
Extending and Embedding Python

Release 3.9.21

Guido van Rossum
and the Python development team

dezembro 09, 2024

Python Software Foundation
Email: docs@python.org

1	Ferramentas de terceiros recomendadas	3
2	Criando extensões sem ferramentas de terceiros	5
2.1	Estendendo Python com C ou C++	5
2.1.1	Um Exemplo Simples	6
2.1.2	Intermezzo: Errors and Exceptions	7
2.1.3	Back to the Example	9
2.1.4	The Module's Method Table and Initialization Function	9
2.1.5	Compilation and Linkage	11
2.1.6	Calling Python Functions from C	12
2.1.7	Extracting Parameters in Extension Functions	14
2.1.8	Keyword Parameters for Extension Functions	15
2.1.9	Building Arbitrary Values	16
2.1.10	Contagens de referências	17
2.1.11	Writing Extensions in C++	21
2.1.12	Providing a C API for an Extension Module	21
2.2	Definindo Tipos de Extensão: Tutorial	24
2.2.1	O básico	24
2.2.2	Adicionando dados e métodos ao exemplo básico	28
2.2.3	Fornecendo controle mais preciso sobre atributos de dados	35
2.2.4	Apoiando a coleta de lixo cíclica	40
2.2.5	Criando subclasses de outros tipos	46
2.3	Defining Extension Types: Assorted Topics	49
2.3.1	Finalization and De-allocation	51
2.3.2	Object Presentation	52
2.3.3	Attribute Management	53
2.3.4	Object Comparison	55
2.3.5	Abstract Protocol Support	56
2.3.6	Weak Reference Support	57
2.3.7	More Suggestions	58
2.4	Construindo extensões C e C++	59
2.4.1	Construindo extensões C e C++ com distutils	59
2.4.2	Distribuindo seus módulos de extensão	61
2.5	Construindo Extensões C e C++ no Windows	61
2.5.1	Uma abordagem de livro de receitas	61
2.5.2	Diferenças entre o Unix e o Windows	62

2.5.3	Usando DLLs na prática	62
3	Incorporando o tempo de execução do CPython em uma aplicação maior	65
3.1	Incorporando o Python numa Outra Aplicação	65
3.1.1	Very High Level Embedding	66
3.1.2	Beyond Very High Level Embedding: An overview	66
3.1.3	Pure Embedding	67
3.1.4	Extending Embedded Python	69
3.1.5	Embedding Python in C++	70
3.1.6	Compiling and Linking under Unix-like systems	70
A	Glossário	71
B	Sobre esses documentos	85
B.1	Contribuidores da Documentação Python	85
C	História e Licença	87
C.1	História do software	87
C.2	Termos e condições para acessar ou usar Python	88
C.2.1	ACORDO DE LICENCIAMENTO DA PSF PARA PYTHON 3.9.21	88
C.2.2	ACORDO DE LICENCIAMENTO DA BEOPEN.COM PARA PYTHON 2.0	89
C.2.3	CONTRATO DE LICENÇA DA CNRI PARA O PYTHON 1.6.1	90
C.2.4	ACORDO DE LICENÇA DA CWI PARA PYTHON 0.9.0 A 1.2	91
C.2.5	LICENÇA BSD DE ZERO CLÁUSULA PARA CÓDIGO NA DOCUMENTAÇÃO DO PYTHON 3.9.21	91
C.3	Licenças e Reconhecimentos para Software Incorporado	92
C.3.1	Mersenne Twister	92
C.3.2	Soquetes	93
C.3.3	Serviços de soquete assíncrono	93
C.3.4	Gerenciamento de cookies	94
C.3.5	Rastreamento de execução	94
C.3.6	Funções UUencode e UUdecode	95
C.3.7	Chamadas de procedimento remoto XML	95
C.3.8	test_epoll	96
C.3.9	kqueue de seleção	96
C.3.10	SipHash24	97
C.3.11	strtod e dtoa	98
C.3.12	OpenSSL	98
C.3.13	expat	100
C.3.14	libffi	101
C.3.15	zlib	102
C.3.16	cfuhash	102
C.3.17	libmpdec	103
C.3.18	Conjunto de testes C14N do W3C	103
D	Direitos autorais	105
	Índice	107

Neste documento descreveremos o desenvolvimento de módulos com C ou C++ para adicionar recursos ao interpretador Python criando novos módulos. Esses módulos podem não somente definir novas funções, mas também novos tipos de objetos e seu conjunto de métodos. O documento também descreve como incorporar o interpretador Python em outro aplicativo, de forma a utilizá-lo como sendo um idiota estendido. Por fim, estudaremos como podemos compilar e fazer a vinculação dos módulos de extensão para que estes possam ser carregados dinamicamente (em tempo de execução) pelo interpretador, caso o sistema operacional subjacente suportar esse recurso.

Este documento pressupõe conhecimentos básicos sobre Python. Para uma introdução informal à linguagem, consulte [tutorial-index](#). [reference-index](#) fornece uma definição mais formal da linguagem. [library-index](#) documenta os tipos, funções e módulos de objetos existentes (embutidos e escritos em Python) que dão à linguagem sua ampla gama de aplicações.

Para uma descrição detalhada de toda a API Python/C, consulte o [c-api-index](#) separado.

Ferramentas de terceiros recomendadas

Esse guia cobre apenas as ferramentas básicas para a criação de extensões fornecidas como parte desta versão do CPython. Ferramentas de terceiros como [Cython](#), [cffi](#), [SWIG](#) e [Numba](#) oferecem abordagens mais simples e sofisticadas para criar extensões C e C++ para Python.

Ver também:

Guia do Usuário de Empacotamento do Python: Extensões Binárias O Guia do Usuário de Empacotamento do Python não abrange apenas várias ferramentas disponíveis que simplificam a criação de extensões binárias, mas também discute os vários motivos pelos quais a criação de um módulo de extensão pode ser desejável em primeiro lugar.

Criando extensões sem ferramentas de terceiros

Esta seção do guia aborda a criação de extensões C e C++ sem assistência de ferramentas de terceiros. Destina-se principalmente aos criadores dessas ferramentas, em vez de ser uma maneira recomendada de criar suas próprias extensões C.

2.1 Estendendo Python com C ou C++

É muito fácil adicionar novos módulos embutidos ao Python, se você souber programar em C. Você pode adicionar *módulos de extensão* para fazer duas coisas que não podem ser feitas diretamente no Python: eles podem implementar novos tipos de objetos embutidos e eles podem chamar funções da biblioteca C e chamadas do sistema.

Para dar suporte a extensões, a API do Python API (Application Programmers Interface) define um conjunto de funções, macros e variáveis que fornecem acesso à maior parte dos aspectos do sistema de tempo de execução do Python. A API do Python pode ser incorporada em um arquivo fonte em C com a inclusão do cabeçalho `"Python.h"`.

A compilação de um módulo de extensão depende do uso pretendido e da configuração do sistema; detalhes serão dados nos próximos capítulos.

Nota: A interface de extensões em C é específica para o CPython, e módulos de extensão não funcionam em outras implementações do Python. Em muitos casos, é possível evitar a criação destas extensões em C e preservar a portabilidade para outras implementações. Por exemplo, se o seu caso de uso for o de fazer chamadas a funções em bibliotecas C ou chamadas de sistema, considere utilizar o módulo `ctypes` ou a biblioteca `ctypes` ao invés de escrever código C personalizado. Esses módulos permitem escrever código Python que pode interoperar com código C e que é mais portátil entre implementações do Python do que escrever e compilar um módulo de extensão em C.

2.1.1 Um Exemplo Simples

Vamos criar um módulo de extensão chamado `spam` (a comida favorita dos fãs de Monty Python...) e digamos que nosso objetivo seja criar uma interface em Python para a função da biblioteca C `system()`¹. Essa função toma uma string de caracteres terminada em nulo como argumento e retorna um número inteiro. Queremos que essa função seja chamável a partir do Python como abaixo:

```
>>> import spam
>>> status = spam.system("ls -l")
```

Comece criando um arquivo chamado `spammodule.c`. (Historicamente, se um módulo for chamado `spam`, o arquivo C contendo sua implementação é chamado `spammodule.c`; se o nome do módulo for muito longo, como `spammyify`, o nome do arquivo pode ser só `spammyify.c`.)

As duas primeiras linhas do nosso arquivo podem ser:

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

o que carrega a API do Python (você pode adicionar um comentário descrevendo o propósito do módulo e uma nota de copyright, se desejar).

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 recomendado sempre definir `PY_SSIZE_T_CLEAN` antes de incluir `Python.h`. Veja [Extracting Parameters in Extension Functions](#) para uma descrição desse macro.

All user-visible symbols defined by `Python.h` have a prefix of `Py` or `PY`, except those defined in standard header files. For convenience, and since they are used extensively by the Python interpreter, "`Python.h`" includes a few standard header files: `<stdio.h>`, `<string.h>`, `<errno.h>`, and `<stdlib.h>`. If the latter header file does not exist on your system, it declares the functions `malloc()`, `free()` and `realloc()` directly.

