFloat_Type**

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*)

Return value: *New reference.* Cria um objeto *PyFloatObject* baseado em uma string de valor “str” ou NULL em falha.

*PyObject** **PyFloat_FromDouble** (double *v*)

Return value: *New reference.* Cria um objeto *PyFloatObject* de *v* ou NULL em falha.

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

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, ele voltará a `__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)

Return value: *New reference.* Retorna uma instância de structseq que contém informações sobre a precisão, os valores mínimo e máximo de um ponto flutuante. É um wrapper fino em torno do arquivo de cabeçalho `float.h`.

double **PyFloat_GetMax** ()

Retorna o ponto flutuante finito máximo representável *DBL_MAX* como double do C.

double **PyFloat_GetMin** ()

Retorna o ponto flutuante positivo mínimo normalizado *DBL_MIN* como double do C.

8.2.4 Objetos de números complexos

Os objetos de 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.

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. É definido 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

PyComplexObject

Este subtipo de *PyObject* representa um objeto Python de número complexo.

PyObject **PyComplex_Type**

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)

Return value: *New reference.* Cria um novo objeto de número complexo Python a partir de um valor C *Py_complex*.

*PyObject** **PyComplex_FromDoubles** (double real, double imag)

Return value: *New reference.* Retorna um novo objeto *PyComplexObject* de real e imag.

double **PyComplex_RealAsDouble** (*PyObject* *op)

Retorna a parte real de *op* como um double C.

double **PyComplex_ImagAsDouble** (*PyObject* *op)

Retorna a parte imaginária de *op* como um double C.

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 `-1.0` como um valor real.

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

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

PyBytesObject

Esta é uma instância de *PyObject* representando o objeto bytes do Python.

PyTypeObject **PyBytes_Type**

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)

Return value: *New reference.* 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)

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

Return value: New reference. 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
%%	<i>n/d</i>	O caractere literal %.
%c	int	Um único byte, representado como um C int.
%d	int	Equivalente a <code>printf("%d").</code> ¹
%u	unsigned int	Equivalente a <code>printf("%u").</code> ¹
%ld	long	Equivalente a <code>printf("%ld").</code> ¹
%lu	unsigned long	Equivalente a <code>printf("%lu").</code> ¹
%zd	<i>Py_ssize_t</i>	Equivalente a <code>printf("%zd").</code> ¹
%zu	size_t	Equivalente a <code>printf("%zu").</code> ¹
%i	int	Equivalente a <code>printf("%i").</code> ¹
%x	int	Equivalente a <code>printf("%x").</code> 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 <code>printf("%p")</code> exceto que é garantido que comece com o literal 0x independentemente do que o <code>printf</code> 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 vars)

Return value: New reference. Idêntico a *PyBytes_FromFormat()* exceto que é preciso exatamente dois argumentos.

*PyObject** **PyBytes_FromObject** (*PyObject* *o)

Return value: New reference. Retorna a representação de bytes do objeto *o* que implementa o protocolo de buffer.

Py_ssize_t **PyBytes_Size** (*PyObject* *o)

Retorna o comprimento dos bytes em objeto bytes *o*.

Py_ssize_t **PyBytes_GET_SIZE** (*PyObject* *o)

Forma macro de *PyBytes_Size()*, mas sem verificação de erro.

char* **PyBytes_AsString** (*PyObject* *o)

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*.

char* **PyBytes_AS_STRING** (*PyObject* *string)

Forma de macro de *PyBytes_AsString()*, mas sem verificação de erro.

int **PyBytes_AsStringAndSize** (*PyObject* *obj, char **buffer, *Py_ssize_t* *length)

Retorna os conteúdos terminados nulos do objeto *obj* através das variáveis de saída *buffer* e *length*.

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.

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.

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 eleva `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)

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)

Cria um novo objeto bytes em **bytes* contendo o conteúdo de *newpart* anexado a *bytes*. Esta versão diminui a contagem de referências de *newpart*.

int **_PyBytes_Resize** (*PyObject* **bytes, *Py_ssize_t* newsize)

Uma maneira de redimensionar um objeto de bytes, mesmo que seja “imutável”. Use isso apenas para construir um novo objeto de bytes; não use isso se os bytes já puderem ser conhecidos em outras partes do código. É um erro invocar essa função se o refcount no objeto de bytes de entrada não for um. 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

PyByteArrayObject

Esse subtipo de *PyObject* representa um objeto Python bytearray.

PyTypeObject **PyByteArray_Type**

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)

Return value: New reference. Retorna um novo objeto bytearray, o, que implementa o *protocolo de buffer*.

*PyObject** **PyByteArray_FromStringAndSize** (const char *string, *Py_ssize_t* len)

Return value: New reference. Cria um novo objeto bytearray a partir de *string* e seu comprimento, *len*. Em caso de falha, NULL é retornado.

*PyObject** **PyByteArray_Concat** (*PyObject* *a, *PyObject* *b)

Return value: New reference. Concatena os bytearrays *a* e *b* e retorna um novo bytearray com o resultado.

Py_ssize_t **PyByteArray_Size** (*PyObject* *bytearray)

Retorna o tamanho de *bytearray* após verificar se há um ponteiro NULL.

char* **PyByteArray_AsString** (*PyObject* *bytearray)

Retorna o conteúdo de *bytearray* como uma matriz de caracteres após verificar um ponteiro NULL. A matriz retornada sempre tem um byte nulo extra acrescentado.

int **PyByteArray_Resize** (*PyObject* *bytearray, *Py_ssize_t* len)

Redimensiona o buffer interno de *bytearray* para o tamanho *len*.

Macros

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

char* **PyByteArray_AS_STRING** (*PyObject* *bytearray)

Versão macro de *PyByteArray_AsString()*.

Py_ssize_t **PyByteArray_GET_SIZE** (*PyObject* *bytearray)

Versão macro de *PyByteArray_Size()*.

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).

*Py_UNICODE** and UTF-8 representations are created on demand and cached in the Unicode object. The *Py_UNICODE** representation is deprecated and inefficient.

Due to the transition between the old APIs and the new APIs, Unicode objects can internally be in two states depending on how they were created:

- “canonical” Unicode objects are all objects created by a non-deprecated Unicode API. They use the most efficient representation allowed by the implementation.
- “legacy” Unicode objects have been created through one of the deprecated APIs (typically *PyUnicode_FromUnicode()*) and only bear the *Py_UNICODE** representation; you will have to call *PyUnicode_READY()* on them before calling any other API.

Nota: The “legacy” Unicode object will be removed in Python 3.12 with deprecated APIs. All Unicode objects will be “canonical” since then. See **PEP 623** for more information.

Unicode Type

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

Py_UCS4

Py_UCS2

Py_UCS1

These types are typedefs for unsigned integer types wide enough to contain characters of 32 bits, 16 bits and 8 bits, respectively. When dealing with single Unicode characters, use *Py_UCS4*.

Novo na versão 3.3.

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.

PyASCIIObject

PyCompactUnicodeObject

PyUnicodeObject

These subtypes of *PyObject* represent a Python Unicode object. In almost all cases, they shouldn’t be used directly, since all API functions that deal with Unicode objects take and return *PyObject* pointers.

Novo na versão 3.3.

PyTypeObject **PyUnicode_Type**

This instance of *PyTypeObject* represents the Python Unicode type. It is exposed to Python code as `str`.

The following APIs are really C macros and can be used to do fast checks and to access internal read-only data of Unicode objects:

int PyUnicode_Check (*PyObject* *o)

Return true if the object *o* is a Unicode object or an instance of a Unicode subtype. This function always succeeds.

int PyUnicode_CheckExact (*PyObject* *o)

Return true if the object *o* is a Unicode object, but not an instance of a subtype. This function always succeeds.

int PyUnicode_READY (*PyObject* *o)

Ensure the string object *o* is in the “canonical” representation. This is required before using any of the access macros described below.

Returns 0 on success and -1 with an exception set on failure, which in particular happens if memory allocation fails.

Novo na versão 3.3.

Deprecated since version 3.10, will be removed in version 3.12: This API will be removed with *PyUnicode_FromUnicode()*.

Py_ssize_t **PyUnicode_GET_LENGTH** (*PyObject* *o)

Return the length of the Unicode string, in code points. *o* has to be a Unicode object in the “canonical” representation (not checked).

Novo na versão 3.3.

*Py_UCS1** **PyUnicode_1BYTE_DATA** (*PyObject* *o)

*Py_UCS2** **PyUnicode_2BYTE_DATA** (*PyObject* *o)

*Py_UCS4** **PyUnicode_4BYTE_DATA** (*PyObject* *o)

Return a pointer to the canonical representation cast to UCS1, UCS2 or UCS4 integer types for direct character access. No checks are performed if the canonical representation has the correct character size; use

PyUnicode_KIND() to select the right macro. Make sure *PyUnicode_READY()* has been called before accessing this.

Novo na versão 3.3.

PyUnicode_WCHAR_KIND

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND

PyUnicode_4BYTE_KIND

Return values of the *PyUnicode_KIND()* macro.

Novo na versão 3.3.

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

unsigned int **PyUnicode_KIND** (*PyObject* **o*)

Return one of the *PyUnicode* kind constants (see above) that indicate how many bytes per character this Unicode object uses to store its data. *o* has to be a Unicode object in the “canonical” representation (not checked).

Novo na versão 3.3.

void* **PyUnicode_DATA** (*PyObject* **o*)

Return a void pointer to the raw Unicode buffer. *o* has to be a Unicode object in the “canonical” representation (not checked).

Novo 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 macro does not do any sanity checks and is intended for usage in loops. The caller should cache the *kind* value and *data* pointer as obtained from other macro calls. *index* is the index in the string (starts at 0) and *value* is the new code point value which should be written to that location.

Novo 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.

Novo na versão 3.3.

Py_UCS4 **PyUnicode_READ_CHAR** (*PyObject* **o*, *Py_ssize_t* *index*)

Read a character from a Unicode object *o*, which must be in the “canonical” representation. This is less efficient than *PyUnicode_READ()* if you do multiple consecutive reads.

Novo na versão 3.3.

PyUnicode_MAX_CHAR_VALUE (*o*)

Return the maximum code point that is suitable for creating another string based on *o*, which must be in the “canonical” representation. This is always an approximation but more efficient than iterating over the string.

Novo na versão 3.3.

Py_ssize_t **PyUnicode_GET_SIZE** (*PyObject* **o*)

Return the size of the deprecated *Py_UNICODE* representation, in code units (this includes surrogate pairs as 2 units). *o* has to be a Unicode object (not checked).

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

Py_ssize_t **PyUnicode_GET_DATA_SIZE** (*PyObject* **o*)

Return the size of the deprecated *Py_UNICODE* representation in bytes. *o* has to be a Unicode object (not checked).

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

`Py_UNICODE*` **PyUnicode_AS_UNICODE** (*PyObject* **o*)

`const char*` **PyUnicode_AS_DATA** (*PyObject* **o*)

Return a pointer to a `Py_UNICODE` representation of the object. The returned buffer is always terminated with an extra null code point. It may also contain embedded null code points, which would cause the string to be truncated when used in most C functions. The `AS_DATA` form casts the pointer to `const char *`. The *o* argument has to be a Unicode object (not checked).

Alterado na versão 3.3: This macro is now inefficient – because in many cases the `Py_UNICODE` representation does not exist and needs to be created – and can fail (return `NULL` with an exception set). Try to port the code to use the new `PyUnicode_nBYTE_DATA()` macros or use `PyUnicode_WRITE()` or `PyUnicode_READ()`.

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

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

Return 1 if the string is a valid identifier according to the language definition, section identifiers. Return 0 otherwise.

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. Nonprintable characters are those characters defined in the Unicode character database as “Other” or “Separator”, excepting the ASCII space (0x20) which is considered printable. (Note that printable characters in this context are those which should not be escaped when

`repr()` is invoked on a string. It has no bearing on the handling of strings written to `sys.stdout` or `sys.stderr`.)

These APIs can be used for fast direct character conversions:

Py_UCS4 **Py_UNICODE_TOLOWER** (*Py_UCS4* *ch*)

Return the character *ch* converted to lower case.

Obsoleto desde a versão 3.3: This function uses simple case mappings.

Py_UCS4 **Py_UNICODE_TOUPPER** (*Py_UCS4* *ch*)

Return the character *ch* converted to upper case.

Obsoleto desde a versão 3.3: This function uses simple case mappings.

Py_UCS4 **Py_UNICODE_TOTITLE** (*Py_UCS4* *ch*)

Return the character *ch* converted to title case.

Obsoleto desde a versão 3.3: This function uses simple case mappings.

int **Py_UNICODE_TODECIMAL** (*Py_UCS4* *ch*)

Return the character *ch* converted to a decimal positive integer. Return `-1` if this is not possible. This macro does not raise exceptions.

int **Py_UNICODE_TODIGIT** (*Py_UCS4* *ch*)

Return the character *ch* converted to a single digit integer. Return `-1` if this is not possible. This macro does not raise exceptions.

double **Py_UNICODE_TONUMERIC** (*Py_UCS4* *ch*)

Return the character *ch* converted to a double. Return `-1.0` if this is not possible. This macro does not raise exceptions.

These APIs can be used to work with surrogates:

Py_UNICODE_IS_SURROGATE (*ch*)

Check if *ch* is a surrogate (`0xD800 <= ch <= 0xDFFF`).

Py_UNICODE_IS_HIGH_SURROGATE (*ch*)

Check if *ch* is a high surrogate (`0xD800 <= ch <= 0xDBFF`).

Py_UNICODE_IS_LOW_SURROGATE (*ch*)

Check if *ch* is a low surrogate (`0xDC00 <= ch <= 0xDFFF`).

Py_UNICODE_JOIN_SURROGATES (*high*, *low*)

Join two surrogate characters and return a single *Py_UCS4* value. *high* and *low* are respectively the leading and trailing surrogates in a surrogate pair.

Creating and accessing Unicode strings

To create Unicode objects and access their basic sequence properties, use these APIs:

*PyObject** **PyUnicode_New** (*Py_ssize_t* *size*, *Py_UCS4* *maxchar*)

Return value: *New reference.* Create a new Unicode object. *maxchar* should be the true maximum code point to be placed in the string. As an approximation, it can be rounded up to the nearest value in the sequence 127, 255, 65535, 1114111.

This is the recommended way to allocate a new Unicode object. Objects created using this function are not resizable.

Novo na versão 3.3.

*PyObject** **PyUnicode_FromKindAndData** (int *kind*, const void **buffer*, *Py_ssize_t* *size*)

Return value: *New reference.* Create a new Unicode object with the given *kind* (possible values are

`PyUnicode_1BYTE_KIND` etc., as returned by `PyUnicode_KIND()`). The *buffer* must point to an array of *size* units of 1, 2 or 4 bytes per character, as given by the kind.

Novo na versão 3.3.

PyObject* PyUnicode_FromStringAndSize (const char **u*, *Py_ssize_t* *size*)

Return value: *New reference.* Create a Unicode object from the char buffer *u*. The bytes will be interpreted as being UTF-8 encoded. The buffer is copied into the new object. If the buffer is not NULL, the return value might be a shared object, i.e. modification of the data is not allowed.

If *u* is NULL, this function behaves like `PyUnicode_FromUnicode()` with the buffer set to NULL. This usage is deprecated in favor of `PyUnicode_New()`, and will be removed in Python 3.12.

PyObject* PyUnicode_FromString (const char **u*)

Return value: *New reference.* Create a Unicode object from a UTF-8 encoded null-terminated char buffer *u*.

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

Return value: *New reference.* Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python Unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* ASCII-encoded string. The following format characters are allowed:

Caracteres Formatados	Tipo	Comentário
%%	<i>n/d</i>	O caractere literal %.
%c	int	A single character, represented as a C int.
%d	int	Equivalente a <code>printf("%d")</code> . ¹
%u	unsigned int	Equivalente a <code>printf("%u")</code> . ¹
%ld	long	Equivalente a <code>printf("%ld")</code> . ¹
%li	long	Equivalent to <code>printf("%li")</code> . ¹
%lu	unsigned long	Equivalente a <code>printf("%lu")</code> . ¹
%lld	long long	Equivalent to <code>printf("%lld")</code> . ¹
%lli	long long	Equivalent to <code>printf("%lli")</code> . ¹
%llu	unsigned long long	Equivalent to <code>printf("%llu")</code> . ¹
%zd	<i>Py_ssize_t</i>	Equivalente a <code>printf("%zd")</code> . ¹
%zi	<i>Py_ssize_t</i>	Equivalent to <code>printf("%zi")</code> . ¹
%zu	<i>size_t</i>	Equivalente a <code>printf("%zu")</code> . ¹
%i	int	Equivalente a <code>printf("%i")</code> . ¹
%x	int	Equivalente a <code>printf("%x")</code> . 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 <code>printf("%p")</code> exceto que é garantido que comece com o literal 0x independentemente do que o <code>printf</code> da plataforma ceda.
%A	PyObject*	The result of calling <code>ascii()</code> .
%U	PyObject*	A Unicode object.
%V	PyObject*, const char*	A Unicode object (which may be NULL) and a null-terminated C character array as a second parameter (which will be used, if the first parameter is NULL).
%S	PyObject*	The result of calling <code>PyObject_Str()</code> .
%R	PyObject*	The result of calling <code>PyObject_Repr()</code> .

¹ For integer specifiers (d, u, ld, li, lu, lld, lli, llu, zd, zi, zu, i, x): the 0-conversion flag has effect even when a precision is given.

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

Nota: The width formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes for "%s" and "%V" (if the `PyObject*` argument is NULL), and a number of characters for "%A", "%U", "%S", "%R" and "%V" (if the `PyObject*` argument is not NULL).

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.

*PyObject** **PyUnicode_FromFormatV** (const char *format, va_list args)

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

*PyObject** **PyUnicode_FromEncodedObject** (*PyObject* *obj, const char *encoding, const char *errors)

Return value: New reference. Decode an encoded object *obj* to a Unicode object.

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

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

The API returns NULL if there was an error. The caller is responsible for decref'ing the returned objects.

Py_ssize_t **PyUnicode_GetLength** (*PyObject* *unicode)

Return the length of the Unicode object, in code points.

Novo 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.

Novo 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.

Novo na versão 3.3.

int **PyUnicode_WriteChar** (*PyObject* *unicode, *Py_ssize_t* index, *Py_UCS4* character)

Write a character to a string. The string must have been created through `PyUnicode_New()`. Since Unicode strings are supposed to be immutable, the string must not be shared, or have been hashed yet.

This function checks that *unicode* is a Unicode object, that the index is not out of bounds, and that the object can be modified safely (i.e. that its reference count is one).

Novo na versão 3.3.

Py_UCS4 **PyUnicode_ReadChar** (*PyObject* *unicode, *Py_ssize_t* index)

Read a character from a string. This function checks that *unicode* is a Unicode object and the index is not out of bounds, in contrast to the macro version `PyUnicode_READ_CHAR()`.

Novo na versão 3.3.

*PyObject** **PyUnicode_Substring** (*PyObject* *str, *Py_ssize_t* start, *Py_ssize_t* end)

Return value: *New reference.* Return a substring of *str*, from character index *start* (included) to character index *end* (excluded). Negative indices are not supported.

Novo na versão 3.3.

*Py_UCS4** **PyUnicode_AsUCS4** (*PyObject* *u, *Py_UCS4* *buffer, *Py_ssize_t* buflen, int copy_null)

Copy the string *u* into a UCS4 buffer, including a null character, if *copy_null* is set. Returns NULL and sets an exception on error (in particular, a `SystemError` if *buflen* is smaller than the length of *u*). *buffer* is returned on success.

Novo na versão 3.3.

*Py_UCS4** **PyUnicode_AsUCS4Copy** (*PyObject* *u)

Copy the string *u* into a new UCS4 buffer that is allocated using `PyMem_Malloc()`. If this fails, NULL is returned with a `MemoryError` set. The returned buffer always has an extra null code point appended.

Novo na versão 3.3.

Deprecated Py_UNICODE APIs

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

These API functions are deprecated with the implementation of [PEP 393](#). Extension modules can continue using them, as they will not be removed in Python 3.x, but need to be aware that their use can now cause performance and memory hits.

*PyObject** **PyUnicode_FromUnicode** (const *Py_UNICODE* *u, *Py_ssize_t* size)

Return value: *New reference.* Create a Unicode object from the `Py_UNICODE` buffer *u* of the given size. *u* may be NULL which causes the contents to be undefined. It is the user's responsibility to fill in the needed data. The buffer is copied into the new object.

If the buffer is not NULL, the return value might be a shared object. Therefore, modification of the resulting Unicode object is only allowed when *u* is NULL.

If the buffer is NULL, `PyUnicode_READY()` must be called once the string content has been filled before using any of the access macros such as `PyUnicode_KIND()`.

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using `PyUnicode_FromKindAndData()`, `PyUnicode_FromWideChar()`, or `PyUnicode_New()`.

*Py_UNICODE** **PyUnicode_AsUnicode** (*PyObject* *unicode)

Return a read-only pointer to the Unicode object's internal `Py_UNICODE` buffer, or NULL on error. This will create the `Py_UNICODE*` representation of the object if it is not yet available. The buffer is always terminated with an extra null code point. Note that the resulting `Py_UNICODE` string may also contain embedded null code points, which would cause the string to be truncated when used in most C functions.

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

Deprecated since version 3.3, will be removed in version 3.10.

*PyObject** **PyUnicode_TransformDecimalToASCII** (*Py_UNICODE* *s, *Py_ssize_t* size)

Return value: *New reference.* Create a Unicode object by replacing all decimal digits in `Py_UNICODE` buffer of the given *size* by ASCII digits 0–9 according to their decimal value. Return NULL if an exception occurs.

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

*Py_UNICODE** **PyUnicode_AsUnicodeAndSize** (*PyObject* *unicode, *Py_ssize_t* *size)

Like *PyUnicode_AsUnicode()*, but also saves the *Py_UNICODE()* array length (excluding the extra null terminator) in *size*. Note that the resulting *Py_UNICODE** string may contain embedded null code points, which would cause the string to be truncated when used in most C functions.

Novo na versão 3.3.

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

*Py_UNICODE** **PyUnicode_AsUnicodeCopy** (*PyObject* *unicode)

Create a copy of a Unicode string ending with a null code point. Return NULL and raise a *MemoryError* exception on memory allocation failure, otherwise return a new allocated buffer (use *PyMem_Free()* to free the buffer). Note that the resulting *Py_UNICODE** string may contain embedded null code points, which would cause the string to be truncated when used in most C functions.

Novo na versão 3.2.

Please migrate to using *PyUnicode_AsUCS4Copy()* or similar new APIs.

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

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

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

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

Return value: *New reference*. Copy an instance of a Unicode subtype to a new true Unicode object if necessary. If *obj* is already a true Unicode object (not a subtype), return the reference with incremented refcount.

Objects other than Unicode or its subtypes will cause a *TypeError*.

Locale Encoding

The current locale encoding can be used to decode text from the operating system.

*PyObject** **PyUnicode_DecodeLocaleAndSize** (const char *str, *Py_ssize_t* len, const char *errors)

Return value: *New reference*. Decode a string from UTF-8 on Android and VxWorks, or from the current locale encoding on other platforms. The supported error handlers are "strict" and "surrogateescape" (**PEP 383**). The decoder uses "strict" error handler if *errors* is NULL. *str* must end with a null character but cannot contain embedded null characters.

Use *PyUnicode_DecodeFSDefaultAndSize()* to decode a string from *Py_FileSystemDefaultEncoding* (the locale encoding read at Python startup).

This function ignores the Python UTF-8 mode.

Ver também:

The *Py_DecodeLocale()* function.

Novo 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)

Return value: New reference. Similar to *PyUnicode_DecodeLocaleAndSize()*, but compute the string length using *strlen()*.

Novo na versão 3.3.

*PyObject** **PyUnicode_EncodeLocale** (*PyObject* *unicode, const char *errors)

Return value: New reference. Encode a Unicode object to UTF-8 on Android and VxWorks, or to the current locale encoding on other platforms. The supported error handlers are "strict" and "surrogateescape" (**PEP 383**). The encoder uses "strict" error handler if *errors* is NULL. Return a bytes object. *unicode* cannot contain embedded null characters.

Use *PyUnicode_EncodeFSDefault()* to encode a string to *Py_FileSystemDefaultEncoding* (the locale encoding read at Python startup).

This function ignores the Python UTF-8 mode.

Ver também:

The *Py_EncodeLocale()* function.

Novo 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

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

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

ParseTuple converter: encode str objects – obtained directly or through the *os.PathLike* interface – to bytes using *PyUnicode_EncodeFSDefault()*; bytes objects are output as-is. *result* must be a *PyBytesObject** which must be released when it is no longer used.

Novo 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)

ParseTuple converter: decode bytes objects – obtained either directly or indirectly through the *os.PathLike* interface – to str using *PyUnicode_DecodeFSDefaultAndSize()*; str objects are output as-is. *result* must be a *PyUnicodeObject** which must be released when it is no longer used.

Novo na versão 3.2.

Alterado na versão 3.6: Aceita um *objeto caminho ou similar*.

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

Return value: New reference. Decode a string using *Py_FileSystemDefaultEncoding* and the *Py_FileSystemDefaultEncodeErrors* error handler.

If *Py_FileSystemDefaultEncoding* is not set, fall back to the locale encoding.

`Py_FileSystemDefaultEncoding` is initialized at startup from the locale encoding and cannot be modified later. If you need to decode a string from the current locale encoding, use `PyUnicode_DecodeLocaleAndSize()`.

Ver também:

The `Py_DecodeLocale()` function.

Alterado na versão 3.6: Use `Py_FileSystemDefaultEncodeErrors` error handler.

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

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

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

Use `PyUnicode_DecodeFSDefaultAndSize()` if you know the string length.

Alterado na versão 3.6: Use `Py_FileSystemDefaultEncodeErrors` error handler.

***PyObject** `PyUnicode_EncodeFSDefault` (*PyObject* *unicode)**

Return value: New reference. Encode a Unicode object to `Py_FileSystemDefaultEncoding` with the `Py_FileSystemDefaultEncodeErrors` error handler, and return bytes. Note that the resulting bytes object may contain null bytes.

If `Py_FileSystemDefaultEncoding` is not set, fall back to the locale encoding.

`Py_FileSystemDefaultEncoding` is initialized at startup from the locale encoding and cannot be modified later. If you need to encode a string to the current locale encoding, use `PyUnicode_EncodeLocale()`.

Ver também:

The `Py_EncodeLocale()` function.

Novo na versão 3.2.

Alterado na versão 3.6: Use `Py_FileSystemDefaultEncodeErrors` error handler.

wchar_t Support

`wchar_t` support for platforms which support it:

***PyObject** `PyUnicode_FromWideChar` (const wchar_t *w, *Py_ssize_t* size)**

Return value: New reference. Create a Unicode object from the `wchar_t` buffer `w` of the given *size*. Passing `-1` as the *size* indicates that the function must itself compute the length, using `wcsl`. Return `NULL` on failure.

***Py_ssize_t* `PyUnicode_AsWideChar` (*PyObject* *unicode, wchar_t *w, *Py_ssize_t* size)**

Copy the Unicode object contents into the `wchar_t` buffer `w`. At most *size* `wchar_t` characters are copied (excluding a possibly trailing null termination character). Return the number of `wchar_t` characters copied or `-1` in case of an error. Note that the resulting `wchar_t*` string may or may not be null-terminated. It is the responsibility of the caller to make sure that the `wchar_t*` string is null-terminated in case this is required by the application. Also, note that the `wchar_t*` string might contain null characters, which would cause the string to be truncated when used with most C functions.

wchar_t* `PyUnicode_AsWideCharString` (*PyObject* *unicode, *Py_ssize_t* *size)

Convert the Unicode object to a wide character string. The output string always ends with a null character. If *size* is not `NULL`, write the number of wide characters (excluding the trailing null termination character) into *size*. Note that the resulting `wchar_t` string might contain null characters, which would cause the string to be truncated when used with most C functions. If *size* is `NULL` and the `wchar_t*` string contains null characters a `ValueError` is raised.

Returns a buffer allocated by `PyMem_Alloc()` (use `PyMem_Free()` to free it) on success. On error, returns `NULL` and `*size` is undefined. Raises a `MemoryError` if memory allocation is failed.

Novo 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 variable `Py_FileSystemDefaultEncoding` internally. This variable should be treated as read-only: on some systems, it will be a pointer to a static string, on others, it will change at run-time (such as when the application invokes `setlocale`).

Error handling is set by errors which may also be set to `NULL` meaning to use the default handling defined for the codec. Default error handling for all built-in codecs is “strict” (`ValueError` is raised).

The codecs all use a similar interface. Only deviations from the following generic ones are documented for simplicity.

Generic Codecs

These are the generic codec APIs:

*PyObject** **PyUnicode_Decode** (const char *s, *Py_ssize_t* size, const char *encoding, const char *errors)

Return value: New reference. Create a Unicode object by decoding *size* bytes of the encoded string *s*. *encoding* and *errors* have the same meaning as the parameters of the same name in the `str()` built-in function. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

*PyObject** **PyUnicode_AsEncodedString** (*PyObject* *unicode, const char *encoding, const char *errors)

Return value: New reference. Encode a Unicode object and return the result as Python bytes object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode `encode()` method. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

*PyObject** **PyUnicode_Encode** (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *encoding, const char *errors)

Return value: New reference. Encode the *Py_UNICODE* buffer *s* of the given *size* and return a Python bytes object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode `encode()` method. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

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

UTF-8 Codecs

These are the UTF-8 codec APIs:

*PyObject** **PyUnicode_DecodeUTF8** (const char *s, *Py_ssize_t* size, const char *errors)

Return value: *New reference.* Create a Unicode object by decoding *size* bytes of the UTF-8 encoded string *s*. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_DecodeUTF8Stateful** (const char *s, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Return value: *New reference.* If *consumed* is NULL, behave like *PyUnicode_DecodeUTF8()*. If *consumed* is not NULL, trailing incomplete UTF-8 byte sequences will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

*PyObject** **PyUnicode_AsUTF8String** (*PyObject* *unicode)

Return value: *New reference.* Encode a Unicode object using UTF-8 and return the result as Python bytes object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

const char* **PyUnicode_AsUTF8AndSize** (*PyObject* *unicode, *Py_ssize_t* *size)

Return a pointer to the UTF-8 encoding of the Unicode object, and store the size of the encoded representation (in bytes) in *size*. The *size* argument can be NULL; in this case no size will be stored. The returned buffer always has an extra null byte appended (not included in *size*), regardless of whether there are any other null code points.

In the case of an error, NULL is returned with an exception set and no *size* is stored.

This caches the UTF-8 representation of the string in the Unicode object, and subsequent calls will return a pointer to the same buffer. The caller is not responsible for deallocating the buffer. The buffer is deallocated and pointers to it become invalid when the Unicode object is garbage collected.

Novo na versão 3.3.

Alterado na versão 3.7: The return type is now `const char *` rather of `char *`.

const char* **PyUnicode_AsUTF8** (*PyObject* *unicode)

As *PyUnicode_AsUTF8AndSize()*, but does not store the size.

Novo na versão 3.3.

Alterado na versão 3.7: The return type is now `const char *` rather of `char *`.

*PyObject** **PyUnicode_EncodeUTF8** (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors)

Return value: *New reference.* Encode the *Py_UNICODE* buffer *s* of the given *size* using UTF-8 and return a Python bytes object. Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsUTF8String()*, *PyUnicode_AsUTF8AndSize()* or *PyUnicode_AsEncodedString()*.

UTF-32 Codecs

These are the UTF-32 codec APIs:

*PyObject** **PyUnicode_DecodeUTF32** (const char *s, *Py_ssize_t* size, const char *errors, int *byteorder)

Return value: *New reference.* Decode *size* bytes from a UTF-32 encoded buffer string and return the corresponding Unicode object. *errors* (if non-NULL) defines the error handling. It defaults to “strict”.

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


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

If `*byteorder` is zero, and the first four bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output.

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

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

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

PyObject* PyUnicode_DecodeUTF32Stateful (const char *s, Py_ssize_t size, const char *errors, int *byteorder, Py_ssize_t *consumed)

Return value: New reference. If `consumed` is `NULL`, behave like `PyUnicode_DecodeUTF32()`. If `consumed` is not `NULL`, `PyUnicode_DecodeUTF32Stateful()` will not treat trailing incomplete UTF-32 byte sequences (such as a number of bytes not divisible by four) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in `consumed`.

PyObject* PyUnicode_AsUTF32String (PyObject *unicode)

Return value: New reference. Return a Python byte string using the UTF-32 encoding in native byte order. The string always starts with a BOM mark. Error handling is “strict”. Return `NULL` if an exception was raised by the codec.

PyObject* PyUnicode_EncodeUTF32 (const Py_UNICODE *s, Py_ssize_t size, const char *errors, int byteorder)

Return value: New reference. Return a Python bytes object holding the UTF-32 encoded value of the Unicode data in `s`. Output is written according to the following byte order:

```
byteorder == -1: little endian
byteorder == 0:  native byte order (writes a BOM mark)
byteorder == 1:  big endian
```

If `byteorder` is `0`, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.

If `Py_UNICODE_WIDE` is not defined, surrogate pairs will be output as a single code point.

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

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsUTF32String()` or `PyUnicode_AsEncodedString()`.

UTF-16 Codecs

These are the UTF-16 codec APIs:

PyObject* PyUnicode_DecodeUTF16 (const char *s, Py_ssize_t size, const char *errors, int *byteorder)

Return value: New reference. Decode `size` bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. `errors` (if non-`NULL`) defines the error handling. It defaults to “strict”.

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

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

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

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

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

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

PyObject* PyUnicode_DecodeUTF16Stateful (const char *s, Py_ssize_t size, const char *errors, int *byteorder, Py_ssize_t *consumed)

Return value: New reference. If *consumed* is `NULL`, behave like `PyUnicode_DecodeUTF16()`. If *consumed* is not `NULL`, `PyUnicode_DecodeUTF16Stateful()` will not treat trailing incomplete UTF-16 byte sequences (such as an odd number of bytes or a split surrogate pair) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject* PyUnicode_AsUTF16String (PyObject *unicode)

Return value: New reference. Return a Python byte string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is “strict”. Return `NULL` if an exception was raised by the codec.

PyObject* PyUnicode_EncodeUTF16 (const Py_UNICODE *s, Py_ssize_t size, const char *errors, int byteorder)

Return value: New reference. Return a Python bytes object holding the UTF-16 encoded value of the Unicode data in *s*. Output is written according to the following byte order:

```
byteorder == -1: little endian
byteorder == 0: native byte order (writes a BOM mark)
byteorder == 1: big endian
```

If `byteorder` is `0`, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.

If `Py_UNICODE_WIDE` is defined, a single `Py_UNICODE` value may get represented as a surrogate pair. If it is not defined, each `Py_UNICODE` value is interpreted as a UCS-2 character.

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

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsUTF16String()` or `PyUnicode_AsEncodedString()`.

UTF-7 Codecs

These are the UTF-7 codec APIs:

PyObject* PyUnicode_DecodeUTF7 (const char *s, Py_ssize_t size, const char *errors)

Return value: New reference. Create a Unicode object by decoding *size* bytes of the UTF-7 encoded string *s*. Return `NULL` if an exception was raised by the codec.

PyObject* PyUnicode_DecodeUTF7Stateful (const char *s, Py_ssize_t size, const char *errors, Py_ssize_t *consumed)

Return value: New reference. If *consumed* is `NULL`, behave like `PyUnicode_DecodeUTF7()`. If *consumed* is not `NULL`, trailing incomplete UTF-7 base-64 sections will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject* PyUnicode_EncodeUTF7 (const Py_UNICODE *s, Py_ssize_t size, int base64SetO, int base64WhiteSpace, const char *errors)

Return value: New reference. Encode the `Py_UNICODE` buffer of the given size using UTF-7 and return a Python bytes object. Return `NULL` if an exception was raised by the codec.

If `base64SetO` is nonzero, “Set O” (punctuation that has no otherwise special meaning) will be encoded in base-64. If `base64WhiteSpace` is nonzero, whitespace will be encoded in base-64. Both are set to zero for the Python “utf-7” codec.

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

Unicode-Escape Codecs

These are the “Unicode Escape” codec APIs:

*PyObject** **PyUnicode_DecodeUnicodeEscape** (const char *s, *Py_ssize_t* size, const char *errors)

Return value: *New reference.* Create a Unicode object by decoding *size* bytes of the Unicode-Escape encoded string *s*. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_AsUnicodeEscapeString** (*PyObject* *unicode)

Return value: *New reference.* Encode a Unicode object using Unicode-Escape and return the result as a bytes object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeUnicodeEscape** (const *Py_UNICODE* *s, *Py_ssize_t* size)

Return value: *New reference.* Encode the `Py_UNICODE` buffer of the given *size* using Unicode-Escape and return a bytes object. Return NULL if an exception was raised by the codec.

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

Raw-Unicode-Escape Codecs

These are the “Raw Unicode Escape” codec APIs:

*PyObject** **PyUnicode_DecodeRawUnicodeEscape** (const char *s, *Py_ssize_t* size, const char *errors)

Return value: *New reference.* Create a Unicode object by decoding *size* bytes of the Raw-Unicode-Escape encoded string *s*. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_AsRawUnicodeEscapeString** (*PyObject* *unicode)

Return value: *New reference.* Encode a Unicode object using Raw-Unicode-Escape and return the result as a bytes object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeRawUnicodeEscape** (const *Py_UNICODE* *s, *Py_ssize_t* size)

Return value: *New reference.* Encode the `Py_UNICODE` buffer of the given *size* using Raw-Unicode-Escape and return a bytes object. Return NULL if an exception was raised by the codec.

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

Latin-1 Codecs

These are the Latin-1 codec APIs: Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

*PyObject** **PyUnicode_DecodeLatin1** (const char *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Create a Unicode object by decoding *size* bytes of the Latin-1 encoded string *s*. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_AsLatin1String** (*PyObject* *unicode)

Return value: New reference. Encode a Unicode object using Latin-1 and return the result as Python bytes object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeLatin1** (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Encode the *Py_UNICODE* buffer of the given *size* using Latin-1 and return a Python bytes object. Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsLatin1String()* or *PyUnicode_AsEncodedString()*.

ASCII Codecs

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

*PyObject** **PyUnicode_DecodeASCII** (const char *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Create a Unicode object by decoding *size* bytes of the ASCII encoded string *s*. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_AsASCIIString** (*PyObject* *unicode)

Return value: New reference. Encode a Unicode object using ASCII and return the result as Python bytes object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeASCII** (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Encode the *Py_UNICODE* buffer of the given *size* using ASCII and return a Python bytes object. Return NULL if an exception was raised by the codec.

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

Character Map Codecs

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the `encodings` package). The codec uses mappings to encode and decode characters. The mapping objects provided must support the `__getitem__()` mapping interface; dictionaries and sequences work well.

These are the mapping codec APIs:

*PyObject** **PyUnicode_DecodeCharmap** (const char *data, *Py_ssize_t* size, *PyObject* *mapping, const char *errors)

Return value: New reference. Create a Unicode object by decoding *size* bytes of the encoded string *s* using the given *mapping* object. Return NULL if an exception was raised by the codec.

If *mapping* is NULL, Latin-1 decoding will be applied. Else *mapping* must map bytes ordinals (integers in the range from 0 to 255) to Unicode strings, integers (which are then interpreted as Unicode ordinals) or None. Unmapped data bytes – ones which cause a `LookupError`, as well as ones which get mapped to None, `0xFFFE` or `'\ufffe'`, are treated as undefined mappings and cause an error.

*PyObject** **PyUnicode_AsCharmapString** (*PyObject* *unicode, *PyObject* *mapping)

Return value: New reference. Encode a Unicode object using the given *mapping* object and return the result as a bytes object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

The *mapping* object must map Unicode ordinal integers to bytes objects, integers in the range from 0 to 255 or None. Unmapped character ordinals (ones which cause a `LookupError`) as well as mapped to None are treated as “undefined mapping” and cause an error.

*PyObject** **PyUnicode_EncodeCharmap** (const *Py_UNICODE* *s, *Py_ssize_t* size, *PyObject* *mapping, const char *errors)

Return value: New reference. Encode the *Py_UNICODE* buffer of the given *size* using the given *mapping* object and return the result as a bytes object. Return NULL if an exception was raised by the codec.

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

The following codec API is special in that maps Unicode to Unicode.

*PyObject** **PyUnicode_Translate** (*PyObject* *str, *PyObject* *table, const char *errors)

Return value: New reference. Translate a string by applying a character mapping table to it and return the resulting Unicode object. Return NULL if an exception was raised by the codec.