The next thing we add to our module file is the C function that will be called when the Python expression `spam.system(string)` is evaluated (we'll see shortly how it ends up being called):

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return PyLong_FromLong(sts);
}
```

There is a straightforward translation from the argument list in Python (for example, the single expression `"ls -l"`) to the arguments passed to the C function. The C function always has two arguments, conventionally named *self* and *args*.

The *self* argument points to the module object for module-level functions; for a method it would point to the object instance.

The *args* argument will be a pointer to a Python tuple object containing the arguments. Each item of the tuple corresponds to an argument in the call's argument list. The arguments are Python objects — in order to do anything with them in our

¹ An interface for this function already exists in the standard module `os` — it was chosen as a simple and straightforward example.

C function we have to convert them to C values. The function `PyArg_ParseTuple()` in the Python API checks the argument types and converts them to C values. It uses a template string to determine the required types of the arguments as well as the types of the C variables into which to store the converted values. More about this later.

`PyArg_ParseTuple()` returns true (nonzero) if all arguments have the right type and its components have been stored in the variables whose addresses are passed. It returns false (zero) if an invalid argument list was passed. In the latter case it also raises an appropriate exception so the calling function can return `NULL` immediately (as we saw in the example).

2.1.2 Intermezzo: Errors and Exceptions

An important convention throughout the Python interpreter is the following: when a function fails, it should set an exception condition and return an error value (usually `-1` or a `NULL` pointer). Exception information is stored in three members of the interpreter's thread state. These are `NULL` if there is no exception. Otherwise they are the C equivalents of the members of the Python tuple returned by `sys.exc_info()`. These are the exception type, exception instance, and a traceback object. It is important to know about them to understand how errors are passed around.

The Python API defines a number of functions to set various types of exceptions.

The most common one is `PyErr_SetString()`. Its arguments are an exception object and a C string. The exception object is usually a predefined object like `PyExc_ZeroDivisionError`. The C string indicates the cause of the error and is converted to a Python string object and stored as the “associated value” of the exception.

Another useful function is `PyErr_SetFromErrno()`, which only takes an exception argument and constructs the associated value by inspection of the global variable `errno`. The most general function is `PyErr_SetObject()`, which takes two object arguments, the exception and its associated value. You don't need to `Py_INCREF()` the objects passed to any of these functions.

You can test non-destructively whether an exception has been set with `PyErr_Occurred()`. This returns the current exception object, or `NULL` if no exception has occurred. You normally don't need to call `PyErr_Occurred()` to see whether an error occurred in a function call, since you should be able to tell from the return value.

When a function *f* that calls another function *g* detects that the latter fails, *f* should itself return an error value (usually `NULL` or `-1`). It should *not* call one of the `PyErr_*` functions — one has already been called by *g*. *f*'s caller is then supposed to also return an error indication to *its* caller, again *without* calling `PyErr_*`, and so on — the most detailed cause of the error was already reported by the function that first detected it. Once the error reaches the Python interpreter's main loop, this aborts the currently executing Python code and tries to find an exception handler specified by the Python programmer.

(There are situations where a module can actually give a more detailed error message by calling another `PyErr_*` function, and in such cases it is fine to do so. As a general rule, however, this is not necessary, and can cause information about the cause of the error to be lost: most operations can fail for a variety of reasons.)

To ignore an exception set by a function call that failed, the exception condition must be cleared explicitly by calling `PyErr_Clear()`. The only time C code should call `PyErr_Clear()` is if it doesn't want to pass the error on to the interpreter but wants to handle it completely by itself (possibly by trying something else, or pretending nothing went wrong).

Every failing `malloc()` call must be turned into an exception — the direct caller of `malloc()` (or `realloc()`) must call `PyErr_NoMemory()` and return a failure indicator itself. All the object-creating functions (for example, `PyLong_FromLong()`) already do this, so this note is only relevant to those who call `malloc()` directly.

Also note that, with the important exception of `PyArg_ParseTuple()` and friends, functions that return an integer status usually return a positive value or zero for success and `-1` for failure, like Unix system calls.

Finally, be careful to clean up garbage (by making `Py_XDECREF()` or `Py_DECREF()` calls for objects you have already created) when you return an error indicator!

The choice of which exception to raise is entirely yours. There are predeclared C objects corresponding to all built-in Python exceptions, such as `PyExc_ZeroDivisionError`, which you can use directly. Of course, you should choose exceptions wisely — don't use `PyExc_TypeError` to mean that a file couldn't be opened (that should probably be `PyExc_IOError`). If something's wrong with the argument list, the `PyArg_ParseTuple()` function usually raises `PyExc_TypeError`. If you have an argument whose value must be in a particular range or must satisfy other conditions, `PyExc_ValueError` is appropriate.

You can also define a new exception that is unique to your module. For this, you usually declare a static object variable at the beginning of your file:

```
static PyObject *SpamError;
```

and initialize it in your module's initialization function (`PyInit_spam()`) with an exception object:

```
PyMODINIT_FUNC
PyInit_spam(void)
{
    PyObject *m;

    m = PyModule_Create(&spammodule);
    if (m == NULL)
        return NULL;

    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    Py_XINCREF(SpamError);
    if (PyModule_AddObject(m, "error", SpamError) < 0) {
        Py_XDECREF(SpamError);
        Py_CLEAR(SpamError);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

Note that the Python name for the exception object is `spam.error`. The `PyErr_NewException()` function may create a class with the base class being `Exception` (unless another class is passed in instead of `NULL`), described in `bltin-exceptions`.

Note also that the `SpamError` variable retains a reference to the newly created exception class; this is intentional! Since the exception could be removed from the module by external code, an owned reference to the class is needed to ensure that it will not be discarded, causing `SpamError` to become a dangling pointer. Should it become a dangling pointer, C code which raises the exception could cause a core dump or other unintended side effects.

We discuss the use of `PyMODINIT_FUNC` as a function return type later in this sample.

The `spam.error` exception can be raised in your extension module using a call to `PyErr_SetString()` as shown below:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
```

(continua na próxima página)

(continuação da página anterior)

```

    if (sts < 0) {
        PyErr_SetString(SpamError, "System command failed");
        return NULL;
    }
    return PyLong_FromLong(sts);
}

```

2.1.3 Back to the Example

Going back to our example function, you should now be able to understand this statement:

```

if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;

```

It returns `NULL` (the error indicator for functions returning object pointers) if an error is detected in the argument list, relying on the exception set by `PyArg_ParseTuple()`. Otherwise the string value of the argument has been copied to the local variable `command`. This is a pointer assignment and you are not supposed to modify the string to which it points (so in Standard C, the variable `command` should properly be declared as `const char *command`).

The next statement is a call to the Unix function `system()`, passing it the string we just got from `PyArg_ParseTuple()`:

```
sts = system(command);
```

Our `spam.system()` function must return the value of `sts` as a Python object. This is done using the function `PyLong_FromLong()`.

```
return PyLong_FromLong(sts);
```

In this case, it will return an integer object. (Yes, even integers are objects on the heap in Python!)

If you have a C function that returns no useful argument (a function returning `void`), the corresponding Python function must return `None`. You need this idiom to do so (which is implemented by the `Py_RETURN_NONE` macro):

```

Py_INCREF(Py_None);
return Py_None;

```

`Py_None` is the C name for the special Python object `None`. It is a genuine Python object rather than a `NULL` pointer, which means “error” in most contexts, as we have seen.

2.1.4 The Module’s Method Table and Initialization Function

I promised to show how `spam_system()` is called from Python programs. First, we need to list its name and address in a “method table”:

```

static PyMethodDef SpamMethods[] = {
    ...
    {"system", spam_system, METH_VARARGS,
     "Execute a shell command."},
    ...
    {NULL, NULL, 0, NULL} /* Sentinel */
};

```

Note the third entry (`METH_VARARGS`). This is a flag telling the interpreter the calling convention to be used for the C function. It should normally always be `METH_VARARGS` or `METH_VARARGS | METH_KEYWORDS`; a value of 0 means that an obsolete variant of `PyArg_ParseTuple()` is used.

When using only `METH_VARARGS`, the function should expect the Python-level parameters to be passed in as a tuple acceptable for parsing via `PyArg_ParseTuple()`; more information on this function is provided below.

The `METH_KEYWORDS` bit may be set in the third field if keyword arguments should be passed to the function. In this case, the C function should accept a third `PyObject *` parameter which will be a dictionary of keywords. Use `PyArg_ParseTupleAndKeywords()` to parse the arguments to such a function.

The method table must be referenced in the module definition structure:

```
static struct PyModuleDef spammodule = {
    PyModuleDef_HEAD_INIT,
    "spam",      /* name of module */
    spam_doc,    /* module documentation, may be NULL */
    -1,          /* size of per-interpreter state of the module,
                  or -1 if the module keeps state in global variables. */
    SpamMethods
};
```

This structure, in turn, must be passed to the interpreter in the module's initialization function. The initialization function must be named `PyInit_name()`, where *name* is the name of the module, and should be the only non-static item defined in the module file:

```
PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModule_Create(&spammodule);
}
```

Note that `PyMODINIT_FUNC` declares the function as `PyObject *` return type, declares any special linkage declarations required by the platform, and for C++ declares the function as `extern "C"`.

When the Python program imports module `spam` for the first time, `PyInit_spam()` is called. (See below for comments about embedding Python.) It calls `PyModule_Create()`, which returns a module object, and inserts built-in function objects into the newly created module based upon the table (an array of `PyMethodDef` structures) found in the module definition. `PyModule_Create()` returns a pointer to the module object that it creates. It may abort with a fatal error for certain errors, or return `NULL` if the module could not be initialized satisfactorily. The init function must return the module object to its caller, so that it then gets inserted into `sys.modules`.

When embedding Python, the `PyInit_spam()` function is not called automatically unless there's an entry in the `PyImport_Inittab` table. To add the module to the initialization table, use `PyImport_AppendInittab()`, optionally followed by an import of the module:

```
int
main(int argc, char *argv[])
{
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }

    /* Add a built-in module, before Py_Initialize */
    if (PyImport_AppendInittab("spam", PyInit_spam) == -1) {
        fprintf(stderr, "Error: could not extend in-built modules table\n");
    }
}
```

(continua na próxima página)

(continuação da página anterior)

```

    exit(1);
}

/* Pass argv[0] to the Python interpreter */
Py_SetProgramName(program);

/* Initialize the Python interpreter. Required.
   If this step fails, it will be a fatal error. */
Py_Initialize();

/* Optionally import the module; alternatively,
   import can be deferred until the embedded script
   imports it. */
PyObject *pmodule = PyImport_ImportModule("spam");
if (!pmodule) {
    PyErr_Print();
    fprintf(stderr, "Error: could not import module 'spam'\n");
}

...

PyMem_RawFree(program);
return 0;
}

```

Nota: Removing entries from `sys.modules` or importing compiled modules into multiple interpreters within a process (or following a `fork()` without an intervening `exec()`) can create problems for some extension modules. Extension module authors should exercise caution when initializing internal data structures.

A more substantial example module is included in the Python source distribution as `Modules/xxmodule.c`. This file may be used as a template or simply read as an example.

Nota: Unlike our `spam` example, `xxmodule` uses *multi-phase initialization* (new in Python 3.5), where a `PyModuleDef` structure is returned from `PyInit_spam`, and creation of the module is left to the import machinery. For details on multi-phase initialization, see [PEP 489](#).

2.1.5 Compilation and Linkage

There are two more things to do before you can use your new extension: compiling and linking it with the Python system. If you use dynamic loading, the details may depend on the style of dynamic loading your system uses; see the chapters about building extension modules (chapter *Construindo extensões C e C++*) and additional information that pertains only to building on Windows (chapter *Construindo Extensões C e C++ no Windows*) for more information about this.

If you can't use dynamic loading, or if you want to make your module a permanent part of the Python interpreter, you will have to change the configuration setup and rebuild the interpreter. Luckily, this is very simple on Unix: just place your file (`spammodule.c` for example) in the `Modules/` directory of an unpacked source distribution, add a line to the file `Modules/Setup.local` describing your file:

```
spam spammodule.o
```

and rebuild the interpreter by running **make** in the toplevel directory. You can also run **make** in the `Modules/` subdirectory, but then you must first rebuild `Makefile` there by running **'make Makefile'**. (This is necessary each time you

change the `Setup` file.)

If your module requires additional libraries to link with, these can be listed on the line in the configuration file as well, for instance:

```
spam spammodule.o -lX11
```

2.1.6 Calling Python Functions from C

So far we have concentrated on making C functions callable from Python. The reverse is also useful: calling Python functions from C. This is especially the case for libraries that support so-called “callback” functions. If a C interface makes use of callbacks, the equivalent Python often needs to provide a callback mechanism to the Python programmer; the implementation will require calling the Python callback functions from a C callback. Other uses are also imaginable.

Fortunately, the Python interpreter is easily called recursively, and there is a standard interface to call a Python function. (I won’t dwell on how to call the Python parser with a particular string as input — if you’re interested, have a look at the implementation of the `-c` command line option in `Modules/main.c` from the Python source code.)

Calling a Python function is easy. First, the Python program must somehow pass you the Python function object. You should provide a function (or some other interface) to do this. When this function is called, save a pointer to the Python function object (be careful to `Py_INCREF()` it!) in a global variable — or wherever you see fit. For example, the following function might be part of a module definition:

```
static PyObject *my_callback = NULL;

static PyObject *
my_set_callback(PyObject *dummy, PyObject *args)
{
    PyObject *result = NULL;
    PyObject *temp;

    if (PyArg_ParseTuple(args, "O:set_callback", &temp)) {
        if (!PyCallable_Check(temp)) {
            PyErr_SetString(PyExc_TypeError, "parameter must be callable");
            return NULL;
        }
        Py_XINCRREF(temp);          /* Add a reference to new callback */
        Py_XDECREF(my_callback);    /* Dispose of previous callback */
        my_callback = temp;         /* Remember new callback */
        /* Boilerplate to return "None" */
        Py_INCREF(Py_None);
        result = Py_None;
    }
    return result;
}
```

This function must be registered with the interpreter using the `METH_VARARGS` flag; this is described in section *The Module’s Method Table and Initialization Function*. The `PyArg_ParseTuple()` function and its arguments are documented in section *Extracting Parameters in Extension Functions*.

The macros `Py_XINCRREF()` and `Py_XDECREF()` increment/decrement the reference count of an object and are safe in the presence of `NULL` pointers (but note that `temp` will not be `NULL` in this context). More info on them in section *Contagens de referências*.

Later, when it is time to call the function, you call the C function `PyObject_CallObject()`. This function has two arguments, both pointers to arbitrary Python objects: the Python function, and the argument list. The argument list must always be a tuple object, whose length is the number of arguments. To call the Python function with no arguments, pass

in NULL, or an empty tuple; to call it with one argument, pass a singleton tuple. `Py_BuildValue()` returns a tuple when its format string consists of zero or more format codes between parentheses. For example:

```
int arg;
PyObject *arglist;
PyObject *result;
...
arg = 123;
...
/* Time to call the callback */
arglist = Py_BuildValue("(i)", arg);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
```

`PyObject_CallObject()` returns a Python object pointer: this is the return value of the Python function. `PyObject_CallObject()` is “reference-count-neutral” with respect to its arguments. In the example a new tuple was created to serve as the argument list, which is `Py_DECREF()`-ed immediately after the `PyObject_CallObject()` call.

The return value of `PyObject_CallObject()` is “new”: either it is a brand new object, or it is an existing object whose reference count has been incremented. So, unless you want to save it in a global variable, you should somehow `Py_DECREF()` the result, even (especially!) if you are not interested in its value.

Before you do this, however, it is important to check that the return value isn’t NULL. If it is, the Python function terminated by raising an exception. If the C code that called `PyObject_CallObject()` is called from Python, it should now return an error indication to its Python caller, so the interpreter can print a stack trace, or the calling Python code can handle the exception. If this is not possible or desirable, the exception should be cleared by calling `PyErr_Clear()`. For example:

```
if (result == NULL)
    return NULL; /* Pass error back */
...use result...
Py_DECREF(result);
```

Depending on the desired interface to the Python callback function, you may also have to provide an argument list to `PyObject_CallObject()`. In some cases the argument list is also provided by the Python program, through the same interface that specified the callback function. It can then be saved and used in the same manner as the function object. In other cases, you may have to construct a new tuple to pass as the argument list. The simplest way to do this is to call `Py_BuildValue()`. For example, if you want to pass an integral event code, you might use the following code:

```
PyObject *arglist;
...
arglist = Py_BuildValue("(l)", eventcode);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

Note the placement of `Py_DECREF(arglist)` immediately after the call, before the error check! Also note that strictly speaking this code is not complete: `Py_BuildValue()` may run out of memory, and this should be checked.

You may also call a function with keyword arguments by using `PyObject_Call()`, which supports arguments and keyword arguments. As in the above example, we use `Py_BuildValue()` to construct the dictionary.

```
PyObject *dict;
...
```

(continua na próxima página)

(continuação da página anterior)

```
dict = Py_BuildValue("{s:i}", "name", val);
result = PyObject_Call(my_callback, NULL, dict);
Py_DECREF(dict);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

2.1.7 Extracting Parameters in Extension Functions

The `PyArg_ParseTuple()` function is declared as follows:

```
int PyArg_ParseTuple(PyObject *arg, const char *format, ...);
```

The *arg* argument must be a tuple object containing an argument list passed from Python to a C function. The *format* argument must be a format string, whose syntax is explained in arg-parsing in the Python/C API Reference Manual. The remaining arguments must be addresses of variables whose type is determined by the format string.

Note that while `PyArg_ParseTuple()` checks that the Python arguments have the required types, it cannot check the validity of the addresses of C variables passed to the call: if you make mistakes there, your code will probably crash or at least overwrite random bits in memory. So be careful!

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

Some example calls:

```
#define PY_SSIZE_T_CLEAN /* Make "s#" use Py_ssize_t rather than int. */
#include <Python.h>
```

```
int ok;
int i, j;
long k, l;
const char *s;
Py_ssize_t size;

ok = PyArg_ParseTuple(args, ""); /* No arguments */
/* Python call: f() */
```

```
ok = PyArg_ParseTuple(args, "s", &s); /* A string */
/* Possible Python call: f('whoops!') */
```

```
ok = PyArg_ParseTuple(args, "lls", &k, &l, &s); /* Two longs and a string */
/* Possible Python call: f(1, 2, 'three') */
```

```
ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size);
/* A pair of ints and a string, whose size is also returned */
/* Possible Python call: f((1, 2), 'three') */
```

```
{
    const char *file;
    const char *mode = "r";
    int bufsize = 0;
```

(continua na próxima página)

(continuação da página anterior)

```

ok = PyArg_ParseTuple(args, "s|si", &file, &mode, &bufsize);
/* A string, and optionally another string and an integer */
/* Possible Python calls:
    f('spam')
    f('spam', 'w')
    f('spam', 'wb', 100000) */
}
    
```

```

{
    int left, top, right, bottom, h, v;
    ok = PyArg_ParseTuple(args, "((ii)(ii))(ii)",
        &left, &top, &right, &bottom, &h, &v);
    /* A rectangle and a point */
    /* Possible Python call:
        f(((0, 0), (400, 300)), (10, 10)) */
}
    
```

```

{
    Py_complex c;
    ok = PyArg_ParseTuple(args, "D:myfunction", &c);
    /* a complex, also providing a function name for errors */
    /* Possible Python call: myfunction(1+2j) */
}
    
```

2.1.8 Keyword Parameters for Extension Functions

The `PyArg_ParseTupleAndKeywords()` function is declared as follows:

```

int PyArg_ParseTupleAndKeywords(PyObject *arg, PyObject *kwdict,
                                const char *format, char *kwlist[], ...);
    
```

The *arg* and *format* parameters are identical to those of the `PyArg_ParseTuple()` function. The *kwdict* parameter is the dictionary of keywords received as the third parameter from the Python runtime. The *kwlist* parameter is a NULL-terminated list of strings which identify the parameters; the names are matched with the type information from *format* from left to right. On success, `PyArg_ParseTupleAndKeywords()` returns true, otherwise it returns false and raises an appropriate exception.

Nota: Nested tuples cannot be parsed when using keyword arguments! Keyword parameters passed in which are not present in the *kwlist* will cause `TypeError` to be raised.

Here is an example module which uses keywords, based on an example by Geoff Philbrick (philbrick@hks.com):

```

#define PY_SSIZE_T_CLEAN /* Make "s#" use Py_ssize_t rather than int. */
#include <Python.h>

static PyObject *
keywdarg_parrot(PyObject *self, PyObject *args, PyObject *keywds)
{
    int voltage;
    const char *state = "a stiff";
    const char *action = "voom";
    const char *type = "Norwegian Blue";
    
```

(continua na próxima página)

(continuação da página anterior)

```

static char *kwlist[] = {"voltage", "state", "action", "type", NULL};

if (!PyArg_ParseTupleAndKeywords(args, keywds, "i|sss", kwlist,
                                &voltage, &state, &action, &type))

    return NULL;

printf("-- This parrot wouldn't %s if you put %i Volts through it.\n",
        action, voltage);
printf("-- Lovely plumage, the %s -- It's %s!\n", type, state);

Py_RETURN_NONE;
}

static PyMethodDef keywdarg_methods[] = {
    /* The cast of the function is necessary since PyCFunction values
     * only take two PyObject* parameters, and keywdarg_parrot() takes
     * three.
     */
    {"parrot", (PyCFunction)(void*)(void)keywdarg_parrot, METH_VARARGS | METH_
↪KEYWORDS,
    "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL} /* sentinel */
};

static struct PyModuleDef keywdargmodule = {
    PyModuleDef_HEAD_INIT,
    "keywdarg",
    NULL,
    -1,
    keywdarg_methods
};

PyMODINIT_FUNC
PyInit_keywdarg(void)
{
    return PyModule_Create(&keywdargmodule);
}

```

2.1.9 Building Arbitrary Values

This function is the counterpart to `PyArg_ParseTuple()`. It is declared as follows:

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

It recognizes a set of format units similar to the ones recognized by `PyArg_ParseTuple()`, but the arguments (which are input to the function, not output) must not be pointers, just values. It returns a new Python object, suitable for returning from a C function called from Python.

One difference with `PyArg_ParseTuple()`: while the latter requires its first argument to be a tuple (since Python argument lists are always represented as tuples internally), `Py_BuildValue()` does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

Examples (to the left the call, to the right the resulting Python value):

<code>Py_BuildValue("")</code>	<code>None</code>
<code>Py_BuildValue("i", 123)</code>	<code>123</code>
<code>Py_BuildValue("iii", 123, 456, 789)</code>	<code>(123, 456, 789)</code>
<code>Py_BuildValue("s", "hello")</code>	<code>'hello'</code>
<code>Py_BuildValue("y", "hello")</code>	<code>b'hello'</code>
<code>Py_BuildValue("ss", "hello", "world")</code>	<code>('hello', 'world')</code>
<code>Py_BuildValue("s#", "hello", 4)</code>	<code>'hell'</code>
<code>Py_BuildValue("y#", "hello", 4)</code>	<code>b'hell'</code>
<code>Py_BuildValue("()")</code>	<code>()</code>
<code>Py_BuildValue("(i)", 123)</code>	<code>(123,)</code>
<code>Py_BuildValue("(ii)", 123, 456)</code>	<code>(123, 456)</code>
<code>Py_BuildValue("(i,i)", 123, 456)</code>	<code>(123, 456)</code>
<code>Py_BuildValue("[i,i]", 123, 456)</code>	<code>[123, 456]</code>
<code>Py_BuildValue("{s:i,s:i}",</code>	
<code> "abc", 123, "def", 456)</code>	<code>{'abc': 123, 'def': 456}</code>
<code>Py_BuildValue("((ii)(ii)) (ii)",</code>	
<code> 1, 2, 3, 4, 5, 6)</code>	<code>(((1, 2), (3, 4)), (5, 6))</code>

2.1.10 Contagens de referências

In languages like C or C++, the programmer is responsible for dynamic allocation and deallocation of memory on the heap. In C, this is done using the functions `malloc()` and `free()`. In C++, the operators `new` and `delete` are used with essentially the same meaning and we'll restrict the following discussion to the C case.

Every block of memory allocated with `malloc()` should eventually be returned to the pool of available memory by exactly one call to `free()`. It is important to call `free()` at the right time. If a block's address is forgotten but `free()` is not called for it, the memory it occupies cannot be reused until the program terminates. This is called a *memory leak*. On the other hand, if a program calls `free()` for a block and then continues to use the block, it creates a conflict with re-use of the block through another `malloc()` call. This is called *using freed memory*. It has the same bad consequences as referencing uninitialized data — core dumps, wrong results, mysterious crashes.

Common causes of memory leaks are unusual paths through the code. For instance, a function may allocate a block of memory, do some calculation, and then free the block again. Now a change in the requirements for the function may add a test to the calculation that detects an error condition and can return prematurely from the function. It's easy to forget to free the allocated memory block when taking this premature exit, especially when it is added later to the code. Such leaks, once introduced, often go undetected for a long time: the error exit is taken only in a small fraction of all calls, and most modern machines have plenty of virtual memory, so the leak only becomes apparent in a long-running process that uses the leaking function frequently. Therefore, it's important to prevent leaks from happening by having a coding convention or strategy that minimizes this kind of errors.

Since Python makes heavy use of `malloc()` and `free()`, it needs a strategy to avoid memory leaks as well as the use of freed memory. The chosen method is called *reference counting*. The principle is simple: every object contains a counter, which is incremented when a reference to the object is stored somewhere, and which is decremented when a reference to it is deleted. When the counter reaches zero, the last reference to the object has been deleted and the object is freed.

An alternative strategy is called *automatic garbage collection*. (Sometimes, reference counting is also referred to as a garbage collection strategy, hence my use of "automatic" to distinguish the two.) The big advantage of automatic garbage collection is that the user doesn't need to call `free()` explicitly. (Another claimed advantage is an improvement in speed or memory usage — this is no hard fact however.) The disadvantage is that for C, there is no truly portable automatic garbage collector, while reference counting can be implemented portably (as long as the functions `malloc()` and `free()` are available — which the C Standard guarantees). Maybe some day a sufficiently portable automatic garbage collector will be available for C. Until then, we'll have to live with reference counts.

While Python uses the traditional reference counting implementation, it also offers a cycle detector that works to detect reference cycles. This allows applications to not worry about creating direct or indirect circular references; these are the

weakness of garbage collection implemented using only reference counting. Reference cycles consist of objects which contain (possibly indirect) references to themselves, so that each object in the cycle has a reference count which is non-zero. Typical reference counting implementations are not able to reclaim the memory belonging to any objects in a reference cycle, or referenced from the objects in the cycle, even though there are no further references to the cycle itself.

The cycle detector is able to detect garbage cycles and can reclaim them. The `gc` module exposes a way to run the detector (the `collect()` function), as well as configuration interfaces and the ability to disable the detector at runtime. The cycle detector is considered an optional component; though it is included by default, it can be disabled at build time using the `--without-cycle-gc` option to the **configure** script on Unix platforms (including Mac OS X). If the cycle detector is disabled in this way, the `gc` module will not be available.

Reference Counting in Python

There are two macros, `Py_INCREF(x)` and `Py_DECREF(x)`, which handle the incrementing and decrementing of the reference count. `Py_DECREF()` also frees the object when the count reaches zero. For flexibility, it doesn't call `free()` directly — rather, it makes a call through a function pointer in the object's *type object*. For this purpose (and others), every object also contains a pointer to its *type object*.

The big question now remains: when to use `Py_INCREF(x)` and `Py_DECREF(x)`? Let's first introduce some terms. Nobody “owns” an object; however, you can *own a reference* to an object. An object's reference count is now defined as the number of owned references to it. The owner of a reference is responsible for calling `Py_DECREF()` when the reference is no longer needed. Ownership of a reference can be transferred. There are three ways to dispose of an owned reference: pass it on, store it, or call `Py_DECREF()`. Forgetting to dispose of an owned reference creates a memory leak.

It is also possible to *borrow*² a reference to an object. The borrower of a reference should not call `Py_DECREF()`. The borrower must not hold on to the object longer than the owner from which it was borrowed. Using a borrowed reference after the owner has disposed of it risks using freed memory and should be avoided completely³.

The advantage of borrowing over owning a reference is that you don't need to take care of disposing of the reference on all possible paths through the code — in other words, with a borrowed reference you don't run the risk of leaking when a premature exit is taken. The disadvantage of borrowing over owning is that there are some subtle situations where in seemingly correct code a borrowed reference can be used after the owner from which it was borrowed has in fact disposed of it.

A borrowed reference can be changed into an owned reference by calling `Py_INCREF()`. This does not affect the status of the owner from which the reference was borrowed — it creates a new owned reference, and gives full owner responsibilities (the new owner must dispose of the reference properly, as well as the previous owner).

Ownership Rules

Whenever an object reference is passed into or out of a function, it is part of the function's interface specification whether ownership is transferred with the reference or not.

Most functions that return a reference to an object pass on ownership with the reference. In particular, all functions whose function it is to create a new object, such as `PyLong_FromLong()` and `Py_BuildValue()`, pass ownership to the receiver. Even if the object is not actually new, you still receive ownership of a new reference to that object. For instance, `PyLong_FromLong()` maintains a cache of popular values and can return a reference to a cached item.

Many functions that extract objects from other objects also transfer ownership with the reference, for instance `PyObject_GetAttrString()`. The picture is less clear, here, however, since a few common routines are exceptions: `PyTuple_GetItem()`, `PyList_GetItem()`, `PyDict_GetItem()`, and `PyDict_GetItemString()` all return references that you borrow from the tuple, list or dictionary.

² The metaphor of “borrowing” a reference is not completely correct: the owner still has a copy of the reference.

³ Checking that the reference count is at least 1 **does not work** — the reference count itself could be in freed memory and may thus be reused for another object!

The function `PyImport_AddModule()` also returns a borrowed reference, even though it may actually create the object it returns: this is possible because an owned reference to the object is stored in `sys.modules`.

When you pass an object reference into another function, in general, the function borrows the reference from you — if it needs to store it, it will use `Py_INCREF()` to become an independent owner. There are exactly two important exceptions to this rule: `PyTuple_SetItem()` and `PyList_SetItem()`. These functions take over ownership of the item passed to them — even if they fail! (Note that `PyDict_SetItem()` and friends don't take over ownership — they are “normal.”)

When a C function is called from Python, it borrows references to its arguments from the caller. The caller owns a reference to the object, so the borrowed reference's lifetime is guaranteed until the function returns. Only when such a borrowed reference must be stored or passed on, it must be turned into an owned reference by calling `Py_INCREF()`.

The object reference returned from a C function that is called from Python must be an owned reference — ownership is transferred from the function to its caller.

Thin Ice

There are a few situations where seemingly harmless use of a borrowed reference can lead to problems. These all have to do with implicit invocations of the interpreter, which can cause the owner of a reference to dispose of it.

The first and most important case to know about is using `Py_DECREF()` on an unrelated object while borrowing a reference to a list item. For instance:

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

This function first borrows a reference to `list[0]`, then replaces `list[1]` with the value 0, and finally prints the borrowed reference. Looks harmless, right? But it's not!

Let's follow the control flow into `PyList_SetItem()`. The list owns references to all its items, so when item 1 is replaced, it has to dispose of the original item 1. Now let's suppose the original item 1 was an instance of a user-defined class, and let's further suppose that the class defined a `__del__()` method. If this class instance has a reference count of 1, disposing of it will call its `__del__()` method.

Since it is written in Python, the `__del__()` method can execute arbitrary Python code. Could it perhaps do something to invalidate the reference to `item` in `bug()`? You bet! Assuming that the list passed into `bug()` is accessible to the `__del__()` method, it could execute a statement to the effect of `del list[0]`, and assuming this was the last reference to that object, it would free the memory associated with it, thereby invalidating `item`.

The solution, once you know the source of the problem, is easy: temporarily increment the reference count. The correct version of the function reads:

```
void
no_bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    Py_INCREF(item);
    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0);
}
```

(continua na próxima página)

(continuação da página anterior)

```
Py_DECREF(item);
}
```

This is a true story. An older version of Python contained variants of this bug and someone spent a considerable amount of time in a C debugger to figure out why his `__del__()` methods would fail...

The second case of problems with a borrowed reference is a variant involving threads. Normally, multiple threads in the Python interpreter can't get in each other's way, because there is a global lock protecting Python's entire object space. However, it is possible to temporarily release this lock using the macro `Py_BEGIN_ALLOW_THREADS`, and to re-acquire it using `Py_END_ALLOW_THREADS`. This is common around blocking I/O calls, to let other threads use the processor while waiting for the I/O to complete. Obviously, the following function has the same problem as the previous one:

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    Py_BEGIN_ALLOW_THREADS
    ...some blocking I/O call...
    Py_END_ALLOW_THREADS
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

NULL Pointers

In general, functions that take object references as arguments do not expect you to pass them `NULL` pointers, and will dump core (or cause later core dumps) if you do so. Functions that return object references generally return `NULL` only to indicate that an exception occurred. The reason for not testing for `NULL` arguments is that functions often pass the objects they receive on to other function — if each function were to test for `NULL`, there would be a lot of redundant tests and the code would run more slowly.

It is better to test for `NULL` only at the “source:” when a pointer that may be `NULL` is received, for example, from `malloc()` or from a function that may raise an exception.

The macros `Py_INCREF()` and `Py_DECREF()` do not check for `NULL` pointers — however, their variants `Py_XINCREF()` and `Py_XDECREF()` do.

The macros for checking for a particular object type (`Pytype_Check()`) don't check for `NULL` pointers — again, there is much code that calls several of these in a row to test an object against various different expected types, and this would generate redundant tests. There are no variants with `NULL` checking.

The C function calling mechanism guarantees that the argument list passed to C functions (`args` in the examples) is never `NULL` — in fact it guarantees that it is always a tuple⁴.

It is a severe error to ever let a `NULL` pointer “escape” to the Python user.

⁴ These guarantees don't hold when you use the “old” style calling convention — this is still found in much existing code.

2.1.11 Writing Extensions in C++

It is possible to write extension modules in C++. Some restrictions apply. If the main program (the Python interpreter) is compiled and linked by the C compiler, global or static objects with constructors cannot be used. This is not a problem if the main program is linked by the C++ compiler. Functions that will be called by the Python interpreter (in particular, module initialization functions) have to be declared using `extern "C"`. It is unnecessary to enclose the Python header files in `extern "C" { ... }` — they use this form already if the symbol `__cplusplus` is defined (all recent C++ compilers define this symbol).

2.1.12 Providing a C API for an Extension Module

Many extension modules just provide new functions and types to be used from Python, but sometimes the code in an extension module can be useful for other extension modules. For example, an extension module could implement a type “collection” which works like lists without order. Just like the standard Python list type has a C API which permits extension modules to create and manipulate lists, this new collection type should have a set of C functions for direct manipulation from other extension modules.

At first sight this seems easy: just write the functions (without declaring them `static`, of course), provide an appropriate header file, and document the C API. And in fact this would work if all extension modules were always linked statically with the Python interpreter. When modules are used as shared libraries, however, the symbols defined in one module may not be visible to another module. The details of visibility depend on the operating system; some systems use one global namespace for the Python interpreter and all extension modules (Windows, for example), whereas others require an explicit list of imported symbols at module link time (AIX is one example), or offer a choice of different strategies (most Unices). And even if symbols are globally visible, the module whose functions one wishes to call might not have been loaded yet!

Portability therefore requires not to make any assumptions about symbol visibility. This means that all symbols in extension modules should be declared `static`, except for the module’s initialization function, in order to avoid name clashes with other extension modules (as discussed in section *The Module’s Method Table and Initialization Function*). And it means that symbols that *should* be accessible from other extension modules must be exported in a different way.

Python provides a special mechanism to pass C-level information (pointers) from one extension module to another one: Capsules. A Capsule is a Python data type which stores a pointer (`void *`). Capsules can only be created and accessed via their C API, but they can be passed around like any other Python object. In particular, they can be assigned to a name in an extension module’s namespace. Other extension modules can then import this module, retrieve the value of this name, and then retrieve the pointer from the Capsule.

There are many ways in which Capsules can be used to export the C API of an extension module. Each function could get its own Capsule, or all C API pointers could be stored in an array whose address is published in a Capsule. And the various tasks of storing and retrieving the pointers can be distributed in different ways between the module providing the code and the client modules.

Whichever method you choose, it’s important to name your Capsules properly. The function `PyCapsule_New()` takes a name parameter (`const char *`); you’re permitted to pass in a `NULL` name, but we strongly encourage you to specify a name. Properly named Capsules provide a degree of runtime type-safety; there is no feasible way to tell one unnamed Capsule from another.

In particular, Capsules used to expose C APIs should be given a name following this convention:

```
modulename.attributename
```

The convenience function `PyCapsule_Import()` makes it easy to load a C API provided via a Capsule, but only if the Capsule’s name matches this convention. This behavior gives C API users a high degree of certainty that the Capsule they load contains the correct C API.

The following example demonstrates an approach that puts most of the burden on the writer of the exporting module, which is appropriate for commonly used library modules. It stores all C API pointers (just one in the example!) in an

array of `void` pointers which becomes the value of a Capsule. The header file corresponding to the module provides a macro that takes care of importing the module and retrieving its C API pointers; client modules only have to call this macro before accessing the C API.

The exporting module is a modification of the `spam` module from section *Um Exemplo Simples*. The function `spam.system()` does not call the C library function `system()` directly, but a function `PySpam_System()`, which would of course do something more complicated in reality (such as adding “spam” to every command). This function `PySpam_System()` is also exported to other extension modules.

The function `PySpam_System()` is a plain C function, declared `static` like everything else:

```
static int
PySpam_System(const char *command)
{
    return system(command);
}
```

The function `spam_system()` is modified in a trivial way:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = PySpam_System(command);
    return PyLong_FromLong(sts);
}
```

In the beginning of the module, right after the line

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

two more lines must be added:

```
#define SPAM_MODULE
#include "spammodule.h"
```

The `#define` is used to tell the header file that it is being included in the exporting module, not a client module. Finally, the module’s initialization function must take care of initializing the C API pointer array:

```
PyMODINIT_FUNC
PyInit_spam(void)
{
    PyObject *m;
    static void *PySpam_API[PySpam_API_pointers];
    PyObject *c_api_object;

    m = PyModule_Create(&spammodule);
    if (m == NULL)
        return NULL;

    /* Initialize the C API pointer array */
    PySpam_API[PySpam_System_NUM] = (void *)PySpam_System;

    /* Create a Capsule containing the API pointer array's address */
```

(continua na próxima página)

(continuação da página anterior)

```

c_api_object = PyCapsule_New((void *)PySpam_API, "spam._C_API", NULL);

if (PyModule_AddObject(m, "_C_API", c_api_object) < 0) {
    Py_XDECREF(c_api_object);
    Py_DECREF(m);
    return NULL;
}

return m;
}
    
```

Note that `PySpam_API` is declared `static`; otherwise the pointer array would disappear when `PyInit_spam()` terminates!