The mapping table must map Unicode ordinal integers to Unicode ordinal integers or None (causing deletion of the character).

Mapping tables need only provide the `__getitem__()` interface; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a `LookupError`) are left untouched and are copied as-is.

errors has the usual meaning for codecs. It may be NULL which indicates to use the default error handling.

*PyObject** **PyUnicode_TranslateCharmap** (const *Py_UNICODE* *s, *Py_ssize_t* size, *PyObject* *mapping, const char *errors)

Return value: New reference. Translate a *Py_UNICODE* buffer of the given *size* by applying a character *mapping* table to it and return the resulting Unicode object. Return NULL when an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_Translate()* or *generic codec based API*

MBCS codecs for Windows

These are the MBCS codec APIs. They are currently only available on Windows and use the Win32 MBCS converters to implement the conversions. Note that MBCS (or DBCS) is a class of encodings, not just one. The target encoding is defined by the user settings on the machine running the codec.

*PyObject** **PyUnicode_DecompileMBCS** (const char *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Create a Unicode object by decoding *size* bytes of the MBCS encoded string *s*. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_DecompileMBCSStateful** (const char *s, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Return value: New reference. If *consumed* is NULL, behave like *PyUnicode_DecompileMBCS()*. If *consumed* is not NULL, *PyUnicode_DecompileMBCSStateful()* will not decode trailing lead byte and the number of bytes that have been decoded will be stored in *consumed*.

*PyObject** **PyUnicode_AsMBCSString** (*PyObject* *unicode)

Return value: New reference. Encode a Unicode object using MBCS and return the result as Python bytes object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

*PyObject** **PyUnicode_EncodeCodePage** (int code_page, *PyObject* *unicode, const char *errors)

Return value: New reference. Encode the Unicode object using the specified code page and return a Python bytes object. Return NULL if an exception was raised by the codec. Use `CP_ACP` code page to get the MBCS encoder.

Novo na versão 3.3.

*PyObject** **PyUnicode_EncodeMBCS** (const *Py_UNICODE* *s, *Py_ssize_t* size, const char *errors)

Return value: New reference. Encode the *Py_UNICODE* buffer of the given *size* using MBCS and return a Python bytes object. Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style *Py_UNICODE* API; please migrate to using *PyUnicode_AsMBCSString()*, *PyUnicode_EncodeCodePage()* or *PyUnicode_AsEncodedString()*.

Methods & Slots

Methods and Slot Functions

The following APIs are capable of handling Unicode objects and strings on input (we refer to them as strings in the descriptions) and return Unicode objects or integers as appropriate.

They all return NULL or -1 if an exception occurs.

*PyObject** **PyUnicode_Concat** (*PyObject* *left, *PyObject* *right)

Return value: New reference. Concat two strings giving a new Unicode string.

*PyObject** **PyUnicode_Split** (*PyObject* *s, *PyObject* *sep, *Py_ssize_t* maxsplit)

Return value: New reference. Split a string giving a list of Unicode strings. If *sep* is NULL, splitting will be done at all whitespace substrings. Otherwise, splits occur at the given separator. At most *maxsplit* splits will be done. If negative, no limit is set. Separators are not included in the resulting list.

*PyObject** **PyUnicode_Splitlines** (*PyObject* *s, int keepend)

Return value: New reference. Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If *keepend* is 0, the line break characters are not included in the resulting strings.

*PyObject** **PyUnicode_Join** (*PyObject* *separator, *PyObject* *seq)

Return value: New reference. Join a sequence of strings using the given *separator* and return the resulting Unicode string.

Py_ssize_t **PyUnicode_Tailmatch** (*PyObject* *str, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Return 1 if *substr* matches *str*[start:end] at the given tail end (*direction* == -1 means to do a prefix match, *direction* == 1 a suffix match), 0 otherwise. Return -1 if an error occurred.

Py_ssize_t **PyUnicode_Find** (*PyObject* *str, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Return the first position of *substr* in *str*[start:end] using the given *direction* (*direction* == 1 means to do a forward search, *direction* == -1 a backward search). The return value is the index of the first match; a value of -1 indicates that no match was found, and -2 indicates that an error occurred and an exception has been set.

Py_ssize_t **PyUnicode_FindChar** (*PyObject* *str, *Py_UCS4* ch, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Return the first position of the character *ch* in *str*[start:end] using the given *direction* (*direction* == 1 means to do a forward search, *direction* == -1 a backward search). The return value is the index of the first match; a value of -1 indicates that no match was found, and -2 indicates that an error occurred and an exception has been set.

Novo na versão 3.3.

Alterado na versão 3.7: *start* and *end* are now adjusted to behave like *str*[start:end].

Py_ssize_t **PyUnicode_Count** (*PyObject* *str, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end)

Return the number of non-overlapping occurrences of *substr* in *str*[start:end]. Return -1 if an error occurred.

*PyObject** **PyUnicode_Replace** (*PyObject* *str, *PyObject* *substr, *PyObject* *replstr, *Py_ssize_t* maxcount)

Return value: New reference. Replace at most *maxcount* occurrences of *substr* in *str* with *replstr* and return the resulting Unicode object. *maxcount* == -1 means replace all occurrences.

int **PyUnicode_Compare** (*PyObject* *left, *PyObject* *right)

Compare two strings and return -1, 0, 1 for less than, equal, and greater than, respectively.

This function returns -1 upon failure, so one should call *PyErr_Occurred()* to check for errors.

int **PyUnicode_CompareWithASCIIString** (*PyObject* *uni, const char *string)

Compare a Unicode object, *uni*, with *string* and return -1, 0, 1 for less than, equal, and greater than, respectively. It is best to pass only ASCII-encoded strings, but the function interprets the input string as ISO-8859-1 if it contains non-ASCII characters.

This function does not raise exceptions.

*PyObject** **PyUnicode_RichCompare** (*PyObject* *left, *PyObject* *right, int op)

Return value: New reference. 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)

Return value: New reference. Return a new string object from *format* and *args*; this is analogous to `format % args`.

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

Check whether *element* is contained in *container* and return true or false accordingly.

element has to coerce to a one element Unicode string. -1 is returned if there was an error.

void **PyUnicode_InternInPlace** (*PyObject* **string)

Intern the argument **string* in place. The argument must be the address of a pointer variable pointing to a Python Unicode string object. If there is an existing interned string that is the same as **string*, it sets **string* to it (decrementing the reference count of the old string object and incrementing the reference count of the interned string object), otherwise it leaves **string* alone and interns it (incrementing its reference count). (Clarification: even though there is a lot of talk about reference counts, think of this function as reference-count-neutral; you own the object after the call if and only if you owned it before the call.)

*PyObject** **PyUnicode_InternFromString** (const char *v)

Return value: New reference. A combination of *PyUnicode_FromString()* and *PyUnicode_InternInPlace()*, returning either a new Unicode string object that has been interned, or a new (“owned”) reference to an earlier interned string object with the same value.

8.3.4 Objeto tupla

PyTupleObject

Este subtipo de *PyObject* representa um objeto tupla em Python.

PyTypeObject **PyTuple_Type**

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)

Return value: New reference. Retorna um novo objeto tupla de tamanho *len*, ou NULL em caso de falha.

*PyObject** **PyTuple_Pack** (*Py_ssize_t* n, ...)

Return value: New reference. Retorna um novo objeto tupla de tamanho *n*, ou NULL 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)

Pega um ponteiro para um objeto tupla e retorna o tamanho dessa tupla.

Py_ssize_t **PyTuple_GET_SIZE** (*PyObject* *p)

Retorna o tamanho da tupla *p*, que deve ser diferente de NULL e apontar para uma tupla; nenhuma verificação de erro é executada.

*PyObject** **PyTuple_GetItem** (*PyObject* *p, *Py_ssize_t* pos)

Return value: Borrowed reference. Retorna o objeto na posição *pos* na tupla apontada por *p*. Se *pos* estiver fora dos limites, retorna NULL e define uma exceção `IndexError`.

*PyObject** **PyTuple_GET_ITEM** (*PyObject* *p, *Py_ssize_t* pos)

Return value: Borrowed reference. Como `PyTuple_GetItem()`, mas faz nenhuma verificação de seus argumentos.

*PyObject** **PyTuple_GetSlice** (*PyObject* *p, *Py_ssize_t* low, *Py_ssize_t* high)

Return value: New reference. Retorna a fatia da tupla apontada por *p* entre *low* (baixo) e *high* (alto), ou NULL em caso de falha. Este é o equivalente da expressão Python `p[low:high]`. A indexação do final da lista não é suportada.

int **PyTuple_SetItem** (*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

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.

Nota: Esta macro “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 assumir 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.

*PyObject** **PyStructSequence_NewType** (*PyStructSequence_Desc* *desc)

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

void **PyStructSequence_InitType** (*PyObject* *type, *PyStructSequence_Desc* *desc)

Inicializa um tipo de sequência de estrutura *type* de *desc* no lugar.

int **PyStructSequence_InitType2** (*PyObject* *type, *PyStructSequence_Desc* *desc)

O mesmo que *PyStructSequence_InitType*, mas retorna 0 em caso de sucesso e -1 em caso de falha.

Novo na versão 3.4.

PyStructSequence_Desc

Contém as metainformações de um tipo de sequência de estrutura a ser criado.

Campo	Tipo em C	Significado
name	const char *	nome do tipo sequência de estrutura
doc	const char *	ponteiro para docstring para o tipo ou NULL para omitir
fields	PyStructSequence_Field *	ponteiro para um vetor terminado em NULL com nomes de campos do novo tipo
n_in_sequence	int	número de campos visíveis para o lado Python (se usado como tupla)

PyStructSequence_Field

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.

Campo	Tipo em C	Significado
name	const char *	nome do campo ou NULL para terminar a lista de campos nomeados; definida para <i>PyStructSequence_UnnamedField</i> para deixar sem nome
doc	const char *	campo docstring ou NULL para omitir

const char * const **PyStructSequence_UnnamedField**

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** (*PyObject* *type)

Return value: *New reference.* Cria um instância de *type*, que deve ser criada com *PyStructSequence_NewType()*.

*PyObject** **PyStructSequence_GetItem** (*PyObject* *p, *Py_ssize_t* pos)

Return value: *Borrowed reference.* Retorna o objeto na posição *pos* na sequência de estrutura apontada por *p*. Nenhuma verificação de limites é executada.

*PyObject** **PyStructSequence_GET_ITEM** (*PyObject* *p, *Py_ssize_t* pos)

Return value: *Borrowed reference.* Macro equivalente de *PyStructSequence_GetItem()*.

void **PyStructSequence_SetItem** (*PyObject* **p*, *Py_ssize_t* *pos*, *PyObject* **o*)

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.

Nota: Esta função “rouba” uma referência a *o*.

void **PyStructSequence_SET_ITEM** (*PyObject* **p*, *Py_ssize_t* **pos*, *PyObject* **o*)

Macro equivalente de *PyStructSequence_SetItem()*.

Nota: Esta função “rouba” uma referência a *o*.

8.3.6 Objeto List

PyListObject

Este subtipo de *PyObject* representa um objeto de lista Python.

PyTypeObject **PyList_Type**

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*)

Return value: *New reference*. 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 de lista retornado 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()*.

Py_ssize_t **PyList_Size** (*PyObject* **list*)

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*)

Forma macro de *PyList_Size()* sem verificação de erros.

*PyObject** **PyList_GetItem** (*PyObject* **list*, *Py_ssize_t* *index*)

Return value: *Borrowed reference*. 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 ou >=len(list)), retorna NULL e levanta uma exceção *IndexError*.

*PyObject** **PyList_GET_ITEM** (*PyObject* **list*, *Py_ssize_t* *i*)

Return value: *Borrowed reference*. Forma macro de *PyList_GetItem()* sem verificação de erros.

int **PyList_SetItem** (*PyObject* **list*, *Py_ssize_t* *index*, *PyObject* **item*)

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.

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)

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)

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)

Return value: *New reference*. 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)

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_Sort** (*PyObject* *list)

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)

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)

Return value: *New reference*. Retorna um novo objeto tupla contendo os conteúdos de *list*; equivale a “tuple(list)”.

8.4 Coleções

8.4.1 Objetos dicionários

PyDictObject

Este subtipo do *PyObject* representa um objeto dicionário Python.

PyTypeObject **PyDict_Type**

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** ()

Return value: *New reference.* Retorna um novo dicionário vazio ou NULL em caso de falha.

*PyObject** **PyDictProxy_New** (*PyObject* *mapping)

Return value: *New reference.* 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)

Esvazia um dicionário existente de todos os pares chave-valor.

int **PyDict_Contains** (*PyObject* *p, *PyObject* *key)

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`.

*PyObject** **PyDict_Copy** (*PyObject* *p)

Return value: *New reference.* Retorna um novo dicionário que contém o mesmo chave-valor como *p*.

int **PyDict_SetItem** (*PyObject* *p, *PyObject* *key, *PyObject* *val)

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)

Insere *val* no dicionário *p* usando *key* como uma chave. *key* deve ser a `const char*`. O objeto chave é criado usando `PyUnicode_FromString(key)`. Retorna 0 em caso de sucesso ou -1 em caso de falha. Esta função *não* rouba uma referência a *val*.

int **PyDict_DelItem** (*PyObject* *p, *PyObject* *key)

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)

Remove a entrada no dicionário *p* que tem uma chave especificada pela string *key*. Se *key* não estiver no dicionário, `KeyError` é levantada. Retorna 0 em caso de sucesso ou -1 em caso de falha.

*PyObject** **PyDict_GetItem** (*PyObject* *p, *PyObject* *key)

Return value: *Borrowed reference.* Retorna o objeto do dicionário *p* que possui uma chave *key*. Retorna NULL se a chave *key* não estiver presente, mas *sem* definir uma exceção.

Observe que as exceções que ocorrem ao chamar os métodos `__hash__()` e `__eq__()` serão suprimidas. Para obter o relatório de erros, use `PyDict_GetItemWithError()`.

*PyObject** **PyDict_GetItemWithError** (*PyObject* *p, *PyObject* *key)

Return value: *Borrowed reference.* 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)

Return value: *Borrowed reference.* É o mesmo que `PyDict_GetItem()`, mas *key* é especificada como um `const char*`, em vez de um *PyObject**.

Observe que as exceções que ocorrem ao chamar os métodos `__hash__()` e `__eq__()` e criar um objeto string temporário serão suprimidas. Para obter o relatório de erros, use `PyDict_GetItemWithError()`.

*PyObject** **PyDict_SetDefault** (*PyObject* *p, *PyObject* *key, *PyObject* *defaultobj)

Return value: *Borrowed reference.* 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.

Novo na versão 3.4.

*PyObject** **PyDict_Items** (*PyObject* **p*)

Return value: *New reference.* Retorna um *PyListObject* contendo todos os itens do dicionário.

*PyObject** **PyDict_Keys** (*PyObject* **p*)

Return value: *New reference.* Retorna um *PyListObject* contendo todas as chaves do dicionário.

*PyObject** **PyDict_Values** (*PyObject* **p*)

Return value: *New reference.* Retorna um *PyListObject* contendo todos os valores do dicionário *p*.

Py_ssize_t **PyDict_Size** (*PyObject* **p*)

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*)

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)) {
    /* do something interesting with the values... */
    ...
}
```

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);
}
```

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

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*)

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*)

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
```

8.4.2 Objeto Set

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

PySetObject

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

PyTypeObject **PySet_Type**

Essa é uma instância de *PyTypeObject* representando o tipo Python set

PyTypeObject **PyFrozenSet_Type**

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 **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*)

Return value: New reference. 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*)

Return value: New reference. 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*)

Retorna o comprimento de um objeto `set` ou `frozenset`. Equivalente a `len(anyset)`. Levanta um `PyExc_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*)

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 hasháveis em `frozensets` temporários. Levanta um `TypeError` se a *key* não for hashável. Levanta `PyExc_SystemError` se *anyset* não é um `set`, `frozenset`, ou uma instância de um subtipo.

int **PySet_Add** (*PyObject* **set*, *PyObject* **key*)

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

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*)

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 hashável. Ao contrário do método Python `discard()`, esta função não converte automaticamente conjuntos não hasháveis em `frozensets` temporários. Levanta `PyExc_SystemError` se *set* não é uma instância de `set` ou seu subtipo.

*PyObject** **PySet_Pop** (*PyObject* **set*)

Return value: New reference. 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*)

Limpa todos os elementos de um conjunto existente

8.5 Objetos Função

8.5.1 Objetos Função

Existem algumas funções específicas para as funções do Python.

PyFunctionObject

A estrutura C usada para funções.

PyTypeObject **PyFunction_Type**

Esta é uma instância de *PyTypeObject* e representa o tipo de função Python. Está exposto aos programadores Python como `types.FunctionType`.

int PyFunction_Check (*PyObject* *o)

Retorna verdadeiro se *o* é um objeto de função (tem tipo *PyFunction_Type*). O parâmetro não deve ser NULL. Esta função sempre tem sucesso.

*PyObject** **PyFunction_New** (*PyObject* *code, *PyObject* *globals)

Return value: *New reference.* Retorna um novo objeto função associado ao código objeto *code*. *globals* deve ser um dicionário com as variáveis globais acessíveis à função.

A docstring e o nome da função são recuperados do objeto de código. `__module__` * é recuperado de *globals*. Os padrões de argumento, as anotações e o encerramento são definidos como NULL. `__qualname__` está definido para o mesmo valor que o nome da função.

*PyObject** **PyFunction_NewWithQualName** (*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)

Return value: *New reference.* Como *PyFunction_New()*, mas também permite configurar o atributo `__qualname__` do objeto da função. *qualname* deve ser um objeto unicode ou NULL; Se NULL, o atributo `__qualname__` é definido como o mesmo valor que o atributo `__name__`.

Novo na versão 3.3.

*PyObject** **PyFunction_GetCode** (*PyObject* *op)

Return value: *Borrowed reference.* Retorna o objeto de código associado ao objeto função *op*.

*PyObject** **PyFunction_GetGlobals** (*PyObject* *op)

Return value: *Borrowed reference.* Retorna o dicionário global associado ao objeto função *op*.

*PyObject** **PyFunction_GetModule** (*PyObject* *op)

Return value: *Borrowed reference.* Retorna o atributo `__module__` do objeto função *op*. Esta é normalmente uma string contendo o nome do módulo, mas pode ser configurada para qualquer outro objeto pelo código Python.

*PyObject** **PyFunction_GetDefaults** (*PyObject* *op)

Return value: *Borrowed reference.* Retorna o argumento os valores padrão do objeto função *op*. Isso pode ser uma tupla de argumentos ou NULL.

int PyFunction_SetDefaults (*PyObject* *op, *PyObject* *defaults)

Define o argumento valores padrão para o objeto função *op*. *defaults* deve ser `Py_None` ou uma tupla.

Levanta `SystemError` e retorna -1 em falha.

*PyObject** **PyFunction_GetClosure** (*PyObject* *op)

Return value: *Borrowed reference.* Retorna o fechamento associado ao objeto função *op*. Isso pode ser NULL ou uma tupla de objetos de 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 falha.

*PyObject** **PyFunction_GetAnnotations** (*PyObject** *op*)

Return value: Borrowed reference. 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 falha.

8.5.2 Objetos de Método de Instância

Um método de instância é um wrapper para um *PyCFunction* e a nova maneira de vincular um *PyCFunction* a um objeto de classe. Ele substitui a chamada anterior `PyMethod_New(func, NULL, class)`.

PyObject **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*)

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

*PyObject** **PyInstanceMethod_Function** (*PyObject** *im*)

Return value: Borrowed reference. Retorna o objeto função associado ao método de instância *im*.

*PyObject** **PyInstanceMethod_GET_FUNCTION** (*PyObject** *im*)

Return value: Borrowed reference. 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.

PyObject **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*)

Return value: New reference. 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*)

Return value: Borrowed reference. Retorna o objeto função associado ao método *meth*.

*PyObject** **PyMethod_GET_FUNCTION** (*PyObject** *meth*)

Return value: Borrowed reference. Versão macro de *PyMethod_Function()* que evita a verificação de erros.

*PyObject** **PyMethod_Self** (*PyObject** *meth*)

Return value: Borrowed reference. Retorna a instância associada com o método *meth*.

*PyObject** **PyMethod_GET_SELF** (*PyObject* *meth)

Return value: Borrowed reference. 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.

PyCellObject

A estrutura C usada para objetos célula.

PyTypeObject **PyCell_Type**

O objeto de tipo correspondente aos objetos célula.

int **PyCell_Check** (ob)

Retorna verdadeiro se *ob* for um objeto célula; *ob* não deve ser NULL. Esta função sempre tem sucesso.

*PyObject** **PyCell_New** (*PyObject* *ob)

Return value: New reference. Cria e retorna um novo objeto célula contendo o valor *ob*. O parâmetro pode ser NULL.

*PyObject** **PyCell_Get** (*PyObject* *cell)

Return value: New reference. Retorna o conteúdo da célula *cell*.

*PyObject** **PyCell_GET** (*PyObject* *cell)

Return value: Borrowed reference. 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; se não for um objeto célula, -1 será retornado. Em caso de sucesso, 0 será retornado.

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.

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 tipo Python code.

int **PyCode_Check** (*PyObject* *co)

Retorna verdadeiro se *co* for um objeto code. Esta função sempre tem sucesso.

int **PyCode_GetNumFree** (*PyCodeObject* *co)

Retorna o número de variáveis livres em *co*.

*PyCodeObject** **PyCode_New** (int *argcount*, int *kwnonlyargcount*, int *nlocals*, int *stacksize*, int *flags*, *PyObject* **code*, *PyObject* **consts*, *PyObject* **names*, *PyObject* **varnames*, *PyObject* **freevars*, *PyObject* **cellvars*, *PyObject* **filename*, *PyObject* **name*, int *firstlineno*, *PyObject* **notab*)

Return value: *New reference.* Retorna um novo objeto código. Se você precisa de um objeto código fictício para criar um quadro, use `PyCode_NewEmpty()` no caso. Chamar `PyCode_New()` diretamente pode vinculá-lo a uma versão precisa do Python, uma vez que a definição do bytecode muda frequentemente.

*PyCodeObject** **PyCode_NewWithPosOnlyArgs** (int *argcount*, int *posonlyargcount*, int *kwnonlyargcount*, int *nlocals*, int *stacksize*, int *flags*, *PyObject* **code*, *PyObject* **consts*, *PyObject* **names*, *PyObject* **varnames*, *PyObject* **freevars*, *PyObject* **cellvars*, *PyObject* **filename*, *PyObject* **name*, int *firstlineno*, *PyObject* **notab*)

Return value: *New reference.* Semelhante a `PyCode_New()`, mas com um “posonlyargcount” extra para argumentos apenas posicionais.

Novo na versão 3.8.

*PyCodeObject** **PyCode_NewEmpty** (const char **filename*, const char **funcname*, int *firstlineno*)

Return value: *New reference.* Retorna um novo objeto código vazio com o nome do arquivo especificado, o nome da função e o número da primeira linha. É ilegal executar `exec()` ou `eval()` no objeto código resultante.

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 wrappers C de conveniência 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*)

Return value: *New reference.* 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*)

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*)

Return value: *New reference.* 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 manipulador é uma função do tipo `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` com nenhum argumento.

Novo na versão 3.8.

int **PyFile_WriteObject** (`PyObject *obj`, `PyObject *p`, int *flags*)

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`)

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 Objeto Module

PyObject **PyModule_Type**

Esta instância de `PyObject` 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`)

Return value: *New reference.* Retorna um novo objeto de módulo com o atributo `__name__` definido como **nome**. Os atributos de módulo `:attr: '__name__':attr: '__doc__':attr: '__package__'` e `__loader__` são preenchidos (todos exceto `__name__` são definidos como None); O caller é responsável por providenciar um atributo `__file__`.

Novo na versão 3.3.

Alterado na versão 3.4: `__package__` e `__loader__` são definidos como None.

*PyObject** **PyModule_New** (const char *name)

Return value: New reference. 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)

Return value: Borrowed reference. Return the dictionary object that implements *module*'s namespace; this object is the same as the `__dict__` attribute of the module object. If *module* is not a module object (or a subtype of a module object), `SystemError` is raised and `NULL` is returned.

It is recommended extensions use other *PyModule_**() and *PyObject_**() functions rather than directly manipulate a module's `__dict__`.

*PyObject** **PyModule_GetNameObject** (*PyObject* *module)

Return value: New reference. Return *module*'s `__name__` value. If the module does not provide one, or if it is not a string, `SystemError` is raised and `NULL` is returned.

Novo na versão 3.3.

const char* **PyModule_GetName** (*PyObject* *module)

Semelhante a *PyModule_GetNameObject()* mas retorna o nome codificado em 'utf-8'

void* **PyModule_GetState** (*PyObject* *module)

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)

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)

Return value: New reference. Retorna o nome do arquivo do qual o *módulo* foi carregado usando o atributo `__file__` do *módulo*. 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.

Novo na versão 3.2.

const char* **PyModule_GetFilename** (*PyObject* *module)

Semelhante a *PyModule_GetFilenameObject()* mas retorna o nome do arquivo codificado em 'utf-8'.

Obsoleto desde a versão 3.2: *PyModule_GetFilename()* raises `UnicodeEncodeError` on unencodable filenames, use *PyModule_GetFilenameObject()* instead.

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.

PyModuleDef

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 for the module; usually a docstring variable created with `PyDoc_STRVAR` is used.

`Py_ssize_t m_size`

Module state may be kept in a per-module memory area that can be retrieved with `PyModule_GetState()`, rather than in static globals. This makes modules safe for use in multiple sub-interpreters.

This memory area is allocated based on `m_size` on module creation, and freed when the module object is deallocated, after the `m_free` function has been called, if present.

Setting `m_size` to `-1` means that the module does not support sub-interpreters, because it has global state.

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`

A pointer to a table of module-level functions, described by `PyMethodDef` values. Can be `NULL` if no functions are present.

`PyModuleDef_Slot* m_slots`

An array of slot definitions for multi-phase initialization, terminated by a `{0, NULL}` entry. When using single-phase initialization, `m_slots` must be `NULL`.

Alterado na versão 3.5: Prior to version 3.5, this member was always set to `NULL`, and was defined as:

inquiry `m_reload`

traverseproc `m_traverse`

A traversal function to call during GC traversal of the module object, or `NULL` if not needed.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (`Py_mod_exec` function). More precisely, this function is not called if `m_size` is greater than 0 and the module state (as returned by `PyModule_GetState()`) is `NULL`.

Alterado na versão 3.9: Não é mais chamado antes que o estado do módulo seja alocado.

inquiry `m_clear`

A clear function to call during GC clearing of the module object, or `NULL` if not needed.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (`Py_mod_exec` function). More precisely, this function is not called if `m_size` is greater than 0 and the module state (as returned by `PyModule_GetState()`) is `NULL`.

Like `PyTypeObject.tp_clear`, this function is not *always* called before a module is deallocated. For example, when reference counting is enough to determine that an object is no longer used, the cyclic garbage collector is not involved and `m_free` is called directly.

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.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (`Py_mod_exec` function). More precisely, this function is not called if `m_size` is greater than 0 and the module state (as returned by `PyModule_GetState()`) is `NULL`.

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)

Return value: *New reference.* 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*)

Return value: *New reference.* Create a new module object, given the definition in *def*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

Nota: A maioria dos usos dessa função deve ser feita com: `c:func: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_AddObject()*.

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)

Return value: *Borrowed reference.* 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.

Novo na versão 3.5.

The *m_slots* member of the module definition must point to an array of *PyModuleDef_Slot* structures:

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.

Novo 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`.

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*)

Return value: *New reference.* Create a new module object, given the definition in *module* and the `ModuleSpec` *spec*. This behaves like `PyModule_FromDefAndSpec2()` with `module_api_version` set to `PYTHON_API_VERSION`.

Novo na versão 3.5.

PyObject * **PyModule_FromDefAndSpec2** (*PyModuleDef* **def*, *PyObject* **spec*, int *module_api_version*)

Return value: New reference. Create a new module object, given the definition in *module* and the ModuleSpec *spec*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a RuntimeWarning is emitted.

Nota: Most uses of this function should be using `PyModule_FromDefAndSpec()` instead; only use this if you are sure you need it.

Novo na versão 3.5.

int **PyModule_ExecDef** (*PyObject* **module*, *PyModuleDef* **def*)

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

Novo na versão 3.5.

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

Set the docstring for *module* to *docstring*. This function is called automatically when creating a module from *PyModuleDef*, using either `PyModule_Create` or `PyModule_FromDefAndSpec`.

Novo na versão 3.5.

int **PyModule_AddFunctions** (*PyObject* **module*, *PyMethodDef* **functions*)

Add the functions from the NULL terminated *functions* array to *module*. Refer to the *PyMethodDef* documentation for details on individual entries (due to the lack of a shared module namespace, module level “functions” implemented in C typically receive the module as their first parameter, making them similar to instance methods on Python classes). This function is called automatically when creating a module from *PyModuleDef*, using either `PyModule_Create` or `PyModule_FromDefAndSpec`.

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

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

Nota: Unlike other functions that steal references, `PyModule_AddObject()` only decrements the reference count of *value* on success.

This means that its return value must be checked, and calling code must `Py_DECREF()` *value* manually on error. Example usage:

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

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

Add an integer constant to *module* as *name*. This convenience function can be used from the module’s initialization function. Return `-1` on error, `0` on success.

int **PyModule_AddStringConstant** (*PyObject* *module, const char *name, const char *value)

Add a string constant to *module* as *name*. This convenience function can be used from the module's initialization function. The string *value* must be NULL-terminated. Return -1 on error, 0 on success.

int **PyModule_AddIntMacro** (*PyObject* *module, macro)

Add an int constant to *module*. The name and the value are taken from *macro*. For example `PyModule_AddIntMacro(module, AF_INET)` adds the int constant `AF_INET` with the value of `AF_INET` to *module*. Return -1 on error, 0 on success.

int **PyModule_AddStringMacro** (*PyObject* *module, macro)

Add a string constant to *module*.

int **PyModule_AddType** (*PyObject* *module, *PyTypeObject* *type)

Add a type object to *module*. The type object is finalized by calling internally `PyType_Ready()`. The name of the type object is taken from the last component of *tp_name* after dot. Return -1 on error, 0 on success.

Novo na versão 3.9.

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)

Return value: Borrowed reference. Returns the module object that was created from *def* for the current interpreter. This method requires that the module object has been attached to the interpreter state with `PyState_AddModule()` beforehand. In case the corresponding module object is not found or has not been attached to the interpreter state yet, it returns NULL.

int **PyState_AddModule** (*PyObject* *module, *PyModuleDef* *def)

Attaches the module object passed to the function to the interpreter state. This allows the module object to be accessible via `PyState_FindModule()`.

Only effective on modules created using single-phase initialization.

Python calls `PyState_AddModule` automatically after importing a module, so it is unnecessary (but harmless) to call it from module initialization code. An explicit call is needed only if the module's own init code subsequently calls `PyState_FindModule`. The function is mainly intended for implementing alternative import mechanisms (either by calling it directly, or by referring to its implementation for details of the required state updates).

The caller must hold the GIL.

Return 0 on success or -1 on failure.

Novo na versão 3.3.

int **PyState_RemoveModule** (*PyModuleDef* *def)

Removes the module object created from *def* from the interpreter state. Return 0 on success or -1 on failure.

The caller must hold the GIL.

Novo 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.

PyObject **PySeqIter_Type**

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** (op)

Retorna true se o tipo de *op* for *PySeqIter_Type*. Esta função sempre é bem-sucedida.

*PyObject** **PySeqIter_New** (*PyObject* *seq)

Return value: New reference. Retorna um iterador que funcione com um objeto de sequência geral, *seq*. A iteração termina quando a sequência levanta *IndexError* para a operação de assinatura.

PyObject **PyCallIter_Type**

Objeto de tipo para objetos iteradores retornados por *PyCallIter_New()* e a forma de dois argumentos da função embutida *iter()*.

int **PyCallIter_Check** (op)

Retorna true se o tipo de *op* for *PyCallIter_Type*. Esta função sempre é bem-sucedida.

*PyObject** **PyCallIter_New** (*PyObject* *callable, *PyObject* *sentinel)

Return value: New reference. Retorna um novo iterador. O primeiro parâmetro, *callable*, pode ser qualquer objeto chamável do Python que possa ser chamado sem parâmetros; cada chamada deve retornar o próximo item na iteração. Quando *callable* retorna um valor igual a *sentinel*, a iteração será encerrada.

8.6.4 Objetos Descritores

“Descritores” são objetos que descrevem algum atributo de um objeto. Eles são encontrados no dicionário de objetos de tipo.

PyObject **PyProperty_Type**

O tipo de objeto para os tipos de descritores embutidos.

*PyObject** **PyDescr_NewGetSet** (*PyObject* *type, struct *PyGetSetDef* *getset)

Return value: New reference.

*PyObject** **PyDescr_NewMember** (*PyObject* *type, struct *PyMemberDef* *meth)

Return value: New reference.

*PyObject** **PyDescr_NewMethod** (*PyObject* *type, struct *PyMethodDef* *meth)

Return value: New reference.

*PyObject** **PyDescr_NewWrapper** (*PyObject* *type, struct wrapperbase *wrapper, void *wrapped)

Return value: New reference.

*PyObject** **PyDescr_NewClassMethod** (*PyObject* *type, *PyMethodDef* *method)

Return value: New reference.

int **PyDescr_IsData** (*PyObject* *descr)

Retorna True se os objetos descritores *descr* descrevem um atributo de dados, ou False se os mesmos descrevem um método. *descr* deve ser um objeto descritor; não há verificação de erros.

*PyObject** **PyWrapper_New** (*PyObject* *, *PyObject* *)

Return value: New reference.

8.6.5 Objetos Slice

PyObject **PySlice_Type**

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)

Return value: *New reference.* 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 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)

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)

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 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) {  
    // return error  
}
```

substituído por

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {  
    // return error  
}  
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.

Obsoleto 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)`

Extraí 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 em caso de erro, 0 em caso de sucesso.

Novo 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)`

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.

Novo na versão 3.6.1.

8.6.6 Objeto Ellipsis

`PyObject *Py_Ellipsis`

O objeto Python `Ellipsis`. Este objeto não possui métodos. Ele precisa ser tratado como qualquer outro objeto no que diz respeito às contagens de referências. Como `Py_None`, é um objeto singleton.

8.6.7 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)`

Return value: *New reference.* 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.

`PyObject *PyMemoryView_FromMemory (char *mem, Py_ssize_t size, int flags)`

Return value: *New reference.* Cria um objeto `memoryview` usando *mem* como o buffer subjacente. *flags* pode ser um dos seguintes `PYBUF_READ` ou `PYBUF_WRITE`.

Novo na versão 3.3.

`PyObject *PyMemoryView_FromBuffer (Py_buffer *view)`

Return value: *New reference.* 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)`

Return value: *New reference.* Cria um objeto `memoryview` para um pedaço *contíguo* de memória (na ordem 'C' ou 'Fortran', 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.

`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.

Py_buffer ***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*.

8.6.8 Objetos de 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** (ob)

Retorna verdadeiro se *ob* for um objeto referência ou um objeto intermediário. Esta função sempre tem sucesso.

int **PyWeakref_CheckRef** (ob)

Retorna verdadeiro se *ob* for um objeto referência. Esta função sempre tem sucesso.

int **PyWeakref_CheckProxy** (ob)

Retorna verdadeiro se *ob* for um objeto intermediário. Esta função sempre tem sucesso.

*PyObject** **PyWeakref_NewRef** (*PyObject* *ob, *PyObject* *callback)

Return value: New reference. 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)

Return value: New reference. Retorna um objeto de proxy 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 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 fracamente referenciável, ou se *callback* não for um chamável, None, ou NULL, isso retornará NULL e levantará a *TypeError*.

*PyObject** **PyWeakref_GetObject** (*PyObject* *ref)

Return value: Borrowed reference. Retorna o objeto referenciado de uma referência fraca, *ref*. Se o referente não estiver mais em tempo real, retorna *Py_None*.

Nota: Esta função retorna **referência emprestada** ao objeto referenciado. Isso significa que você deve sempre chamar *Py_INCREF()* no objeto, exceto se você souber que não pode ser destruído enquanto você ainda está usando.

*PyObject** **PyWeakref_GET_OBJECT** (*PyObject* *ref)

Return value: Borrowed reference. Semelhante a *PyWeakref_GetObject()*, mas implementado como uma macro que não verifica erros.

8.6.9 Capsules

Consulte `using-capsules` para obter mais informações sobre o uso desses objetos.

Novo na versão 3.1.

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.

PyCapsule_Destructor

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*)

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

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*)

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*)

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*)

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*)

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)`

Importa um ponteiro para um objeto C de um atributo capsule em um módulo. O parâmetro *name* deve especificar o nome completo do atributo, como em `module.attribute`. O nome armazenado na cápsula deve corresponder exatamente a essa sequência. Se *no_block* for verdadeiro, importa o módulo sem bloquear (usando `PyImport_ImportModuleNoBlock()`). Se *no_block* for falso, importa o módulo convencionalmente (usando `PyImport_ImportModule()`).

Retorna o ponteiro interno *pointer* da cápsula com sucesso. Em caso de falha, define uma exceção e retorna NULL.

`int PyCapsule_IsValid (PyObject *capsule, const char *name)`

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)`

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)`

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)`

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)`

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.

8.6.10 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()`.

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)

Return value: *New reference.* 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)

Return value: *New reference.* 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

Novo na versão 3.5.

Os objetos corrotina são aquelas funções declaradas com um retorno de palavra-chave `async`.

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)

Return value: *New reference.* 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

Nota: Alterado na versão 3.7.1: 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:

```
// in 3.7.0:
PyContext *PyContext_New(void);

// in 3.7.1+:
PyObject *PyContext_New(void);
```

Veja [bpo-34762](#) para mais detalhes.

Novo na versão 3.7.

Esta seção detalha a API C pública para o módulo `contextvars`.

PyContext

A estrutura C usada para representar um objeto `contextvars.Context`.

PyContextVar

A estrutura C usada para representar um objeto `contextvars.ContextVar`.

PyContextToken

A estrutura C usada para representar um objeto `contextvars.Token`

***PyObject* PyContext_Type**

O objeto de tipo que representa o tipo de *contexto*.

***PyObject* PyContextVar_Type**

O objeto de tipo que representa o tipo de *variável de contexto*.

***PyObject* 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)

Return value: New reference. Cria um novo objeto de contexto vazio. Retorna NULL se um erro ocorreu.

PyObject ***PyContext_Copy** (*PyObject* *ctx)

Return value: New reference. Cria uma cópia rasa do objeto de contexto *ctx* passado. Retorna NULL se um erro ocorreu.

PyObject ***PyContext_CopyCurrent** (void)

Return value: New reference. Cria uma cópia rasa do contexto da thread atual. Retorna NULL se um erro ocorreu.

int **PyContext_Enter** (*PyObject* *ctx)

Define *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)

Return value: New reference. 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)

Return value: New reference. 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.

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`.

Novo na versão 3.7.

Macros de verificação de tipo:

int PyDate_Check (*PyObject* *ob)

Retorna true 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 true se *ob* for do tipo `PyDateTime_DateType`. *ob* não deve ser NULL. Esta função sempre tem sucesso.

int PyDateTime_Check (*PyObject* *ob)

Retorna true 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 true se *ob* for do tipo `PyDateTime_DateTimeType`. *ob* não deve ser NULL. Esta função sempre tem sucesso.

int PyTime_Check (*PyObject* *ob)

Retorna true 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 true se *ob* for do tipo `PyDateTime_TimeType`. *ob* não deve ser NULL. Esta função sempre tem sucesso.

int PyDelta_Check (*PyObject* *ob)

Retorna true se *ob* é do tipo `PyDateTime_DeltaType` ou um subtipo de `PyDateTime_DeltaType`. *ob* não deve ser NULL. Esta função sempre tem sucesso.

int PyDelta_CheckExact (*PyObject* *ob)

Retorna true se *ob* for do tipo `PyDateTime_DeltaType`. *ob* não deve ser NULL. Esta função sempre tem sucesso.

int PyTZInfo_Check (*PyObject* *ob)

Retorna true se *ob* é do tipo `PyDateTime_TZInfoType` ou um subtipo de `PyDateTime_TZInfoType`. *ob* não deve ser NULL. Esta função sempre tem sucesso.

int **PyTZInfo_CheckExact** (*PyObject* **ob*)

Retorna true 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*)

Return value: *New reference*. 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*)

Return value: *New reference*. 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*)

Return value: *New reference*. Retorna um objeto `datetime.datetime` com o ano, mês, dia, hora, minuto, segundo, microssegundo e a dobra especificados.