The bulk of the work is in the header file `spammodule.h`, which looks like this:

```

#ifndef Py_SPAMMODULE_H
#define Py_SPAMMODULE_H
#ifdef __cplusplus
extern "C" {
#endif

/* Header file for spammodule */

/* C API functions */
#define PySpam_System_NUM 0
#define PySpam_System_RETURN int
#define PySpam_System_PROTO (const char *command)

/* Total number of C API pointers */
#define PySpam_API_pointers 1

#ifdef SPAM_MODULE
/* This section is used when compiling spammodule.c */

static PySpam_System_RETURN PySpam_System PySpam_System_PROTO;

#else
/* This section is used in modules that use spammodule's API */

static void **PySpam_API;

#define PySpam_System \
    (*(PySpam_System_RETURN (*)(PySpam_System_PROTO) PySpam_API[PySpam_System_NUM])

/* Return -1 on error, 0 on success.
 * PyCapsule_Import will set an exception if there's an error.
 */
static int
import_spam(void)
{
    PySpam_API = (void **)PyCapsule_Import("spam._C_API", 0);
    return (PySpam_API != NULL) ? 0 : -1;
}

#endif
    
```

(continua na próxima página)

(continuação da página anterior)

```
#ifdef __cplusplus
}
#endif

#endif /* !defined(Py_SPAMMODULE_H) */
```

All that a client module must do in order to have access to the function `PySpam_System()` is to call the function (or rather macro) `import_spam()` in its initialization function:

```
PyMODINIT_FUNC
PyInit_client(void)
{
    PyObject *m;

    m = PyModule_Create(&clientmodule);
    if (m == NULL)
        return NULL;
    if (import_spam() < 0)
        return NULL;
    /* additional initialization can happen here */
    return m;
}
```

The main disadvantage of this approach is that the file `spammodule.h` is rather complicated. However, the basic structure is the same for each function that is exported, so it has to be learned only once.

Finally it should be mentioned that Capsules offer additional functionality, which is especially useful for memory allocation and deallocation of the pointer stored in a Capsule. The details are described in the Python/C API Reference Manual in the section capsules and in the implementation of Capsules (files `Include/pycapsule.h` and `Objects/pycapsule.c` in the Python source code distribution).

2.2 Definindo Tipos de Extensão: Tutorial

O Python permite que o gravador de um módulo de extensão C defina novos tipos que podem ser manipulados a partir do código Python, da mesma forma que os tipos embutidos `str` e `list`. O código para todos os tipos de extensão segue um padrão, mas há alguns detalhes que você precisa entender antes de começar. Este documento é uma introdução suave ao tópico.

2.2.1 O básico

The *CPython* runtime sees all Python objects as variables of type `PyObject*`, which serves as a “base type” for all Python objects. The `PyObject` structure itself only contains the object’s *reference count* and a pointer to the object’s “type object”. This is where the action is; the type object determines which (C) functions get called by the interpreter when, for instance, an attribute gets looked up on an object, a method called, or it is multiplied by another object. These C functions are called “type methods”.

Então, se você quiser definir um novo tipo de extensão, você precisa criar um novo objeto de tipo.

This sort of thing can only be explained by example, so here’s a minimal, but complete, module that defines a new type named `Custom` inside a C extension module `custom`:

Nota: What we're showing here is the traditional way of defining *static* extension types. It should be adequate for most uses. The C API also allows defining heap-allocated extension types using the `PyType_FromSpec()` function, which isn't covered in this tutorial.

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

typedef struct {
    PyObject_HEAD
    /* Type-specific fields go here. */
} CustomObject;

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = PyType_GenericNew,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

Agora isso é um pouco para ser absorvido de uma só vez, mas esperamos que os bits pareçam familiares no capítulo anterior. Este arquivo define três coisas:

1. O que o **objeto** `Custom` contém: esta é a estrutura do `CustomObject`, que é alocada uma vez a cada instância `Custom`.
2. How the `Custom` **type** behaves: this is the `CustomType` struct, which defines a set of flags and function pointers

that the interpreter inspects when specific operations are requested.

3. How to initialize the custom module: this is the `PyInit_custom` function and the associated `custommodule` struct.

O primeiro bit é

```
typedef struct {
    PyObject_HEAD
} CustomObject;
```

This is what a Custom object will contain. `PyObject_HEAD` is mandatory at the start of each object struct and defines a field called `ob_base` of type `PyObject`, containing a pointer to a type object and a reference count (these can be accessed using the macros `Py_TYPE` and `Py_REFCNT` respectively). The reason for the macro is to abstract away the layout and to enable additional fields in debug builds.

Nota: There is no semicolon above after the `PyObject_HEAD` macro. Be wary of adding one by accident: some compilers will complain.

Of course, objects generally store additional data besides the standard `PyObject_HEAD` boilerplate; for example, here is the definition for standard Python floats:

```
typedef struct {
    PyObject_HEAD
    double ob_fval;
} PyFloatObject;
```

O segundo bit é a definição do objeto de tipo.

```
static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = PyType_GenericNew,
};
```

Nota: We recommend using C99-style designated initializers as above, to avoid listing all the `PyTypeObject` fields that you don't care about and also to avoid caring about the fields' declaration order.

The actual definition of `PyTypeObject` in `object.h` has many more fields than the definition above. The remaining fields will be filled with zeros by the C compiler, and it's common practice to not specify them explicitly unless you need them.

Vamos separá-lo, um campo de cada vez

```
PyVarObject_HEAD_INIT(NULL, 0)
```

This line is mandatory boilerplate to initialize the `ob_base` field mentioned above.

```
.tp_name = "custom.Custom",
```

The name of our type. This will appear in the default textual representation of our objects and in some error messages, for example:

```
>>> "" + custom.Custom()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "custom.Custom") to str
```

Note that the name is a dotted name that includes both the module name and the name of the type within the module. The module in this case is `custom` and the type is `Custom`, so we set the type name to `custom.Custom`. Using the real dotted import path is important to make your type compatible with the `pydoc` and `pickle` modules.

```
.tp_basicsize = sizeof(CustomObject),
.tp_itemsize = 0,
```

This is so that Python knows how much memory to allocate when creating new `Custom` instances. `tp_itemsize` is only used for variable-sized objects and should otherwise be zero.

Nota: If you want your type to be subclassable from Python, and your type has the same `tp_basicsize` as its base type, you may have problems with multiple inheritance. A Python subclass of your type will have to list your type first in its `__bases__`, or else it will not be able to call your type's `__new__()` method without getting an error. You can avoid this problem by ensuring that your type has a larger value for `tp_basicsize` than its base type does. Most of the time, this will be true anyway, because either your base type will be `object`, or else you will be adding data members to your base type, and therefore increasing its size.

We set the class flags to `Py_TPFLAGS_DEFAULT`.

```
.tp_flags = Py_TPFLAGS_DEFAULT,
```

All types should include this constant in their flags. It enables all of the members defined until at least Python 3.3. If you need further members, you will need to OR the corresponding flags.

We provide a doc string for the type in `tp_doc`.

```
.tp_doc = PyDoc_STR("Custom objects"),
```

To enable object creation, we have to provide a `tp_new` handler. This is the equivalent of the Python method `__new__()`, but has to be specified explicitly. In this case, we can just use the default implementation provided by the API function `PyType_GenericNew()`.

```
.tp_new = PyType_GenericNew,
```

Everything else in the file should be familiar, except for some code in `PyInit_custom()`:

```
if (PyType_Ready(&CustomType) < 0)
    return;
```

This initializes the `Custom` type, filling in a number of members to the appropriate default values, including `ob_type` that we initially set to `NULL`.

```
Py_INCREF(&CustomType);
if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
    Py_DECREF(&CustomType);
    Py_DECREF(m);
    return NULL;
}
```

This adds the type to the module dictionary. This allows us to create `Custom` instances by calling the `Custom` class:

```
>>> import custom
>>> mycustom = custom.Custom()
```

That's it! All that remains is to build it; put the above code in a file called `custom.c` and:

```
from distutils.core import setup, Extension
setup(name="custom", version="1.0",
      ext_modules=[Extension("custom", ["custom.c"])])
```

in a file called `setup.py`; then typing

```
$ python setup.py build
```

at a shell should produce a file `custom.so` in a subdirectory; move to that directory and fire up Python — you should be able to import `custom` and play around with `Custom` objects.

Isso não foi tão difícil, foi?

Naturalmente, o tipo personalizado atual é bastante desinteressante. Não tem dados e não faz nada. Não pode nem ser subclassificado.

Nota: While this documentation showcases the standard `distutils` module for building C extensions, it is recommended in real-world use cases to use the newer and better-maintained `setuptools` library. Documentation on how to do this is out of scope for this document and can be found in the [Python Packaging User's Guide](#).

2.2.2 Adicionando dados e métodos ao exemplo básico

Let's extend the basic example to add some data and methods. Let's also make the type usable as a base class. We'll create a new module, `custom2` that adds these capabilities:

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

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
```

(continua na próxima página)

(continuação da página anterior)

```

    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist,
                                     &first, &last,
                                     &self->number))

        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"first", T_OBJECT_EX, offsetof(CustomObject, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(CustomObject, last), 0,
     "last name"},
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
    }
}
    
```

(continua na próxima página)

```

        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom2.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_new = Custom_new,
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom2",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom2(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}

```

Esta versão do módulo possui várias alterações.

Nós adicionamos uma inclusão extra

```
#include <structmember.h>
```

Esta inclusão fornece declarações que usamos para manipular atributos, conforme descrito um pouco mais tarde.

The Custom type now has three data attributes in its C struct, *first*, *last*, and *number*. The *first* and *last* variables are Python strings containing first and last names. The *number* attribute is a C integer.

A estrutura do objeto é atualizada de acordo

```
typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last; /* last name */
    int number;
} CustomObject;
```

Because we now have data to manage, we have to be more careful about object allocation and deallocation. At a minimum, we need a deallocation method:

```
static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

which is assigned to the `tp_dealloc` member:

```
.tp_dealloc = (destructor) Custom_dealloc,
```

This method first clears the reference counts of the two Python attributes. `Py_XDECREF()` correctly handles the case where its argument is `NULL` (which might happen here if `tp_new` failed midway). It then calls the `tp_free` member of the object's type (computed by `Py_TYPE(self)`) to free the object's memory. Note that the object's type might not be `CustomType`, because the object may be an instance of a subclass.

Nota: The explicit cast to destructor above is needed because we defined `Custom_dealloc` to take a `CustomObject *` argument, but the `tp_dealloc` function pointer expects to receive a `PyObject *` argument. Otherwise, the compiler will emit a warning. This is object-oriented polymorphism, in C!

We want to make sure that the first and last names are initialized to empty strings, so we provide a `tp_new` implementation:

```
static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwargs)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
    }
}
```

(continua na próxima página)

(continuação da página anterior)

```

    }
    self->last = PyUnicode_FromString("");
    if (self->last == NULL) {
        Py_DECREF(self);
        return NULL;
    }
    self->number = 0;
}
return (PyObject *) self;
}

```

and install it in the `tp_new` member:

```

.tp_new = Custom_new,

```

The `tp_new` handler is responsible for creating (as opposed to initializing) objects of the type. It is exposed in Python as the `__new__()` method. It is not required to define a `tp_new` member, and indeed many extension types will simply reuse `PyType_GenericNew()` as done in the first version of the `Custom` type above. In this case, we use the `tp_new` handler to initialize the `first` and `last` attributes to non-NULL default values.

`tp_new` is passed the type being instantiated (not necessarily `CustomType`, if a subclass is instantiated) and any arguments passed when the type was called, and is expected to return the instance created. `tp_new` handlers always accept positional and keyword arguments, but they often ignore the arguments, leaving the argument handling to initializer (a.k.a. `tp_init` in C or `__init__` in Python) methods.

Nota: `tp_new` shouldn't call `tp_init` explicitly, as the interpreter will do it itself.

The `tp_new` implementation calls the `tp_alloc` slot to allocate memory:

```

self = (CustomObject *) type->tp_alloc(type, 0);

```

Since memory allocation may fail, we must check the `tp_alloc` result against `NULL` before proceeding.

Nota: We didn't fill the `tp_alloc` slot ourselves. Rather `PyType_Ready()` fills it for us by inheriting it from our base class, which is `object` by default. Most types use the default allocation strategy.

Nota: If you are creating a co-operative `tp_new` (one that calls a base type's `tp_new` or `__new__()`), you must *not* try to determine what method to call using method resolution order at runtime. Always statically determine what type you are going to call, and call its `tp_new` directly, or via `type->tp_base->tp_new`. If you do not do this, Python subclasses of your type that also inherit from other Python-defined classes may not work correctly. (Specifically, you may not be able to create instances of such subclasses without getting a `TypeError`.)

We also define an initialization function which accepts arguments to provide initial values for our instance:

```

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|OOi", kwlist,

```

(continua na próxima página)

(continuação da página anterior)

```

                                &first, &last,
                                &self->number))

    return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }
    return 0;
}

```

by filling the `tp_init` slot.

```
.tp_init = (initproc) Custom_init,
```

The `tp_init` slot is exposed in Python as the `__init__()` method. It is used to initialize an object after it's created. Initializers always accept positional and keyword arguments, and they should return either 0 on success or -1 on error.

Unlike the `tp_new` handler, there is no guarantee that `tp_init` is called at all (for example, the `pickle` module by default doesn't call `__init__()` on unpickled instances). It can also be called multiple times. Anyone can call the `__init__()` method on our objects. For this reason, we have to be extra careful when assigning the new attribute values. We might be tempted, for example to assign the `first` member like this:

```

if (first) {
    Py_XDECREF(self->first);
    Py_INCREF(first);
    self->first = first;
}

```

But this would be risky. Our type doesn't restrict the type of the `first` member, so it could be any kind of object. It could have a destructor that causes code to be executed that tries to access the `first` member; or that destructor could release the *Global interpreter Lock* and let arbitrary code run in other threads that accesses and modifies our object.

Para sermos paranoicos e nos protegemos contra essa possibilidade, quase sempre realocamos os membros antes de decrementar suas contagens de referência. Quando não temos que fazer isso?

- quando sabemos absolutamente que a contagem de referência é maior que 1;
- when we know that deallocation of the object¹ will neither release the *GIL* nor cause any calls back into our type's code;
- when decrementing a reference count in a `tp_dealloc` handler on a type which doesn't support cyclic garbage collection².

We want to expose our instance variables as attributes. There are a number of ways to do that. The simplest way is to define member definitions:

¹ Isso é verdade quando sabemos que o objeto é um tipo básico, como uma string ou um float.

² We relied on this in the `tp_dealloc` handler in this example, because our type doesn't support garbage collection.

```
static PyMemberDef Custom_members[] = {
    {"first", T_OBJECT_EX, offsetof(CustomObject, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(CustomObject, last), 0,
     "last name"},
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};
```

and put the definitions in the `tp_members` slot:

```
.tp_members = Custom_members,
```

Each member definition has a member name, type, offset, access flags and documentation string. See the [Generic Attribute Management](#) section below for details.

A disadvantage of this approach is that it doesn't provide a way to restrict the types of objects that can be assigned to the Python attributes. We expect the first and last names to be strings, but any Python objects can be assigned. Further, the attributes can be deleted, setting the C pointers to NULL. Even though we can make sure the members are initialized to non-NULL values, the members can be set to NULL if the attributes are deleted.

We define a single method, `Custom.name()`, that outputs the objects name as the concatenation of the first and last names.

```
static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}
```

The method is implemented as a C function that takes a `Custom` (or `Custom` subclass) instance as the first argument. Methods always take an instance as the first argument. Methods often take positional and keyword arguments as well, but in this case we don't take any and don't need to accept a positional argument tuple or keyword argument dictionary. This method is equivalent to the Python method:

```
def name(self):
    return "%s %s" % (self.first, self.last)
```

Note that we have to check for the possibility that our `first` and `last` members are NULL. This is because they can be deleted, in which case they are set to NULL. It would be better to prevent deletion of these attributes and to restrict the attribute values to be strings. We'll see how to do that in the next section.

Agora que definimos o método, precisamos criar uma array de definições de métodos:

```
static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    },
```

(continua na próxima página)

(continuação da página anterior)

```
{NULL} /* Sentinel */
};
```

(note that we used the METH_NOARGS flag to indicate that the method is expecting no arguments other than *self*) and assign it to the `tp_methods` slot:

```
.tp_methods = Custom_methods,
```

Finally, we'll make our type usable as a base class for subclassing. We've written our methods carefully so far so that they don't make any assumptions about the type of the object being created or used, so all we need to do is to add the `Py_TPFLAGS_BASETYPE` to our class flag definition:

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
```

We rename `PyInit_custom()` to `PyInit_custom2()`, update the module name in the `PyModuleDef` struct, and update the full class name in the `PyTypeObject` struct.

Finally, we update our `setup.py` file to build the new module:

```
from distutils.core import setup, Extension
setup(name="custom", version="1.0",
      ext_modules=[
          Extension("custom", ["custom.c"]),
          Extension("custom2", ["custom2.c"]),
      ])
```

2.2.3 Fornecendo controle mais preciso sobre atributos de dados

In this section, we'll provide finer control over how the `first` and `last` attributes are set in the `Custom` example. In the previous version of our module, the instance variables `first` and `last` could be set to non-string values or even deleted. We want to make sure that these attributes always contain strings.

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

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last; /* last name */
    int number;
} CustomObject;

static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{

```

(continua na próxima página)

(continuação da página anterior)

```

CustomObject *self;
self = (CustomObject *) type->tp_alloc(type, 0);
if (self != NULL) {
    self->first = PyUnicode_FromString("");
    if (self->first == NULL) {
        Py_DECREF(self);
        return NULL;
    }
    self->last = PyUnicode_FromString("");
    if (self->last == NULL) {
        Py_DECREF(self);
        return NULL;
    }
    self->number = 0;
}
return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwargs)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwargs, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))

        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

```

(continua na próxima página)

(continuação da página anterior)

```

static int
Custom_setfirst(CustomObject *self, PyObject *value, void *closure)
{
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
            "The first attribute value must be a string");
        return -1;
    }
    tmp = self->first;
    Py_INCREF(value);
    self->first = value;
    Py_DECREF(tmp);
    return 0;
}

static PyObject *
Custom_getlast(CustomObject *self, void *closure)
{
    Py_INCREF(self->last);
    return self->last;
}

static int
Custom_setlast(CustomObject *self, PyObject *value, void *closure)
{
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
            "The last attribute value must be a string");
        return -1;
    }
    tmp = self->last;
    Py_INCREF(value);
    self->last = value;
    Py_DECREF(tmp);
    return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))

```

(continua na próxima página)

```

{
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },
    {NULL} /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom3.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_new = Custom_new,
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
    .tp_getset = Custom_getsetters,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom3",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom3(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}

```

To provide greater control, over the `first` and `last` attributes, we'll use custom getter and setter functions. Here are the functions for getting and setting the `first` attribute:

```
static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(CustomObject *self, PyObject *value, void *closure)
{
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }
    tmp = self->first;
    Py_INCREF(value);
    self->first = value;
    Py_DECREF(tmp);
    return 0;
}
```

The getter function is passed a `Custom` object and a “closure”, which is a void pointer. In this case, the closure is ignored. (The closure supports an advanced usage in which definition data is passed to the getter and setter. This could, for example, be used to allow a single set of getter and setter functions that decide the attribute to get or set based on data in the closure.)