Novo na versão 3.6.

*PyObject** **PyTime_FromTime** (int *hour*, int *minute*, int *second*, int *usecond*)

Return value: *New reference*. 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*)

Return value: *New reference*. Retorna um objeto `datetime.time` com a hora, minuto, segundo, microssegundo e a dobra especificados.

Novo na versão 3.6.

*PyObject** **PyDelta_FromDSU** (int *days*, int *seconds*, int *useconds*)

Return value: *New reference*. 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** (*PyDateTime_DeltaType** *offset*)

Return value: *New reference*. Retorna um objeto `datetime.timezone` com um deslocamento fixo sem nome representado pelo argumento *offset*.

Novo na versão 3.7.

*PyObject** **PyTimeZone_FromOffsetAndName** (*PyDateTime_DeltaType** *offset*, *PyUnicode** *name*)

Return value: *New reference*. Retorna um objeto `datetime.timezone` com um deslocamento fixo representado pelo argumento *offset* e com *tzname name*.

Novo na versão 3.7.

Macros para extrair campos de objetos de data. O argumento deve ser uma instância de `PyDateTime_Date`, incluindo subclasses (como `PyDateTime_DateTime`). O argumento não deve ser NULL e o tipo não está marcado:

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 de data e hora. O argumento deve ser uma instância de `PyDateTime_DateTime`, incluindo 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)

Return the fold, as an int from 0 through 1.

Novo na versão 3.6.

Macros para extrair campos de objetos de tempo. O argumento deve ser uma instância de `PyDateTime_Time`, incluindo 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)

Return the fold, as an int from 0 through 1.

Novo na versão 3.6.

Macros para extrair campos de objetos time delta. O argumento deve ser uma instância de `PyDateTime_Delta`, incluindo subclasses. O argumento não deve ser `NULL`, e o tipo não é checado:

int PyDateTime_DELTA_GET_DAYS (PyDateTime_Delta *o)

Retorna o número de dias, como um inteiro de -999999999 a 999999999.

Novo na versão 3.3.

int PyDateTime_DELTA_GET_SECONDS (PyDateTime_Delta *o)

Retorna o número de segundos, como um inteiro de 0 a 86399.

Novo na versão 3.3.

int PyDateTime_DELTA_GET_MICROSECONDS (PyDateTime_Delta *o)

Retorna o número de microssegundos, como um inteiro de 0 a 999999.

Novo na versão 3.3.

Macros para a conveniência de módulos implementando a API de DB:

*PyObject** **PyDateTime_FromTimestamp** (*PyObject* *args)

Return value: *New reference.* Cria e retorna um novo objeto `datetime.datetime`, com uma tupla de argumentos adequada para passar para `datetime.datetime.fromtimestamp()`.

*PyObject** **PyDate_FromTimestamp** (*PyObject* *args)

Return value: *New reference.* Cria e retorna um novo objeto `datetime.date`, com uma tupla de argumentos adequada para passar para `datetime.date.fromtimestamp()`.

8.6.14 Objetos de indicação de tipos

Vários tipos embutidos para indicação de tipos são fornecidos. Apenas `GenericAlias` é exposto a C.

*PyObject** **Py_GenericAlias** (*PyObject* **origin*, *PyObject* **args*)

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[] = {
    // Other methods.
    ...
    {"__class_getitem__", (PyCFunction)Py_GenericAlias, METH_O|METH_CLASS, "See_
↪ PEP 585"}
    ...
}
```

Ver também:

O método de modelo de dados `__class_getitem__()`.

Novo na versão 3.9.

PyTypeObject **Py_GenericAliasType**

O tipo C do objeto retornado por `Py_GenericAlias()`. Equivalente a `types.GenericAlias` no Python.

Novo na versão 3.9.

Inicialização, Finalização e Threads

Consulte também *Configuração de Inicialização do Python*.

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 de Configuração
 - `PyImport_AppendInittab()`
 - `PyImport_ExtendInittab()`
 - `PyInitFrozenExtensions()`
 - `PyMem_SetAllocator()`
 - `PyMem_SetupDebugHooks()`
 - `PyObject_SetArenaAllocator()`
 - `Py_SetPath()`
 - `Py_SetProgramName()`
 - `Py_SetPythonHome()`
 - `Py_SetStandardStreamEncoding()`
 - `PySys_AddWarnOption()`
 - `PySys_AddXOption()`
 - `PySys_ResetWarnOptions()`
- Funções Informativas:

- `Py_IsInitialized()`
- `PyMem_GetAllocator()`
- `PyObject_GetArenaAllocator()`
- `Py_GetBuildInfo()`
- `Py_GetCompiler()`
- `Py_GetCopyright()`
- `Py_GetPlatform()`
- `Py_GetVersion()`

- Utilitários:

- `Py_DecodeLocale()`

- Alocadores de memória:

- `PyMem_RawMalloc()`
 - `PyMem_RawRealloc()`
 - `PyMem_RawCalloc()`
 - `PyMem_RawFree()`

Nota: As seguintes funções **não devem ser chamadas** antes `Py_Initialize()`: `Py_EncodeLocale()`, `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, `Py_GetProgramFullPath()`, `Py_GetPythonHome()`, `Py_GetProgramName()` e `PyEval_InitThreads()`.

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`

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`.

int `Py_DebugFlag`

Ativa a saída de depuração do analisador sintático (somente para especialistas, dependendo das opções de compilação).

Definida pela opção `-d` e a variável de ambiente `PYTHONDEBUG`.

int `Py_DontWriteBytecodeFlag`

Se definido 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`.

int `Py_FrozenFlag`

Suprime mensagens de erro ao calcular o caminho de pesquisa do módulo em `Py_GetPath()`.

Sinalizador privado usado pelos programas `_freeze_importlib` e `frozenmain`.

int `Py_HashRandomizationFlag`

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.

int `Py_IgnoreEnvironmentFlag`

Ignora todas as variáveis de ambiente `PYTHON*`, por exemplo `PYTHONPATH` e `PYTHONHOME`, que pode ser definido.

Definida pelas opções `-E` e `-I`.

int `Py_InspectFlag`

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`.

int `Py_InteractiveFlag`

Definida pela opção `-i`.

int `Py_IsolatedFlag`

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`.

Novo na versão 3.4.

int `Py_LegacyWindowsFSEncodingFlag`

If the flag is non-zero, use the `mbcs` encoding instead of the UTF-8 encoding for the filesystem encoding.

Definida como 1 se a variável de ambiente `PYTHONLEGACYWINDOWSFSENCODING` estiver definida como uma string não vazia.

Veja [PEP 529](#) para mais detalhes.

Disponibilidade: Windows.

int `Py_LegacyWindowsStdioFlag`

Se o sinalizador for diferente de zero, usa `io.FileIO` em vez de `WindowsConsoleIO` para fluxos padrão `sys`.

Definida como 1 se a variável de ambiente `PYTHONLEGACYWINDOWSSTDIO` estiver definida como uma string não vazia.

Veja [PEP 528](#) para mais detalhes.

Disponibilidade: Windows.

int `Py_NoSiteFlag`

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`.

int `Py_NoUserSiteDirectory`

Não adiciona o diretório `site-packages` de usuário a `sys.path`.

Definida pelas opções `-s` e `-I`, e pela variável de ambiente `PYTHONNOUSERSITE`.

int `Py_OptimizeFlag`

Definida pela opção `-O` e pela variável de ambiente `PYTHONOPTIMIZE`.

int `Py_QuietFlag`

Não exibe as mensagens de copyright e de versão nem mesmo no modo interativo.

Definida pela opção `-q`.

Novo na versão 3.2.

int `Py_UnbufferedStdioFlag`

Força os fluxos `stdout` e `stderr` a não serem armazenados em buffer.

Definida pela opção `-u` e pela variável de ambiente `PYTHONUNBUFFERED`.

int `Py_VerboseFlag`

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 opção `-v` e a variável de ambiente `PYTHONVERBOSE`.

9.3 Inicializando e encerrando o interpretador

void `Py_Initialize()`

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.

This initializes the table of loaded modules (`sys.modules`), and creates the fundamental modules `builtins`, `__main__` and `sys`. It also initializes the module search path (`sys.path`). It does not set `sys.argv`; use `PySys_SetArgvEx()` for that. This is a no-op when called for a second time (without calling `Py_FinalizeEx()` first). There is no return value; it is a fatal error if the initialization fails.

Nota: On Windows, changes the console mode from `O_TEXT` to `O_BINARY`, which will also affect non-Python uses of the console using the C Runtime.

void `Py_InitializeEx(int initsigs)`

This function works like `Py_Initialize()` if `initsigs` is 1. If `initsigs` is 0, it skips initialization registration of signal handlers, which might be useful when Python is embedded.

int `Py_IsInitialized()`

Return true (nonzero) when the Python interpreter has been initialized, false (zero) if not. After `Py_FinalizeEx()` is called, this returns false until `Py_Initialize()` is called again.

int `Py_FinalizeEx()`

Undo all initializations made by `Py_Initialize()` and subsequent use of Python/C API functions, and destroy all sub-interpreters (see `Py_NewInterpreter()` below) that were created and not yet destroyed since the last call to `Py_Initialize()`. Ideally, this frees all memory allocated by the Python interpreter. This is a no-op when called for a second time (without calling `Py_Initialize()` again first). Normally the return value is 0. If there were errors during finalization (flushing buffered data), `-1` is returned.

This function is provided for a number of reasons. An embedding application might want to restart Python without having to restart the application itself. An application that has loaded the Python interpreter from a dynamically loadable library (or DLL) might want to free all memory allocated by Python before unloading the DLL. During

a hunt for memory leaks in an application a developer might want to free all memory allocated by Python before exiting from the application.

Bugs and caveats: The destruction of modules and objects in modules is done in random order; this may cause destructors (`__del__()` methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once; this can happen if an application calls `Py_Initialize()` and `Py_FinalizeEx()` more than once.

Raises an auditing event `cpython._PySys_ClearAuditHooks` with no arguments.

Novo na versão 3.6.

void **Py_Finalize()**

This is a backwards-compatible version of `Py_FinalizeEx()` that disregards the return value.

9.4 Process-wide parameters

int **Py_SetStandardStreamEncoding**(const char **encoding*, const char **errors*)

This function should be called before `Py_Initialize()`, if it is called at all. It specifies which encoding and error handling to use with standard IO, with the same meanings as in `str.encode()`.

It overrides `PYTHONIOENCODING` values, and allows embedding code to control IO encoding when the environment variable does not work.

encoding and/or *errors* may be `NULL` to use `PYTHONIOENCODING` and/or default values (depending on other settings).

Note that `sys.stderr` always uses the “backslashreplace” error handler, regardless of this (or any other) setting.

If `Py_FinalizeEx()` is called, this function will need to be called again in order to affect subsequent calls to `Py_Initialize()`.

Returns 0 if successful, a nonzero value on error (e.g. calling after the interpreter has already been initialized).

Novo na versão 3.4.

void **Py_SetProgramName**(const wchar_t **name*)

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.

Utiliza `Py_DecodeLocale()` para decodificar uma string de bytes para obter uma string tipo `wchar_t*`.

wchar_t* **Py_GetProgramName**()

Return the program name set with `Py_SetProgramName()`, or the default. The returned string points into static storage; the caller should not modify its value.

wchar_t* **Py_GetPrefix**()

Return the *prefix* for installed platform-independent files. This is derived through a number of complicated rules from the program name set with `Py_SetProgramName()` and some environment variables; for example, if the program name is `'/usr/local/bin/python'`, the prefix is `'/usr/local'`. The returned string points

into static storage; the caller should not modify its value. This corresponds to the `prefix` variable in the top-level Makefile and the `--prefix` argument to the `configure` script at build time. The value is available to Python code as `sys.prefix`. It is only useful on Unix. See also the next function.

wchar_t* **Py_GetExecPrefix** ()

Return the *exec-prefix* for installed platform-dependent files. This is derived through a number of complicated rules from the program name set with `Py_SetProgramName()` and some environment variables; for example, if the program name is `"/usr/local/bin/python"`, the *exec-prefix* is `"/usr/local"`. The returned string points into static storage; the caller should not modify its value. This corresponds to the `exec_prefix` variable in the top-level Makefile and the `--exec-prefix` argument to the `configure` script at build time. The value is available to Python code as `sys.exec_prefix`. It is only useful on Unix.

Background: The *exec-prefix* differs from the *prefix* when platform dependent files (such as executables and shared libraries) are installed in a different directory tree. In a typical installation, platform dependent files may be installed in the `/usr/local/plat` subtree while platform independent may be installed in `/usr/local`.

Generally speaking, a platform is a combination of hardware and software families, e.g. Sparc machines running the Solaris 2.x operating system are considered the same platform, but Intel machines running Solaris 2.x are another platform, and Intel machines running Linux are yet another platform. Different major revisions of the same operating system generally also form different platforms. Non-Unix operating systems are a different story; the installation strategies on those systems are so different that the *prefix* and *exec-prefix* are meaningless, and set to the empty string. Note that compiled Python bytecode files are platform independent (but not independent from the Python version by which they were compiled!).

System administrators will know how to configure the `mount` or `automount` programs to share `/usr/local` between platforms while having `/usr/local/plat` be a different filesystem for each platform.

wchar_t* **Py_GetProgramFullPath** ()

Return the full program name of the Python executable; this is computed as a side-effect of deriving the default module search path from the program name (set by `Py_SetProgramName()` above). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.executable`.

wchar_t* **Py_GetPath** ()

Return the default module search path; this is computed from the program name (set by `Py_SetProgramName()` above) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is `:` on Unix and macOS, `;` on Windows. The returned string points into static storage; the caller should not modify its value. The list `sys.path` is initialized with this value on interpreter startup; it can be (and usually is) modified later to change the search path for loading modules.

void **Py_SetPath** (const wchar_t *)

Set the default module search path. If this function is called before `Py_Initialize()`, then `Py_GetPath()` won't attempt to compute a default search path but uses the one provided instead. This is useful if Python is embedded by an application that has full knowledge of the location of all modules. The path components should be separated by the platform dependent delimiter character, which is `:` on Unix and macOS, `;` on Windows.

This also causes `sys.executable` to be set to the program full path (see `Py_GetProgramFullPath()`) and for `sys.prefix` and `sys.exec_prefix` to be empty. It is up to the caller to modify these if required after calling `Py_Initialize()`.

Utiliza `Py_DecodeLocale()` para decodificar uma string de bytes para obter uma string tipo `wchar_t*`.

O argumento caminho é copiado internamente, então o chamador pode liberá-lo depois da finalização da chamada.

Alterado na versão 3.8: O caminho completo do programa agora é utilizado para `sys.executable`, em vez do nome do programa.

const char* **Py_GetVersion** ()

Retorna a versão 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`.

const char* **Py_GetPlatform** ()

Return the platform identifier for the current platform. On Unix, this is formed from the “official” name of the operating system, converted to lower case, followed by the major revision number; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is 'sunos5'. On macOS, it is 'darwin'. On Windows, it is 'win'. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.platform`.

const char* **Py_GetCopyright** ()

Retorna a string oficial de direitos autorais 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** ()

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** ()

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*)

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 (".").

Utiliza `Py_DecodeLocale()` para decodificar uma string de bytes para obter uma string tipo `wchar_t*`.

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");
```

Novo na versão 3.1.3.

void **PySys_SetArgv** (int *argc*, wchar_t ***argv*)

This function works like `PySys_SetArgvEx()` with `updatepath` set to 1 unless the **python** interpreter was started with the `-I`.

Utiliza `Py_DecodeLocale()` para decodificar uma string de bytes para obter uma string tipo `wchar_*`.

Alterado na versão 3.4: The `updatepath` value depends on `-I`.

void **Py_SetPythonHome** (const wchar_t **home*)

Set the default “home” directory, that is, the location of the standard Python libraries. See `PYTHONHOME` for the meaning of the argument string.

The argument should point to a zero-terminated character string in static storage whose contents will not change for the duration of the program’s execution. No code in the Python interpreter will change the contents of this storage.

Utiliza `Py_DecodeLocale()` para decodificar uma string de bytes para obter uma string tipo `wchar_*`.

w_char* **Py_GetPythonHome** ()

Return the default “home”, that is, the value set by a previous call to `Py_SetPythonHome()`, or the value of the `PYTHONHOME` environment variable if it is set.

9.5 Thread State and the Global Interpreter Lock

The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there’s a global lock, called the *global interpreter lock* or *GIL*, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

Therefore, the rule exists that only the thread that has acquired the *GIL* may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads (see `sys.setswitchinterval()`). The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.

The Python interpreter keeps some thread-specific bookkeeping information inside a data structure called *PyThreadState*. There’s also one global variable pointing to the current *PyThreadState*: it can be retrieved using `PyThreadState_Get()`.

9.5.1 Releasing the GIL from extension code

A maioria dos códigos de extensão que manipulam o *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
```

The `Py_BEGIN_ALLOW_THREADS` macro opens a new block and declares a hidden local variable; the `Py_END_ALLOW_THREADS` macro closes the block.

The block above expands to the following code:

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

Here is how these functions work: the global interpreter lock is used to protect the pointer to the current thread state. When releasing the lock and saving the thread state, the current thread state pointer must be retrieved before the lock is released (since another thread could immediately acquire the lock and store its own thread state in the global variable). Conversely, when acquiring the lock and restoring the thread state, the lock must be acquired before storing the thread state pointer.

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:

PyInterpreterState

This data structure represents the state shared by a number of cooperating threads. Threads belonging to the same interpreter share their module administration and a few other internal items. There are no public members in this structure.

Threads belonging to different interpreters initially share nothing, except process state like available memory, open file descriptors and such. The global interpreter lock is also shared by all threads, regardless of to which interpreter they belong.

PyThreadState

Esta estrutura de dados representa o estado de uma tarefa. O único membro de dados público é `interp` (`PyInterpreterState *`), que aponta para o estado do interpretador desta tarefa.

void **PyEval_InitThreads**()

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()`.

Deprecated since version 3.9, will be removed in version 3.11.

int **PyEval_ThreadsInitialized**()

Returns a non-zero value if `PyEval_InitThreads()` has been called. This function can be called without holding the GIL, and therefore can be used to avoid calls to the locking API when running single-threaded.

Alterado na versão 3.7: The *GIL* is now initialized by `Py_Initialize()`.

Deprecated since version 3.9, will be removed in version 3.11.

*PyThreadState** **PyEval_SaveThread** ()

Release the global interpreter lock (if it has been created) and reset the thread state to `NULL`, returning the previous thread state (which is not `NULL`). If the lock has been created, the current thread must have acquired it.

void **PyEval_RestoreThread** (*PyThreadState* *tstate)

Acquire the global interpreter lock (if it has been created) and set the thread state to *tstate*, which must not be `NULL`. If the lock has been created, the current thread must not have acquired it, otherwise deadlock ensues.

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** ()

Return the current thread state. The global interpreter lock must be held. When the current thread state is `NULL`, this issues a fatal error (so that the caller needn't check for `NULL`).

*PyThreadState** **PyThreadState_Swap** (*PyThreadState* *tstate)

Swap the current thread state with the thread state given by the argument *tstate*, which may be `NULL`. The global interpreter lock must be held and is not released.

The following functions use thread-local storage, and are not compatible with sub-interpreters:

PyGILState_STATE **PyGILState_Ensure** ()

Ensure that the current thread is ready to call the Python C API regardless of the current state of Python, or of the global interpreter lock. This may be called as many times as desired by a thread as long as each call is matched with a call to `PyGILState_Release()`. In general, other thread-related APIs may be used between `PyGILState_Ensure()` and `PyGILState_Release()` calls as long as the thread state is restored to its previous state before the `Release()`. For example, normal usage of the `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS` macros is acceptable.

The return value is an opaque “handle” to the thread state when `PyGILState_Ensure()` was called, and must be passed to `PyGILState_Release()` to ensure Python is left in the same state. Even though recursive calls are allowed, these handles *cannot* be shared - each unique call to `PyGILState_Ensure()` must save the handle for its call to `PyGILState_Release()`.

When the function returns, the current thread will hold the GIL and be able to call arbitrary Python code. Failure is a fatal error.

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*)

Release any resources previously acquired. After this call, Python's state will be the same as it was prior to the corresponding `PyGILState_Ensure()` call (but generally this state will be unknown to the caller, hence the use of the `GILState` API).

Every call to `PyGILState_Ensure()` must be matched by a call to `PyGILState_Release()` on the same thread.

*PyThreadState** **PyGILState_GetThisThreadState** ()

Get the current thread state for this thread. May return `NULL` if no `GILState` API has been used on the current thread. Note that the main thread always has such a thread-state, even if no auto-thread-state call has been made on the main thread. This is mainly a helper/diagnostic function.

int **PyGILState_Check** ()

Return 1 if the current thread is holding the GIL and 0 otherwise. This function can be called from any thread at any time. Only if it has had its Python thread state initialized and currently is holding the GIL will it return 1. This is mainly a helper/diagnostic function. It can be useful for example in callback contexts or memory allocation functions when knowing that the GIL is locked can allow the caller to perform sensitive actions or otherwise behave differently.

Novo 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

This macro expands to `{ PyThreadState *_save; _save = PyEval_SaveThread();`. Note that it contains an opening brace; it must be matched with a following `Py_END_ALLOW_THREADS` macro. See above for further discussion of this macro.

Py_END_ALLOW_THREADS

This macro expands to `PyEval_RestoreThread(_save); }`. Note that it contains a closing brace; it must be matched with an earlier `Py_BEGIN_ALLOW_THREADS` macro. See above for further discussion of this macro.

Py_BLOCK_THREADS

This macro expands to `PyEval_RestoreThread(_save);`; it is equivalent to `Py_END_ALLOW_THREADS` without the closing brace.

Py_UNBLOCK_THREADS

This macro expands to `_save = PyEval_SaveThread();`; it is equivalent to `Py_BEGIN_ALLOW_THREADS` without the opening brace and variable declaration.

9.5.5 Low-level API

All of the following functions must be called after `Py_Initialize()`.

Alterado na versão 3.7: `Py_Initialize()` now initializes the *GIL*.

*PyInterpreterState** **PyInterpreterState_New** ()

Create a new interpreter state object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

Raises an auditing event `cpython.PyInterpreterState_New` with no arguments.

void **PyInterpreterState_Clear** (*PyInterpreterState* *interp)

Reset all information in an interpreter state object. The global interpreter lock must be held.

Raises an auditing event `cpython.PyInterpreterState_Clear` with no arguments.

void **PyInterpreterState_Delete** (*PyInterpreterState* *interp)

Destroy an interpreter state object. The global interpreter lock need not be held. The interpreter state must have been reset with a previous call to `PyInterpreterState_Clear()`.

*PyThreadState** **PyThreadState_New** (*PyInterpreterState* *interp)

Create a new thread state object belonging to the given interpreter object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

void **PyThreadState_Clear** (*PyThreadState* *tstate)

Reset all information in a thread state object. The global interpreter lock must be held.

Alterado na versão 3.9: This function now calls the `PyThreadState.on_delete` callback. Previously, that happened in `PyThreadState_Delete()`.

void **PyThreadState_Delete** (*PyThreadState* *tstate)

Destroy a thread state object. The global interpreter lock need not be held. The thread state must have been reset with a previous call to *PyThreadState_Clear()*.

void **PyThreadState_DeleteCurrent** (void)

Destroy the current thread state and release the global interpreter lock. Like *PyThreadState_Delete()*, the global interpreter lock need not be held. The thread state must have been reset with a previous call to *PyThreadState_Clear()*.

*PyFrameObject** **PyThreadState_GetFrame** (*PyThreadState* *tstate)

Get the current frame of the Python thread state *tstate*.

Return a strong reference. Return NULL if no frame is currently executing.

See also *PyEval_GetFrame()*.

tstate must not be NULL.

Novo na versão 3.9.

uint64_t **PyThreadState_GetID** (*PyThreadState* *tstate)

Get the unique thread state identifier of the Python thread state *tstate*.

tstate must not be NULL.

Novo na versão 3.9.

*PyInterpreterState** **PyThreadState_GetInterpreter** (*PyThreadState* *tstate)

Get the interpreter of the Python thread state *tstate*.

tstate must not be NULL.

Novo na versão 3.9.

*PyInterpreterState** **PyInterpreterState_Get** (void)

Get the current interpreter.

Issue a fatal error if there no current Python thread state or no current interpreter. It cannot return NULL.

The caller must hold the GIL.

Novo na versão 3.9.

int64_t **PyInterpreterState_GetID** (*PyInterpreterState* *interp)

Return the interpreter's unique ID. If there was any error in doing so then -1 is returned and an error is set.

The caller must hold the GIL.

Novo na versão 3.7.

*PyObject** **PyInterpreterState_GetDict** (*PyInterpreterState* *interp)

Return a dictionary in which interpreter-specific data may be stored. If this function returns NULL then no exception has been raised and the caller should assume no interpreter-specific dict is available.

This is not a replacement for *PyModule_GetState()*, which extensions should use to store interpreter-specific state information.

Novo na versão 3.8.

*PyObject** (***_PyFrameEvalFunction**) (*PyThreadState* *tstate, *PyFrameObject* *frame, int throwflag)

Type of a frame evaluation function.

The *throwflag* parameter is used by the *throw()* method of generators: if non-zero, handle the current exception.

Alterado na versão 3.9: The function now takes a *tstate* parameter.

PyFrameEvalFunction **_PyInterpreterState_GetEvalFrameFunc** (*PyInterpreterState* *interp)

Get the frame evaluation function.

See the [PEP 523](#) “Adding a frame evaluation API to CPython”.

Novo 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”.

Novo na versão 3.9.

*PyObject** **PyThreadState_GetDict** ()

Return value: Borrowed reference. Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no current thread state is available. If this function returns NULL, no exception has been raised and the caller should assume no current thread state is available.

int **PyThreadState_SetAsyncExc** (unsigned long id, *PyObject* *exc)

Asynchronously raise an exception in a thread. The *id* argument is the thread id of the target thread; *exc* is the exception object to be raised. This function does not steal any references to *exc*. To prevent naive misuse, you must write your own C extension to call this. Must be called with the GIL held. Returns the number of thread states modified; this is normally one, but will be zero if the thread id isn't found. If *exc* is NULL, the pending exception (if any) for the thread is cleared. This raises no exceptions.

Alterado na versão 3.7: The type of the *id* parameter changed from long to unsigned long.

void **PyEval_AcquireThread** (*PyThreadState* *tstate)

Acquire the global interpreter lock and set the current thread state to *tstate*, which must not be NULL. The lock must have been created earlier. If this thread already has the lock, deadlock ensues.

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)

Reset the current thread state to NULL and release the global interpreter lock. The lock must have been created earlier and must be held by the current thread. The *tstate* argument, which must not be NULL, is only used to check that it represents the current thread state — if it isn't, a fatal error is reported.

`PyEval_SaveThread()` is a higher-level function which is always available (even when threads have not been initialized).

void **PyEval_AcquireLock** ()

Acquire the global interpreter lock. The lock must have been created earlier. If this thread already has the lock, a deadlock ensues.

Obsoleto desde a versão 3.2: This function does not update the current thread state. Please use `PyEval_RestoreThread()` or `PyEval_AcquireThread()` instead.

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.

void **PyEval_ReleaseLock** ()

Release the global interpreter lock. The lock must have been created earlier.

Obsoleto desde a versão 3.2: This function does not update the current thread state. Please use `PyEval_SaveThread()` or `PyEval_ReleaseThread()` instead.

9.6 Sub-interpreter support

While in most uses, you will only embed a single Python interpreter, there are cases where you need to create several independent interpreters in the same process and perhaps even in the same thread. Sub-interpreters allow you to do that.

The “main” interpreter is the first one created when the runtime initializes. It is usually the only Python interpreter in a process. Unlike sub-interpreters, the main interpreter has unique process-global responsibilities like signal handling. It is also responsible for execution during runtime initialization and is usually the active interpreter during runtime finalization. The `PyInterpreterState_Main()` function returns a pointer to its state.

You can switch between sub-interpreters using the `PyThreadState_Swap()` function. You can create and destroy them using the following functions:

*PyThreadState** **Py_NewInterpreter** ()

Create a new sub-interpreter. This is an (almost) totally separate environment for the execution of Python code. In particular, the new interpreter has separate, independent versions of all imported modules, including the fundamental modules `builtins`, `__main__` and `sys`. The table of loaded modules (`sys.modules`) and the module search path (`sys.path`) are also separate. The new environment has no `sys.argv` variable. It has new standard I/O stream file objects `sys.stdin`, `sys.stdout` and `sys.stderr` (however these refer to the same underlying file descriptors).

The return value points to the first thread state created in the new sub-interpreter. This thread state is made in the current thread state. Note that no actual thread is created; see the discussion of thread states below. If creation of the new interpreter is unsuccessful, `NULL` is returned; no exception is set since the exception state is stored in the current thread state and there may not be a current thread state. (Like all other Python/C API functions, the global interpreter lock must be held before calling this function and is still held when it returns; however, unlike most other Python/C API functions, there needn't be a current thread state on entry.)

Extension modules are shared between (sub-)interpreters as follows:

- For modules using multi-phase initialization, e.g. `PyModule_FromDefAndSpec()`, a separate module object is created and initialized for each interpreter. Only C-level static and global variables are shared between these module objects.
- For modules using single-phase initialization, e.g. `PyModule_Create()`, the first time a particular extension is imported, it is initialized normally, and a (shallow) copy of its module's dictionary is squirreled away. When the same extension is imported by another (sub-)interpreter, a new module is initialized and filled with the contents of this copy; the extension's `init` function is not called. Objects in the module's dictionary thus end up shared across (sub-)interpreters, which might cause unwanted behavior (see *Bugs and caveats* below).

Note that this is different from what happens when an extension is imported after the interpreter has been completely re-initialized by calling `Py_FinalizeEx()` and `Py_Initialize()`; in that case, the ex-

tension's `initmodule` function is called again. As with multi-phase initialization, this means that only C-level static and global variables are shared between these modules.

void **Py_EndInterpreter** (*PyThreadState* *tstate)

Destroy the (sub-)interpreter represented by the given thread state. The given thread state must be the current thread state. See the discussion of thread states below. When the call returns, the current thread state is `NULL`. All thread states associated with this interpreter are destroyed. (The global interpreter lock must be held before calling this function and is still held when it returns.) `Py_FinalizeEx()` will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

9.6.1 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)

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](#).

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.

Novo na versão 3.1.

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.

`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 <i>what</i>	Meaning of <i>arg</i>
<code>PyTrace_CALL</code>	Always Py_None .
<code>PyTrace_EXCEPTION</code>	Exception information as returned by <code>sys.exc_info()</code> .
<code>PyTrace_LINE</code>	Always Py_None .
<code>PyTrace_RETURN</code>	Value being returned to the caller, or <code>NULL</code> if caused by an exception.
<code>PyTrace_C_CALL</code>	Function object being called.
<code>PyTrace_C_EXCEPTION</code>	Function object being called.
<code>PyTrace_C_RETURN</code>	Function object being called.
<code>PyTrace_OPCODE</code>	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`.

The caller must hold the *GIL*.

void **PyEval_SetTrace** (*Py_tracefunc func*, *PyObject *obj*)

Set the tracing function to *func*. This is similar to *PyEval_SetProfile()*, except the tracing function does receive line-number events and per-opcode events, but does not receive any event related to C function objects being called. Any trace function registered using *PyEval_SetTrace()* will not receive `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION` or `PyTrace_C_RETURN` as a value for the *what* parameter.

The caller must hold the *GIL*.

9.9 Advanced Debugger Support

These functions are only intended to be used by advanced debugging tools.

*PyInterpreterState** **PyInterpreterState_Head** ()

Return the interpreter state object at the head of the list of all such objects.

*PyInterpreterState** **PyInterpreterState_Main** ()

Return the main interpreter state object.

*PyInterpreterState** **PyInterpreterState_Next** (*PyInterpreterState *interp*)

Return the next interpreter state object after *interp* from the list of all such objects.

*PyThreadState** **PyInterpreterState_ThreadHead** (*PyInterpreterState *interp*)

Return the pointer to the first *PyThreadState* object in the list of threads associated with the interpreter *interp*.

*PyThreadState** **PyThreadState_Next** (*PyThreadState *tstate*)

Return the next thread state object after *tstate* from the list of all such objects belonging to the same *PyInterpreterState* object.

9.10 Thread Local Storage Support

The Python interpreter provides low-level support for thread-local storage (TLS) which wraps the underlying native TLS implementation to support the Python-level thread local storage API (`threading.local`). The CPython C level APIs are similar to those offered by pthreads and Windows: use a thread key and functions to associate a `void*` value per thread.

The GIL does *not* need to be held when calling these functions; they supply their own locking.

Note that `Python.h` does not include the declaration of the TLS APIs, you need to include `pthread.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.

9.10.1 Thread Specific Storage (TSS) API

TSS API is introduced to supersede the use of the existing TLS API within the CPython interpreter. This API uses a new type `Py_tss_t` instead of `int` to represent thread keys.

Novo na versão 3.7.

Ver também:

“A New C-API for Thread-Local Storage in CPython” ([PEP 539](#))

`Py_tss_t`

This data structure represents the state of a thread key, the definition of which may depend on the underlying TLS implementation, and it has an internal field representing the key's initialization state. There are no public members in this structure.

When `Py_LIMITED_API` is not defined, static allocation of this type by `Py_tss_NEEDS_INIT` is allowed.

`Py_tss_NEEDS_INIT`

This macro expands to the initializer for `Py_tss_t` variables. Note that this macro won't be defined with `Py_LIMITED_API`.

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()`

Return a value which is the same state as a value initialized with `Py_tss_NEEDS_INIT`, or `NULL` in the case of dynamic allocation failure.

`void` `PyThread_tss_free(Py_tss_t *key)`

Free the given `key` allocated by `PyThread_tss_alloc()`, after first calling `PyThread_tss_delete()` to ensure any associated thread locals have been unassigned. This is a no-op if the `key` argument is `NULL`.

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)

Return a non-zero value if the given `Py_tss_t` has been initialized by `PyThread_tss_create()`.

int **PyThread_tss_create** (*Py_tss_t* *key)

Return a zero value on successful initialization of a TSS key. The behavior is undefined if the value pointed to by the *key* argument is not initialized by `Py_tss_NEEDS_INIT`. This function can be called repeatedly on the same key – calling it on an already initialized key is a no-op and immediately returns success.

void **PyThread_tss_delete** (*Py_tss_t* *key)

Destroy a TSS key to forget the values associated with the key across all threads, and change the key's initialization state to uninitialized. A destroyed key is able to be initialized again by `PyThread_tss_create()`. This function can be called repeatedly on the same key – calling it on an already destroyed key is a no-op.

int **PyThread_tss_set** (*Py_tss_t* *key, void *value)

Return a zero value to indicate successfully associating a `void*` value with a TSS key in the current thread. Each thread has a distinct mapping of the key to a `void*` value.

void* **PyThread_tss_get** (*Py_tss_t* *key)

Return the `void*` value associated with a TSS key in the current thread. This returns `NULL` if no value is associated with the key in the current thread.

9.10.2 Thread Local Storage (TLS) API

Obsoleto 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** ()

void **PyThread_delete_key** (int key)

int **PyThread_set_key_value** (int key, void *value)

void* **PyThread_get_key_value** (int key)

void **PyThread_delete_key_value** (int key)

void **PyThread_ReInitTLS** ()

Configuração de Inicialização do Python

Novo na versão 3.8.

Estruturas:

- *PyConfig*
- *PyPreConfig*
- *PyStatus*
- *PyWideStringList*

Funções:

- *PyConfig_Clear()*
- *PyConfig_InitIsolatedConfig()*
- *PyConfig_InitPythonConfig()*
- *PyConfig_Read()*
- *PyConfig_SetArgv()*
- *PyConfig_SetBytesArgv()*
- *PyConfig_SetBytesString()*
- *PyConfig_SetString()*
- *PyConfig_SetWideStringList()*
- *PyPreConfig_InitIsolatedConfig()*
- *PyPreConfig_InitPythonConfig()*
- *PyStatus_Error()*
- *PyStatus_Exception()*
- *PyStatus_Exit()*
- *PyStatus_IsError()*

- `PyStatus_IsExit()`
- `PyStatus_NoMemory()`
- `PyStatus_Ok()`
- `PyWideStringList_Append()`
- `PyWideStringList_Insert()`
- `Py_ExitStatusException()`
- `Py_InitializeFromConfig()`
- `Py_PreInitialize()`
- `Py_PreInitializeFromArgs()`
- `Py_PreInitializeFromBytesArgs()`
- `Py_RunMain()`
- `Py_GetArgcArgv()`

A pré-configuração (tipo `PyPreConfig`) é armazenado em `_PyRuntime.preconfig` e a configuração (tipo `PyConfig`) é armazenado em `PyInterpreterState.config`.

Veja também *Inicialização, Finalização e Threads*.

Ver também:

PEP 587 “Configuração da inicialização do Python”.

10.1 PyWideStringList

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.

10.2 PyStatus

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.

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;
}

```

10.3 PyPreConfig

PyPreConfig

Estrutura usada para pré-inicializar o Python:

- Define o alocador de memória do Python
- Configura a localidade LC_CTYPE
- Define o modo UTF-8

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 do alocador de memória:

- 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): força o uso de `malloc()`
- PYMEM_ALLOCATOR_MALLOC_DEBUG (4): força o 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 do Python pymalloc* com ganchos de depuração

PYMEM_ALLOCATOR_PYMALLOC e PYMEM_ALLOCATOR_PYMALLOC_DEBUG não são suportados se Python estiver configurado usando `--without-pymalloc`

Veja *Gerenciamento de memória*.

int **configure_locale**

Definir a localidade LC_CTYPE para a localidade preferida do usuário? Se for igual a 0, define `coerce_c_locale` e `coerce_c_locale_warn` para 0.

int **coerce_c_locale**

Se for igual a 2, força a localidade C; se for igual a 1, lê a localidade LC_CTYPE para decidir se deve ser forçado.

int **coerce_c_locale_warn**

Se diferente de zero, emite um aviso se a localidade C for forçada.

int **dev_mode**

Veja *PyConfig.dev_mode*.

int **isolated**

Veja *PyConfig.isolated*.

int **legacy_windows_fs_encoding** (Windows *only*)

Se diferente de zero, desabilita o modo UTF-8, define a codificação do sistema de arquivos Python para mbcS, define o tratador de erros do sistema de arquivos para replace.

Disponível apenas no Windows. A macro `#ifdef MS_WINDOWS` pode ser usada para código específico do Windows.

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.

int **use_environment**

Veja *PyConfig.use_environment*.

int **utf8_mode**

Se não zero, habilita o modo UTF-8.

10.4 Preinitialization with PyPreConfig

Functions to preinitialize Python:

PyStatus **Py_PreInitialize** (const *PyPreConfig* **preconfig*)

Preinitialize Python from *preconfig* preconfiguration.

PyStatus **Py_PreInitializeFromBytesArgs** (const *PyPreConfig* **preconfig*, int *argc*, char * const **argv*)

Preinitialize Python from *preconfig* preconfiguration and command line arguments (bytes strings).

PyStatus **Py_PreInitializeFromArgs** (const *PyPreConfig* **preconfig*, int *argc*, wchar_t * const **argv*)

Preinitialize Python from *preconfig* preconfiguration and command line arguments (wide strings).

The caller is responsible to handle exceptions (error or exit) using *PyStatus_Exception()* and *Py_ExitStatusException()*.

For *Python Configuration* (*PyPreConfig_InitPythonConfig()*), if Python is initialized with command line arguments, the command line arguments must also be passed to preinitialize Python, since they have an effect on the pre-configuration like encodings. For example, the `-X utf8` command line option enables the UTF-8 Mode.

PyMem_SetAllocator() can be called after *Py_PreInitialize()* and before *Py_InitializeFromConfig()* to install a custom memory allocator. It can be called before *Py_PreInitialize()* if *PyPreConfig.allocator* is set to `PYMEM_ALLOCATOR_NOT_SET`.

Python memory allocation functions like *PyMem_RawMalloc()* must not be used before Python preinitialization, whereas calling directly *malloc()* and *free()* is always safe. *Py_DecodeLocale()* must not be called before the preinitialization.

Example using the preinitialization to enable the UTF-8 Mode:


```
PyStatus status;
PyPreConfig preconfig;
PyPreConfig_InitPythonConfig(&preconfig);

preconfig.utf8_mode = 1;

status = Py_PreInitialize(&preconfig);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}

/* at this point, Python will speak UTF-8 */

Py_Initialize();
/* ... use Python API here ... */
Py_Finalize();
```

10.5 PyConfig

PyConfig

Structure containing most parameters to configure Python.

Structure methods:

void **PyConfig_InitPythonConfig** (*PyConfig* **config*)

Initialize configuration with *Python Configuration*.

void **PyConfig_InitIsolatedConfig** (*PyConfig* **config*)

Initialize configuration with *Isolated Configuration*.

PyStatus **PyConfig_SetString** (*PyConfig* **config*, wchar_t * const **config_str*, const wchar_t **str*)

Copy the wide character string *str* into **config_str*.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesString** (*PyConfig* **config*, wchar_t * const **config_str*, const char **str*)

Decode *str* using *Py_DecodeLocale*() and set the result into **config_str*.

Preinitialize Python if needed.

PyStatus **PyConfig_SetArgv** (*PyConfig* **config*, int *argc*, wchar_t * const **argv*)

Set command line arguments from wide character strings.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesArgv** (*PyConfig* **config*, int *argc*, char * const **argv*)

Set command line arguments: decode bytes using *Py_DecodeLocale*() .

Preinitialize Python if needed.

PyStatus **PyConfig_SetWideStringList** (*PyConfig* **config*, *PyWideStringList* **list*,
Py_ssize_t *length*, wchar_t ***items*)

Set the list of wide strings *list* to *length* and *items*.

Preinitialize Python if needed.

PyStatus **PyConfig_Read** (*PyConfig* **config*)

Read all Python configuration.

Fields which are already initialized are left unchanged.

Preinitialize Python if needed.

void **PyConfig_Clear** (*PyConfig *config*)
Release configuration memory.

Most `PyConfig` methods preinitialize Python if needed. In that case, the Python preinitialization configuration is based on the `PyConfig`. If configuration fields which are in common with `PyPreConfig` are tuned, they must be set before calling a `PyConfig` method:

- `dev_mode`
- `isolated`
- `parse_argv`
- `use_environment`

Moreover, if `PyConfig_SetArgv()` or `PyConfig_SetBytesArgv()` is used, this method must be called first, before other methods, since the preinitialization configuration depends on command line arguments (if `parse_argv` is non-zero).

The caller of these methods is responsible to handle exceptions (error or exit) using `PyStatus_Exception()` and `Py_ExitStatusException()`.

Campos de estrutura

PyWideStringList **argv**

Command line arguments, `sys.argv`. See `parse_argv` to parse `argv` the same way the regular Python parses Python command line arguments. If `argv` is empty, an empty string is added to ensure that `sys.argv` always exists and is never empty.

wchar_t* **base_exec_prefix**
`sys.base_exec_prefix`.

wchar_t* **base_executable**
`sys._base_executable`: `__PYENVN_LAUNCHER__` environment variable value, or copy of `PyConfig.executable`.

wchar_t* **base_prefix**
`sys.base_prefix`.

wchar_t* **platlibdir**
`sys.platlibdir`: platform library directory name, set at configure time by `--with-platlibdir`, overrideable by the `PYTHONPLATLIBDIR` environment variable.

Novo na versão 3.9.

int **buffered_stdio**
If equals to 0, enable unbuffered mode, making the stdout and stderr streams unbuffered.
stdin is always opened in buffered mode.

int **bytes_warning**
If equals to 1, issue a warning when comparing bytes or bytearray with str, or comparing bytes with int. If equal or greater to 2, raise a `BytesWarning` exception.

wchar_t* **check_hash_pycs_mode**
Control the validation behavior of hash-based .pyc files (see [PEP 552](#)): `--check-hash-based-pycs` command line option value.

Valid values: `always`, `never` and `default`.

The default value is: `default`.

int **configure_c_stdio**

If non-zero, configure C standard streams (`stdio`, `stdout`, `stderr`). For example, set their mode to `O_BINARY` on Windows.

int **dev_mode**

If non-zero, enable the Python Development Mode.

int **dump_refs**

If non-zero, dump all objects which are still alive at exit.

`Py_TRACE_REFS` macro must be defined in build.

wchar_t* **exec_prefix**

`sys.exec_prefix`.

wchar_t* **executable**

`sys.executable`.

int **faulthandler**

If non-zero, call `faulthandler.enable()` at startup.

wchar_t* **filesystem_encoding**

Filesystem encoding, `sys.getfilesystemencoding()`.

wchar_t* **filesystem_errors**

Filesystem encoding errors, `sys.getfilesystemencodeerrors()`.

unsigned long **hash_seed**

int **use_hash_seed**

Randomized hash function seed.

If `use_hash_seed` is zero, a seed is chosen randomly at Python startup, and `hash_seed` is ignored.

wchar_t* **home**

Python home directory.

Initialized from `PYTHONHOME` environment variable value by default.

int **import_time**

If non-zero, profile import time.

int **inspect**

Enter interactive mode after executing a script or a command.

int **install_signal_handlers**

Install signal handlers?

int **interactive**

Interactive mode.

int **isolated**

If greater than 0, enable isolated mode:

- `sys.path` contains neither the script's directory (computed from `argv[0]` or the current directory) nor the user's site-packages directory.
- Python REPL doesn't import `readline` nor enable default `readline` configuration on interactive prompts.
- Set `use_environment` and `user_site_directory` to 0.

int **legacy_windows_stdio**

If non-zero, use `io.FileIO` instead of `io.WindowsConsoleIO` for `sys.stdin`, `sys.stdout` and `sys.stderr`.

Disponível apenas no Windows. A macro `#ifdef MS_WINDOWS` pode ser usada para código específico do Windows.

int **malloc_stats**

If non-zero, dump statistics on *Python pymalloc memory allocator* at exit.

The option is ignored if Python is built using `--without-pymalloc`.

wchar_t* **pythonpath_env**

Module search paths as a string separated by `DELIM(os.path.pathsep)`.

Initialized from `PYTHONPATH` environment variable value by default.

PyWideStringList **module_search_paths**

int **module_search_paths_set**

`sys.path`. If *module_search_paths_set* is equal to 0, the *module_search_paths* is overridden by the function calculating the *Path Configuration*.

int **optimization_level**

Compilation optimization level:

- 0: Peephole optimizer (and `__debug__` is set to True)
- 1: Remove assertions, set `__debug__` to False
- 2: Strip docstrings

int **parse_argv**

If non-zero, parse *argv* the same way the regular Python command line arguments, and strip Python arguments from *argv*: see Command Line Arguments.

int **parser_debug**

If non-zero, turn on parser debugging output (for expert only, depending on compilation options).

int **pathconfig_warnings**

If equal to 0, suppress warnings when calculating the *Path Configuration* (Unix only, Windows does not log any warning). Otherwise, warnings are written into `stderr`.

wchar_t* **prefix**

`sys.prefix`.

wchar_t* **program_name**

Program name. Used to initialize *executable*, and in early error messages.

wchar_t* **pycache_prefix**

`sys.pycache_prefix`: `.pyc` cache prefix.

If NULL, `sys.pycache_prefix` is set to None.

int **quiet**

Quiet mode. For example, don't display the copyright and version messages in interactive mode.

wchar_t* **run_command**

`python3 -c COMMAND` argument. Used by *Py_RunMain()*.

wchar_t* **run_filename**

`python3 FILENAME` argument. Used by *Py_RunMain()*.

wchar_t* **run_module**

`python3 -m MODULE` argument. Used by *Py_RunMain()*.

int **show_ref_count**

Show total reference count at exit?

Set to 1 by `-X showrefcount` command line option.

Need a debug build of Python (`Py_REF_DEBUG` macro must be defined).

int **site_import**

Import the `site` module at startup?

int **skip_source_first_line**

Skip the first line of the source?

wchar_t* **stdio_encoding**

wchar_t* **stdio_errors**

Encoding and encoding errors of `sys.stdin`, `sys.stdout` and `sys.stderr`.

int **tracemalloc**

If non-zero, call `tracemalloc.start()` at startup.

int **use_environment**

If greater than 0, use environment variables.

int **user_site_directory**

If non-zero, add user site directory to `sys.path`.

int **verbose**

If non-zero, enable verbose mode.

PyWideStringList **warnoptions**

`sys.warnoptions`: options of the `warnings` module to build warnings filters: lowest to highest priority.

The `warnings` module adds `sys.warnoptions` in the reverse order: the last *PyConfig.warnoptions* item becomes the first item of `warnings.filters` which is checked first (highest priority).

int **write_bytecode**

If non-zero, write `.pyc` files.

`sys.dont_write_bytecode` is initialized to the inverted value of *write_bytecode*.

PyWideStringList **xoptions**

`sys._xoptions`.

int **_use_peg_parser**

Enable PEG parser? Default: 1.

Set to 0 by `-X oldparser` and `PYTHONOLDPARSER`.

See also **PEP 617**.

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

If `parse_argv` is non-zero, `argv` arguments are parsed the same way the regular Python parses command line arguments, and Python arguments are stripped from `argv`: see [Command Line Arguments](#).

The `xoptions` options are parsed to set other options: see `-X` option.

Alterado na versão 3.9: The `show_alloc_count` field has been removed.

10.6 Initialization with PyConfig

Function to initialize Python:

PyStatus Py_InitializeFromConfig (const *PyConfig* **config*)
Initialize Python from *config* configuration.

The caller is responsible to handle exceptions (error or exit) using *PyStatus_Exception()* and *Py_ExitStatusException()*.

If *PyImport_FrozenModules()*, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* are used, they must be set or called after Python preinitialization and before the Python initialization. If Python is initialized multiple times, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* must be called before each Python initialization.

Example setting the program name:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name. Implicitly preinitialize Python. */
    status = PyConfig_SetString(&config, &config.program_name,
                               L"/path/to/my_program");
    if (PyStatus_Exception(status)) {
        goto fail;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto fail;
    }
    PyConfig_Clear(&config);
    return;

fail:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}
```

More complete example modifying the default configuration, read the configuration, and then override some parameters:

```
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);
```

(continua na próxima página)

```

    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Read all configuration at once */
    status = PyConfig_Read(&config);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Append our custom search path to sys.path */
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/more/modules");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Override executable computed by PyConfig_Read() */
    status = PyConfig_SetString(&config, &config.executable,
                               L"/path/to/my_executable");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    status = Py_InitializeFromConfig(&config);

done:
    PyConfig_Clear(&config);
    return status;
}

```

10.7 Isolated Configuration

`PyPreConfig_InitIsolatedConfig()` and `PyConfig_InitIsolatedConfig()` functions create a configuration to isolate Python from the system. For example, to embed Python into an application.

This configuration ignores global configuration variables, environment variables, command line arguments (`PyConfig.argv` is not parsed) and user site directory. The C standard streams (ex: `stdout`) and the `LC_CTYPE` locale are left unchanged. Signal handlers are not installed.

Configuration files are still used with this configuration. Set the *Path Configuration* (“output fields”) to ignore these configuration files and avoid the function computing the default path configuration.

10.8 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 UTF-8 Mode (**PEP 540**) depending on the `LC_CTYPE` locale, `PYTHONUTF8` and `PYTHONCOERCECLOCALE` environment variables.

Exemplo de Python personalizado rodando 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). */
    status = PyConfig_SetBytesArgv(&config, argc, argv);
    if (PyStatus_Exception(status)) {
        goto fail;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto fail;
    }
    PyConfig_Clear(&config);

    return Py_RunMain();
fail:
    PyConfig_Clear(&config);
    if (PyStatus_IsExit(status)) {
        return status.exitcode;
    }
    /* Display the error message and exit the process with
       non-zero exit code */
    Py_ExitStatusException(status);
}
```

10.9 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*)
 - `__PYENVN_LAUNCHER__` environment variable
 - (Windows only) Application paths in the registry under “SoftwarePythonPythonCoreX.YPythonPath” of HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE (where X.Y is the Python version).

- Path configuration output fields:

- `PyConfig.base_exec_prefix`
- `PyConfig.base_executable`
- `PyConfig.base_prefix`
- `PyConfig.exec_prefix`
- `PyConfig.executable`
- `PyConfig.module_search_paths_set, PyConfig.module_search_paths`
- `PyConfig.prefix`

If at least one “output field” is not set, Python calculates the path configuration to fill unset fields. If `module_search_paths_set` is equal to 0, `module_search_paths` is overridden and `module_search_paths_set` is set to 1.

It is possible to completely ignore the function calculating the default path configuration by setting explicitly all path configuration output fields listed above. A string is considered as set even if it is non-empty. `module_search_paths` is considered as set if `module_search_paths_set` is set to 1. In this case, path configuration input fields are ignored as well.

Set `pathconfig_warnings` to 0 to suppress warnings when calculating the path configuration (Unix only, Windows does not log any warning).

If `base_prefix` or `base_exec_prefix` fields are not set, they inherit their value from `prefix` and `exec_prefix` respectively.

`Py_RunMain()` and `Py_Main()` modify `sys.path`:

- If `run_filename` is set and is a directory which contains a `__main__.py` script, prepend `run_filename` to `sys.path`.
- If `isolated` is zero:
 - If `run_module` is set, prepend the current directory to `sys.path`. Do nothing if the current directory cannot be read.
 - If `run_filename` is set, prepend the directory of the filename to `sys.path`.
 - Otherwise, prepend an empty string to `sys.path`.

If `site_import` is non-zero, `sys.path` can be modified by the `site` module. If `user_site_directory` is non-zero and the user’s site-package directory exists, the `site` module appends the user’s site-package directory to `sys.path`.

The following configuration files are used by the path configuration:

- `pyvenv.cfg`
- `python.__pth` (Windows only)
- `pybuilddir.txt` (Unix only)

The `__PYENVN_LAUNCHER__` environment variable is used to set `PyConfig.base_executable`

10.10 Py_RunMain()

int **Py_RunMain** (void)

Execute the command (*PyConfig.run_command*), the script (*PyConfig.run_filename*) or the module (*PyConfig.run_module*) specified on the command line or in the configuration.

By default and when if `-i` option is used, run the REPL.

Finally, finalizes Python and returns an exit status that can be passed to the `exit()` function.

See *Python Configuration* for an example of customized Python always running in isolated mode using *Py_RunMain()*.

10.11 Py_GetArgcArgv()

void **Py_GetArgcArgv** (int *argc, wchar_t ***argv)

Get the original command line arguments, before Python modified them.

10.12 Multi-Phase Initialization Private Provisional API

This section is a private provisional API introducing multi-phase initialization, the core feature of **PEP 432**:

- “Core” initialization phase, “bare minimum Python”:
 - Builtin types;
 - Builtin exceptions;
 - Builtin and frozen modules;
 - The `sys` module is only partially initialized (ex: `sys.path` doesn’t exist yet).
- “Main” initialization phase, Python is fully initialized:
 - Install and configure `importlib`;
 - Apply the *Path Configuration*;
 - Install signal handlers;
 - Finish `sys` module initialization (ex: create `sys.stdout` and `sys.path`);
 - Enable optional features like `faulthandler` and `tracemalloc`;
 - Import the `site` module;
 - etc.

Private provisional API:

- `PyConfig._init_main`: if set to 0, *Py_InitializeFromConfig()* stops at the “Core” initialization phase.
- `PyConfig._isolated_interpreter`: if non-zero, disallow threads, subprocesses and fork.

PyStatus **Py_InitializeMain** (void)

Move to the “Main” initialization phase, finish the Python initialization.

No module is imported during the “Core” phase and the `importlib` module is not configured: the *Path Configuration* is only applied during the “Main” phase. It may allow to customize Python in Python to override or tune the *Path Configuration*, maybe install a custom `sys.meta_path` importer or an import hook, etc.

It may become possible to 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);
    }
}
```

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);
```

(continua na próxima página)

(continuação da página anterior)

```
free(buf); /* malloc'ed */  
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the 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 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.

Novo na versão 3.4.

`void* PyMem_RawMalloc (size_t n)`

Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawMalloc(1)` had been called instead. The memory will not have been initialized in any way.

`void* PyMem_RawCalloc (size_t nelem, size_t elsize)`

Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawCalloc(1, 1)` had been called instead.

Novo na versão 3.5.

`void* PyMem_RawRealloc (void *p, size_t n)`

Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyMem_RawMalloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to `PyMem_RawMalloc()`, `PyMem_RawRealloc()` or `PyMem_RawCalloc()`.

If the request fails, `PyMem_RawRealloc()` returns NULL and `p` remains a valid pointer to the previous memory area.

void **PyMem_RawFree** (void **p*)

Frees the memory block pointed to by `p`, which must have been returned by a previous call to `PyMem_RawMalloc()`, `PyMem_RawRealloc()` or `PyMem_RawCalloc()`. Otherwise, or if `PyMem_RawFree(p)` has been called before, undefined behavior occurs.

If `p` is NULL, no operation is performed.

11.3 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*)

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*)

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.

Novo na versão 3.5.

void* **PyMem_Realloc** (void **p*, size_t *n*)

Resizes the memory block pointed to by `p` to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If `p` is NULL, the call is equivalent to `PyMem_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-NULL.

Unless `p` is NULL, it must have been returned by a previous call to `PyMem_Malloc()`, `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*)

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.

TYPE* **PyMem_New** (TYPE, size_t *n*)

Same as `PyMem_Malloc()`, but allocates $(n * \text{sizeof}(\text{TYPE}))$ bytes of memory. Returns a pointer cast to TYPE*. The memory will not have been initialized in any way.

TYPE* **PyMem_Resize** (void **p*, TYPE, size_t *n*)

Same as `PyMem_Realloc()`, but the memory block is resized to $(n * \text{sizeof}(\text{TYPE}))$ 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)`

11.4 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.

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*)

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*)

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.

Novo na versão 3.5.

void* **PyObject_Realloc** (void **p*, size_t *n*)

Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyObject_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to `PyObject_Malloc()`, `PyObject_Realloc()` or `PyObject_Calloc()`.

If the request fails, `PyObject_Realloc()` returns `NULL` and *p* remains a valid pointer to the previous memory area.

void **PyObject_Free** (void **p*)

Frees the memory block pointed to by *p*, which must have been returned by a previous call to `PyObject_Malloc()`, `PyObject_Realloc()` or `PyObject_Calloc()`. Otherwise, or if `PyObject_Free(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

11.5 Alocadores de memória padrão

Alocadores de memória padrão:

Configuração	Nome	Py-Mem_RawMalloc	PyMem_Malloc	PyObject_Malloc
Release build	"pymalloc"	malloc	pymalloc	pymalloc
Debug build	"pymalloc_debug"	malloc + debug	pymalloc + debug	pymalloc + debug
Release build, without pymalloc	"malloc"	malloc	malloc	malloc
Debug build, without pymalloc	"malloc_debug"	malloc + debug	malloc + debug	malloc + debug

Legend:

- Name: value for `PYTHONMALLOC` environment variable
- malloc: system allocators from the standard C library, C functions: `malloc()`, `calloc()`, `realloc()` and `free()`
- pymalloc: *pymalloc memory allocator*
- "+ debug": with debug hooks installed by `PyMem_SetupDebugHooks()`

11.6 Alocadores de memória

Novo na versão 3.4.

PyMemAllocatorEx

Structure used to describe a memory block allocator. The structure has the following fields:

Campo	Significado
<code>void *ctx</code>	user context passed as first argument
<code>void* malloc(void *ctx, size_t size)</code>	allocate a memory block
<code>void* calloc(void *ctx, size_t nelem, size_t elsize)</code>	allocate a memory block initialized with zeros
<code>void* realloc(void *ctx, void *ptr, size_t new_size)</code>	allocate or resize a memory block
<code>void free(void *ctx, void *ptr)</code>	free a memory block

Alterado na versão 3.5: The `PyMemAllocator` structure was renamed to `PyMemAllocatorEx` and a new `calloc` field was added.

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.

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.

void **PyMem_SetupDebugHooks** (void)

Setup hooks to detect bugs in the Python memory allocator functions.

Newly allocated memory is filled with the byte 0xCD (CLEANBYTE), freed memory is filled with the byte 0xDD (DEADBYTE). Memory blocks are surrounded by “forbidden bytes” (FORBIDDENBYTE: byte 0xFD).

Checagens em Tempo de Execução:

- Detect API violations, ex: `PyObject_Free()` called on a buffer 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.

These hooks are *installed by default* if Python is compiled in debug mode. The `PYTHONMALLOC` environment variable can be used to install debug hooks on a Python compiled in release mode.

Alterado na versão 3.6: This 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 (CLEANBYTE), 0xDB (DEADBYTE) and 0xFB (FORBIDDENBYTE) have been replaced with 0xCD, 0xDD and 0xFD to use the same values than Windows CRT debug `malloc()` and `free()`.

11.7 The pymalloc allocator

Python has a *pymalloc* allocator optimized for small objects (smaller or equal to 512 bytes) with a short lifetime. It uses memory mappings called “arenas” with a fixed size of 256 KiB. It falls back to `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()` e `VirtualFree()` no Windows,
- `mmap()` e `munmap()` se disponível,
- `malloc()` e `free()` do contrário.

11.7.1 Customize pymalloc Arena Allocator

Novo na versão 3.4.

PyObjectArenaAllocator

Structure used to describe an arena allocator. The structure has three fields:

Campo	Significado
<code>void *ctx</code>	user context passed as first argument
<code>void* alloc(void *ctx, size_t size)</code>	allocate an arena of size bytes
<code>void free(void *ctx, void *ptr, size_t size)</code>	free an arena

`void PyObject_GetArenaAllocator (PyObjectArenaAllocator *allocator)`

Get the arena allocator.

`void PyObject_SetArenaAllocator (PyObjectArenaAllocator *allocator)`

Set the arena allocator.

11.8 tracemalloc C API

Novo 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.

11.9 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.

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)

Return value: New reference.

*PyVarObject** **_PyObject_NewVar** (*PyTypeObject* *type, *Py_ssize_t* size)

Return value: New reference.

*PyObject** **PyObject_Init** (*PyObject* *op, *PyTypeObject* *type)

Return value: Borrowed reference. Inicializa um objeto *op* recém-alocado com seu tipo e referência inicial. Retorna o objeto inicializado. Se o *type* indica que o objeto participa no detector de lixo cíclico ele é adicionado ao grupo do detector de objetos observados. Outros campos do objeto não são afetados.

*PyVarObject** **PyObject_InitVar** (*PyVarObject* *op, *PyTypeObject* *type, *Py_ssize_t* size)

Return value: Borrowed reference. Isto faz tudo que o *PyObject_Init()* faz e também inicializa a informação de comprimento para um objeto de tamanho variável.

TYPE* **PyObject_New** (**TYPE**, *PyTypeObject* *type)

Return value: New reference. Aloca um novo objeto Python usando o tipo de estrutura do C *TYPE* e o objeto Python do tipo *type*. Campos não definidos pelo cabeçalho do objeto Python não são inicializados; a contagem de referências do objeto será um. O tamanho da alocação de memória é determinado do campo *tp_basicsize* do objeto tipo.

TYPE* **PyObject_NewVar** (**TYPE**, *PyTypeObject* *type, *Py_ssize_t* size)

Return value: New reference. Aloca um novo objeto Python usando o tipo de estrutura do C *TYPE* e o objeto Python do tipo *type*. Campos não definidos pelo cabeçalho do objeto Python não são inicializados. A memória alocada permite a estrutura *TYPE* e os campos *size* do tamanho dado pelo campo *tp_itemsize* do tipo *type*. Isto é útil para implementar objetos como tuplas, 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.

void **PyObject_De1** (void *op)

Libera memória alocada a um objeto usando `PyObject_New()` ou `PyObject_NewVar()`. Isto é normalmente chamado pelo `tp_dealloc` manipulador 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

There are a large number of structures which are used in the definition of object types for Python. This section describes these structures and how they are used.

12.2.1 Base object types and macros

All Python objects ultimately share a small number of fields at the beginning of the object's representation in memory. These are represented by the *PyObject* and *PyVarObject* types, which are defined, in turn, by the expansions of some macros also used, whether directly or indirectly, in the definition of all other Python objects.

PyObject

All object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In a normal “release” build, it contains only the object's reference count and a pointer to the corresponding type object. Nothing is actually declared to be a *PyObject*, but every pointer to a Python object can be cast to a *PyObject**. Access to the members must be done by using the macros `Py_REFCNT` and `Py_TYPE`.

PyVarObject

This is an extension of *PyObject* that adds the `ob_size` field. This is only used for objects that have some notion of *length*. This type does not often appear in the Python/C API. Access to the members must be done by using the macros `Py_REFCNT`, `Py_TYPE`, and `Py_SIZE`.

PyObject_HEAD

This is a macro used when declaring new types which represent objects without a varying length. The `PyObject_HEAD` macro expands to:

```
PyObject ob_base;
```

See documentation of *PyObject* above.

PyObject_VAR_HEAD

This is a macro used when declaring new types which represent objects with a length that varies from instance to instance. The `PyObject_VAR_HEAD` macro expands to:

```
PyVarObject ob_base;
```

See documentation of *PyVarObject* above.

Py_TYPE(o)

This macro is used to access the `ob_type` member of a Python object. It expands to:

```
((PyObject*) (o)) -> ob_type)
```

int **Py_IS_TYPE** (*PyObject* *o, *PyTypeObject* *type)

Return non-zero if the object *o* type is *type*. Return zero otherwise. Equivalent to: `Py_TYPE(o) == type`.

Novo na versão 3.9.

void **Py_SET_TYPE** (*PyObject* *o, *PyTypeObject* *type)

Set the object *o* type to *type*.

Novo na versão 3.9.

Py_REFCNT (o)

This macro is used to access the `ob_refcnt` member of a Python object. It expands to:

```
((PyObject*) (o)) -> ob_refcnt)
```

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

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

Novo na versão 3.9.

Py_SIZE (o)

This macro is used to access the `ob_size` member of a Python object. It expands to:

```
((PyVarObject*) (o)) -> ob_size)
```

void **Py_SET_SIZE** (*PyVarObject* *o, *Py_ssize_t* size)

Set the object *o* size to *size*.

Novo 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

PyCFunction

Type of the functions used to implement most Python callables in C. Functions of this type take two *PyObject* * parameters and return one such value. If the return value is `NULL`, an exception shall have been set. If not `NULL`, the return value is interpreted as the return value of the function as exposed in Python. The function must return a new reference.

The function signature is:

```
PyObject *PyCFunction(PyObject *self,
                      PyObject *args);
```


PyCFunctionWithKeywords

Type of the functions used to implement Python callables in C with signature `METH_VARARGS | METH_KEYWORDS`. The function signature is:

```
PyObject *PyCFunctionWithKeywords(PyObject *self,
                                   PyObject *args,
                                   PyObject *kwargs);
```

_PyCFunctionFast

Type of the functions used to implement Python callables in C with signature `METH_FASTCALL`. The function signature is:

```
PyObject *_PyCFunctionFast(PyObject *self,
                           PyObject *const *args,
                           Py_ssize_t nargs);
```

_PyCFunctionFastWithKeywords

Type of the functions used to implement Python callables in C with signature `METH_FASTCALL | METH_KEYWORDS`. The function signature is:

```
PyObject *_PyCFunctionFastWithKeywords(PyObject *self,
                                        PyObject *const *args,
                                        Py_ssize_t nargs,
                                        PyObject *kwnames);
```

PyCMethod

Type of the functions used to implement Python callables in C with signature `METH_METHOD | METH_FASTCALL | METH_KEYWORDS`. The function signature is:

```
PyObject *PyCMethod(PyObject *self,
                    PyTypeObject *defining_class,
                    PyObject *const *args,
                    Py_ssize_t nargs,
                    PyObject *kwnames)
```

Novo na versão 3.9.

PyMethodDef

Structure used to describe a method of an extension type. This structure has four fields:

Campo	Tipo em C	Significado
ml_name	const char *	name of the method
ml_meth	PyCFunction	pointer to the C implementation
ml_flags	int	flag bits indicating how the call should be constructed
ml_doc	const char *	points to the contents of the docstring

The `ml_meth` is a C function pointer. The functions may be of different types, but they always return `PyObject *`. If the function is not of the `PyCFunction`, the compiler will require a cast in the method table. Even though `PyCFunction` defines the first parameter as `PyObject *`, it is common that the method implementation uses the specific C type of the `self` object.

The `ml_flags` field is a bitfield which can include the following flags. The individual flags indicate either a calling convention or a binding convention.

There are these calling conventions:

METH_VARARGS

This is the typical calling convention, where the methods have the type `PyCFunction`. The function expects two

`PyObject*` values. The first one is the *self* object for methods; for module functions, it is the module object. The second parameter (often called *args*) is a tuple object representing all arguments. This parameter is typically processed using `PyArg_ParseTuple()` or `PyArg_UnpackTuple()`.

METH_VARARGS | METH_KEYWORDS

Methods with these flags must be of type `PyCFunctionWithKeywords`. The function expects three parameters: *self*, *args*, *kwargs* where *kwargs* is a dictionary of all the keyword arguments or possibly NULL if there are no keyword arguments. The parameters are typically processed using `PyArg_ParseTupleAndKeywords()`.

METH_FASTCALL

Fast calling convention supporting only positional arguments. The methods have the type `_PyCFunctionFast`. The first parameter is *self*, the second parameter is a C array of `PyObject*` values indicating the arguments and the third parameter is the number of arguments (the length of the array).

This is not part of the *limited API*.

Novo na versão 3.7.

METH_FASTCALL | METH_KEYWORDS

Extension of `METH_FASTCALL` supporting also keyword arguments, with methods of type `_PyCFunctionFastWithKeywords`. Keyword arguments are passed the same way as in the *vector-call protocol*: there is an additional fourth `PyObject*` parameter which is a tuple representing the names of the keyword arguments (which are guaranteed to be strings) or possibly NULL if there are no keywords. The values of the keyword arguments are stored in the *args* array, after the positional arguments.

This is not part of the *limited API*.

Novo na versão 3.7.

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*.

Novo 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.

METH_O

Methods with a single object argument can be listed with the `METH_O` flag, instead of invoking `PyArg_ParseTuple()` with a "O" argument. They have the type `PyCFunction`, with the *self* parameter, and a `PyObject*` parameter representing the single argument.

These two constants are not used to indicate the calling convention but the binding when use with methods of classes. These may not be used for functions defined for modules. At most one of these flags may be set for any given method.

METH_CLASS

The method will be passed the type object as the first parameter rather than an instance of the type. This is used to create *class methods*, similar to what is created when using the `classmethod()` built-in function.

METH_STATIC

The method will be passed NULL as the first parameter rather than an instance of the type. This is used to create *static methods*, similar to what is created when using the `staticmethod()` built-in function.

One other constant controls whether a method is loaded in place of another definition with the same method name.

METH_COEXIST

The method will be loaded in place of existing definitions. Without *METH_COEXIST*, the default is to skip repeated definitions. Since slot wrappers are loaded before the method table, the existence of a *sq_contains* slot, for example, would generate a wrapped method named `__contains__()` and preclude the loading of a corresponding PyCFunction with the same name. With the flag defined, the PyCFunction will be loaded in place of the wrapper object and will co-exist with the slot. This is helpful because calls to PyCFunctions are optimized more than wrapper object calls.

12.2.3 Accessing attributes of extension types

PyMemberDef

Structure which describes an attribute of a type which corresponds to a C struct member. Its fields are:

Campo	Tipo em C	Significado
name	const char *	name of the member
type	int	the type of the member in the C struct
offset	Py_ssize_t	the offset in bytes that the member is located on the type's object struct
flags	int	flag bits indicating if the field should be read-only or writable
doc	const char *	points to the contents of the docstring

`type` can be one of many `T_` macros corresponding to various C types. When the member is accessed in Python, it will be converted to the equivalent Python type.

Macro name	C type
<code>T_SHORT</code>	short
<code>T_INT</code>	int
<code>T_LONG</code>	long
<code>T_FLOAT</code>	float
<code>T_DOUBLE</code>	double
<code>T_STRING</code>	const char *
<code>T_OBJECT</code>	PyObject *
<code>T_OBJECT_EX</code>	PyObject *
<code>T_CHAR</code>	char
<code>T_BYTE</code>	char
<code>T_UBYTE</code>	unsigned char
<code>T_UINT</code>	unsigned int
<code>T_USHORT</code>	unsigned short
<code>T_ULONG</code>	unsigned long
<code>T_BOOL</code>	char
<code>T_LONGLONG</code>	long long
<code>T_ULONGLONG</code>	unsigned long long
<code>T_PYSSIZET</code>	Py_ssize_t

`T_OBJECT` and `T_OBJECT_EX` differ in that `T_OBJECT` returns `None` if the member is `NULL` and `T_OBJECT_EX` raises an `AttributeError`. Try to use `T_OBJECT_EX` over `T_OBJECT` because `T_OBJECT_EX` handles use of the `del` statement on that attribute more correctly than `T_OBJECT`.

`flags` can be 0 for write and read access or `READONLY` for read-only access. Using `T_STRING` for `type` implies `READONLY`. `T_STRING` data is interpreted as UTF-8. Only `T_OBJECT` and `T_OBJECT_EX` members can be deleted. (They are set to `NULL`).

Heap allocated types (created using `PyType_FromSpec()` or similar), `PyMemberDef` may contain definitions for the special members `__dictoffset__`, `__weaklistoffset__` and `__vectorcalloffset__`, corresponding to `tp_dictoffset`, `tp_weaklistoffset` and `tp_vectorcall_offset` in type objects. These must be defined with `T_PYSSIZET` and `READONLY`, for example:

```
static PyMemberDef spam_type_members[] = {
    {"__dictoffset__", T_PYSSIZET, offsetof(Spam_object, dict), READONLY},
    {NULL} /* Sentinel */
};
```

*PyObject** **PyMember_GetOne** (const char **obj_addr*, struct *PyMemberDef* **m*)

Get an attribute belonging to the object at address *obj_addr*. The attribute is described by `PyMemberDef` *m*. Returns `NULL` on error.

int **PyMember_SetOne** (char **obj_addr*, struct *PyMemberDef* **m*, *PyObject* **o*)

Set an attribute belonging to the object at address *obj_addr* to object *o*. The attribute to set is described by `PyMemberDef` *m*. Returns 0 if successful and a negative value on failure.

PyGetSetDef

Structure to define property-like access for a type. See also description of the `PyTypeObject.tp_getset` slot.

Campo	Tipo em C	Significado
nome	const char *	attribute name
get	getter	C function to get the attribute
set	setter	optional C function to set or delete the attribute, if omitted the attribute is readonly
doc	const char *	optional docstring
closure	void *	optional function pointer, providing additional data for getter and setter

The `get` function takes one *PyObject** parameter (the instance) and a function pointer (the associated closure):

```
typedef PyObject *(*getter)(PyObject *, void *);
```

It should return a new reference on success or `NULL` with a set exception on failure.

`set` functions take two *PyObject** parameters (the instance and the value to be set) and a function pointer (the associated closure):

```
typedef int (*setter)(PyObject *, PyObject *, void *);
```

In case the attribute should be deleted the second parameter is `NULL`. Should return 0 on success or `-1` with a set exception on failure.

12.3 Objetos tipo

Talvez uma das estruturas mais importantes do sistema de objetos Python seja a estrutura que define um novo tipo: a estrutura `PyTypeObject`. Objetos de tipo podem ser manipulados usando qualquer uma das funções `PyObject_*()` ou `PyType_*()`, mas não oferecem muita coisa interessante para a maioria dos aplicativos Python. Esses objetos são fundamentais para o comportamento dos objetos, portanto, são muito importantes para o próprio interpretador e para qualquer módulo de extensão que implemente novos tipos.

Os objetos de tipo são bastante grandes em comparação com a maioria dos tipos padrão. A razão para o tamanho é que cada objeto de tipo armazena um grande número de valores, principalmente indicadores de função C, cada um dos quais

implementa uma pequena parte da funcionalidade do tipo. Os campos do objeto de tipo são examinados em detalhes nesta seção. Os campos serão descritos na ordem em que ocorrem na estrutura.

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 ¹	Type	métodos/atributos especiais	Info ²				
			O	T	D	I	
<R> <code>tp_name</code>	const char *	<code>__name__</code>	X	X			
<code>tp_basicsize</code>	<code>Py_ssize_t</code>		X	X			X
<code>tp_itemsize</code>	<code>Py_ssize_t</code>			X			X
<code>tp_dealloc</code>	destructor		X	X			X
<code>tp_vectorcall_offset</code>	<code>Py_ssize_t</code>			X			X
(<code>tp_getattr</code>)	<code>getattrfunc</code>	<code>__getattribute__</code> , <code>__getattr__</code>					G
(<code>tp_setattr</code>)	<code>setattrfunc</code>	<code>__setattr__</code> , <code>__delattr__</code>					G
<code>tp_as_async</code>	<code>PyAsyncMethods</code> *	<i>sub-slots</i>					%
<code>tp_repr</code>	<code>reprfunc</code>	<code>__repr__</code>	X	X			X
<code>tp_as_number</code>	<code>PyNumberMethods</code> *	<i>sub-slots</i>					%
<code>tp_as_sequence</code>	<code>PySequenceMethods</code> *	<i>sub-slots</i>					%
<code>tp_as_mapping</code>	<code>PyMappingMethods</code> *	<i>sub-slots</i>					%
<code>tp_hash</code>	<code>hashfunc</code>	<code>__hash__</code>	X				G
<code>tp_call</code>	<code>ternaryfunc</code>	<code>__call__</code>		X			X
<code>tp_str</code>	<code>reprfunc</code>	<code>__str__</code>	X				X
<code>tp_getattro</code>	<code>getattrofunc</code>	<code>__getattribute__</code> , <code>__getattr__</code>	X	X			G
<code>tp_setattro</code>	<code>setattrofunc</code>	<code>__setattr__</code> , <code>__delattr__</code>	X	X			G
<code>tp_as_buffer</code>	<code>PyBufferProcs</code> *						%
<code>tp_flags</code>	unsigned long		X	X			?
<code>tp_doc</code>	const char *	<code>__doc__</code>	X	X			
<code>tp_traverse</code>	<code>traverseproc</code>			X			G
<code>tp_clear</code>	<code>inquiry</code>			X			G
<code>tp_richcompare</code>	<code>richcmpfunc</code>	<code>__lt__</code> , <code>__le__</code> , <code>__eq__</code> , <code>__ne__</code> , <code>__gt__</code> , <code>__ge__</code>	X				G
<code>tp_weaklistoffset</code>	<code>Py_ssize_t</code>			X			?
<code>tp_iter</code>	<code>getiterfunc</code>	<code>__iter__</code>					X
<code>tp_ternext</code>	<code>iternextfunc</code>	<code>__next__</code>					X
<code>tp_methods</code>	<code>PyMethodDef</code> []		X	X			
<code>tp_members</code>	<code>PyMemberDef</code> []			X			
<code>tp_getset</code>	<code>PyGetSetDef</code> []		X	X			
<code>tp_base</code>	<code>PyTypeObject</code> *	<code>__base__</code>				X	
<code>tp_dict</code>	<code>PyObject</code> *	<code>__dict__</code>				?	
<code>tp_descr_get</code>	<code>descrgetfunc</code>	<code>__get__</code>					X
<code>tp_descr_set</code>	<code>descrsetfunc</code>	<code>__set__</code> , <code>__delete__</code>					X
<code>tp_dictoffset</code>	<code>Py_ssize_t</code>			X			?
<code>tp_init</code>	<code>initproc</code>	<code>__init__</code>	X	X			X
<code>tp_alloc</code>	<code>allocfunc</code>		X			?	?
<code>tp_new</code>	<code>newfunc</code>	<code>__new__</code>	X	X		?	?
<code>tp_free</code>	<code>freefunc</code>		X	X		?	?

Continuação na próxima página

Tabela 1 – continuação da página anterior

Slot de PyTypeObject ¹	<i>Type</i>	métodos/atributos especiais	Info ²			
			O	T	D	I
<i>tp_is_gc</i>	<i>inquiry</i>			X		X
< <i>tp_bases</i> >	<i>PyObject *</i>	<i>__bases__</i>			~	
< <i>tp_mro</i> >	<i>PyObject *</i>	<i>__mro__</i>			~	
[<i>tp_cache</i>]	<i>PyObject *</i>					
[<i>tp_subclasses</i>]	<i>PyObject *</i>	<i>__subclasses__</i>				
[<i>tp_weaklist</i>]	<i>PyObject *</i>					
(<i>tp_del</i>)	<i>destructor</i>					
[<i>tp_version_tag</i>]	<i>unsigned int</i>					
<i>tp_finalize</i>	<i>destructor</i>	<i>__del__</i>				X
<i>tp_vectorcall</i>	<i>vectorcallfunc</i>					

sub-slots

Slot	<i>Type</i>	special methods
<i>am_await</i>	<i>unaryfunc</i>	<i>__await__</i>
<i>am_aiter</i>	<i>unaryfunc</i>	<i>__aiter__</i>
<i>am_anext</i>	<i>unaryfunc</i>	<i>__anext__</i>
<i>nb_add</i>	<i>binaryfunc</i>	<i>__add__</i> <i>__radd__</i>
<i>nb_inplace_add</i>	<i>binaryfunc</i>	<i>__iadd__</i>
<i>nb_subtract</i>	<i>binaryfunc</i>	<i>__sub__</i> <i>__rsub__</i>
<i>nb_inplace_subtract</i>	<i>binaryfunc</i>	<i>__isub__</i>
<i>nb_multiply</i>	<i>binaryfunc</i>	<i>__mul__</i> <i>__rmul__</i>
<i>nb_inplace_multiply</i>	<i>binaryfunc</i>	<i>__imul__</i>
<i>nb_remainder</i>	<i>binaryfunc</i>	<i>__mod__</i> <i>__rmod__</i>
<i>nb_inplace_remainder</i>	<i>binaryfunc</i>	<i>__imod__</i>
<i>nb_divmod</i>	<i>binaryfunc</i>	<i>__divmod__</i> <i>__rdivmod__</i>
<i>nb_power</i>	<i>ternaryfunc</i>	<i>__pow__</i> <i>__rpow__</i>
<i>nb_inplace_power</i>	<i>ternaryfunc</i>	<i>__ipow__</i>

Continuação na próxima página

¹ A slot name in parentheses indicates it is (effectively) deprecated. Names in angle brackets should be 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	Type	special methods
<i>nb_negative</i>	<i>unaryfunc</i>	<i>__neg__</i>
<i>nb_positive</i>	<i>unaryfunc</i>	<i>__pos__</i>
<i>nb_absolute</i>	<i>unaryfunc</i>	<i>__abs__</i>
<i>nb_bool</i>	<i>inquiry</i>	<i>__bool__</i>
<i>nb_invert</i>	<i>unaryfunc</i>	<i>__invert__</i>
<i>nb_lshift</i>	<i>binaryfunc</i>	<i>__lshift__</i> <i>__rlshift__</i>
<i>nb_inplace_lshift</i>	<i>binaryfunc</i>	<i>__ilshift__</i>
<i>nb_rshift</i>	<i>binaryfunc</i>	<i>__rshift__</i> <i>__rrshift__</i>
<i>nb_inplace_rshift</i>	<i>binaryfunc</i>	<i>__irshift__</i>
<i>nb_and</i>	<i>binaryfunc</i>	<i>__and__</i> <i>__rand__</i>
<i>nb_inplace_and</i>	<i>binaryfunc</i>	<i>__iand__</i>
<i>nb_xor</i>	<i>binaryfunc</i>	<i>__xor__</i> <i>__rxor__</i>
<i>nb_inplace_xor</i>	<i>binaryfunc</i>	<i>__ixor__</i>
<i>nb_or</i>	<i>binaryfunc</i>	<i>__or__</i> <i>__ror__</i>
<i>nb_inplace_or</i>	<i>binaryfunc</i>	<i>__ior__</i>
<i>nb_int</i>	<i>unaryfunc</i>	<i>__int__</i>
<i>nb_reserved</i>	<i>void *</i>	
<i>nb_float</i>	<i>unaryfunc</i>	<i>__float__</i>
<i>nb_floor_divide</i>	<i>binaryfunc</i>	<i>__floordiv__</i>
<i>nb_inplace_floor_divide</i>	<i>binaryfunc</i>	<i>__ifloordiv__</i>
<i>nb_true_divide</i>	<i>binaryfunc</i>	<i>__truediv__</i>
<i>nb_inplace_true_divide</i>	<i>binaryfunc</i>	<i>__itruediv__</i>
<i>nb_index</i>	<i>unaryfunc</i>	<i>__index__</i>
<i>nb_matrix_multiply</i>	<i>binaryfunc</i>	<i>__matmul__</i> <i>__rmatmul__</i>
<i>nb_inplace_matrix_multiply</i>	<i>binaryfunc</i>	<i>__imatmul__</i>
<i>mp_length</i>	<i>lenfunc</i>	<i>__len__</i>
<i>mp_subscript</i>	<i>binaryfunc</i>	<i>__getitem__</i>
<i>mp_ass_subscript</i>	<i>objobjargproc</i>	<i>__setitem__</i> , <i>__delitem__</i>
<i>sq_length</i>	<i>lenfunc</i>	<i>__len__</i>
<i>sq_concat</i>	<i>binaryfunc</i>	<i>__add__</i>
<i>sq_repeat</i>	<i>ssizeargfunc</i>	<i>__mul__</i>
<i>sq_item</i>	<i>ssizeargfunc</i>	<i>__getitem__</i>
<i>sq_ass_item</i>	<i>ssizeobjargproc</i>	<i>__setitem__</i> <i>__delitem__</i>
<i>sq_contains</i>	<i>objobjproc</i>	<i>__contains__</i>
<i>sq_inplace_concat</i>	<i>binaryfunc</i>	<i>__iadd__</i>
<i>sq_inplace_repeat</i>	<i>ssizeargfunc</i>	<i>__imul__</i>
<i>bf_getbuffer</i>	<i>getbufferproc()</i>	
<i>bf_releasebuffer</i>	<i>releasebufferproc()</i>	

slot typedefs

typedef	Parameter Types	Return Type
<i>allocfunc</i>	<i>PyTypeObject</i> * <i>Py_ssize_t</i>	<i>PyObject</i> *
<i>destructor</i>	void *	void
<i>freefunc</i>	void *	void
<i>traverseproc</i>	void * <i>visitproc</i> void *	int
<i>newfunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
<i>initproc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	int
<i>reprfunc</i>	<i>PyObject</i> *	<i>PyObject</i> *
<i>getattrfunc</i>	<i>PyObject</i> * const char *	<i>PyObject</i> *
<i>setattrfunc</i>	<i>PyObject</i> * const char * <i>PyObject</i> *	int
<i>getattrofunc</i>	<i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
<i>setattrofunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	int
<i>descrgetfunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
208 <i>descrsetfunc</i>	<i>PyObject</i> * <i>PyObject</i> *	int

See *Slot Type typedefs* below for more detail.

12.3.2 PyTypeObject Definition

The structure definition for *PyTypeObject* can be found in `Include/object.h`. For convenience of reference, this repeats the definition found there:

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrofunc tp_getattr;
    setattrofunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                   or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;

    /* Iterators */
}
```

(continua na próxima página)

```

getiterfunc tp_iter;
iternextfunc tp_iternext;

/* Attribute descriptor and subclassing stuff */
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
struct _typeobject *tp_base;
PyObject *tp_dict;
descrgetfunc tp_descr_get;
descrsetfunc tp_descr_set;
Py_ssize_t tp_dictoffset;
initproc tp_init;
allocfunc tp_alloc;
newfunc tp_new;
freefunc tp_free; /* Low-level free-memory routine */
inquiry tp_is_gc; /* For PyObject_IS_GC */
PyObject *tp_bases;
PyObject *tp_mro; /* method resolution order */
PyObject *tp_cache;
PyObject *tp_subclasses;
PyObject *tp_weaklist;
destructor tp_del;

/* Type attribute cache version tag. Added in version 2.6 */
unsigned int tp_version_tag;

destructor tp_finalize;
} PyTypeObject;

```

12.3.3 PyObject Slots

The type object structure extends the *PyVarObject* structure. The *ob_size* field is used for dynamic types (created by *type_new()*, usually called from a class statement). Note that *PyType_Type* (the metatype) initializes *tp_itemsize*, which means that its instances (i.e. type objects) *must* have the *ob_size* field.

*PyObject** **PyObject._ob_next**

*PyObject** **PyObject._ob_prev**

These fields are only present when the macro *Py_TRACE_REFS* is defined. Their initialization to *NULL* is taken care of by the *PyObject_HEAD_INIT* macro. For statically allocated objects, these fields always remain *NULL*. For dynamically allocated objects, these two fields are used to link the object into a doubly-linked list of *all* live objects on the heap. This could be used for various debugging purposes; currently the only use is to print the objects that are still alive at the end of a run when the environment variable *PYTHONDUMPREFS* is set.

Inheritance:

These fields are not inherited by subtypes.

Py_ssize_t **PyObject.ob_refcnt**

This is the type object's reference count, initialized to 1 by the *PyObject_HEAD_INIT* macro. Note that for statically allocated type objects, the type's instances (objects whose *ob_type* points back to the type) do *not* count as references. But for dynamically allocated type objects, the instances *do* count as references.

Inheritance:

This field is not inherited by subtypes.

***PyTypeObject** PyObject.ob_type**

This is the type's type, in other words its metatype. It is initialized by the argument to the `PyObject_HEAD_INIT` macro, and its value should normally be `&PyType_Type`. However, for dynamically loadable extension modules that must be usable on Windows (at least), the compiler complains that this is not a valid initializer. Therefore, the convention is to pass `NULL` to the `PyObject_HEAD_INIT` macro and to initialize this field explicitly at the start of the module's initialization function, before doing anything else. This is typically done like this:

```
Foo_Type.ob_type = &PyType_Type;
```

This should be done before any instances of the type are created. `PyType_Ready()` checks if `ob_type` is `NULL`, and if so, initializes it to the `ob_type` field of the base class. `PyType_Ready()` will not change this field if it is non-zero.

Inheritance:

This field is inherited by subtypes.

12.3.4 PyVarObject Slots

***Py_ssize_t* PyVarObject.ob_size**

For statically allocated type objects, this should be initialized to zero. For dynamically allocated type objects, this field has a special internal meaning.

Inheritance:

This field is not inherited by subtypes.

12.3.5 PyTypeObject Slots

Each slot has a section describing inheritance. If `PyType_Ready()` may set a value when the field is set to `NULL` then there will also be a “Default” section. (Note that many fields set on `PyBaseObject_Type` and `PyType_Type` effectively act as defaults.)

const char* PyTypeObject.tp_name

Pointer to a NUL-terminated string containing the name of the type. For types that are accessible as module globals, the string should be the full module name, followed by a dot, followed by the type name; for built-in types, it should be just the type name. If the module is a submodule of a package, the full package name is part of the full module name. For example, a type named `T` defined in module `M` in subpackage `Q` in package `P` should have the `tp_name` initializer `"P.Q.M.T"`.

For dynamically allocated type objects, this should just be the type name, and the module name explicitly stored in the type dict as the value for key `'__module__'`.

For statically allocated type objects, the `tp_name` field should contain a dot. Everything before the last dot is made accessible as the `__module__` attribute, and everything after the last dot is made accessible as the `__name__` attribute.

If no dot is present, the entire `tp_name` field is made accessible as the `__name__` attribute, and the `__module__` attribute is undefined (unless explicitly set in the dictionary, as explained above). This means your type will be impossible to pickle. Additionally, it will not be listed in module documentations created with `pydoc`.

This field must not be `NULL`. It is the only required field in `PyTypeObject()` (other than potentially `tp_itemsize`).

Inheritance:

This field is not inherited by subtypes.

Py_ssize_t **PyTypeObject.tp_basicsize**

Py_ssize_t **PyTypeObject.tp_itemsize**

These fields allow calculating the size in bytes of instances of the type.

There are two kinds of types: types with fixed-length instances have a zero *tp_itemsize* field, types with variable-length instances have a non-zero *tp_itemsize* field. For a type with fixed-length instances, all instances have the same size, given in *tp_basicsize*.

For a type with variable-length instances, the instances must have an *ob_size* field, and the instance size is *tp_basicsize* plus N times *tp_itemsize*, where N is the “length” of the object. The value of N is typically stored in the instance’s *ob_size* field. There are exceptions: for example, ints use a negative *ob_size* to indicate a negative number, and N is `abs(ob_size)` there. Also, the presence of an *ob_size* field in the instance layout doesn’t mean that the instance structure is variable-length (for example, the structure for the list type has fixed-length instances, yet those instances have a meaningful *ob_size* field).

The basic size includes the fields in the instance declared by the macro *PyObject_HEAD* or *PyObject_VAR_HEAD* (whichever is used to declare the instance struct) and this in turn includes the *_ob_prev* and *_ob_next* fields if they are present. This means that the only correct way to get an initializer for the *tp_basicsize* is to use the `sizeof` operator on the struct used to declare the instance layout. The basic size does not include the GC header size.

A note about alignment: if the variable items require a particular alignment, this should be taken care of by the value of *tp_basicsize*. Example: suppose a type implements an array of double. *tp_itemsize* is `sizeof(double)`. It is the programmer’s responsibility that *tp_basicsize* is a multiple of `sizeof(double)` (assuming this is the alignment requirement for double).

For any type with variable-length instances, this field must not be NULL.

Inheritance:

These fields are inherited separately by subtypes. If the base type has a non-zero *tp_itemsize*, it is generally not safe to set *tp_itemsize* to a different non-zero value in a subtype (though this depends on the implementation of the base type).

destructor **PyTypeObject.tp_dealloc**

A pointer to the instance destructor function. This function must be defined unless the type guarantees that its instances will never be deallocated (as is the case for the singletons `None` and `Ellipsis`). The function signature is:

```
void tp_dealloc(PyObject *self);
```

The destructor function is called by the *Py_DECREF()* and *Py_XDECREF()* macros when the new reference count is zero. At this point, the instance is still in existence, but there are no references to it. The destructor function should free all references which the instance owns, free all memory buffers owned by the instance (using the freeing function corresponding to the allocation function used to allocate the buffer), and call the type’s *tp_free* function. If the type is not subtypable (doesn’t have the *Py_TPFLAGS_BASETYPE* flag bit set), it is permissible to call the object deallocator directly instead of via *tp_free*. The object deallocator should be the one used to allocate the instance; this is normally *PyObject_Del()* if the instance was allocated using *PyObject_New()* or *PyObject_VarNew()*, or *PyObject_GC_Del()* if the instance was allocated using *PyObject_GC_New()* or *PyObject_GC_NewVar()*.

If the type supports garbage collection (has the *Py_TPFLAGS_HAVE_GC* flag bit set), the destructor should call *PyObject_GC_UnTrack()* before clearing any member fields.

```
static void foo_dealloc(foo_object *self) {
    PyObject_GC_UnTrack(self);
    Py_CLEAR(self->ref);
}
```

(continua na próxima página)

(continuação da página anterior)

```
Py_TYPE(self)->tp_free((PyObject *)self);
}
```

Finally, if the type is heap allocated (`Py_TPFLAGS_HEAPTYPE`), the deallocator should decrement the reference count for its type object after calling the type deallocator. In order to avoid dangling pointers, the recommended way to achieve this is:

```
static void foo_dealloc(foo_object *self) {
    PyTypeObject *tp = Py_TYPE(self);
    // free references and buffers here
    tp->tp_free(self);
    Py_DECREF(tp);
}
```

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()`.

Aviso: It is not recommended for *heap types* to implement the vectorcall protocol. When a user sets `__call__` in Python code, only *tp_call* is updated, likely making it inconsistent with the vectorcall function.

Nota: The semantics of the `tp_vectorcall_offset` slot are provisional and expected to be finalized in Python 3.9. If you use vectorcall, plan for updating your code for Python 3.9.

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.

Inheritance:

This field is always inherited. However, the `Py_TPFLAGS_HAVE_VECTORCALL` flag is not always inherited. If it's not, then the subclass won't use *vectorcall*, except when `PyVectorcall_Call()` is explicitly called. This is in particular the case for *heap types* (including subclasses defined in Python).

`getattrfunc PyTypeObject.tp_getattr`

An optional pointer to the get-attribute-string function.

This field is deprecated. When it is defined, it should point to a function that acts the same as the *tp_getattro* function, but taking a C string instead of a Python string object to give the attribute name.

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 **PyObject.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** **PyObject.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.

Novo 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 **PyObject.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.

Default:

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** **PyObject.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** **PyObject.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** PyObject.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* PyObject.tp_hash**

An optional pointer to a function that implements the built-in function `hash()`.

The signature is the same as for *PyObject_Hash()*:

```
Py_hash_t tp_hash(PyObject *);
```

The value `-1` should not be returned as a normal return value; when an error occurs during the computation of the hash value, the function should set an exception and return `-1`.

When this field is not set (and *tp_richcompare* is not set), an attempt to take the hash of the object raises `TypeError`. This is the same as setting it to *PyObject_HashNotImplemented()*.

This field can be set explicitly to *PyObject_HashNotImplemented()* to block inheritance of the hash method from a parent type. This is interpreted as the equivalent of `__hash__ = None` at the Python level, causing `isinstance(o, collections.Hashable)` to correctly return `False`. Note that the converse is also true - setting `__hash__ = None` on a class at the Python level will result in the *tp_hash* slot being set to *PyObject_HashNotImplemented()*.

Inheritance:

Group: *tp_hash*, *tp_richcompare*

This field is inherited by subtypes together with *tp_richcompare*: a subtype inherits both of *tp_richcompare* and *tp_hash*, when the subtype's *tp_richcompare* and *tp_hash* are both `NULL`.

***ternaryfunc* PyObject.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* PyObject.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.

Default:

When this field is not set, *PyObject_Repr()* is called to return a string representation.

***getattrofunc* PyObject.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.

Default:

PyBaseObject_Type uses *PyObject_GenericGetAttr()*.

***setattrofunc* PyObject.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.

Default:

PyBaseObject_Type uses *PyObject_GenericSetAttr()*.

***PyBufferProcs** PyObject.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 PyObject.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.

Default:

`PyBaseObject_Type` uses `Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE`.

Bit Masks:

The following bit masks are currently defined; these can be ORed together using the `|` operator to form the value of the `tp_flags` field. The macro `PyType_HasFeature()` takes a type and a flags value, `tp` and `f`, and checks whether `tp->tp_flags & f` is non-zero.

Py_TPFLAGS_HEAPTYPE

This bit is set when the type object itself is allocated on the heap, for example, types created dynamically using `PyType_FromSpec()`. In this case, the `ob_type` field of its instances is considered a reference to the type, and the type object is INCREf'ed when a new instance is created, and DECREf'ed when an instance is destroyed (this does not apply to instances of subtypes; only the type referenced by the instance's `ob_type` gets INCREf'ed or DECREf'ed).

Inheritance:

???

Py_TPFLAGS_BASETYPE

This bit is set when the type can be used as the base type of another type. If this bit is clear, the type cannot be subtyped (similar to a “final” class in Java).

Inheritance:

???

Py_TPFLAGS_READY

This bit is set when the type object has been fully initialized by `PyType_Ready()`.

Inheritance:

???

Py_TPFLAGS_READYING

This bit is set while `PyType_Ready()` is in the process of initializing the type object.

Inheritance:

???

Py_TPFLAGS_HAVE_GC

This bit is set when the object supports garbage collection. If this bit is set, instances must be created using `PyObject_GC_New()` and destroyed using `PyObject_GC_Del()`. More information in section *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`, `Py_TPFLAGS_HAVE_VERSION_TAG`.

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, **kwargs)` (with `obj` not `None`) must be equivalent to `meth(obj, *args, **kwargs)`.
- `meth.__get__(None, cls)(*args, **kwargs)` must be equivalent to `meth(*args, **kwargs)`.

This flag enables an optimization for typical method calls like `obj.meth()`: it avoids creating a temporary “bound method” object for `obj.meth`.

Novo na versão 3.8.

Inheritance:

This flag is never inherited by heap types. For extension types, it is inherited whenever `tp_descr_get` is inherited.

Py_TPFLAGS_LONG_SUBCLASS**Py_TPFLAGS_LIST_SUBCLASS****Py_TPFLAGS_TUPLE_SUBCLASS****Py_TPFLAGS_BYTES_SUBCLASS****Py_TPFLAGS_UNICODE_SUBCLASS****Py_TPFLAGS_DICT_SUBCLASS****Py_TPFLAGS_BASE_EXC_SUBCLASS****Py_TPFLAGS_TYPE_SUBCLASS**

These flags are used by functions such as `PyLong_Check()` to quickly determine if a type is a subclass of a built-in type; such specific checks are faster than a generic check, like `PyObject_IsInstance()`. Custom types that inherit from built-ins should have their `tp_flags` set appropriately, or the code that interacts with such types will behave differently depending on what kind of check is used.

Py_TPFLAGS_HAVE_FINALIZE

This bit is set when the `tp_finalize` slot is present in the type structure.

Novo na versão 3.4.

Obsoleto 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 for *static* subtypes if `tp_call` is also inherited. *Heap types* do not inherit `Py_TPFLAGS_HAVE_VECTORCALL`.

Novo na versão 3.9.

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.

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.

Heap-allocated types (`Py_TPFLAGS_HEAPTYPE`, such as those created with `PyType_FromSpec()` and similar APIs) hold a reference to their type. Their traversal function must therefore either visit `Py_TYPE(self)`, or delegate this responsibility by calling `tp_traverse` of another heap-allocated type (such as a heap-allocated superclass). If they do not, the type object may not be garbage-collected.

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 **`PyObject.tp_clear`**

An optional pointer to a clear function for the garbage collector. This is only used if the `Py_TPFLAGS_HAVE_GC` flag bit is set. The signature is:

```
int tp_clear(PyObject *);
```

The `tp_clear` member function is used to break reference cycles in cyclic garbage detected by the garbage collector. Taken together, all `tp_clear` functions in the system must combine to break all reference cycles. This is subtle, and if in any doubt supply a `tp_clear` function. For example, the tuple type does not implement a `tp_clear` function, because it's possible to prove that no reference cycle can be composed entirely of tuples. Therefore the `tp_clear` functions of other types must be sufficient to break any cycle containing a tuple. This isn't immediately obvious, and there's rarely a good reason to avoid implementing `tp_clear`.

Implementations of `tp_clear` should drop the instance's references to those of its members that may be Python objects, and set its pointers to those members to NULL, as in the following example:

```
static int
local_clear(localobject *self)
{
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

The `Py_CLEAR()` macro should be used, because clearing references is delicate: the reference to the contained object must not be decremented until after the pointer to the contained object is set to NULL. This is because decrementing the reference count may cause the contained object to become trash, triggering a chain of reclamation activity that may include invoking arbitrary Python code (due to finalizers, or weakref callbacks, associated with the contained object). If it's possible for such code to reference *self* again, it's important that the pointer to the contained object be NULL at that time, so that *self* knows the contained object can no longer be used. The `Py_CLEAR()` macro performs the operations in a safe order.

Note that `tp_clear` is not *always* called before an instance is deallocated. For example, when reference counting is enough to determine that an object is no longer used, the cyclic garbage collector is not involved and `tp_dealloc` is called directly.

Because the goal of `tp_clear` functions is to break reference cycles, it's not necessary to clear contained objects like Python strings or Python integers, which can't participate in reference cycles. On the other hand, it may be convenient to clear all contained Python objects, and write the type's `tp_dealloc` function to invoke `tp_clear`.

More information about Python's garbage collection scheme can be found in section *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 **`PyObject.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
<i>Py_LT</i>	<
<i>Py_LE</i>	<=
<i>Py_EQ</i>	==
<i>Py_NE</i>	!=
<i>Py_GT</i>	>
<i>Py_GE</i>	>=

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 return value's reference count is properly incremented.

On error, sets an exception and returns *NULL* from the function.

Novo 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*.

Default:

PyBaseObject_Type provides a *tp_richcompare* implementation, which may be inherited. However, if only *tp_hash* is defined, not even the inherited function is used and instances of the type will not be able to participate in any comparisons.

Py_ssize_t *PyTypeObject.tp_weaklistoffset*

If the instances of this type are weakly referenceable, this field is greater than zero and contains the offset in the instance structure of the weak reference list head (ignoring the GC header, if present); this offset is used by *PyObject_ClearWeakRefs()* and the *PyWeakref_*()* functions. The instance structure needs to include a field of type *PyObject** which is initialized to *NULL*.

Do not confuse this field with *tp_weaklist*; that is the list head for weak references to the type object itself.

Inheritance:

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype uses a different weak reference list head than the base type. Since the list head is always found via *tp_weaklistoffset*, this should not be a problem.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types are weakly referenceable, the type is made weakly referenceable by adding a weak reference list head slot to the instance layout and setting the `tp_weaklistoffset` of that slot's offset.

When a type's `__slots__` declaration contains a slot named `__weakref__`, that slot becomes the weak reference list head for instances of the type, and the slot's offset is stored in the type's `tp_weaklistoffset`.

When a type's `__slots__` declaration does not contain a slot named `__weakref__`, the type inherits its `tp_weaklistoffset` from its base type.

getterfunc **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).

*PyObject** **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).

Default:

This field defaults to *&PyBaseObject_Type* (which to Python programmers is known as the type object).

*PyObject** **PyTypeObject.tp_dict**

The type’s dictionary is stored here by *PyType_Ready()*.

This field should normally be initialized to NULL before *PyType_Ready* is called; it may also be initialized to a dictionary containing initial attributes for the type. Once *PyType_Ready()* has initialized the type, extra attributes for the type may be added to this dictionary only if they don’t correspond to overloaded operations (like *__add__()*).

Inheritance:

This field is not inherited by subtypes (though the attributes defined in here are inherited through a different mechanism).

Default:

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.

The function signature is:

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *type);
```

Inheritance:

This field is inherited by subtypes.

***descrsetfunc* PyTypeObject.tp_descr_set**

An optional pointer to a function for setting and deleting a descriptor's value.

The function signature is:

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

The *value* argument is set to NULL to delete the value.

Inheritance:

This field is inherited by subtypes.

***Py_ssize_t* PyTypeObject.tp_dictoffset**

If the instances of this type have a dictionary containing instance variables, this field is non-zero and contains the offset in the instances of the type of the instance variable dictionary; this offset is used by *PyObject_GenericGetAttr()*.

Do not confuse this field with *tp_dict*; that is the dictionary for attributes of the type object itself.

If the value of this field is greater than zero, it specifies the offset from the start of the instance structure. If the value is less than zero, it specifies the offset from the *end* of the instance structure. A negative offset is more expensive to use, and should only be used when the instance structure contains a variable-length part. This is used for example to add an instance variable dictionary to subtypes of *str* or *tuple*. Note that the *tp_basicsize* field should account for the dictionary added to the end in that case, even though the dictionary is not included in the basic object layout. On a system with a pointer size of 4 bytes, *tp_dictoffset* should be set to -4 to indicate that the dictionary is at the very end of the structure.

The real dictionary offset in an instance can be computed from a negative *tp_dictoffset* as follows:

```
dictoffset = tp_basicsize + abs(ob_size)*tp_itemsize + tp_dictoffset
if dictoffset is not aligned on sizeof(void*):
    round up to sizeof(void*)
```

where *tp_basicsize*, *tp_itemsize* and *tp_dictoffset* are taken from the type object, and *ob_size* is taken from the instance. The absolute value is taken because ints use the sign of *ob_size* to store the sign of the number. (There's never a need to do this calculation yourself; it is done for you by *_PyObject_GetDictPtr()*.)

Inheritance:

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype instances store the dictionary at a difference offset than the base type. Since the dictionary is always found via *tp_dictoffset*, this should not be a problem.

When a type defined by a class statement has no *__slots__* declaration, and none of its base types has an instance variable dictionary, a dictionary slot is added to the instance layout and the *tp_dictoffset* is set to that slot's offset.

When a type defined by a class statement has a *__slots__* declaration, the type inherits its *tp_dictoffset* from its base type.

(Adding a slot named *__dict__* to the *__slots__* declaration does not have the expected effect, it just causes confusion. Maybe this should be added as a feature just like *__weakref__* though.)

Default:

This slot has no default. For static types, if the field is NULL then no *__dict__* gets created for instances.

***initproc* PyTypeObject.tp_init**

An optional pointer to an instance initialization function.

This function corresponds to the `__init__()` method of classes. Like `__init__()`, it is possible to create an instance without calling `__init__()`, and it is possible to reinitialize an instance by calling its `__init__()` method again.

The function signature is:

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwargs);
```

The *self* argument is the instance to be initialized; the *args* and *kwargs* 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.

Default:

For static types this field does not have a default.

allocfunc **PyTypeObject.tp_alloc**

An optional pointer to an instance allocation function.

The function signature is:

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems);
```

Inheritance:

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement).

Default:

For dynamic subtypes, this field is always set to *PyType_GenericAlloc()*, to force a standard heap allocation strategy.

For static subtypes, *PyBaseObject_Type* uses *PyType_GenericAlloc()*. That is the recommended value for all statically defined types.

newfunc **PyTypeObject.tp_new**

An optional pointer to an instance creation function.

The function signature is:

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwargs);
```

The *subtype* argument is the type of the object being created; the *args* and *kwargs* 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*.

Inheritance:

This field is inherited by subtypes, except it is not inherited by static types whose `tp_base` is `NULL` or `&PyBaseObject_Type`.

Default:

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)

Default:

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.

Default:

This slot has no default. If this field is `NULL`, `Py_TPFLAGS_HAVE_GC` is used as the functional equivalent.

*PyObject** **PyTypeObject.tp_bases**

Tuple of base types.

This is set for types created by a class statement. It should be `NULL` for statically defined types.

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.

Inheritance:

This field is not inherited; it is calculated fresh by `PyType_Ready()`.

*PyObject** **PyTypeObject.tp_cache**

Unused. Internal use only.

Inheritance:

This field is not inherited.

*PyObject** **PyTypeObject.tp_subclasses**

List of weak references to subclasses. Internal use only.

Inheritance:

This field is not inherited.

*PyObject** **PyTypeObject.tp_weaklist**

Weak reference list head, for weak references to this type object. Not inherited. Internal use only.

Inheritance:

This field is not inherited.

destructor **PyTypeObject.tp_del**

This field is deprecated. Use *tp_finalize* instead.

unsigned int **PyTypeObject.tp_version_tag**

Used to index into the method cache. Internal use only.

Inheritance:

This field is not inherited.

destructor **PyTypeObject.tp_finalize**

An optional pointer to an instance finalization function. Its signature is:

```
void tp_finalize(PyObject *self);
```

If *tp_finalize* is set, the interpreter calls it once when finalizing an instance. It is called either from the garbage collector (if the instance is part of an isolated reference cycle) or just before the object is deallocated. Either way, it is guaranteed to be called before attempting to break reference cycles, ensuring that it finds the object in a sane state.

tp_finalize should not mutate the current exception status; therefore, a recommended way to write a non-trivial finalizer is:

```
static void
local_finalize(PyObject *self)
{
    PyObject *error_type, *error_value, *error_traceback;

    /* Save the current exception, if any. */
    PyErr_Fetch(&error_type, &error_value, &error_traceback);

    /* ... */

    /* Restore the saved exception. */
    PyErr_Restore(error_type, error_value, error_traceback);
}
```

For this field to be taken into account (even through inheritance), you must also set the *Py_TPFLAGS_HAVE_FINALIZE* flags bit.

Also, note that, in a garbage collected Python, *tp_dealloc* may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a

garbage collection on any thread). This is not a problem for Python API calls, since the thread on which `tp_dealloc` is called will own the Global Interpreter Lock (GIL). However, if the object being destroyed in turn destroys objects from some other C or C++ library, care should be taken to ensure that destroying those objects on the thread which called `tp_dealloc` will not violate any assumptions of the library.

Inheritance:

This field is inherited by subtypes.

Novo na versão 3.4.

Ver também:

“Safe object finalization” ([PEP 442](#))

vectorcallfunc `PyTypeObject.tp_vectorcall`

Vectorcall function to use for calls of this type object. In other words, it is used to implement *vectorcall* for `type.__call__`. If `tp_vectorcall` is `NULL`, the default call implementation using `__new__` and `__init__` is used.

Inheritance:

This field is never inherited.

Novo na versão 3.9: (the field exists since 3.8 but it's only used since 3.9)

12.3.6 Heap Types

Traditionally, types defined in C code are *static*, that is, a static *PyTypeObject* structure is defined directly in code and initialized using *PyType_Ready()*.

This results in types that are limited relative to types defined in Python:

- Static types are limited to one base, i.e. they cannot use multiple inheritance.
- Static type objects (but not necessarily their instances) are immutable. It is not possible to add or modify the type object's attributes from Python.
- Static type objects are shared across *sub-interpreters*, so they should not include any subinterpreter-specific state.

Also, since *PyTypeObject* is not part of the *stable ABI*, any extension modules using static types must be compiled for a specific Python minor version.

An alternative to static types is *heap-allocated types*, or *heap types* for short, which correspond closely to classes created by Python's `class` statement.

This is done by filling a *PyType_Spec* structure and calling *PyType_FromSpecWithBases()*.

12.4 Number Object Structures

PyNumberMethods

This structure holds pointers to the functions which an object uses to implement the number protocol. Each function is used by the function of similar name documented in the *Protocolo de número* section.

Here is the structure definition:

```
typedef struct {  
    binaryfunc nb_add;  
    binaryfunc nb_subtract;
```

(continua na próxima página)

(continuação da página anterior)

```

    binaryfunc nb_multiply;
    binaryfunc nb_remainder;
    binaryfunc nb_divmod;
    ternaryfunc nb_power;
    unaryfunc nb_negative;
    unaryfunc nb_positive;
    unaryfunc nb_absolute;
    inquiry nb_bool;
    unaryfunc nb_invert;
    binaryfunc nb_lshift;
    binaryfunc nb_rshift;
    binaryfunc nb_and;
    binaryfunc nb_xor;
    binaryfunc nb_or;
    unaryfunc nb_int;
    void *nb_reserved;
    unaryfunc nb_float;

    binaryfunc nb_inplace_add;
    binaryfunc nb_inplace_subtract;
    binaryfunc nb_inplace_multiply;
    binaryfunc nb_inplace_remainder;
    ternaryfunc nb_inplace_power;
    binaryfunc nb_inplace_lshift;
    binaryfunc nb_inplace_rshift;
    binaryfunc nb_inplace_and;
    binaryfunc nb_inplace_xor;
    binaryfunc nb_inplace_or;

    binaryfunc nb_floor_divide;
    binaryfunc nb_true_divide;
    binaryfunc nb_inplace_floor_divide;
    binaryfunc nb_inplace_true_divide;

    unaryfunc nb_index;

    binaryfunc nb_matrix_multiply;
    binaryfunc nb_inplace_matrix_multiply;
} PyNumberMethods;

```

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`

12.5 Mapping Object Structures

PyMappingMethods

This structure holds pointers to the functions which an object uses to implement the mapping protocol. It has three members:

lenfunc **PyMappingMethods.mp_length**

This function is used by *PyMapping_Size()* and *PyObject_Size()*, and has the same signature. This slot may be set to NULL if the object has no defined length.

binaryfunc **PyMappingMethods.mp_subscript**

This function is used by *PyObject_GetItem()* and *PySequence_GetSlice()*, and has the same signature as *PyObject_GetItem()*. This slot must be filled for the *PyMapping_Check()* function to return 1, it can be NULL otherwise.

objobjargproc **PyMappingMethods.mp_ass_subscript**

This function is used by *PyObject_SetItem()*, *PyObject_DelItem()*, *PyObject_SetSlice()* and *PyObject_DelSlice()*. It has the same signature as *PyObject_SetItem()*, but *v* can also be set to NULL to delete an item. If this slot is NULL, the object does not support item assignment and deletion.

12.6 Sequence Object Structures

PySequenceMethods

This structure holds pointers to the functions which an object uses to implement the sequence protocol.

lenfunc **PySequenceMethods.sq_length**

This function is used by *PySequence_Size()* and *PyObject_Size()*, and has the same signature. It is also used for handling negative indices via the *sq_item* and the *sq_ass_item* slots.

binaryfunc **PySequenceMethods.sq_concat**

This function is used by *PySequence_Concat()* and has the same signature. It is also used by the *+* operator, after trying the numeric addition via the *nb_add* slot.

ssizeargfunc **PySequenceMethods.sq_repeat**

This function is used by *PySequence_Repeat()* and has the same signature. It is also used by the *** operator, after trying numeric multiplication via the *nb_multiply* slot.

ssizeargfunc **PySequenceMethods.sq_item**

This function is used by *PySequence_GetItem()* and has the same signature. It is also used by *PyObject_GetItem()*, after trying the subscription via the *mp_subscript* slot. This slot must be filled for the *PySequence_Check()* function to return 1, it can be NULL otherwise.

Negative indexes are handled as follows: if the *sq_length* slot is filled, it is called and the sequence length is used to compute a positive index which is passed to *sq_item*. If *sq_length* is NULL, the index is passed as is to the function.

ssizeobjargproc **PySequenceMethods.sq_ass_item**

This function is used by *PySequence_SetItem()* and has the same signature. It is also used by *PyObject_SetItem()* and *PyObject_DelItem()*, after trying the item assignment and deletion via the *mp_ass_subscript* slot. This slot may be left to NULL if the object does not support item assignment and deletion.

objobjproc **PySequenceMethods.sq_contains**

This function may be used by *PySequence_Contains()* and has the same signature. This slot may be left to NULL, in this case *PySequence_Contains()* simply traverses the sequence until it finds a match.

***binaryfunc* PySequenceMethods.sq_inplace_concat**

This function is used by *PySequence_InPlaceConcat()* and has the same signature. It should modify its first operand, and return it. This slot may be left to NULL, in this case *PySequence_InPlaceConcat()* will fall back to *PySequence_Concat()*. It is also used by the augmented assignment `+=`, after trying numeric in-place addition via the *nb_inplace_add* slot.

***ssizeargfunc* PySequenceMethods.sq_inplace_repeat**

This function is used by *PySequence_InPlaceRepeat()* and has the same signature. It should modify its first operand, and return it. This slot may be left to NULL, in this case *PySequence_InPlaceRepeat()* will fall back to *PySequence_Repeat()*. It is also used by the augmented assignment `*=`, after trying numeric in-place multiplication via the *nb_inplace_multiply* slot.

12.7 Buffer Object Structures

PyBufferProcs

This structure holds pointers to the functions required by the *Buffer protocol*. The protocol defines how an exporter object can expose its internal data to consumer objects.

***getbufferproc* PyBufferProcs.bf_getbuffer**

The signature of this function is:

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

Handle a request to *exporter* to fill in *view* as specified by *flags*. Except for point (3), an implementation of this function MUST take these steps:

- (1) Check if the request can be met. If not, raise *PyExc_BufferError*, set *view->obj* to NULL and return -1.
- (2) Fill in the requested fields.
- (3) Increment an internal counter for the number of exports.
- (4) Set *view->obj* to *exporter* and increment *view->obj*.
- (5) Return 0.

If *exporter* is part of a chain or tree of buffer providers, two main schemes can be used:

- Re-export: Each member of the tree acts as the exporting object and sets *view->obj* to a new reference to itself.
- Redirect: The buffer request is redirected to the root object of the tree. Here, *view->obj* will be a new reference to the root object.

The individual fields of *view* are described in section *Buffer structure*, the rules how an exporter must react to specific requests are in section *Buffer request types*.

All memory pointed to in the *Py_buffer* structure belongs to the exporter and must remain valid until there are no consumers left. *format*, *shape*, *strides*, *suboffsets* and *internal* are read-only for the consumer.

PyBuffer_FillInfo() provides an easy way of exposing a simple bytes buffer while dealing correctly with all request types.

PyObject_GetBuffer() is the interface for the consumer that wraps this function.

***releasebufferproc* PyBufferProcs.bf_releasebuffer**

The signature of this function is:

```
void (PyObject *exporter, Py_buffer *view);
```

Handle a request to release the resources of the buffer. If no resources need to be released, *PyBufferProcs.bf_releasebuffer* may be NULL. Otherwise, a standard implementation of this function will take these optional steps:

- (1) Decrement an internal counter for the number of exports.
- (2) If the counter is 0, free all memory associated with *view*.

The exporter MUST use the *internal* field to keep track of buffer-specific resources. This field is guaranteed to remain constant, while a consumer MAY pass a copy of the original buffer as the *view* argument.

This function MUST NOT decrement *view->obj*, since that is done automatically in *PyBuffer_Release()* (this scheme is useful for breaking reference cycles).

PyBuffer_Release() is the interface for the consumer that wraps this function.

12.8 Async Object Structures

Novo na versão 3.5.

PyAsyncMethods

This structure holds pointers to the functions required to implement *awaitable* and *asynchronous iterator* objects.

Here is the structure definition:

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
} PyAsyncMethods;
```

unaryfunc **PyAsyncMethods.am_await**

The signature of this function is:

```
PyObject *am_await(PyObject *self);
```

The returned object must be an iterator, i.e. *PyIter_Check()* must return 1 for it.

This slot may be set to NULL if an object is not an *awaitable*.

unaryfunc **PyAsyncMethods.am_aiter**

The signature of this function is:

```
PyObject *am_aiter(PyObject *self);
```

Must return an *asynchronous iterator* object. See *__anext__()* for details.

This slot may be set to NULL if an object does not implement asynchronous iteration protocol.

unaryfunc **PyAsyncMethods.am_anext**

The signature of this function is:

```
PyObject *am_anext(PyObject *self);
```

Must return an *awaitable* object. See *__anext__()* for details. This slot may be set to NULL.

12.9 Slot Type typedefs

PyObject * (***allocfunc**) (*PyTypeObject* *cls, *Py_ssize_t* nitems)

The purpose of this function is to separate memory allocation from memory initialization. It should return a pointer to a block of memory of adequate length for the instance, suitably aligned, and initialized to zeros, but with `ob_refcnt` set to 1 and `ob_type` set to the type argument. If the type's `tp_itemsize` is non-zero, the object's `ob_size` field should be initialized to `nitems` and the length of the allocated memory block should be `tp_basicsize + nitems*tp_itemsize`, rounded up to a multiple of `sizeof(void*)`; otherwise, `nitems` is not used and the length of the block should be `tp_basicsize`.

This function should not do any other instance initialization, not even to allocate additional memory; that should be done by `tp_new`.

void (***destructor**) (*PyObject* *)

void (***freefunc**) (void *)

See `tp_free`.

PyObject * (***newfunc**) (*PyObject* *, *PyObject* *, *PyObject* *)

See `tp_new`.

int (***initproc**) (*PyObject* *, *PyObject* *, *PyObject* *)

See `tp_init`.

PyObject * (***reprfunc**) (*PyObject* *)

See `tp_repr`.

PyObject * (***getattrfunc**) (*PyObject* *self, char *attr)

Return the value of the named attribute for the object.

int (***setattrfunc**) (*PyObject* *self, char *attr, *PyObject* *value)

Set the value of the named attribute for the object. The value argument is set to `NULL` to delete the attribute.

PyObject * (***getattrofunc**) (*PyObject* *self, *PyObject* *attr)

Return the value of the named attribute for the object.

See `tp_getattro`.

int (***setattrofunc**) (*PyObject* *self, *PyObject* *attr, *PyObject* *value)

Set the value of the named attribute for the object. The value argument is set to `NULL` to delete the attribute.

See `tp_setattro`.

PyObject * (***descrgetfunc**) (*PyObject* *, *PyObject* *, *PyObject* *)

See `tp_descrget`.

int (***descrsetfunc**) (*PyObject* *, *PyObject* *, *PyObject* *)

See `tp_descrset`.

Py_hash_t (***hashfunc**) (*PyObject* *)

See `tp_hash`.

PyObject * (***richcmpfunc**) (*PyObject* *, *PyObject* *, int)

See `tp_richcompare`.

PyObject * (***getiterfunc**) (*PyObject* *)

See `tp_iter`.

PyObject * (***iternextfunc**) (*PyObject* *)

See `tp_iternext`.

Py_ssize_t (***lenfunc**) (*PyObject* *)

```

int (*getbufferproc) (PyObject *, Py_buffer *, int)
void (*releasebufferproc) (PyObject *, Py_buffer *)
PyObject* (*unaryfunc) (PyObject *)
PyObject* (*binaryfunc) (PyObject *, PyObject *)
PyObject* (*ternaryfunc) (PyObject *, PyObject *, PyObject *)
PyObject* (*ssizeargfunc) (PyObject *, Py_ssize_t)
int (*ssizeobjargproc) (PyObject *, Py_ssize_t)
int (*objobjproc) (PyObject *, PyObject *)
int (*objobjargproc) (PyObject *, PyObject *, PyObject *)

```

12.10 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;
} PyObject;

static PyTypeObject PyObject_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 PyObject_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 */
};

```

(continua na próxima página)

(continuação da página anterior)

```

0,                /* tp_str */
0,                /* tp_getattro */
0,                /* tp_setattro */
0,                /* tp_as_buffer */
0,                /* tp_flags */
PyDoc_STR("My objects"), /* tp_doc */
0,                /* tp_traverse */
0,                /* tp_clear */
0,                /* tp_richcompare */
0,                /* tp_weaklistoffset */
0,                /* tp_iter */
0,                /* tp_iternext */
0,                /* tp_methods */
0,                /* tp_members */
0,                /* tp_getset */
0,                /* tp_base */
0,                /* tp_dict */
0,                /* tp_descr_get */
0,                /* tp_descr_set */
0,                /* tp_dictoffset */
0,                /* tp_init */
0,                /* tp_alloc */
myobj_new,        /* tp_new */
};

```

A type that supports weakrefs, instance dicts, and hashing:

```

typedef struct {
    PyObject_HEAD
    const char *data;
    PyObject *inst_dict;
    PyObject *weakreflist;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_weaklistoffset = offsetof(MyObject, weakreflist),
    .tp_dictoffset = offsetof(MyObject, inst_dict),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
    .tp_new = myobj_new,
    .tp_traverse = (traverseproc)myobj_traverse,
    .tp_clear = (inquiry)myobj_clear,
    .tp_alloc = PyType_GenericNew,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
    .tp_hash = (hashfunc)myobj_hash,
    .tp_richcompare = PyBaseObject_Type.tp_richcompare,
};

```

A str subclass that cannot be subclassed and cannot be called to create instances (e.g. uses a separate factory func):

```

typedef struct {
    PyUnicodeObject raw;
    char *extra;
};

```

(continua na próxima página)

(continuação da página anterior)

```

} MyStr;

static PyObject MyStr_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyStr",
    .tp_basicsize = sizeof(MyStr),
    .tp_base = NULL, // set to &PyUnicode_Type in module init
    .tp_doc = PyDoc_STR("my custom str"),
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = NULL,
    .tp_repr = (reprfunc)myobj_repr,
};

```

The simplest static type (with fixed-length instances):

```

typedef struct {
    PyObject_HEAD
} MyObject;

static PyObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
};

```

The simplest static type (with variable-length instances):

```

typedef struct {
    PyObject_VAR_HEAD
    const char *data[1];
} MyObject;

static PyObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject) - sizeof(char *),
    .tp_itemsize = sizeof(char *),
};

```

12.11 Suporte a Coleta Cíclica de Lixo

O suporte do Python para detectar e coletar o lixo, que envolve referências circulares, requer suporte dos tipos de objetos que são “contêineres” para outros objetos que também podem ser contêineres. 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 explícito para coleta de lixo.

Para criar um tipo container, o `tp_flags` campo do tipo do objeto deve incluir o `Py_TPFLAGS_HAVE_GC` e fornecer uma implementação do `c:member:~PyObject.tp_traverse` manipulador. Se as instâncias do tipo forem mutáveis, uma `c:member:~PyObject.tp_clear` implementação também deverá ser fornecida.

Py_TPFLAGS_HAVE_GC

Objetos com esse tipo de sinalizador definido devem seguir as regras documentadas aqui. Por conveniência esses objetos serão referenciados como objetos de contêiner.

Construtores para tipos de contêiner devem obedecer a duas regras:

1. A memória para o objeto deve ser alocada usando `PyObject_GC_New()` ou `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.

TYPE* **PyObject_GC_New**(TYPE, *PyTypeObject* *type)

Analogous to `PyObject_New()` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

TYPE* **PyObject_GC_NewVar**(TYPE, *PyTypeObject* *type, *Py_ssize_t* size)

Analogous to `PyObject_NewVar()` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

TYPE* **PyObject_GC_Resize**(TYPE, *PyVarObject* *op, *Py_ssize_t* newsize)

Resize an object allocated by `PyObject_NewVar()`. Returns the resized object or NULL on failure. *op* must not be tracked by the collector yet.

void **PyObject_GC_Track**(*PyObject* *op)

Adds the object *op* to the set of container objects tracked by the collector. The collector can run at unexpected times so objects must be valid while being tracked. This should be called once all the fields followed by the `tp_traverse` handler become valid, usually near the end of the constructor.

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)

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()`.

Novo na versão 3.9.

int **PyObject_GC_IsFinalized**(*PyObject* *op)

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()`.

Novo na versão 3.9.

void **PyObject_GC_Del**(void *op)

Releases memory allocated to an object using `PyObject_GC_New()` or `PyObject_GC_NewVar()`.

void **PyObject_GC_UnTrack** (void **op*)

Remove the object *op* from the set of container objects tracked by the collector. Note that `PyObject_GC_Track()` can be called again on this object to add it back to the set of tracked objects. The deallocator (`tp_dealloc` handler) should call this for the object before any of the fields used by the `tp_traverse` handler become invalid.

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:

int (***visitproc**) (*PyObject* **object*, void **arg*)

Type of the visitor function passed to the `tp_traverse` handler. The function should be called with an object to traverse as *object* and the third parameter to the `tp_traverse` handler as *arg*. The Python core uses several visitor functions to implement cyclic garbage detection; it's not expected that users will need to write their own visitor functions.

The `tp_traverse` handler must have the following type:

int (***traverseproc**) (*PyObject* **self*, *visitproc* *visit*, void **arg*)

Traversal function for a container object. Implementations must call the *visit* function for each object directly contained by *self*, with the parameters to *visit* being the contained object and the *arg* value passed to the handler. The *visit* function must not be called with a NULL object argument. If *visit* returns a non-zero value that value should be returned immediately.

To simplify writing `tp_traverse` handlers, a `Py_VISIT()` macro is provided. In order to use this macro, the `tp_traverse` implementation must name its arguments exactly *visit* and *arg*:

void **Py_VISIT** (*PyObject* **o*)

If *o* is not NULL, call the *visit* callback, with arguments *o* and *arg*. If *visit* returns a non-zero value, then return it. Using this macro, `tp_traverse` handlers look like:

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

The `tp_clear` handler must be of the `inquiry` type, or NULL if the object is immutable.

int (***inquiry**) (*PyObject* **self*)

Drop references that may have created reference cycles. Immutable objects do not have to define this method since they can never directly create reference cycles. Note that the object must still be valid after calling this method (don't just call `Py_DECREF()` on a reference). The collector will call this method if it detects that this object is involved in a reference cycle.

API e Versionamento ABI

`PY_VERSION_HEX` É o número da versão do Python codificado em um único inteiro.

Por exemplo se o `PY_VERSION_HEX` está configurado para `0x030401a2`, a informações de versão subjacente pode ser encontrada tratando-a como um numero de 32 bit da seguinte maneira:

By-tes	Bits (big endian order)	Significado
1	1-8	<code>PY_MAJOR_VERSION</code> (O 3 em 3.4.1a2)
2	9-16	<code>PY_MINOR_VERSION</code> (O 4 em 3.4.1a2)
3	17-24	<code>PY_MICRO_VERSION</code> (O 1 em 3.4.1a2)
4	25-28	<code>PY_RELEASE_LEVEL</code> (0xA para alfa, 0xB para beta, 0xC para release candidate e 0xF para final), que nesse caso é alfa.
	29-32	<code>PY_RELEASE_SERIAL</code> (O 2 em 3.4.1a2, zero para lançamentos finais)

Portanto “3.4.1a2” é em versão hexa “0x03041a2”.

Todas as macros dadas estão definidas em [Include/patchlevel.h](#).

>>> 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 shell 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`.

2to3 Uma ferramenta que tenta converter código Python 2.x em código Python 3.x tratando a maioria das incompatibilidades que podem ser detectadas com análise do código-fonte e navegação na árvore sintática.

O 2to3 está disponível na biblioteca padrão como `lib2to3`; um ponto de entrada é disponibilizado como `Tools/scripts/2to3`. Veja [2to3-reference](#).

classe base abstrata 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). CBAs 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 CBAs 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 CBAs 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.

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 do local (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 referenciado pelo nome separado por um ponto. Por exemplo, se um objeto *o* tem um atributo *a* esse seria referenciado como *o.a*.

aguardável Um objeto que pode ser usado em uma expressão `await`. Pode ser uma *corrotina* ou um objeto com um método `__await__()`. Veja também [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 byte 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`.

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`.

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.

class 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).

coerção A conversão implícita de uma instância de um tipo para outro durante uma operação que envolve dois argumentos do mesmo tipo. Por exemplo, `int(3.15)` converte o número do ponto flutuante no número inteiro 3, mas em `3+4.5`, cada argumento é de um tipo diferente (um `int`, um `float`), e ambos devem ser convertidos para o mesmo tipo antes de poderem ser adicionados ou isso levantará um `TypeError`. Sem coerção, todos os argumentos de tipos compatíveis teriam que ser normalizados com o mesmo valor pelo programador, por exemplo, `float(3)+4.5` em vez de apenas `3+4.5`.

número complexo Uma extensão ao familiar sistema de números reais em que todos os números são expressos como uma soma de uma parte real e uma parte imaginária. Números imaginários são múltiplos reais da unidade imaginária (a raiz quadrada de -1), normalmente escrita como `i` em matemática ou `j` em engenharia. O Python tem suporte nativo para números complexos, que são escritos com esta última notação; a parte imaginária escrita com um sufixo `j`, p.ex., `3+1j`. Para ter acesso aos equivalentes para números complexos do módulo `math`, utilize `cmath`. O uso de números complexos é uma funcionalidade matemática bastante avançada. Se você não sabe se irá precisar deles, é quase certo que você pode ignorá-los sem problemas.

gerenciador de contexto Um objeto que controla o ambiente visto numa instrução `with` por meio da definição dos métodos `__enter__()` e `__exit__()`. Veja [PEP 343](#).

variável de contexto Uma variável que pode ter valores diferentes, dependendo do seu contexto. Isso é semelhante ao armazenamento local de threads, no qual cada thread pode ter um valor diferente para uma variável. No entanto, com variáveis de contexto, pode haver vários contextos em uma thread e o principal uso para variáveis de contexto é acompanhar as variáveis em tarefas assíncronas simultâneas. Veja `contextvars`.

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.

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.

descriptor Qualquer objeto que define os métodos `__get__()`, `__set__()` ou `__delete__()`. Quando um atributo de classe é um descriptor, 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 descriptor, o respectivo método descriptor é chamado. Compreender descriptors 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 descriptors, veja: `descriptors` ou o Guia de Descriptors.

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__()`. Dicionários são estruturas chamadas de hash na linguagem 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 em Python assume a existência de chaves ou atributos válidos e captura exceções caso essa premissa se prove falsa. Este estilo limpo e rápido se caracteriza pela presença de várias instruções `try` e `except`. A técnica diverge do estilo *LBYL*, comum em outras linguagens como C, por exemplo.

expressão 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*.

localizador Um objeto que tenta encontrar o *carregador* para um módulo que está sendo importado.

Desde o Python 3.3, 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 [PEP 302](#), [PEP 420](#) e [PEP 451](#) para mais informações.

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 `-2.75` arredondado *para baixo*. Consulte a [PEP 238](#).

função Uma série de instruções que retorna algum valor para um chamador. Também pode ser passado zero ou mais *argumentos* que podem ser usados na execução do corpo. Veja também *parâmetro*, *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 sum_two_numbers(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 essa funcionalidade.

__future__ A future statement, `from __future__ import <feature>`, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The

`__future__` module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:

```
>>> 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 local (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 loop, um `range`, e uma cláusula `if` opcional. A expressão combinada gera valores para uma função encapsuladora:

```
>>> sum(i*i for i in range(10))           # sum of squares 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 A *type* that can be parameterized; typically a container class such as `list` or `dict`. Used for *type hints* and *annotations*.

For more details, see generic alias types, **PEP 483**, **PEP 484**, **PEP 585**, and the `typing` module.

GIL Veja *bloqueio global do interpretador*.

bloqueio global do interpretador O mecanismo utilizado pelo interpretador *CPython* para garantir que apenas uma thread execute o *bytecode* Python por vez. Isto simplifica a implementação do CPython ao fazer com que o modelo de objetos (incluindo tipos 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 o GIL ao realizar tarefas computacionalmente muito intensas, como compactação ou cálculos de hash. Além disso, o GIL é sempre liberado nas operações de E/S.

No passado, esforços para criar um interpretador que lidasse plenamente com threads (travando dados compartilhados numa granularidade bem mais fina) não foram bem sucedidos devido a queda no desempenho ao serem executados em processadores de apenas um núcleo. Acredita-se que superar essa questão de desempenho acabaria tornando a implementação muito mais complicada e bem mais difícil de manter.

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 integrado para Python. IDLE é um editor básico e um ambiente interpretador que vem junto com a distribuição padrão do Python.

imutável Um objeto que possui um valor fixo. Objetos imutáveis incluem números, strings e tuplas. Estes objetos não podem ser alterados. Um novo objeto deve ser criado se um valor diferente tiver de ser armazenado. Objetos imutáveis têm um papel importante em lugares onde um valor constante de hash seja necessário, como por exemplo uma chave em um dicionário.

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)`).

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.

iterable 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 não sequenciais como `dict`, *objeto arquivo*, e objetos de qualquer classe que você definir com um método `__iter__()` ou com um método `__getitem__()` que implemente 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 nativa `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` exception 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`.

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 em 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, o módulo `operator` dispõe de três construtores para funções chave: `attrgetter()`, `itemgetter()` e o `methodcaller()`. Consulte o *HowTo* 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 bloqueios ou usando a abordagem *EAFP*.

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. Deve definir um método chamado `load_module()`. Um carregador é normalmente devolvido por um *localizador*. Veja a **PEP 302** para detalhes e `importlib.abc.Loader` para um *classe base abstrata*.

método mágico Um sinônimo informal para um *método especial*.

mapeamento Um objeto contêiner que suporta buscas por chaves arbitrárias e implementa os métodos especificados em classes base abstratas `Mapping` ou `MutableMapping`. 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.

metaclasses A classe de uma classe. Definições de classe criam um nome de classe, um dicionário de classe e uma lista de classes base. A metaclasses 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 *A ordem de resolução de métodos do Python 2.3* 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*.

módulo spec Um espaço de nomes que contém as informações relacionadas à importação usadas para carregar um módulo. Uma instância de `importlib.machinery.ModuleSpec`.

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]           # indexed access
1024
>>> sys.float_info.max_exp      # named field access
1024
>>> isinstance(sys.float_info, tuple) # kind of tuple
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 com uma função fábrica `collections.namedtuple()`. A segunda técnica também adiciona 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 tornar 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* da [PEP 420](#) que serve apenas como container para sub pacotes. 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`.

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, `__getattr__()`, métodos de classe, e métodos estáticos.

object Qualquer dado que tenha estado (atributos ou valores) e comportamento definidos (métodos). Também a última classe base de qualquer *classe estilo novo*.

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 especifica 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 *posonly1* e *posonly2* a seguir:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *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 *kw_only1* and *kw_only2* a seguir:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *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 o termo *argumento* no glossário, a pergunta 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_hook` 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* 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 os idiomas mais comuns da linguagem Python, ao invés de implementar códigos usando conceitos comuns a outros idiomas. Por exemplo, um idioma comum em Python é fazer um loop sobre todos os elementos de uma iterável usando a instrução `for`. Muitas outras linguagens não têm esse tipo de construção, então as pessoas que não estão familiarizadas com o Python usam um contador numérico:

```
for i in range(len(food)):
    print(food[i])
```

Ao contrário do método limpo, ou então, Pythônico:

```
for piece in food:
    print(piece)
```

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 meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

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 para um objeto. Quando a contagem de referências de um objeto atinge zero, ele é desalocado. Contagem de referências geralmente não é visível no código Python, mas é um elemento chave da implementação *CPython*. O módulo `sys` define a função `getrefcount()` que programadores podem chamar para retornar a contagem de referências para um objeto em particular.

pacote regular Um *pacote* tradicional, como um diretório contendo um arquivo `__init__.py`.

Veja também *pacote de espaço de nomes*.

__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.

sequence 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 mapa e não uma sequência porque a busca usa uma chave *imutável* 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()`.

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 [compreensions](#).

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.

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`.

codificador de texto A string in Python is a sequence of Unicode code points (in range U+0000–U+10FFFF). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as “encoding”, and recreating the string from the sequence of bytes is known as “decoding”.

There are a variety of different text serialization codecs, which are collectively referred to as “text encodings”.

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`.

Vea também *arquivo binário* para um objeto arquivo apto a ler e escrever *objetos byte ou similar*.

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 tipo de objeto ele é; cada objeto tem um tipo. Um tipo de objeto é acessível pelo atributo `__class__` ou pode ser recuperado com `type(obj)`.

tipo alias Um sinônimo para um tipo, criado através da atribuição do tipo para um identificador.

Tipos alias são úteis para simplificar *dicas de tipo*. Por exemplo:

```
def remove_gray_shades(
    colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

pode tornar-se mais legível desta forma:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

Vea `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 ferramentas de análise de tipos estático, e 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:
    field: 'annotation'
```

Anotações de variáveis são normalmente usadas para *dicas de tipo*: por exemplo, espera-se que esta variável receba valores do tipo `int`:

```
count: 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.

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.

Sobre esses documentos

Esses documentos são gerados a partir de [reStructuredText](#) pelo [Sphinx](#), um processador de documentos especificamente escrito para documentação Python.

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 escritor de boa parte do conteúdo;
- O projeto [Docutils](#) por criar [reStructuredText](#) e o pacote [Docutils](#);
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.

B.1 Contribuidores da Documentação 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!

História e Licença

C.1 História do software

O Python foi criado no início dos anos 1990 por Guido van Rossum na Stichting Mathematisch Centrum (CWI, veja <https://www.cwi.nl/>) na Holanda como um sucessor de uma linguagem chamada ABC. Guido continua a ser o principal autor de Python, embora inclua muitas contribuições de outros.

Em 1995, Guido continuou seu trabalho em Python na Corporação para Iniciativas Nacionais de Pesquisa (CNRI, veja <https://www.cnri.reston.va.us/>) em Reston, Virgínia, onde lançou várias versões do software.

Em maio de 2000, Guido e a equipe principal de desenvolvimento do Python mudaram-se para o BeOpen.com para formar a equipe BeOpen PythonLabs. Em outubro do mesmo ano, a equipe da PythonLabs mudou para a Digital Creations (agora Zope Corporation; veja <https://www.zope.org/>). Em 2001, formou-se a Python Software Foundation (PSF, veja <https://www.python.org/psf/>), uma organização sem fins lucrativos criada especificamente para possuir propriedade intelectual relacionada a Python. A Zope Corporation é um membro patrocinador do PSF.

Todas as versões do Python são de código aberto (consulte <https://opensource.org/> para a definição de código aberto). Historicamente, a maioria, mas não todas, versões do Python também são compatíveis com GPL; a tabela abaixo resume os vários lançamentos.

Versão	Derivada de	Ano	Proprietário	Compatível com a GPL?
0.9.0 a 1.2	n/a	1991-1995	CWI	sim
1.3 a 1.5.2	1.2	1995-1999	CNRI	sim
1.6	1.5.2	2000	CNRI	não
2.0	1.6	2000	BeOpen.com	não
1.6.1	1.6	2001	CNRI	não
2.1	2.0+1.6.1	2001	PSF	não
2.0.1	2.0+1.6.1	2001	PSF	sim
2.1.1	2.1+2.0.1	2001	PSF	sim
2.1.2	2.1.1	2002	PSF	sim
2.1.3	2.1.2	2002	PSF	sim
2.2 e acima	2.1.1	2001-agora	PSF	sim

Nota: Compatível com 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.

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 o *Acordo de Licenciamento PSF*.

A partir do Python 3.8.6, exemplos, receitas e outros códigos na documentação são licenciados duplamente sob o Acordo de Licenciamento PSF 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 ACORDO DE LICENCIAMENTO DA PSF PARA PYTHON 3.9.18

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),
→and
the Individual or Organization ("Licensee") accessing and otherwise using.
→Python
3.9.18 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby
grants Licensee a nonexclusive, royalty-free, world-wide license to.
→reproduce,
analyze, test, perform and/or display publicly, prepare derivative works,
distribute, and otherwise use Python 3.9.18 alone or in any derivative
version, provided, however, that PSF's License Agreement and PSF's notice.
→of
copyright, i.e., "Copyright © 2001–2023 Python Software Foundation; All.
→Rights
Reserved" are retained in Python 3.9.18 alone or in any derivative version
prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or
incorporates Python 3.9.18 or any part thereof, and wants to make the
derivative work available to others as provided herein, then Licensee.
→hereby
agrees to include in any such work a brief summary of the changes made to.
→Python
3.9.18.
4. PSF is making Python 3.9.18 available to Licensee on an "AS IS" basis.
PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF
EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION.
→OR
WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT.
→THE
USE OF PYTHON 3.9.18 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.9.18 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.9.18, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 3.9.18, 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.

(continua na próxima página)

(continuação da página anterior)

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

(continua na próxima página)

(continuação da página anterior)

its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.4 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 LICENÇA BSD DE ZERO CLÁUSULA PARA CÓDIGO NA DOCUMENTAÇÃO DO PYTHON 3.9.18

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

(continua na próxima página)

(continuação da página anterior)

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

O módulo `_random` inclui código baseado em um download de <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html>. A seguir estão os comentários literais do código original:

A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using `init_genrand(seed)`
or `init_by_array(init_key, key_length)`.

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.

(continua na próxima página)

(continuação da página anterior)

```
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, <http://www.wide.ad.jp/>.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.
```

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

```
THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Serviços de soquete assíncrono

Os módulos `asynchat` e `asyncore` contêm o seguinte aviso:

```
Copyright 1996 by Sam Rushing
```

```
All Rights Reserved
```

```
Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.
```

```
SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
```

(continua na próxima página)

(continuação da página anterior)

```
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  
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
```

(continua na próxima página)

(continuação da página anterior)

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 módulo uu contém o seguinte aviso:

Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Lance Ellinghouse not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:

- Use binascii module to do the actual line-by-line conversion between ascii and binary. This results in a 1000-fold speedup. The C 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

(continua na próxima página)

(continuação da página anterior)

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 MERCHANTABILITY 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_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:

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

(continua na próxima página)

(continuação da página anterior)

```

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.
</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 <http://www.netlib.org/fp/>. O arquivo original, conforme recuperado em 16 de março de 2009, contém os seguintes avisos de direitos autorais e de licenciamento:

```
/* *****  
 *  
 * The author of this software is David M. Gay.  
 *  
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.  
 *  
 * Permission to use, copy, modify, and distribute this software for any  
 * purpose without fee is hereby granted, provided that this entire notice  
 * is included in all copies of any software which is or includes a copy  
 * or modification of this software and in all copies of the supporting  
 * documentation for such software.  
 *  
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED  
 * WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY  
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY  
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.  
 *  
 ***** */
```

C.3.12 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

LICENSE ISSUES

=====

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

OpenSSL License

```
/* =====  
 * Copyright (c) 1998-2008 The OpenSSL Project. All rights reserved.  
 *  
 * Redistribution and use in source and binary forms, with or without  
 * modification, are permitted provided that the following conditions  
 * are met:  
 *  
 * 1. Redistributions of source code must retain the above copyright  
 * notice, this list of conditions and the following disclaimer.  
 *  
 * 2. Redistributions in binary form must reproduce the above copyright  
 * notice, this list of conditions and the following disclaimer in
```

(continua na próxima página)

(continuação da página anterior)

```

*   the documentation and/or other materials provided with the
*   distribution.
*
* 3. All advertising materials mentioning features or use of this
*   software must display the following acknowledgment:
*   "This product includes software developed by the OpenSSL Project
*   for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
*
* 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
*   endorse or promote products derived from this software without
*   prior written permission. For written permission, please contact
*   openssl-core@openssl.org.
*
* 5. Products derived from this software may not be called "OpenSSL"
*   nor may "OpenSSL" appear in their names without prior written
*   permission of the OpenSSL Project.
*
* 6. Redistributions of any form whatsoever must retain the following
*   acknowledgment:
*   "This product includes software developed by the OpenSSL Project
*   for use in the OpenSSL Toolkit (http://www.openssl.org/)"
*
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com).  This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

```

Original SSLeay License

```

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*
* This library is free for commercial and non-commercial use as long as
* the following conditions are aheared to. The following conditions
* apply to all code found in this distribution, be it the RC4, RSA,
* lhash, DES, etc., code; not just the SSL code. The SSL documentation
* included with this distribution is covered by the same copyright terms

```

(continua na próxima página)

(continuação da página anterior)

```

* except that the holder is Tim Hudson (tjh@cryptsoft.com).
*
* Copyright remains Eric Young's, and as such any Copyright notices in
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
*   notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*   notice, this list of conditions and the following disclaimer in the
*   documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
*   must display the following acknowledgement:
*   "This product includes cryptographic software written by
*    Eric Young (eay@cryptsoft.com)"
*   The word 'cryptographic' can be left out if the rouines from the library
*   being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
*   the apps directory (application code) you must include an acknowledgement:
*   "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/

```

C.3.13 expat

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

```

(continua na próxima página)

(continuação da página anterior)

```
"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 `_ctypes` é construída usando uma cópia incluída das fontes libffi, a menos que a compilação esteja configurada `--with-system-libffi`:

```
Copyright (c) 1996-2008 Red Hat, Inc and others.
```

```
Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
`Software'), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:
```

```
The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.
```

```
THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 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 compilação:

```
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
```

(continua na próxima página)

(continuação da página anterior)

```
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

O módulo `_decimal` é construído usando uma cópia incluída da biblioteca `libmpdec`, a menos que a compilação esteja configurada `--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,
```

(continua na próxima página)

(continuação da página anterior)

```
this list of conditions and the following disclaimer.  
* Redistributions in binary form must reproduce the original copyright  
  notice, this list of conditions and the following disclaimer in the  
  documentation and/or other materials provided with the distribution.  
* Neither the name of the W3C nor the names of its contributors may be  
  used to endorse or promote products derived from this work without  
  specific prior written permission.  
  
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS  
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT  
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR  
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT  
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,  
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT  
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,  
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY  
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT  
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE  
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

APÊNDICE D

Direitos autorais

Python e essa documentação é:

Copyright © 2001-2023 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. Todos os direitos reservados.

Copyright © 1995-2000 Corporation for National Research Initiatives. Todos os direitos reservados.

Copyright © 1991-1995 Stichting Mathematisch Centrum. Todos os direitos reservados.

Veja: *História e Licença* para informações completas de licença e permissões.

Não alfabético

..., [243](#)
 2to3, [243](#)
 >>>, [243](#)
 __all__ (*package variable*), [44](#)
 __dict__ (*module attribute*), [133](#)
 __doc__ (*module attribute*), [132](#)
 __file__ (*module attribute*), [132](#), [133](#)
 __future__, [247](#)
 __import__
 função interna, [44](#)
 __loader__ (*module attribute*), [132](#)
 __main__
 módulo, [12](#), [154](#), [165](#)
 __name__ (*module attribute*), [132](#), [133](#)
 __package__ (*module attribute*), [132](#)
 __slots__, [254](#)
 _frozen (*tipo C*), [47](#)
 _inittab (*tipo C*), [47](#)
 _Py_c_diff (*função C*), [96](#)
 _Py_c_neg (*função C*), [96](#)
 _Py_c_pow (*função C*), [96](#)
 _Py_c_prod (*função C*), [96](#)
 _Py_c_quot (*função C*), [96](#)
 _Py_c_sum (*função C*), [96](#)
 _Py_InitializeMain (*função C*), [185](#)
 _Py_NoneStruct (*variável C*), [198](#)
 _PyBytes_Resize (*função C*), [99](#)
 _PyCFunctionFast (*tipo C*), [200](#)
 _PyCFunctionFastWithKeywords (*tipo C*), [200](#)
 _PyFrameEvalFunction (*tipo C*), [163](#)
 _PyInterpreterState_GetEvalFrameFunc (*função C*), [163](#)
 _PyInterpreterState_SetEvalFrameFunc (*função C*), [164](#)
 _PyObject_New (*função C*), [197](#)
 _PyObject_NewVar (*função C*), [197](#)
 _PyTuple_Resize (*função C*), [120](#)
 _thread

módulo, [160](#)

A

abort(), [43](#)
 abs
 função interna, [73](#)
 aguardável, [244](#)
 allocfunc (*tipo C*), [234](#)
 ambiente virtual, [256](#)
 anotação, [243](#)
 anotação de função, [247](#)
 anotação de variável, [256](#)
 API provisória, [253](#)
 argumento, [243](#)
 argumento nomeado, [250](#)
 argumento posicional, [253](#)
 argv (*in module sys*), [157](#)
 arquivo binário, [244](#)
 arquivo texto, [255](#)
 ascii
 função interna, [65](#)
 aspas triplas, [255](#)
 atributo, [244](#)

B

BDFL, [244](#)
 binaryfunc (*tipo C*), [235](#)
 bloqueio global do interpretador, [248](#)
 buffer interface
 (see buffer protocol), [78](#)
 buffer object
 (see buffer protocol), [78](#)
 buffer protocol, [78](#)
 builtins
 módulo, [12](#), [154](#), [165](#)
 bytearray
 objeto, [99](#)
 bytecode, [245](#)
 bytes
 função interna, [65](#)

objeto, 97

C

`calloc()`, 187

caminho de importação, 249

Capsule

objeto, 143

carregador, 250

C-contiguous, 82, 245

class, 245

classe base abstrata, 243

classe estilo novo, 252

classmethod

função interna, 201

cleanup functions, 44

`close()` (*in module os*), 166

`CO_FUTURE_DIVISION` (*variável C*), 21

code object, 130

codificador de texto, 255

coerção, 245

coleta de lixo, 248

compile

função interna, 45

complex number

objeto, 96

compreensão de conjunto, 255

compreensão de dicionário, 246

compreensão de lista, 250

contagem de referências, 254

contíguo, 245

contiguous, 82

copyright (*in module sys*), 157

corrotina, 246

CPython, 246

`create_module` (*função C*), 136

D

decorador, 246

`descrgetfunc` (*tipo C*), 234

descritor, 246

`descrsetfunc` (*tipo C*), 234

desligamento do interpretador, 249

despacho único, 255

destructor (*tipo C*), 234

dica de tipo, 256

dicionário, 246

dictionary

objeto, 123

divisão pelo piso, 247

divmod

função interna, 72

docstring, 246

E

EAFP, 247

entrada de caminho, 253

`EOFError` (*built-in exception*), 131

escopo aninhado, 252

espaço de nomes, 251

`exc_info()` (*in module sys*), 11

`exec_module` (*função C*), 136

`exec_prefix`, 4

executable (*in module sys*), 156

`exit()`, 44

expressão, 247

expressão geradora, 248

F

f-string, 247

fatia, 255

file

objeto, 131

float

função interna, 74

floating point

objeto, 95

Fortran contiguous, 82, 245

`free()`, 187

`freefunc` (*tipo C*), 234

freeze utility, 47

frozenset

objeto, 126

função, 247

função chave, 250

função de corrotina, 246

função de retorno, 245

função genérica, 248

função interna

`__import__`, 44

`abs`, 73

`ascii`, 65

`bytes`, 65

classmethod, 201

compile, 45

`divmod`, 72

`float`, 74

`hash`, 66, 215

`int`, 74

`len`, 66, 75, 77, 122, 125, 127

`pow`, 72, 74

`repr`, 65, 214

staticmethod, 201

`tuple`, 76, 123

`type`, 66

function

objeto, 128

G

gancho de entrada de caminho, **253**
 generator, **248**
 generator expression, **248**
 gerador, **248**
 gerador assíncrono, **244**
 gerenciador de contexto, **245**
 gerenciador de contexto assíncrono, **244**
 getattrfunc (*tipo C*), **234**
 getattrofunc (*tipo C*), **234**
 getbufferproc (*tipo C*), **234**
 getiterfunc (*tipo C*), **234**
 GIL, **248**
 global interpreter lock, **158**

H

hash
 função interna, **66**, **215**
 hasheável, **249**
 hashfunc (*tipo C*), **234**

I

IDLE, **249**
 importação, **249**
 importador, **249**
 imutável, **249**
 incr_item(), **11**, **12**
 initproc (*tipo C*), **234**
 inquiry (*tipo C*), **239**
 instancemethod
 objeto, **129**
 instrução, **255**
 int
 função interna, **74**
 integer
 objeto, **91**
 interativo, **249**
 interpretado, **249**
 interpreter lock, **158**
 iterable, **249**
 iterador, **250**
 iterador assíncrono, **244**
 iterador gerador, **248**
 iterador gerador assíncrono, **244**
 iterável assíncrono, **244**
 iternextfunc (*tipo C*), **234**

K

KeyboardInterrupt (*built-in exception*), **31**

L

lambda, **250**
 LBYL, **250**

len
 função interna, **66**, **75**, **77**, **122**, **125**, **127**
 lenfunc (*tipo C*), **234**
 list
 objeto, **122**
 lista, **250**
 localizador, **247**
 localizador baseado no caminho, **253**
 localizador de entrada de caminho, **253**
 localizador de metacaminho, **251**
 lock, interpreter, **158**
 long integer
 objeto, **91**
 LONG_MAX, **92**

M

magic
 method, **250**
 main(), **155**, **157**
 malloc(), **187**
 mapeamento, **251**
 mapping
 objeto, **123**
 máquina virtual, **256**
 memoryview
 objeto, **141**
 metaclasses, **251**
 METH_CLASS (*variável interna*), **201**
 METH_COEXIST (*variável interna*), **201**
 METH_FASTCALL (*variável interna*), **201**
 METH_NOARGS (*variável interna*), **201**
 METH_O (*variável interna*), **201**
 METH_STATIC (*variável interna*), **201**
 METH_VARARGS (*variável interna*), **200**
 method
 magic, **250**
 objeto, **129**
 special, **255**
 MethodType (*in module types*), **128**, **129**
 método, **251**
 método especial, **255**
 método mágico, **250**
 module
 objeto, **132**
 search path, **12**, **154**, **156**
 modules (*in module sys*), **44**, **154**
 ModuleType (*in module types*), **132**
 módulo, **251**
 __main__, **12**, **154**, **165**
 _thread, **160**
 builtins, **12**, **154**, **165**
 signal, **31**
 sys, **12**, **154**, **165**
 módulo de extensão, **247**

módulo spec, [251](#)

MRO, [251](#)

mutável, [251](#)

N

newfunc (*tipo C*), [234](#)

nome qualificado, [254](#)

None

objeto, [91](#)

novas linhas universais, [256](#)

numeric

objeto, [91](#)

número complexo, [245](#)

O

object, [252](#)

code, [130](#)

objeto

bytearray, [99](#)

bytes, [97](#)

Capsule, [143](#)

complex number, [96](#)

dictionary, [123](#)

file, [131](#)

floating point, [95](#)

frozenset, [126](#)

function, [128](#)

instancemethod, [129](#)

integer, [91](#)

list, [122](#)

long integer, [91](#)

mapping, [123](#)

memoryview, [141](#)

method, [129](#)

module, [132](#)

None, [91](#)

numeric, [91](#)

sequence, [97](#)

set, [126](#)

tuple, [119](#)

type, [6](#), [87](#)

objeto arquivo, [247](#)

objeto arquivo ou similar, [247](#)

objeto byte ou similar, [245](#)

objeto caminho ou similar, [253](#)

objobjargproc (*tipo C*), [235](#)

objobjproc (*tipo C*), [235](#)

ordem de resolução de métodos, [251](#)

OverflowError (*built-in exception*), [92](#), [93](#)

P

package variable

__all__, [44](#)

pacote, [252](#)

pacote de espaço de nomes, [252](#)

pacote provisório, [253](#)

pacote regular, [254](#)

parâmetro, [252](#)

PATH, [12](#)

path

module search, [12](#), [154](#), [156](#)

path (*in module sys*), [12](#), [154](#), [156](#)

PEP, [253](#)

platform (*in module sys*), [157](#)

porção, [253](#)

pow

função interna, [72](#), [74](#)

prefix, [4](#)

Propostas Estendidas Python

PEP 1, [253](#)

PEP 7, [4](#), [6](#)

PEP 238, [21](#), [247](#)

PEP 278, [256](#)

PEP 302, [247](#), [250](#)

PEP 343, [245](#)

PEP 353, [10](#)

PEP 362, [244](#), [253](#)

PEP 383, [108](#), [109](#)

PEP 384, [15](#)

PEP 393, [100](#), [107](#)

PEP 411, [253](#)

PEP 420, [247](#), [252](#), [253](#)

PEP 432, [185](#), [186](#)

PEP 442, [228](#)

PEP 443, [248](#)

PEP 451, [136](#), [247](#)

PEP 483, [248](#)

PEP 484, [243](#), [247](#), [248](#), [255](#), [256](#)

PEP 489, [136](#)

PEP 492, [244](#), [246](#)

PEP 498, [247](#)

PEP 519, [253](#)

PEP 523, [164](#)

PEP 525, [244](#)

PEP 526, [243](#), [256](#)

PEP 528, [153](#)

PEP 529, [109](#), [153](#)

PEP 538, [182](#)

PEP 539, [169](#)

PEP 540, [182](#)

PEP 552, [177](#)

PEP 578, [43](#)

PEP 585, [248](#)

PEP 587, [172](#)

PEP 590, [67](#)

PEP 617, [180](#)

PEP 623, [100](#)

PEP 3116, [256](#)

- PEP 3119, 65, 66
- PEP 3121, 134
- PEP 3147, 46
- PEP 3151, 36
- PEP 3155, 254
- Py_ABS (macro C), 5
- Py_AddPendingCall (função C), 166
- Py_AddPendingCall(), 166
- Py_AtExit (função C), 44
- Py_BEGIN_ALLOW_THREADS, 159
- Py_BEGIN_ALLOW_THREADS (macro C), 162
- Py_BLOCK_THREADS (macro C), 162
- Py_buffer (tipo C), 79
- Py_buffer.buf (membro C), 79
- Py_buffer.format (membro C), 80
- Py_buffer.internal (membro C), 81
- Py_buffer.itemsize (membro C), 80
- Py_buffer.len (membro C), 79
- Py_buffer.ndim (membro C), 80
- Py_buffer.obj (membro C), 79
- Py_buffer.readonly (membro C), 79
- Py_buffer.shape (membro C), 80
- Py_buffer.strides (membro C), 80
- Py_buffer.suboffsets (membro C), 80
- Py_BuildValue (função C), 55
- Py_BytesMain (função C), 17
- Py_BytesWarningFlag (variável C), 152
- Py_CHARMASK (macro C), 5
- Py_CLEAR (função C), 23
- Py_CompileString (função C), 20
- Py_CompileString(), 21
- Py_CompileStringExFlags (função C), 20
- Py_CompileStringFlags (função C), 20
- Py_CompileStringObject (função C), 20
- Py_complex (tipo C), 96
- Py_DebugFlag (variável C), 152
- Py_DecodeLocale (função C), 40
- Py_DECREF (função C), 23
- Py_DECREF(), 7
- Py_DEPRECATED (macro C), 5
- Py_DontWriteBytecodeFlag (variável C), 152
- Py_Ellipsis (variável C), 141
- Py_EncodeLocale (função C), 41
- Py_END_ALLOW_THREADS, 159
- Py_END_ALLOW_THREADS (macro C), 162
- Py_EndInterpreter (função C), 166
- Py_EnterRecursiveCall (função C), 34
- Py_eval_input (variável C), 21
- Py_Exit (função C), 43
- Py_ExitStatusException (função C), 173
- Py_False (variável C), 94
- Py_FatalError (função C), 43
- Py_FatalError(), 157
- Py_FdIsInteractive (função C), 39
- Py_file_input (variável C), 21
- Py_Finalize (função C), 155
- Py_FinalizeEx (função C), 154
- Py_FinalizeEx(), 44, 154, 165, 166
- Py_FrozenFlag (variável C), 152
- Py_GenericAlias (função C), 150
- Py_GenericAliasType (variável C), 150
- Py_GetArgcArgv (função C), 185
- Py_GetBuildInfo (função C), 157
- Py_GetCompiler (função C), 157
- Py_GetCopyright (função C), 157
- Py_GETENV (macro C), 5
- Py_GetExecPrefix (função C), 156
- Py_GetExecPrefix(), 12
- Py_GetPath (função C), 156
- Py_GetPath(), 12, 155, 156
- Py_GetPlatform (função C), 157
- Py_GetPrefix (função C), 155
- Py_GetPrefix(), 12
- Py_GetProgramFullPath (função C), 156
- Py_GetProgramFullPath(), 12
- Py_GetProgramName (função C), 155
- Py_GetPythonHome (função C), 158
- Py_GetVersion (função C), 156
- Py_HashRandomizationFlag (variável C), 153
- Py_IgnoreEnvironmentFlag (variável C), 153
- Py_INCREF (função C), 23
- Py_INCREF(), 7
- Py_Initialize (função C), 154
- Py_Initialize(), 12, 155, 165
- Py_InitializeEx (função C), 154
- Py_InitializeFromConfig (função C), 181
- Py_InspectFlag (variável C), 153
- Py_InteractiveFlag (variável C), 153
- Py_IS_TYPE (função C), 199
- Py_IsInitialized (função C), 154
- Py_IsInitialized(), 12
- Py_IsolatedFlag (variável C), 153
- Py_LeaveRecursiveCall (função C), 34
- Py_LegacyWindowsFSEncodingFlag (variável C), 153
- Py_LegacyWindowsStdioFlag (variável C), 153
- Py_Main (função C), 17
- Py_MAX (macro C), 5
- Py_MEMBER_SIZE (macro C), 5
- Py_MIN (macro C), 5
- Py_mod_create (macro C), 136
- Py_mod_exec (macro C), 136
- Py_NewInterpreter (função C), 165
- Py_None (variável C), 91
- Py_NoSiteFlag (variável C), 153
- Py_NotImplemented (variável C), 63
- Py_NoUserSiteDirectory (variável C), 153
- Py_OptimizeFlag (variável C), 154

- Py_PreInitialize (função C), 175
- Py_PreInitializeFromArgs (função C), 175
- Py_PreInitializeFromBytesArgs (função C), 175
- Py_PRINT_RAW, 132
- Py_QuietFlag (variável C), 154
- Py_REFCNT (macro C), 199
- Py_ReprEnter (função C), 34
- Py_ReprLeave (função C), 34
- Py_RETURN_FALSE (macro C), 94
- Py_RETURN_NONE (macro C), 91
- Py_RETURN_NOTIMPLEMENTED (macro C), 63
- Py_RETURN_RICHCOMPARE (macro C), 221
- Py_RETURN_TRUE (macro C), 94
- Py_RunMain (função C), 185
- Py_SET_REFCNT (função C), 199
- Py_SET_SIZE (função C), 199
- Py_SET_TYPE (função C), 199
- Py_SetPath (função C), 156
- Py_SetPath(), 156
- Py_SetProgramName (função C), 155
- Py_SetProgramName(), 12, 154156
- Py_SetPythonHome (função C), 158
- Py_SetStandardStreamEncoding (função C), 155
- Py_single_input (variável C), 21
- Py_SIZE (macro C), 199
- Py_ssize_t (tipo C), 10
- PY_SSIZE_T_MAX, 93
- Py_STRINGIFY (macro C), 5
- Py_TPFLAGS_BASE_EXC_SUBCLASS (variável interna), 218
- Py_TPFLAGS_BASETYPE (variável interna), 217
- Py_TPFLAGS_BYTES_SUBCLASS (variável interna), 218
- Py_TPFLAGS_DEFAULT (variável interna), 217
- Py_TPFLAGS_DICT_SUBCLASS (variável interna), 218
- Py_TPFLAGS_HAVE_FINALIZE (variável interna), 218
- Py_TPFLAGS_HAVE_GC (variável interna), 217
- Py_TPFLAGS_HAVE_VECTORCALL (variável interna), 218
- Py_TPFLAGS_HEAPTYPE (variável interna), 217
- Py_TPFLAGS_LIST_SUBCLASS (variável interna), 218
- Py_TPFLAGS_LONG_SUBCLASS (variável interna), 218
- Py_TPFLAGS_METHOD_DESCRIPTOR (variável interna), 218
- Py_TPFLAGS_READY (variável interna), 217
- Py_TPFLAGS_READYING (variável interna), 217
- Py_TPFLAGS_TUPLE_SUBCLASS (variável interna), 218
- Py_TPFLAGS_TYPE_SUBCLASS (variável interna), 218
- Py_TPFLAGS_UNICODE_SUBCLASS (variável interna), 218
- Py_tracefunc (tipo C), 167
- Py_True (variável C), 94
- Py_tss_NEEDS_INIT (macro C), 169
- Py_tss_t (tipo C), 169
- Py_TYPE (macro C), 198
- Py_UCS1 (tipo C), 101
- Py_UCS2 (tipo C), 101
- Py_UCS4 (tipo C), 101
- Py_UNBLOCK_THREADS (macro C), 162
- Py_UnbufferedStdioFlag (variável C), 154
- Py_UNICODE (tipo C), 101
- Py_UNICODE_IS_HIGH_SURROGATE (macro C), 104
- Py_UNICODE_IS_LOW_SURROGATE (macro C), 104
- Py_UNICODE_IS_SURROGATE (macro C), 104
- Py_UNICODE_ISALNUM (função C), 103
- Py_UNICODE_ISALPHA (função C), 103
- Py_UNICODE_ISDECIMAL (função C), 103
- Py_UNICODE_ISDIGIT (função C), 103
- Py_UNICODE_ISLINEBREAK (função C), 103
- Py_UNICODE_ISLOWER (função C), 103
- Py_UNICODE_ISNUMERIC (função C), 103
- Py_UNICODE_ISPRINTABLE (função C), 103
- Py_UNICODE_ISSPACE (função C), 103
- Py_UNICODE_ISTITLE (função C), 103
- Py_UNICODE_ISUPPER (função C), 103
- Py_UNICODE_JOIN_SURROGATES (macro C), 104
- Py_UNICODE_TODecimal (função C), 104
- Py_UNICODE_TODIGIT (função C), 104
- Py_UNICODE_TOLOWER (função C), 104
- Py_UNICODE_TONUMERIC (função C), 104
- Py_UNICODE_TOTITLE (função C), 104
- Py_UNICODE_TOUPPER (função C), 104
- Py_UNREACHABLE (macro C), 5
- Py_UNUSED (macro C), 5
- Py_VaBuildValue (função C), 56
- Py_VECTORCALL_ARGUMENTS_OFFSET (macro C), 68
- Py_VerboseFlag (variável C), 154
- Py_VISIT (função C), 239
- Py_XDECREF (função C), 23
- Py_XDECREF(), 12
- Py_XINCREf (função C), 23
- PyAnySet_Check (função C), 126
- PyAnySet_CheckExact (função C), 126
- PyArg_Parse (função C), 54
- PyArg_ParseTuple (função C), 54
- PyArg_ParseTupleAndKeywords (função C), 54
- PyArg_UnpackTuple (função C), 54
- PyArg_ValidateKeywordArguments (função C), 54

- PyArg_VaParse (*função C*), 54
- PyArg_VaParseTupleAndKeywords (*função C*), 54
- PyASCIIObject (*tipo C*), 101
- PyAsyncMethods (*tipo C*), 233
- PyAsyncMethods.am_aiter (*membro C*), 233
- PyAsyncMethods.am_anext (*membro C*), 233
- PyAsyncMethods.am_await (*membro C*), 233
- PyBool_Check (*função C*), 94
- PyBool_FromLong (*função C*), 94
- PyBUF_ANY_CONTIGUOUS (*macro C*), 82
- PyBUF_C_CONTIGUOUS (*macro C*), 82
- PyBUF_CONTIG (*macro C*), 83
- PyBUF_CONTIG_RO (*macro C*), 83
- PyBUF_F_CONTIGUOUS (*macro C*), 82
- PyBUF_FORMAT (*macro C*), 81
- PyBUF_FULL (*macro C*), 83
- PyBUF_FULL_RO (*macro C*), 83
- PyBUF_INDIRECT (*macro C*), 82
- PyBUF_ND (*macro C*), 82
- PyBUF_RECORDS (*macro C*), 83
- PyBUF_RECORDS_RO (*macro C*), 83
- PyBUF_SIMPLE (*macro C*), 82
- PyBUF_STRIDED (*macro C*), 83
- PyBUF_STRIDED_RO (*macro C*), 83
- PyBUF_STRIDES (*macro C*), 82
- PyBUF_WRITABLE (*macro C*), 81
- PyBuffer_FillContiguousStrides (*função C*), 85
- PyBuffer_FillInfo (*função C*), 85
- PyBuffer_FromContiguous (*função C*), 85
- PyBuffer_GetPointer (*função C*), 85
- PyBuffer_IsContiguous (*função C*), 85
- PyBuffer_Release (*função C*), 85
- PyBuffer_SizeFromFormat (*função C*), 85
- PyBuffer_ToContiguous (*função C*), 85
- PyBufferProcs, 78
- PyBufferProcs (*tipo C*), 232
- PyBufferProcs.bf_getbuffer (*membro C*), 232
- PyBufferProcs.bf_releasebuffer (*membro C*), 232
- PyByteArray_AS_STRING (*função C*), 100
- PyByteArray_AsString (*função C*), 100
- PyByteArray_Check (*função C*), 99
- PyByteArray_CheckExact (*função C*), 99
- PyByteArray_Concat (*função C*), 100
- PyByteArray_FromObject (*função C*), 100
- PyByteArray_FromStringAndSize (*função C*), 100
- PyByteArray_GET_SIZE (*função C*), 100
- PyByteArray_Resize (*função C*), 100
- PyByteArray_Size (*função C*), 100
- PyByteArray_Type (*variável C*), 99
- PyByteArrayObject (*tipo C*), 99
- PyBytes_AS_STRING (*função C*), 98
- PyBytes_AsString (*função C*), 98
- PyBytes_AsStringAndSize (*função C*), 98
- PyBytes_Check (*função C*), 97
- PyBytes_CheckExact (*função C*), 97
- PyBytes_Concat (*função C*), 99
- PyBytes_ConcatAndDel (*função C*), 99
- PyBytes_FromFormat (*função C*), 98
- PyBytes_FromFormatV (*função C*), 98
- PyBytes_FromObject (*função C*), 98
- PyBytes_FromString (*função C*), 97
- PyBytes_FromStringAndSize (*função C*), 97
- PyBytes_GET_SIZE (*função C*), 98
- PyBytes_Size (*função C*), 98
- PyBytes_Type (*variável C*), 97
- PyBytesObject (*tipo C*), 97
- pyc baseado em hash, 249
- PyCallable_Check (*função C*), 72
- PyCallIter_Check (*função C*), 139
- PyCallIter_New (*função C*), 139
- PyCallIter_Type (*variável C*), 139
- PyCapsule (*tipo C*), 143
- PyCapsule_CheckExact (*função C*), 143
- PyCapsule_Destructor (*tipo C*), 143
- PyCapsule_GetContext (*função C*), 143
- PyCapsule_GetDestructor (*função C*), 143
- PyCapsule_GetName (*função C*), 143
- PyCapsule_GetPointer (*função C*), 143
- PyCapsule_Import (*função C*), 143
- PyCapsule_IsValid (*função C*), 144
- PyCapsule_New (*função C*), 143
- PyCapsule_SetContext (*função C*), 144
- PyCapsule_SetDestructor (*função C*), 144
- PyCapsule_SetName (*função C*), 144
- PyCapsule_SetPointer (*função C*), 144
- PyCell_Check (*função C*), 130
- PyCell_GET (*função C*), 130
- PyCell_Get (*função C*), 130
- PyCell_New (*função C*), 130
- PyCell_SET (*função C*), 130
- PyCell_Set (*função C*), 130
- PyCell_Type (*variável C*), 130
- PyCellobject (*tipo C*), 130
- PyCFunction (*tipo C*), 199
- PyCFunctionWithKeywords (*tipo C*), 199
- PyCMethod (*tipo C*), 200
- PyCode_Check (*função C*), 130
- PyCode_GetNumFree (*função C*), 130
- PyCode_New (*função C*), 130
- PyCode_NewEmpty (*função C*), 131
- PyCode_NewWithPosOnlyArgs (*função C*), 131
- PyCode_Type (*variável C*), 130
- PyCodec_BackslashReplaceErrors (*função C*), 60

- PyCodec_Decode (*função C*), 59
- PyCodec_Decoder (*função C*), 60
- PyCodec_Encode (*função C*), 59
- PyCodec_Encoder (*função C*), 60
- PyCodec_IgnoreErrors (*função C*), 60
- PyCodec_IncrementalDecoder (*função C*), 60
- PyCodec_IncrementalEncoder (*função C*), 60
- PyCodec_KnownEncoding (*função C*), 59
- PyCodec_LookupError (*função C*), 60
- PyCodec_NameReplaceErrors (*função C*), 60
- PyCodec_Register (*função C*), 59
- PyCodec_RegisterError (*função C*), 60
- PyCodec_ReplaceErrors (*função C*), 60
- PyCodec_StreamReader (*função C*), 60
- PyCodec_StreamWriter (*função C*), 60
- PyCodec_StrictErrors (*função C*), 60
- PyCodec_XMLCharRefReplaceErrors (*função C*), 60
- PyCodeObject (*tipo C*), 130
- PyCompactUnicodeObject (*tipo C*), 101
- PyCompilerFlags (*tipo C*), 21
- PyCompilerFlags.cf_feature_version (*membro C*), 21
- PyCompilerFlags.cf_flags (*membro C*), 21
- PyComplex_AsCComplex (*função C*), 97
- PyComplex_Check (*função C*), 96
- PyComplex_CheckExact (*função C*), 96
- PyComplex_FromCComplex (*função C*), 97
- PyComplex_FromDoubles (*função C*), 97
- PyComplex_ImagAsDouble (*função C*), 97
- PyComplex_RealAsDouble (*função C*), 97
- PyComplex_Type (*variável C*), 96
- PyComplexObject (*tipo C*), 96
- PyConfig (*tipo C*), 176
- PyConfig_Clear (*função C*), 177
- PyConfig_InitIsolatedConfig (*função C*), 176
- PyConfig_InitPythonConfig (*função C*), 176
- PyConfig_Read (*função C*), 176
- PyConfig_SetArgv (*função C*), 176
- PyConfig_SetBytesArgv (*função C*), 176
- PyConfig_SetBytesString (*função C*), 176
- PyConfig_SetString (*função C*), 176
- PyConfig_SetWideStringList (*função C*), 176
- PyConfig._use_peg_parser (*membro C*), 180
- PyConfig.argv (*membro C*), 177
- PyConfig.base_exec_prefix (*membro C*), 177
- PyConfig.base_executable (*membro C*), 177
- PyConfig.base_prefix (*membro C*), 177
- PyConfig.buffered_stdio (*membro C*), 177
- PyConfig.bytes_warning (*membro C*), 177
- PyConfig.check_hash_pycs_mode (*membro C*), 177
- PyConfig.configure_c_stdio (*membro C*), 177
- PyConfig.dev_mode (*membro C*), 178
- PyConfig.dump_refs (*membro C*), 178
- PyConfig.exec_prefix (*membro C*), 178
- PyConfig.executable (*membro C*), 178
- PyConfig.faulthandler (*membro C*), 178
- PyConfig.filesystem_encoding (*membro C*), 178
- PyConfig.filesystem_errors (*membro C*), 178
- PyConfig.hash_seed (*membro C*), 178
- PyConfig.home (*membro C*), 178
- PyConfig.import_time (*membro C*), 178
- PyConfig.inspect (*membro C*), 178
- PyConfig.install_signal_handlers (*membro C*), 178
- PyConfig.interactive (*membro C*), 178
- PyConfig.isolated (*membro C*), 178
- PyConfig.legacy_windows_stdio (*membro C*), 178
- PyConfig.malloc_stats (*membro C*), 179
- PyConfig.module_search_paths (*membro C*), 179
- PyConfig.module_search_paths_set (*membro C*), 179
- PyConfig.optimization_level (*membro C*), 179
- PyConfig.parse_argv (*membro C*), 179
- PyConfig.parser_debug (*membro C*), 179
- PyConfig.pathconfig_warnings (*membro C*), 179
- PyConfig.platlibdir (*membro C*), 177
- PyConfig.prefix (*membro C*), 179
- PyConfig.program_name (*membro C*), 179
- PyConfig.pycache_prefix (*membro C*), 179
- PyConfig.pythonpath_env (*membro C*), 179
- PyConfig.quiet (*membro C*), 179
- PyConfig.run_command (*membro C*), 179
- PyConfig.run_filename (*membro C*), 179
- PyConfig.run_module (*membro C*), 179
- PyConfig.show_ref_count (*membro C*), 179
- PyConfig.site_import (*membro C*), 180
- PyConfig.skip_source_first_line (*membro C*), 180
- PyConfig.stdio_encoding (*membro C*), 180
- PyConfig.stdio_errors (*membro C*), 180
- PyConfig.tracemalloc (*membro C*), 180
- PyConfig.use_environment (*membro C*), 180
- PyConfig.use_hash_seed (*membro C*), 178
- PyConfig.user_site_directory (*membro C*), 180
- PyConfig.verbose (*membro C*), 180
- PyConfig.warnoptions (*membro C*), 180
- PyConfig.write_bytecode (*membro C*), 180
- PyConfig.xoptions (*membro C*), 180
- PyContext (*tipo C*), 145
- PyContext_CheckExact (*função C*), 146
- PyContext_Copy (*função C*), 146

- PyContext_CopyCurrent (*função C*), 146
- PyContext_Enter (*função C*), 146
- PyContext_Exit (*função C*), 146
- PyContext_New (*função C*), 146
- PyContext_Type (*variável C*), 145
- PyContextToken (*tipo C*), 145
- PyContextToken_CheckExact (*função C*), 146
- PyContextToken_Type (*variável C*), 146
- PyContextVar (*tipo C*), 145
- PyContextVar_CheckExact (*função C*), 146
- PyContextVar_Get (*função C*), 146
- PyContextVar_New (*função C*), 146
- PyContextVar_Reset (*função C*), 147
- PyContextVar_Set (*função C*), 146
- PyContextVar_Type (*variável C*), 146
- PyCoro_CheckExact (*função C*), 145
- PyCoro_New (*função C*), 145
- PyCoro_Type (*variável C*), 145
- PyCoroObject (*tipo C*), 145
- PyDate_Check (*função C*), 147
- PyDate_CheckExact (*função C*), 147
- PyDate_FromDate (*função C*), 148
- PyDate_FromTimestamp (*função C*), 149
- PyDateTime_Check (*função C*), 147
- PyDateTime_CheckExact (*função C*), 147
- PyDateTime_DATE_GET_FOLD (*função C*), 149
- PyDateTime_DATE_GET_HOUR (*função C*), 148
- PyDateTime_DATE_GET_MICROSECOND (*função C*), 149
- PyDateTime_DATE_GET_MINUTE (*função C*), 149
- PyDateTime_DATE_GET_SECOND (*função C*), 149
- PyDateTime_DELTA_GET_DAYS (*função C*), 149
- PyDateTime_DELTA_GET_MICROSECONDS (*função C*), 149
- PyDateTime_DELTA_GET_SECONDS (*função C*), 149
- PyDateTime_FromDateAndTime (*função C*), 148
- PyDateTime_FromDateAndTimeAndFold (*função C*), 148
- PyDateTime_FromTimestamp (*função C*), 149
- PyDateTime_GET_DAY (*função C*), 148
- PyDateTime_GET_MONTH (*função C*), 148
- PyDateTime_GET_YEAR (*função C*), 148
- PyDateTime_TIME_GET_FOLD (*função C*), 149
- PyDateTime_TIME_GET_HOUR (*função C*), 149
- PyDateTime_TIME_GET_MICROSECOND (*função C*), 149
- PyDateTime_TIME_GET_MINUTE (*função C*), 149
- PyDateTime_TIME_GET_SECOND (*função C*), 149
- PyDateTime_TimeZone_UTC (*variável C*), 147
- PyDelta_Check (*função C*), 147
- PyDelta_CheckExact (*função C*), 147
- PyDelta_FromDSU (*função C*), 148
- PyDescr_IsData (*função C*), 139
- PyDescr_NewClassMethod (*função C*), 139
- PyDescr_NewGetSet (*função C*), 139
- PyDescr_NewMember (*função C*), 139
- PyDescr_NewMethod (*função C*), 139
- PyDescr_NewWrapper (*função C*), 139
- PyDict_Check (*função C*), 123
- PyDict_CheckExact (*função C*), 123
- PyDict_Clear (*função C*), 124
- PyDict_Contains (*função C*), 124
- PyDict_Copy (*função C*), 124
- PyDict_DelItem (*função C*), 124
- PyDict_DelItemString (*função C*), 124
- PyDict_GetItem (*função C*), 124
- PyDict_GetItemString (*função C*), 124
- PyDict_GetItemWithError (*função C*), 124
- PyDict_Items (*função C*), 125
- PyDict_Keys (*função C*), 125
- PyDict_Merge (*função C*), 125
- PyDict_MergeFromSeq2 (*função C*), 126
- PyDict_New (*função C*), 124
- PyDict_Next (*função C*), 125
- PyDict_SetDefault (*função C*), 124
- PyDict_SetItem (*função C*), 124
- PyDict_SetItemString (*função C*), 124
- PyDict_Size (*função C*), 125
- PyDict_Type (*variável C*), 123
- PyDict_Update (*função C*), 126
- PyDict_Values (*função C*), 125
- PyDictObject (*tipo C*), 123
- PyDictProxy_New (*função C*), 124
- PyDoc_STR (*macro C*), 6
- PyDoc_STRVAR (*macro C*), 6
- PyErr_BadArgument (*função C*), 26
- PyErr_BadInternalCall (*função C*), 28
- PyErr_CheckSignals (*função C*), 31
- PyErr_Clear (*função C*), 26
- PyErr_Clear(), 10, 12
- PyErr_ExceptionMatches (*função C*), 30
- PyErr_ExceptionMatches(), 12
- PyErr_Fetch (*função C*), 30
- PyErr_Format (*função C*), 26
- PyErr_FormatV (*função C*), 26
- PyErr_GetExcInfo (*função C*), 31
- PyErr_GivenExceptionMatches (*função C*), 30
- PyErr_NewException (*função C*), 32
- PyErr_NewExceptionWithDoc (*função C*), 32
- PyErr_NoMemory (*função C*), 27
- PyErr_NormalizeException (*função C*), 30
- PyErr_Occurred (*função C*), 30
- PyErr_Occurred(), 10
- PyErr_Print (*função C*), 26
- PyErr_PrintEx (*função C*), 26
- PyErr_ResourceWarning (*função C*), 29
- PyErr_Restore (*função C*), 30

- [PyErr_SetExcFromWindowsErr \(função C\), 27](#)
- [PyErr_SetExcFromWindowsErrWithFilename \(função C\), 28](#)
- [PyErr_SetExcFromWindowsErrWithFilenameObject \(função C\), 27](#)
- [PyErr_SetExcFromWindowsErrWithFilenameObjectEx \(função C\), 28](#)
- [PyErr_SetExcInfo \(função C\), 31](#)
- [PyErr_SetFromErrno \(função C\), 27](#)
- [PyErr_SetFromErrnoWithFilename \(função C\), 27](#)
- [PyErr_SetFromErrnoWithFilenameObject \(função C\), 27](#)
- [PyErr_SetFromErrnoWithFilenameObjects \(função C\), 27](#)
- [PyErr_SetFromWindowsErr \(função C\), 27](#)
- [PyErr_SetFromWindowsErrWithFilename \(função C\), 27](#)
- [PyErr_SetImportError \(função C\), 28](#)
- [PyErr_SetImportErrorSubclass \(função C\), 28](#)
- [PyErr_SetInterrupt \(função C\), 31](#)
- [PyErr_SetNone \(função C\), 26](#)
- [PyErr_SetObject \(função C\), 26](#)
- [PyErr_SetString \(função C\), 26](#)
- [PyErr_SetString\(\), 10](#)
- [PyErr_SyntaxLocation \(função C\), 28](#)
- [PyErr_SyntaxLocationEx \(função C\), 28](#)
- [PyErr_SyntaxLocationObject \(função C\), 28](#)
- [PyErr_WarnEx \(função C\), 29](#)
- [PyErr_WarnExplicit \(função C\), 29](#)
- [PyErr_WarnExplicitObject \(função C\), 29](#)
- [PyErr_WarnFormat \(função C\), 29](#)
- [PyErr_WriteUnraisable \(função C\), 26](#)
- [PyEval_AcquireLock \(função C\), 164](#)
- [PyEval_AcquireThread \(função C\), 164](#)
- [PyEval_AcquireThread\(\), 160](#)
- [PyEval_EvalCode \(função C\), 20](#)
- [PyEval_EvalCodeEx \(função C\), 20](#)
- [PyEval_EvalFrame \(função C\), 21](#)
- [PyEval_EvalFrameEx \(função C\), 21](#)
- [PyEval_GetBuiltins \(função C\), 58](#)
- [PyEval_GetFrame \(função C\), 58](#)
- [PyEval_GetFuncDesc \(função C\), 59](#)
- [PyEval_GetFuncName \(função C\), 59](#)
- [PyEval_GetGlobals \(função C\), 58](#)
- [PyEval_GetLocals \(função C\), 58](#)
- [PyEval_InitThreads \(função C\), 160](#)
- [PyEval_InitThreads\(\), 154](#)
- [PyEval_MergeCompilerFlags \(função C\), 21](#)
- [PyEval_ReleaseLock \(função C\), 165](#)
- [PyEval_ReleaseThread \(função C\), 164](#)
- [PyEval_ReleaseThread\(\), 160](#)
- [PyEval_RestoreThread \(função C\), 161](#)
- [PyEval_RestoreThread\(\), 159, 160](#)
- [PyEval_SaveThread \(função C\), 160](#)
- [PyEval_SaveThread\(\), 159, 160](#)
- [PyEval_SetProfile \(função C\), 168](#)
- [PyEval_SetTrace \(função C\), 168](#)
- [PyEval_ThreadsInitialized \(função C\), 160](#)
- [PyExc_ArithmeticError, 35](#)
- [PyExc_AssertionError, 35](#)
- [PyExc_AttributeError, 35](#)
- [PyExc_BaseException, 35](#)
- [PyExc_BlockingIOError, 35](#)
- [PyExc_BrokenPipeError, 35](#)
- [PyExc_BufferError, 35](#)
- [PyExc_BytesWarning, 36](#)
- [PyExc_ChildProcessError, 35](#)
- [PyExc_ConnectionAbortedError, 35](#)
- [PyExc_ConnectionError, 35](#)
- [PyExc_ConnectionRefusedError, 35](#)
- [PyExc_ConnectionResetError, 35](#)
- [PyExc_DeprecationWarning, 36](#)
- [PyExc_EnvironmentError, 36](#)
- [PyExc_EOFError, 35](#)
- [PyExc_Exception, 35](#)
- [PyExc_FileExistsError, 35](#)
- [PyExc_FileNotFoundError, 35](#)
- [PyExc_FloatingPointError, 35](#)
- [PyExc_FutureWarning, 36](#)
- [PyExc_GeneratorExit, 35](#)
- [PyExc_ImportError, 35](#)
- [PyExc_ImportWarning, 36](#)
- [PyExc_IndentationError, 35](#)
- [PyExc_IndexError, 35](#)
- [PyExc_InterruptedError, 35](#)
- [PyExc_IOError, 36](#)
- [PyExc_IsADirectoryError, 35](#)
- [PyExc_KeyboardInterrupt, 35](#)
- [PyExc_KeyError, 35](#)
- [PyExc_LookupError, 35](#)
- [PyExc_MemoryError, 35](#)
- [PyExc_ModuleNotFoundError, 35](#)
- [PyExc_NameError, 35](#)
- [PyExc_NotADirectoryError, 35](#)
- [PyExc_NotImplementedError, 35](#)
- [PyExc_OSError, 35](#)
- [PyExc_OverflowError, 35](#)
- [PyExc_PendingDeprecationWarning, 36](#)
- [PyExc_PermissionError, 35](#)
- [PyExc_ProcessLookupError, 35](#)
- [PyExc_RecursionError, 35](#)
- [PyExc_ReferenceError, 35](#)
- [PyExc_ResourceWarning, 36](#)
- [PyExc_RuntimeError, 35](#)
- [PyExc_RuntimeWarning, 36](#)
- [PyExc_StopAsyncIteration, 35](#)
- [PyExc_StopIteration, 35](#)

- PyExc_SyntaxError, 35
- PyExc_SyntaxWarning, 36
- PyExc_SystemError, 35
- PyExc_SystemExit, 35
- PyExc_TabError, 35
- PyExc_TimeoutError, 35
- PyExc_TypeError, 35
- PyExc_UnboundLocalError, 35
- PyExc_UnicodeDecodeError, 35
- PyExc_UnicodeEncodeError, 35
- PyExc_UnicodeError, 35
- PyExc_UnicodeTranslateError, 35
- PyExc_UnicodeWarning, 36
- PyExc_UserWarning, 36
- PyExc_ValueError, 35
- PyExc_Warning, 36
- PyExc_WindowsError, 36
- PyExc_ZeroDivisionError, 35
- PyException_GetCause (função C), 32
- PyException_GetContext (função C), 32
- PyException_GetTraceback (função C), 32
- PyException_SetCause (função C), 32
- PyException_SetContext (função C), 32
- PyException_SetTraceback (função C), 32
- PyFile_FromFd (função C), 131
- PyFile_GetLine (função C), 131
- PyFile_SetOpenCodeHook (função C), 132
- PyFile_WriteObject (função C), 132
- PyFile_WriteString (função C), 132
- PyFloat_AS_DOUBLE (função C), 95
- PyFloat_AsDouble (função C), 95
- PyFloat_Check (função C), 95
- PyFloat_CheckExact (função C), 95
- PyFloat_FromDouble (função C), 95
- PyFloat_FromString (função C), 95
- PyFloat_GetInfo (função C), 95
- PyFloat_GetMax (função C), 95
- PyFloat_GetMin (função C), 95
- PyFloat_Type (variável C), 95
- PyFloatObject (tipo C), 95
- PyFrame_GetBack (função C), 58
- PyFrame_GetCode (função C), 59
- PyFrame_GetLineNumber (função C), 59
- PyFrameObject (tipo C), 21
- PyFrozenSet_Check (função C), 126
- PyFrozenSet_CheckExact (função C), 127
- PyFrozenSet_New (função C), 127
- PyFrozenSet_Type (variável C), 126
- PyFunction_Check (função C), 128
- PyFunction_GetAnnotations (função C), 128
- PyFunction_GetClosure (função C), 128
- PyFunction_GetCode (função C), 128
- PyFunction_GetDefaults (função C), 128
- PyFunction_GetGlobals (função C), 128
- PyFunction_GetModule (função C), 128
- PyFunction_New (função C), 128
- PyFunction_NewWithQualName (função C), 128
- PyFunction_SetAnnotations (função C), 129
- PyFunction_SetClosure (função C), 128
- PyFunction_SetDefaults (função C), 128
- PyFunction_Type (variável C), 128
- PyFunctionObject (tipo C), 128
- PyGen_Check (função C), 144
- PyGen_CheckExact (função C), 144
- PyGen_New (função C), 144
- PyGen_NewWithQualName (função C), 145
- PyGen_Type (variável C), 144
- PyGenObject (tipo C), 144
- PyGetSetDef (tipo C), 203
- PyGILState_Check (função C), 161
- PyGILState_Ensure (função C), 161
- PyGILState_GetThisThreadState (função C), 161
- PyGILState_Release (função C), 161
- PyImport_AddModule (função C), 45
- PyImport_AddModuleObject (função C), 45
- PyImport_AppendInittab (função C), 47
- PyImport_ExecCodeModule (função C), 45
- PyImport_ExecCodeModuleEx (função C), 45
- PyImport_ExecCodeModuleObject (função C), 46
- PyImport_ExecCodeModuleWithPathnames (função C), 46
- PyImport_ExtendInittab (função C), 47
- PyImport_FrozenModules (variável C), 47
- PyImport_GetImporter (função C), 46
- PyImport_GetMagicNumber (função C), 46
- PyImport_GetMagicTag (função C), 46
- PyImport_GetModule (função C), 46
- PyImport_GetModuleDict (função C), 46
- PyImport_Import (função C), 45
- PyImport_ImportFrozenModule (função C), 47
- PyImport_ImportFrozenModuleObject (função C), 46
- PyImport_ImportModule (função C), 44
- PyImport_ImportModuleEx (função C), 44
- PyImport_ImportModuleLevel (função C), 44
- PyImport_ImportModuleLevelObject (função C), 44
- PyImport_ImportModuleNoBlock (função C), 44
- PyImport_ReloadModule (função C), 45
- PyIndex_Check (função C), 74
- PyInstanceMethod_Check (função C), 129
- PyInstanceMethod_Function (função C), 129
- PyInstanceMethod_GET_FUNCTION (função C), 129
- PyInstanceMethod_New (função C), 129
- PyInstanceMethod_Type (variável C), 129

- PyInterpreterState (*tipo C*), 160
- PyInterpreterState_Clear (*função C*), 162
- PyInterpreterState_Delete (*função C*), 162
- PyInterpreterState_Get (*função C*), 163
- PyInterpreterState_GetDict (*função C*), 163
- PyInterpreterState_GetID (*função C*), 163
- PyInterpreterState_Head (*função C*), 168
- PyInterpreterState_Main (*função C*), 168
- PyInterpreterState_New (*função C*), 162
- PyInterpreterState_Next (*função C*), 168
- PyInterpreterState_ThreadHead (*função C*), 168
- PyIter_Check (*função C*), 78
- PyIter_Next (*função C*), 78
- PyList_Append (*função C*), 123
- PyList_AsTuple (*função C*), 123
- PyList_Check (*função C*), 122
- PyList_CheckExact (*função C*), 122
- PyList_GET_ITEM (*função C*), 122
- PyList_GET_SIZE (*função C*), 122
- PyList_GetItem (*função C*), 122
- PyList_GetItem(), 9
- PyList_GetSlice (*função C*), 123
- PyList_Insert (*função C*), 123
- PyList_New (*função C*), 122
- PyList_Reverse (*função C*), 123
- PyList_SET_ITEM (*função C*), 123
- PyList_SetItem (*função C*), 122
- PyList_SetItem(), 8
- PyList_SetSlice (*função C*), 123
- PyList_Size (*função C*), 122
- PyList_Sort (*função C*), 123
- PyList_Type (*variável C*), 122
- PyListObject (*tipo C*), 122
- PyLong_AsDouble (*função C*), 94
- PyLong_AsLong (*função C*), 92
- PyLong_AsLongAndOverflow (*função C*), 92
- PyLong_AsLongLong (*função C*), 92
- PyLong_AsLongLongAndOverflow (*função C*), 93
- PyLong_AsSize_t (*função C*), 93
- PyLong_AsSsize_t (*função C*), 93
- PyLong_AsUnsignedLong (*função C*), 93
- PyLong_AsUnsignedLongLong (*função C*), 93
- PyLong_AsUnsignedLongLongMask (*função C*), 94
- PyLong_AsUnsignedLongMask (*função C*), 93
- PyLong_AsVoidPtr (*função C*), 94
- PyLong_Check (*função C*), 91
- PyLong_CheckExact (*função C*), 91
- PyLong_FromDouble (*função C*), 92
- PyLong_FromLong (*função C*), 91
- PyLong_FromLongLong (*função C*), 91
- PyLong_FromSize_t (*função C*), 91
- PyLong_FromSsize_t (*função C*), 91
- PyLong_FromString (*função C*), 92
- PyLong_FromUnicode (*função C*), 92
- PyLong_FromUnicodeObject (*função C*), 92
- PyLong_FromUnsignedLong (*função C*), 91
- PyLong_FromUnsignedLongLong (*função C*), 92
- PyLong_FromVoidPtr (*função C*), 92
- PyLong_Type (*variável C*), 91
- PyLongObject (*tipo C*), 91
- PyMapping_Check (*função C*), 76
- PyMapping_DelItem (*função C*), 77
- PyMapping_DelItemString (*função C*), 77
- PyMapping_GetItemString (*função C*), 77
- PyMapping_HasKey (*função C*), 77
- PyMapping_HasKeyString (*função C*), 77
- PyMapping_Items (*função C*), 77
- PyMapping_Keys (*função C*), 77
- PyMapping_Length (*função C*), 76
- PyMapping_SetItemString (*função C*), 77
- PyMapping_Size (*função C*), 76
- PyMapping_Values (*função C*), 77
- PyMappingMethods (*tipo C*), 231
- PyMappingMethods.mp_ass_subscript (*membro C*), 231
- PyMappingMethods.mp_length (*membro C*), 231
- PyMappingMethods.mp_subscript (*membro C*), 231
- PyMarshal_ReadLastObjectFromFile (*função C*), 48
- PyMarshal_ReadLongFromFile (*função C*), 48
- PyMarshal_ReadObjectFromFile (*função C*), 48
- PyMarshal_ReadObjectFromString (*função C*), 48
- PyMarshal_ReadShortFromFile (*função C*), 48
- PyMarshal_WriteLongToFile (*função C*), 48
- PyMarshal_WriteObjectToFile (*função C*), 48
- PyMarshal_WriteObjectToString (*função C*), 48
- PyMem_Calloc (*função C*), 189
- PyMem_Del (*função C*), 190
- PYMEM_DOMAIN_MEM (*macro C*), 192
- PYMEM_DOMAIN_OBJ (*macro C*), 192
- PYMEM_DOMAIN_RAW (*macro C*), 192
- PyMem_Free (*função C*), 189
- PyMem_GetAllocator (*função C*), 192
- PyMem_Malloc (*função C*), 189
- PyMem_New (*função C*), 189
- PyMem_RawCalloc (*função C*), 188
- PyMem_RawFree (*função C*), 189
- PyMem_RawMalloc (*função C*), 188
- PyMem_RawRealloc (*função C*), 188
- PyMem_Realloc (*função C*), 189
- PyMem_Resize (*função C*), 190
- PyMem_SetAllocator (*função C*), 192
- PyMem_SetupDebugHooks (*função C*), 192

- PyMemAllocatorDomain (*tipo C*), 192
- PyMemAllocatorEx (*tipo C*), 191
- PyMember_GetOne (*função C*), 203
- PyMember_SetOne (*função C*), 203
- PyMemberDef (*tipo C*), 202
- PyMemoryView_Check (*função C*), 141
- PyMemoryView_FromBuffer (*função C*), 141
- PyMemoryView_FromMemory (*função C*), 141
- PyMemoryView_FromObject (*função C*), 141
- PyMemoryView_GET_BASE (*função C*), 141
- PyMemoryView_GET_BUFFER (*função C*), 141
- PyMemoryView_GetContiguous (*função C*), 141
- PyMethod_Check (*função C*), 129
- PyMethod_Function (*função C*), 129
- PyMethod_GET_FUNCTION (*função C*), 129
- PyMethod_GET_SELF (*função C*), 130
- PyMethod_New (*função C*), 129
- PyMethod_Self (*função C*), 129
- PyMethod_Type (*variável C*), 129
- PyMethodDef (*tipo C*), 200
- PyModule_AddFunctions (*função C*), 137
- PyModule_AddIntConstant (*função C*), 137
- PyModule_AddIntMacro (*função C*), 138
- PyModule_AddObject (*função C*), 137
- PyModule_AddStringConstant (*função C*), 137
- PyModule_AddStringMacro (*função C*), 138
- PyModule_AddType (*função C*), 138
- PyModule_Check (*função C*), 132
- PyModule_CheckExact (*função C*), 132
- PyModule_Create (*função C*), 135
- PyModule_Create2 (*função C*), 135
- PyModule_ExecDef (*função C*), 137
- PyModule_FromDefAndSpec (*função C*), 136
- PyModule_FromDefAndSpec2 (*função C*), 136
- PyModule_GetDef (*função C*), 133
- PyModule_GetDict (*função C*), 133
- PyModule_GetFilename (*função C*), 133
- PyModule_GetFilenameObject (*função C*), 133
- PyModule_GetName (*função C*), 133
- PyModule_GetNameObject (*função C*), 133
- PyModule_GetState (*função C*), 133
- PyModule_New (*função C*), 132
- PyModule_NewObject (*função C*), 132
- PyModule_SetDocString (*função C*), 137
- PyModule_Type (*variável C*), 132
- PyModuleDef (*tipo C*), 133
- PyModuleDef_Init (*função C*), 135
- PyModuleDef_Slot (*tipo C*), 135
- PyModuleDef_Slot.slot (*membro C*), 135
- PyModuleDef_Slot.value (*membro C*), 136
- PyModuleDef.m_base (*membro C*), 133
- PyModuleDef.m_clear (*membro C*), 134
- PyModuleDef.m_doc (*membro C*), 133
- PyModuleDef.m_free (*membro C*), 134
- PyModuleDef.m_methods (*membro C*), 134
- PyModuleDef.m_name (*membro C*), 133
- PyModuleDef.m_reload (*membro C*), 134
- PyModuleDef.m_size (*membro C*), 134
- PyModuleDef.m_slots (*membro C*), 134
- PyModuleDef.m_traverse (*membro C*), 134
- PyNumber_Absolute (*função C*), 73
- PyNumber_Add (*função C*), 72
- PyNumber_And (*função C*), 73
- PyNumber_AsSsize_t (*função C*), 74
- PyNumber_Check (*função C*), 72
- PyNumber_Divmod (*função C*), 72
- PyNumber_Float (*função C*), 74
- PyNumber_FloorDivide (*função C*), 72
- PyNumber_Index (*função C*), 74
- PyNumber_InPlaceAdd (*função C*), 73
- PyNumber_InPlaceAnd (*função C*), 74
- PyNumber_InPlaceFloorDivide (*função C*), 73
- PyNumber_InPlaceLshift (*função C*), 74
- PyNumber_InPlaceMatrixMultiply (*função C*), 73
- PyNumber_InPlaceMultiply (*função C*), 73
- PyNumber_InPlaceOr (*função C*), 74
- PyNumber_InPlacePower (*função C*), 74
- PyNumber_InPlaceRemainder (*função C*), 73
- PyNumber_InPlaceRshift (*função C*), 74
- PyNumber_InPlaceSubtract (*função C*), 73
- PyNumber_InPlaceTrueDivide (*função C*), 73
- PyNumber_InPlaceXor (*função C*), 74
- PyNumber_Invert (*função C*), 73
- PyNumber_Long (*função C*), 74
- PyNumber_Lshift (*função C*), 73
- PyNumber_MatrixMultiply (*função C*), 72
- PyNumber_Multiply (*função C*), 72
- PyNumber_Negative (*função C*), 72
- PyNumber_Or (*função C*), 73
- PyNumber_Positive (*função C*), 72
- PyNumber_Power (*função C*), 72
- PyNumber_Remainder (*função C*), 72
- PyNumber_Rshift (*função C*), 73
- PyNumber_Subtract (*função C*), 72
- PyNumber_ToBase (*função C*), 74
- PyNumber_TrueDivide (*função C*), 72
- PyNumber_Xor (*função C*), 73
- PyNumberMethods (*tipo C*), 228
- PyNumberMethods.nb_absolute (*membro C*), 230
- PyNumberMethods.nb_add (*membro C*), 229
- PyNumberMethods.nb_and (*membro C*), 230
- PyNumberMethods.nb_bool (*membro C*), 230
- PyNumberMethods.nb_divmod (*membro C*), 230
- PyNumberMethods.nb_float (*membro C*), 230
- PyNumberMethods.nb_floor_divide (*membro C*), 230
- PyNumberMethods.nb_index (*membro C*), 230

`PyNumberMethods.nb_inplace_add` (*membro C*), 230

`PyNumberMethods.nb_inplace_and` (*membro C*), 230

`PyNumberMethods.nb_inplace_floor_divide` (*membro C*), 230

`PyNumberMethods.nb_inplace_lshift` (*membro C*), 230

`PyNumberMethods.nb_inplace_matrix_multiply` (*membro C*), 230

`PyNumberMethods.nb_inplace_multiply` (*membro C*), 230

`PyNumberMethods.nb_inplace_or` (*membro C*), 230

`PyNumberMethods.nb_inplace_power` (*membro C*), 230

`PyNumberMethods.nb_inplace_remainder` (*membro C*), 230

`PyNumberMethods.nb_inplace_rshift` (*membro C*), 230

`PyNumberMethods.nb_inplace_subtract` (*membro C*), 230

`PyNumberMethods.nb_inplace_true_divide` (*membro C*), 230

`PyNumberMethods.nb_inplace_xor` (*membro C*), 230

`PyNumberMethods.nb_int` (*membro C*), 230

`PyNumberMethods.nb_invert` (*membro C*), 230

`PyNumberMethods.nb_lshift` (*membro C*), 230

`PyNumberMethods.nb_matrix_multiply` (*membro C*), 230

`PyNumberMethods.nb_multiply` (*membro C*), 229

`PyNumberMethods.nb_negative` (*membro C*), 230

`PyNumberMethods.nb_or` (*membro C*), 230

`PyNumberMethods.nb_positive` (*membro C*), 230

`PyNumberMethods.nb_power` (*membro C*), 230

`PyNumberMethods.nb_remainder` (*membro C*), 229

`PyNumberMethods.nb_reserved` (*membro C*), 230

`PyNumberMethods.nb_rshift` (*membro C*), 230

`PyNumberMethods.nb_subtract` (*membro C*), 229

`PyNumberMethods.nb_true_divide` (*membro C*), 230

`PyNumberMethods.nb_xor` (*membro C*), 230

`PyObject` (*tipo C*), 198

`PyObject_AsCharBuffer` (*função C*), 86

`PyObject_ASCII` (*função C*), 65

`PyObject_AsFileDescriptor` (*função C*), 131

`PyObject_AsReadBuffer` (*função C*), 86

`PyObject_AsWriteBuffer` (*função C*), 86

`PyObject_Bytes` (*função C*), 65

`PyObject_Call` (*função C*), 69

`PyObject_CallFunction` (*função C*), 70

`PyObject_CallFunctionObjArgs` (*função C*), 70

`PyObject_CallMethod` (*função C*), 70

`PyObject_CallMethodNoArgs` (*função C*), 70

`PyObject_CallMethodObjArgs` (*função C*), 70

`PyObject_CallMethodOneArg` (*função C*), 71

`PyObject_CallNoArgs` (*função C*), 69

`PyObject_CallObject` (*função C*), 70

`PyObject_Calloc` (*função C*), 190

`PyObject_CallOneArg` (*função C*), 70

`PyObject_CheckBuffer` (*função C*), 84

`PyObject_CheckReadBuffer` (*função C*), 86

`PyObject_Del` (*função C*), 197

`PyObject_DelAttr` (*função C*), 64

`PyObject_DelAttrString` (*função C*), 64

`PyObject_DelItem` (*função C*), 66

`PyObject_Dir` (*função C*), 66

`PyObject_Free` (*função C*), 191

`PyObject_GC_Del` (*função C*), 238

`PyObject_GC_IsFinalized` (*função C*), 238

`PyObject_GC_IsTracked` (*função C*), 238

`PyObject_GC_New` (*função C*), 238

`PyObject_GC_NewVar` (*função C*), 238

`PyObject_GC_Resize` (*função C*), 238

`PyObject_GC_Track` (*função C*), 238

`PyObject_GC_UnTrack` (*função C*), 238

`PyObject_GenericGetAttr` (*função C*), 64

`PyObject_GenericGetDict` (*função C*), 64

`PyObject_GenericSetAttr` (*função C*), 64

`PyObject_GenericSetDict` (*função C*), 64

`PyObject_GetArenaAllocator` (*função C*), 194

`PyObject_GetAttr` (*função C*), 64

`PyObject_GetAttrString` (*função C*), 64

`PyObject_GetBuffer` (*função C*), 84

`PyObject_GetItem` (*função C*), 66

`PyObject_GetIter` (*função C*), 67

`PyObject_HasAttr` (*função C*), 63

`PyObject_HasAttrString` (*função C*), 63

`PyObject_Hash` (*função C*), 66

`PyObject_HashNotImplemented` (*função C*), 66

`PyObject_HEAD` (*macro C*), 198

`PyObject_HEAD_INIT` (*macro C*), 199

`PyObject_Init` (*função C*), 197

`PyObject_InitVar` (*função C*), 197

`PyObject_IS_GC` (*função C*), 238

`PyObject_IsInstance` (*função C*), 65

`PyObject_IsSubclass` (*função C*), 65

`PyObject_IsTrue` (*função C*), 66

`PyObject_Length` (*função C*), 66

`PyObject_LengthHint` (*função C*), 66

`PyObject_Malloc` (*função C*), 190

`PyObject_New` (*função C*), 197

`PyObject_NewVar` (*função C*), 197

`PyObject_Not` (*função C*), 66

`PyObject._ob_next` (*membro C*), 210

`PyObject._ob_prev` (*membro C*), 210

- PyObject_Print (*função C*), 63
- PyObject_Realloc (*função C*), 190
- PyObject_Repr (*função C*), 65
- PyObject_RichCompare (*função C*), 64
- PyObject_RichCompareBool (*função C*), 65
- PyObject_SetArenaAllocator (*função C*), 194
- PyObject_SetAttr (*função C*), 64
- PyObject_SetAttrString (*função C*), 64
- PyObject_SetItem (*função C*), 66
- PyObject_Size (*função C*), 66
- PyObject_Str (*função C*), 65
- PyObject_Type (*função C*), 66
- PyObject_TypeCheck (*função C*), 66
- PyObject_VAR_HEAD (*macro C*), 198
- PyObject_Vectorcall (*função C*), 71
- PyObject_VectorcallDict (*função C*), 71
- PyObject_VectorcallMethod (*função C*), 71
- PyObjectArenaAllocator (*tipo C*), 194
- PyObject.ob_refcnt (*membro C*), 210
- PyObject.ob_type (*membro C*), 210
- PyOS_AfterFork (*função C*), 40
- PyOS_AfterFork_Child (*função C*), 40
- PyOS_AfterFork_Parent (*função C*), 39
- PyOS_BeforeFork (*função C*), 39
- PyOS_CheckStack (*função C*), 40
- PyOS_double_to_string (*função C*), 58
- PyOS_FSPath (*função C*), 39
- PyOS_getsig (*função C*), 40
- PyOS_InputHook (*variável C*), 18
- PyOS_ReadlineFunctionPointer (*variável C*), 19
- PyOS_setsig (*função C*), 40
- PyOS_snprintf (*função C*), 57
- PyOS_stricmp (*função C*), 58
- PyOS_string_to_double (*função C*), 57
- PyOS_strnicmp (*função C*), 58
- PyOS_vsnprintf (*função C*), 57
- PyParser_SimpleParseFile (*função C*), 19
- PyParser_SimpleParseFileFlags (*função C*), 19
- PyParser_SimpleParseString (*função C*), 19
- PyParser_SimpleParseStringFlags (*função C*), 19
- PyParser_SimpleParseStringFlagsFilename (*função C*), 19
- PyPreConfig (*tipo C*), 174
- PyPreConfig_InitIsolatedConfig (*função C*), 174
- PyPreConfig_InitPythonConfig (*função C*), 174
- PyPreConfig.allocator (*membro C*), 174
- PyPreConfig.coerce_c_locale (*membro C*), 175
- PyPreConfig.coerce_c_locale_warn (*membro C*), 175
- PyPreConfig.configure_locale (*membro C*), 174
- PyPreConfig.dev_mode (*membro C*), 175
- PyPreConfig.isolated (*membro C*), 175
- PyPreConfig.legacy_windows_fs_encoding (*membro C*), 175
- PyPreConfig.parse_argv (*membro C*), 175
- PyPreConfig.use_environment (*membro C*), 175
- PyPreConfig.utf8_mode (*membro C*), 175
- PyProperty_Type (*variável C*), 139
- PyRun_AnyFile (*função C*), 17
- PyRun_AnyFileEx (*função C*), 17
- PyRun_AnyFileExFlags (*função C*), 18
- PyRun_AnyFileFlags (*função C*), 17
- PyRun_File (*função C*), 19
- PyRun_FileEx (*função C*), 20
- PyRun_FileExFlags (*função C*), 20
- PyRun_FileFlags (*função C*), 20
- PyRun_InteractiveLoop (*função C*), 18
- PyRun_InteractiveLoopFlags (*função C*), 18
- PyRun_InteractiveOne (*função C*), 18
- PyRun_InteractiveOneFlags (*função C*), 18
- PyRun_SimpleFile (*função C*), 18
- PyRun_SimpleFileEx (*função C*), 18
- PyRun_SimpleFileExFlags (*função C*), 18
- PyRun_SimpleString (*função C*), 18
- PyRun_SimpleStringFlags (*função C*), 18
- PyRun_String (*função C*), 19
- PyRun_StringFlags (*função C*), 19
- PySeqIter_Check (*função C*), 139
- PySeqIter_New (*função C*), 139
- PySeqIter_Type (*variável C*), 139
- PySequence_Check (*função C*), 75
- PySequence_Concat (*função C*), 75
- PySequence_Contains (*função C*), 75
- PySequence_Count (*função C*), 75
- PySequence_DelItem (*função C*), 75
- PySequence_DelSlice (*função C*), 75
- PySequence_Fast (*função C*), 76
- PySequence_Fast_GET_ITEM (*função C*), 76
- PySequence_Fast_GET_SIZE (*função C*), 76
- PySequence_Fast_ITEMS (*função C*), 76
- PySequence_GetItem (*função C*), 75
- PySequence_GetItem(), 9
- PySequence_GetSlice (*função C*), 75
- PySequence_Index (*função C*), 76
- PySequence_InPlaceConcat (*função C*), 75
- PySequence_InPlaceRepeat (*função C*), 75
- PySequence_ITEM (*função C*), 76
- PySequence_Length (*função C*), 75
- PySequence_List (*função C*), 76
- PySequence_Repeat (*função C*), 75
- PySequence_SetItem (*função C*), 75
- PySequence_SetSlice (*função C*), 75

- PySequence_Size (*função C*), 75
- PySequence_Tuple (*função C*), 76
- PySequenceMethods (*tipo C*), 231
- PySequenceMethods.sq_ass_item (*membro C*), 231
- PySequenceMethods.sq_concat (*membro C*), 231
- PySequenceMethods.sq_contains (*membro C*), 231
- PySequenceMethods.sq_inplace_concat (*membro C*), 231
- PySequenceMethods.sq_inplace_repeat (*membro C*), 232
- PySequenceMethods.sq_item (*membro C*), 231
- PySequenceMethods.sq_length (*membro C*), 231
- PySequenceMethods.sq_repeat (*membro C*), 231
- PySet_Add (*função C*), 127
- PySet_Check (*função C*), 126
- PySet_Clear (*função C*), 127
- PySet_Contains (*função C*), 127
- PySet_Discard (*função C*), 127
- PySet_GET_SIZE (*função C*), 127
- PySet_New (*função C*), 127
- PySet_Pop (*função C*), 127
- PySet_Size (*função C*), 127
- PySet_Type (*variável C*), 126
- PySetObject (*tipo C*), 126
- PySignal_SetWakeupFd (*função C*), 31
- PySlice_AdjustIndices (*função C*), 141
- PySlice_Check (*função C*), 140
- PySlice_GetIndices (*função C*), 140
- PySlice_GetIndicesEx (*função C*), 140
- PySlice_New (*função C*), 140
- PySlice_Type (*variável C*), 140
- PySlice_Unpack (*função C*), 140
- PyState_AddModule (*função C*), 138
- PyState_FindModule (*função C*), 138
- PyState_RemoveModule (*função C*), 138
- PyStatus (*tipo C*), 173
- PyStatus_Error (*função C*), 173
- PyStatus_Exception (*função C*), 173
- PyStatus_Exit (*função C*), 173
- PyStatus_IsError (*função C*), 173
- PyStatus_IsExit (*função C*), 173
- PyStatus_NoMemory (*função C*), 173
- PyStatus_Ok (*função C*), 173
- PyStatus.err_msg (*membro C*), 173
- PyStatus.exitcode (*membro C*), 173
- PyStatus.func (*membro C*), 173
- PyStructSequence_Desc (*tipo C*), 121
- PyStructSequence_Field (*tipo C*), 121
- PyStructSequence_GET_ITEM (*função C*), 121
- PyStructSequence_GetItem (*função C*), 121
- PyStructSequence_InitType (*função C*), 121
- PyStructSequence_InitType2 (*função C*), 121
- PyStructSequence_New (*função C*), 121
- PyStructSequence_NewType (*função C*), 121
- PyStructSequence_SET_ITEM (*função C*), 122
- PyStructSequence_SetItem (*função C*), 121
- PyStructSequence_UnnamedField (*variável C*), 121
- PySys_AddAuditHook (*função C*), 43
- PySys_AddWarnOption (*função C*), 42
- PySys_AddWarnOptionUnicode (*função C*), 42
- PySys_AddXOption (*função C*), 42
- PySys_Audit (*função C*), 43
- PySys_FormatStderr (*função C*), 42
- PySys_FormatStdout (*função C*), 42
- PySys_GetObject (*função C*), 42
- PySys_GetXOptions (*função C*), 42
- PySys_ResetWarnOptions (*função C*), 42
- PySys_SetArgv (*função C*), 158
- PySys_SetArgv(), 154
- PySys_SetArgvEx (*função C*), 157
- PySys_SetArgvEx(), 12, 154
- PySys_SetObject (*função C*), 42
- PySys_SetPath (*função C*), 42
- PySys_WriteStderr (*função C*), 42
- PySys_WriteStdout (*função C*), 42
- Python 3000, 253
- PYTHON*, 153
- PYTHONCOERCECLOCALE, 182
- PYTHONDEBUG, 152
- PYTHONDONTWRITEBYTECODE, 152
- PYTHONDUMPREFS, 210
- PYTHONHASHSEED, 153
- PYTHONHOME, 12, 153, 158, 178
- Pythônico, 254
- PYTHONINSPECT, 153
- PYTHONIOENCODING, 155
- PYTHONLEGACYWINDOWSFSENCODING, 153
- PYTHONLEGACYWINDOWSSTDIO, 153
- PYTHONMALLOC, 188, 191, 193
- PYTHONMALLOCSTATS, 188
- PYTHONNOUSERSITE, 154
- PYTHONOLDPARSER, 180
- PYTHONOPTIMIZE, 154
- PYTHONPATH, 12, 153, 179
- PYTHONUNBUFFERED, 154
- PYTHONUTF8, 182
- PYTHONVERBOSE, 154
- PyThread_create_key (*função C*), 170
- PyThread_delete_key (*função C*), 170
- PyThread_delete_key_value (*função C*), 170
- PyThread_get_key_value (*função C*), 170
- PyThread_ReInitTLS (*função C*), 170
- PyThread_set_key_value (*função C*), 170
- PyThread_tss_alloc (*função C*), 169
- PyThread_tss_create (*função C*), 170

- PyThread_tss_delete (*função C*), 170
- PyThread_tss_free (*função C*), 169
- PyThread_tss_get (*função C*), 170
- PyThread_tss_is_created (*função C*), 170
- PyThread_tss_set (*função C*), 170
- PyThreadState, 158
- PyThreadState (*tipo C*), 160
- PyThreadState_Clear (*função C*), 162
- PyThreadState_Delete (*função C*), 162
- PyThreadState_DeleteCurrent (*função C*), 163
- PyThreadState_Get (*função C*), 161
- PyThreadState_GetDict (*função C*), 164
- PyThreadState_GetFrame (*função C*), 163
- PyThreadState_GetID (*função C*), 163
- PyThreadState_GetInterpreter (*função C*), 163
- PyThreadState_New (*função C*), 162
- PyThreadState_Next (*função C*), 168
- PyThreadState_SetAsyncExc (*função C*), 164
- PyThreadState_Swap (*função C*), 161
- PyTime_Check (*função C*), 147
- PyTime_CheckExact (*função C*), 147
- PyTime_FromTime (*função C*), 148
- PyTime_FromTimeAndFold (*função C*), 148
- PyTimeZone_FromOffset (*função C*), 148
- PyTimeZone_FromOffsetAndName (*função C*), 148
- PyTrace_C_CALL (*variável C*), 168
- PyTrace_C_EXCEPTION (*variável C*), 168
- PyTrace_C_RETURN (*variável C*), 168
- PyTrace_CALL (*variável C*), 167
- PyTrace_EXCEPTION (*variável C*), 167
- PyTrace_LINE (*variável C*), 167
- PyTrace_OPCODE (*variável C*), 168
- PyTrace_RETURN (*variável C*), 168
- PyTraceMalloc_Track (*função C*), 194
- PyTraceMalloc_Untrack (*função C*), 194
- PyTuple_Check (*função C*), 119
- PyTuple_CheckExact (*função C*), 119
- PyTuple_GET_ITEM (*função C*), 120
- PyTuple_GET_SIZE (*função C*), 120
- PyTuple_GetItem (*função C*), 120
- PyTuple_GetSlice (*função C*), 120
- PyTuple_New (*função C*), 120
- PyTuple_Pack (*função C*), 120
- PyTuple_SET_ITEM (*função C*), 120
- PyTuple_SetItem (*função C*), 120
- PyTuple_SetItem(), 8
- PyTuple_Size (*função C*), 120
- PyTuple_Type (*variável C*), 119
- PyTupleObject (*tipo C*), 119
- PyType_Check (*função C*), 87
- PyType_CheckExact (*função C*), 87
- PyType_ClearCache (*função C*), 87
- PyType_FromModuleAndSpec (*função C*), 89
- PyType_FromSpec (*função C*), 89
- PyType_FromSpecWithBases (*função C*), 89
- PyType_GenericAlloc (*função C*), 88
- PyType_GenericNew (*função C*), 88
- PyType_GetFlags (*função C*), 88
- PyType_GetModule (*função C*), 88
- PyType_GetModuleState (*função C*), 89
- PyType_GetSlot (*função C*), 88
- PyType_HasFeature (*função C*), 88
- PyType_IS_GC (*função C*), 88
- PyType_IsSubtype (*função C*), 88
- PyType_Modified (*função C*), 88
- PyType_Ready (*função C*), 88
- PyType_Slot (*tipo C*), 90
- PyType_Slot.PyType_Slot.pfunc (*membro C*), 90
- PyType_Slot.PyType_Slot.slot (*membro C*), 90
- PyType_Spec (*tipo C*), 89
- PyType_Spec.PyType_Spec.basicsize (*membro C*), 89
- PyType_Spec.PyType_Spec.flags (*membro C*), 89
- PyType_Spec.PyType_Spec.itemsize (*membro C*), 89
- PyType_Spec.PyType_Spec.name (*membro C*), 89
- PyType_Spec.PyType_Spec.slots (*membro C*), 90
- PyType_Type (*variável C*), 87
- PyTypeObject (*tipo C*), 87
- PyTypeObject.tp_alloc (*membro C*), 225
- PyTypeObject.tp_as_async (*membro C*), 214
- PyTypeObject.tp_as_buffer (*membro C*), 216
- PyTypeObject.tp_as_mapping (*membro C*), 214
- PyTypeObject.tp_as_number (*membro C*), 214
- PyTypeObject.tp_as_sequence (*membro C*), 214
- PyTypeObject.tp_base (*membro C*), 223
- PyTypeObject.tp_bases (*membro C*), 226
- PyTypeObject.tp_basicsize (*membro C*), 212
- PyTypeObject.tp_cache (*membro C*), 226
- PyTypeObject.tp_call (*membro C*), 215
- PyTypeObject.tp_clear (*membro C*), 220
- PyTypeObject.tp_dealloc (*membro C*), 212
- PyTypeObject.tp_del (*membro C*), 227
- PyTypeObject.tp_descr_get (*membro C*), 223
- PyTypeObject.tp_descr_set (*membro C*), 223
- PyTypeObject.tp_dict (*membro C*), 223
- PyTypeObject.tp_dictoffset (*membro C*), 224
- PyTypeObject.tp_doc (*membro C*), 218
- PyTypeObject.tp_finalize (*membro C*), 227
- PyTypeObject.tp_flags (*membro C*), 216
- PyTypeObject.tp_free (*membro C*), 226

- PyTypeObject.tp_getattr (*membro C*), 213
- PyTypeObject.tp_getattro (*membro C*), 215
- PyTypeObject.tp_getset (*membro C*), 222
- PyTypeObject.tp_hash (*membro C*), 215
- PyTypeObject.tp_init (*membro C*), 224
- PyTypeObject.tp_is_gc (*membro C*), 226
- PyTypeObject.tp_ismember (*membro C*), 212
- PyTypeObject.tp_iter (*membro C*), 222
- PyTypeObject.tp_iternext (*membro C*), 222
- PyTypeObject.tp_members (*membro C*), 222
- PyTypeObject.tp_methods (*membro C*), 222
- PyTypeObject.tp_mro (*membro C*), 226
- PyTypeObject.tp_name (*membro C*), 211
- PyTypeObject.tp_new (*membro C*), 225
- PyTypeObject.tp_repr (*membro C*), 214
- PyTypeObject.tp_richcompare (*membro C*), 220
- PyTypeObject.tp_setattr (*membro C*), 214
- PyTypeObject.tp_setattro (*membro C*), 216
- PyTypeObject.tp_str (*membro C*), 215
- PyTypeObject.tp_subclasses (*membro C*), 227
- PyTypeObject.tp_traverse (*membro C*), 219
- PyTypeObject.tp_vectorcall (*membro C*), 228
- PyTypeObject.tp_vectorcall_offset (*membro C*), 213
- PyTypeObject.tp_version_tag (*membro C*), 227
- PyTypeObject.tp_weaklist (*membro C*), 227
- PyTypeObject.tp_weaklistoffset (*membro C*), 221
- PyTZInfo_Check (*função C*), 147
- PyTZInfo_CheckExact (*função C*), 147
- PyUnicode_1BYTE_DATA (*função C*), 101
- PyUnicode_1BYTE_KIND (*macro C*), 102
- PyUnicode_2BYTE_DATA (*função C*), 101
- PyUnicode_2BYTE_KIND (*macro C*), 102
- PyUnicode_4BYTE_DATA (*função C*), 101
- PyUnicode_4BYTE_KIND (*macro C*), 102
- PyUnicode_AS_DATA (*função C*), 103
- PyUnicode_AS_UNICODE (*função C*), 103
- PyUnicode_AsASCIIString (*função C*), 116
- PyUnicode_AsCharmapString (*função C*), 116
- PyUnicode_AsEncodedString (*função C*), 111
- PyUnicode_AsLatin1String (*função C*), 116
- PyUnicode_AsMBCSString (*função C*), 117
- PyUnicode_AsRawUnicodeEscapeString (*função C*), 115
- PyUnicode_AsUCS4 (*função C*), 107
- PyUnicode_AsUCS4Copy (*função C*), 107
- PyUnicode_AsUnicode (*função C*), 107
- PyUnicode_AsUnicodeAndSize (*função C*), 107
- PyUnicode_AsUnicodeCopy (*função C*), 108
- PyUnicode_AsUnicodeEscapeString (*função C*), 115
- PyUnicode_AsUTF8 (*função C*), 112
- PyUnicode_AsUTF8AndSize (*função C*), 112
- PyUnicode_AsUTF8String (*função C*), 112
- PyUnicode_AsUTF16String (*função C*), 114
- PyUnicode_AsUTF32String (*função C*), 113
- PyUnicode_AsWideChar (*função C*), 110
- PyUnicode_AsWideCharString (*função C*), 110
- PyUnicode_Check (*função C*), 101
- PyUnicode_CheckExact (*função C*), 101
- PyUnicode_Compare (*função C*), 119
- PyUnicode_CompareWithASCIIString (*função C*), 119
- PyUnicode_Concat (*função C*), 118
- PyUnicode_Contains (*função C*), 119
- PyUnicode_CopyCharacters (*função C*), 106
- PyUnicode_Count (*função C*), 118
- PyUnicode_DATA (*função C*), 102
- PyUnicode_Decode (*função C*), 111
- PyUnicode_DecodeASCII (*função C*), 116
- PyUnicode_DecodeCharmap (*função C*), 116
- PyUnicode_DecodeFSDefault (*função C*), 110
- PyUnicode_DecodeFSDefaultAndSize (*função C*), 109
- PyUnicode_DecodeLatin1 (*função C*), 116
- PyUnicode_DecodeLocale (*função C*), 108
- PyUnicode_DecodeLocaleAndSize (*função C*), 108
- PyUnicode_DecodeMBCS (*função C*), 117
- PyUnicode_DecodeMBCSStateful (*função C*), 117
- PyUnicode_DecodeRawUnicodeEscape (*função C*), 115
- PyUnicode_DecodeUnicodeEscape (*função C*), 115
- PyUnicode_DecodeUTF7 (*função C*), 114
- PyUnicode_DecodeUTF7Stateful (*função C*), 114
- PyUnicode_DecodeUTF8 (*função C*), 112
- PyUnicode_DecodeUTF8Stateful (*função C*), 112
- PyUnicode_DecodeUTF16 (*função C*), 113
- PyUnicode_DecodeUTF16Stateful (*função C*), 114
- PyUnicode_DecodeUTF32 (*função C*), 112
- PyUnicode_DecodeUTF32Stateful (*função C*), 113
- PyUnicode_Encode (*função C*), 111
- PyUnicode_EncodeASCII (*função C*), 116
- PyUnicode_EncodeCharmap (*função C*), 117
- PyUnicode_EncodeCodePage (*função C*), 117
- PyUnicode_EncodeFSDefault (*função C*), 110
- PyUnicode_EncodeLatin1 (*função C*), 116
- PyUnicode_EncodeLocale (*função C*), 109
- PyUnicode_EncodeMBCS (*função C*), 118
- PyUnicode_EncodeRawUnicodeEscape (*função C*), 115

- PyUnicode_EncodeUnicodeEscape (*função C*), 115
- PyUnicode_EncodeUTF7 (*função C*), 114
- PyUnicode_EncodeUTF8 (*função C*), 112
- PyUnicode_EncodeUTF16 (*função C*), 114
- PyUnicode_EncodeUTF32 (*função C*), 113
- PyUnicode_Fill (*função C*), 106
- PyUnicode_Find (*função C*), 118
- PyUnicode_FindChar (*função C*), 118
- PyUnicode_Format (*função C*), 119
- PyUnicode_FromEncodedObject (*função C*), 106
- PyUnicode_FromFormat (*função C*), 105
- PyUnicode_FromFormatV (*função C*), 106
- PyUnicode_FromKindAndData (*função C*), 104
- PyUnicode_FromObject (*função C*), 108
- PyUnicode_FromString (*função C*), 105
- PyUnicode_FromString(), 124
- PyUnicode_FromStringAndSize (*função C*), 105
- PyUnicode_FromUnicode (*função C*), 107
- PyUnicode_FromWideChar (*função C*), 110
- PyUnicode_FSConverter (*função C*), 109
- PyUnicode_FSDecoder (*função C*), 109
- PyUnicode_GET_DATA_SIZE (*função C*), 102
- PyUnicode_GET_LENGTH (*função C*), 101
- PyUnicode_GET_SIZE (*função C*), 102
- PyUnicode_GetLength (*função C*), 106
- PyUnicode_GetSize (*função C*), 108
- PyUnicode_InternFromString (*função C*), 119
- PyUnicode_InternInPlace (*função C*), 119
- PyUnicode_IsIdentifier (*função C*), 103
- PyUnicode_Join (*função C*), 118
- PyUnicode_KIND (*função C*), 102
- PyUnicode_MAX_CHAR_VALUE (*macro C*), 102
- PyUnicode_New (*função C*), 104
- PyUnicode_READ (*função C*), 102
- PyUnicode_READ_CHAR (*função C*), 102
- PyUnicode_ReadChar (*função C*), 106
- PyUnicode_READY (*função C*), 101
- PyUnicode_Replace (*função C*), 118
- PyUnicode_RichCompare (*função C*), 119
- PyUnicode_Split (*função C*), 118
- PyUnicode_Splitlines (*função C*), 118
- PyUnicode_Substring (*função C*), 107
- PyUnicode_Tailmatch (*função C*), 118
- PyUnicode_TransformDecimalToASCII (*função C*), 107
- PyUnicode_Translate (*função C*), 117
- PyUnicode_TranslateCharmap (*função C*), 117
- PyUnicode_Type (*variável C*), 101
- PyUnicode_WCHAR_KIND (*macro C*), 102
- PyUnicode_WRITE (*função C*), 102
- PyUnicode_WriteChar (*função C*), 106
- PyUnicodeDecodeError_Create (*função C*), 33
- PyUnicodeDecodeError_GetEncoding (*função C*), 33
- PyUnicodeDecodeError_GetEnd (*função C*), 33
- PyUnicodeDecodeError_GetObject (*função C*), 33
- PyUnicodeDecodeError_GetReason (*função C*), 34
- PyUnicodeDecodeError_GetStart (*função C*), 33
- PyUnicodeDecodeError_SetEnd (*função C*), 33
- PyUnicodeDecodeError_SetReason (*função C*), 34
- PyUnicodeDecodeError_SetStart (*função C*), 33
- PyUnicodeEncodeError_Create (*função C*), 33
- PyUnicodeEncodeError_GetEncoding (*função C*), 33
- PyUnicodeEncodeError_GetEnd (*função C*), 33
- PyUnicodeEncodeError_GetObject (*função C*), 33
- PyUnicodeEncodeError_GetReason (*função C*), 34
- PyUnicodeEncodeError_GetStart (*função C*), 33
- PyUnicodeEncodeError_SetEnd (*função C*), 33
- PyUnicodeEncodeError_SetReason (*função C*), 34
- PyUnicodeEncodeError_SetStart (*função C*), 33
- PyUnicodeObject (*tipo C*), 101
- PyUnicodeTranslateError_Create (*função C*), 33
- PyUnicodeTranslateError_GetEnd (*função C*), 33
- PyUnicodeTranslateError_GetObject (*função C*), 33
- PyUnicodeTranslateError_GetReason (*função C*), 34
- PyUnicodeTranslateError_GetStart (*função C*), 33
- PyUnicodeTranslateError_SetEnd (*função C*), 33
- PyUnicodeTranslateError_SetReason (*função C*), 34
- PyUnicodeTranslateError_SetStart (*função C*), 33
- PyVarObject (*tipo C*), 198
- PyVarObject_HEAD_INIT (*macro C*), 199
- PyVarObject.ob_size (*membro C*), 211
- PyVectorcall_Call (*função C*), 69
- PyVectorcall_Function (*função C*), 68
- PyVectorcall_NARGS (*função C*), 68
- PyWeakref_Check (*função C*), 142
- PyWeakref_CheckProxy (*função C*), 142

PyWeakref_CheckRef (*função C*), 142
PyWeakref_GET_OBJECT (*função C*), 142
PyWeakref_GetObject (*função C*), 142
PyWeakref_NewProxy (*função C*), 142
PyWeakref_NewRef (*função C*), 142
PyWideStringList (*tipo C*), 172
PyWideStringList_Append (*função C*), 172
PyWideStringList_Insert (*função C*), 172
PyWideStringList.items (*membro C*), 172
PyWideStringList.length (*membro C*), 172
PyWrapper_New (*função C*), 139

R

realloc(), 187
releasebufferproc (*tipo C*), 235
repr
 função interna, 65, 214
reprfunc (*tipo C*), 234
richcmpfunc (*tipo C*), 234

S

sdtterr
 stdin stdout, 155
search
 path, module, 12, 154, 156
sequence, 254
 objeto, 97
set
 objeto, 126
set_all(), 9
setattrfunc (*tipo C*), 234
setattrofunc (*tipo C*), 234
setswitchinterval() (*in module sys*), 158
SIGINT, 31
signal
 módulo, 31
SIZE_MAX, 93
special
 method, 255
ssizeargfunc (*tipo C*), 235
ssizeobjargproc (*tipo C*), 235
staticmethod
 função interna, 201
stderr (*in module sys*), 165
stdin
 stdout sdtterr, 155
stdin (*in module sys*), 165
stdout
 sdtterr, stdin, 155
stdout (*in module sys*), 165
strerror(), 27
string
 PyObject_Str (*C function*), 65
sum_list(), 9

sum_sequence(), 10, 11
sys
 módulo, 12, 154, 165
SystemError (*built-in exception*), 133

T

ternaryfunc (*tipo C*), 235
tipagem pato, 246
tipo, 255
tipo alias, 255
tipo genérico, 248
traverseproc (*tipo C*), 239
tupla nomeada, 251
tuple
 função interna, 76, 123
 objeto, 119
type
 função interna, 66
 objeto, 6, 87

U

ULONG_MAX, 93
unaryfunc (*tipo C*), 235

V

váriavel de ambiente
 exec_prefix, 4
 PATH, 12
 prefix, 4
 PYTHON*, 153
 PYTHONCOERCECLOCALE, 182
 PYTHONDEBUG, 152
 PYTHONDONTWRITEBYTECODE, 152
 PYTHONDUMPREFS, 210
 PYTHONHASHSEED, 153
 PYTHONHOME, 12, 153, 158, 178
 PYTHONINSPECT, 153
 PYTHONIOENCODING, 155
 PYTHONLEGACYWINDOWSFSENCODING, 153
 PYTHONLEGACYWINDOWSTDIO, 153
 PYTHONMALLOC, 188, 191, 193
 PYTHONMALLOCSTATS, 188
 PYTHONNOUSERSITE, 154
 PYTHONOLDPARSER, 180
 PYTHONOPTIMIZE, 154
 PYTHONPATH, 12, 153, 179
 PYTHONUNBUFFERED, 154
 PYTHONUTF8, 182
 PYTHONVERBOSE, 154
variável de classe, 245
variável de contexto, 245
vectorcallfunc (*tipo C*), 67
version (*in module sys*), 157
visão de dicionário, 246

visitproc (*tipo C*), [239](#)

Z

Zen do Python, [256](#)