The setter function is passed the `Custom` object, the new value, and the closure. The new value may be `NULL`, in which case the attribute is being deleted. In our setter, we raise an error if the attribute is deleted or if its new value is not a string.

We create an array of `PyGetSetDef` structures:

```
static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL} /* Sentinel */
};
```

e registra isso num slot `tp_getset`:

```
.tp_getset = Custom_getsetters,
```

The last item in a `PyGetSetDef` structure is the “closure” mentioned above. In this case, we aren’t using a closure, so we just pass `NULL`.

Também removemos as definições de membros para esses atributos:

```
static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
};
```

(continua na próxima página)

```
{NULL} /* Sentinel */
};
```

We also need to update the `tp_init` handler to only allow strings³ to be passed:

```
static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))

        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}
```

With these changes, we can assure that the `first` and `last` members are never `NULL` so we can remove checks for `NULL` values in almost all cases. This means that most of the `Py_XDECREF()` calls can be converted to `Py_DECREF()` calls. The only place we can't change these calls is in the `tp_dealloc` implementation, where there is the possibility that the initialization of these members failed in `tp_new`.

We also rename the module initialization function and module name in the initialization function, as we did before, and we add an extra definition to the `setup.py` file.

2.2.4 Apoiando a coleta de lixo cíclica

Python has a *cyclic garbage collector (GC)* that can identify unneeded objects even when their reference counts are not zero. This can happen when objects are involved in cycles. For example, consider:

```
>>> l = []
>>> l.append(l)
>>> del l
```

In this example, we create a list that contains itself. When we delete it, it still has a reference from itself. Its reference count doesn't drop to zero. Fortunately, Python's cyclic garbage collector will eventually figure out that the list is garbage and free it.

³ Agora sabemos que o primeiro e último membros são strings, então talvez pudéssemos ter menos cuidado com a diminuição de suas contagens de referência, no entanto, aceitamos instâncias de subclasses de string. Mesmo que a desalocação de cadeias normais não retorne aos nossos objetos, não podemos garantir que a desalocação de uma instância de uma subclasse de cadeias de caracteres não retornará aos nossos objetos.

In the second version of the `Custom` example, we allowed any kind of object to be stored in the `first` or `last` attributes⁴. Besides, in the second and third versions, we allowed subclassing `Custom`, and subclasses may add arbitrary attributes. For any of those two reasons, `Custom` objects can participate in cycles:

```
>>> import custom3
>>> class Derived(custom3.Custom): pass
...
>>> n = Derived()
>>> n.some_attribute = n
```

To allow a `Custom` instance participating in a reference cycle to be properly detected and collected by the cyclic GC, our `Custom` type needs to fill two additional slots and to enable a flag that enables these slots:

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

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last; /* last name */
    int number;
} CustomObject;

static int
Custom_traverse(CustomObject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}

static int
Custom_clear(CustomObject *self)
{
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}

static void
Custom_dealloc(CustomObject *self)
{
    PyObject_GC_UnTrack(self);
    Custom_clear(self);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
    }
}
```

(continua na próxima página)

⁴ Além disso, mesmo com nossos atributos restritos a instâncias de strings, o usuário poderia passar arbitrariamente subclasses `str` e, portanto, ainda criar ciclos de referência.

(continuação da página anterior)

```

        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))

        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(CustomObject *self, PyObject *value, void *closure)
{
    if (value == NULL) {

```

(continua na próxima página)

(continuação da página anterior)

```

        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
            "The first attribute value must be a string");
        return -1;
    }
    Py_INCREF(value);
    Py_CLEAR(self->first);
    self->first = value;
    return 0;
}

static PyObject *
Custom_getlast(CustomObject *self, void *closure)
{
    Py_INCREF(self->last);
    return self->last;
}

static int
Custom_setlast(CustomObject *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
            "The last attribute value must be a string");
        return -1;
    }
    Py_INCREF(value);
    Py_CLEAR(self->last);
    self->last = value;
    return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"
    },

```

(continua na próxima página)

(continuação da página anterior)

```

{NULL} /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom4.Custom",
    .tp_doc = PyDoc_STR("Custom objects"),
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
    .tp_new = Custom_new,
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_traverse = (traverseproc) Custom_traverse,
    .tp_clear = (inquiry) Custom_clear,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
    .tp_getset = Custom_getsetters,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom4",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom4(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}

```

First, the traversal method lets the cyclic GC know about subobjects that could participate in cycles:

```

static int
Custom_traverse(CustomObject *self, visitproc visit, void *arg)
{
    int vret;
    if (self->first) {
        vret = visit(self->first, arg);
        if (vret != 0)

```

(continua na próxima página)

(continuação da página anterior)

```

        return vret;
    }
    if (self->last) {
        vret = visit(self->last, arg);
        if (vret != 0)
            return vret;
    }
    return 0;
}

```

For each subobject that can participate in cycles, we need to call the `visit()` function, which is passed to the traversal method. The `visit()` function takes as arguments the subobject and the extra argument `arg` passed to the traversal method. It returns an integer value that must be returned if it is non-zero.

Python provides a `Py_VISIT()` macro that automates calling visit functions. With `Py_VISIT()`, we can minimize the amount of boilerplate in `Custom_traverse`:

```

static int
Custom_traverse(CustomObject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}

```

Nota: The `tp_traverse` implementation must name its arguments exactly `visit` and `arg` in order to use `Py_VISIT()`.

Second, we need to provide a method for clearing any subobjects that can participate in cycles:

```

static int
Custom_clear(CustomObject *self)
{
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}

```

Notice the use of the `Py_CLEAR()` macro. It is the recommended and safe way to clear data attributes of arbitrary types while decrementing their reference counts. If you were to call `Py_XDECREF()` instead on the attribute before setting it to `NULL`, there is a possibility that the attribute's destructor would call back into code that reads the attribute again (*especially* if there is a reference cycle).

Nota: You could emulate `Py_CLEAR()` by writing:

```

PyObject *tmp;
tmp = self->first;
self->first = NULL;
Py_XDECREF(tmp);

```

Nevertheless, it is much easier and less error-prone to always use `Py_CLEAR()` when deleting an attribute. Don't try to micro-optimize at the expense of robustness!

The deallocator `Custom_dealloc` may call arbitrary code when clearing attributes. It means the circular GC can be triggered inside the function. Since the GC assumes reference count is not zero, we need to untrack the object from the GC by calling `PyObject_GC_UnTrack()` before clearing members. Here is our reimplemented deallocator using `PyObject_GC_UnTrack()` and `Custom_clear`:

```
static void
Custom_dealloc(CustomObject *self)
{
    PyObject_GC_UnTrack(self);
    Custom_clear(self);
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

Finally, we add the `Py_TPFLAGS_HAVE_GC` flag to the class flags:

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
```

That's pretty much it. If we had written custom `tp_alloc` or `tp_free` handlers, we'd need to modify them for cyclic garbage collection. Most extensions will use the versions automatically provided.

2.2.5 Criando subclasses de outros tipos

It is possible to create new extension types that are derived from existing types. It is easiest to inherit from the built in types, since an extension can easily use the `PyTypeObject` it needs. It can be difficult to share these `PyTypeObject` structures between extension modules.

In this example we will create a `SubList` type that inherits from the built-in `list` type. The new type will be completely compatible with regular lists, but will have an additional `increment()` method that increases an internal counter:

```
>>> import sublist
>>> s = sublist.SubList(range(3))
>>> s.extend(s)
>>> print(len(s))
6
>>> print(s.increment())
1
>>> print(s.increment())
2
```

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

typedef struct {
    PyListObject list;
    int state;
} SubListObject;

static PyObject *
SubList_increment(SubListObject *self, PyObject *unused)
{
    self->state++;
    return PyLong_FromLong(self->state);
}

static PyMethodDef SubList_methods[] = {
    {"increment", (PyCFunction) SubList_increment, METH_NOARGS,
```

(continua na próxima página)

(continuação da página anterior)

```

    PyDoc_STR("increment state counter"}},
    {NULL},
};

static int
SubList_init(SubListObject *self, PyObject *args, PyObject *kwds)
{
    if (PyList_Type.tp_init((PyObject *) self, args, kwds) < 0)
        return -1;
    self->state = 0;
    return 0;
}

static PyTypeObject SubListType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "sublist.SubList",
    .tp_doc = PyDoc_STR("SubList objects"),
    .tp_basicsize = sizeof(SubListObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_init = (initproc) SubList_init,
    .tp_methods = SubList_methods,
};

static PyModuleDef sublistmodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "sublist",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_sublist(void)
{
    PyObject *m;
    SubListType.tp_base = &PyList_Type;
    if (PyType_Ready(&SubListType) < 0)
        return NULL;

    m = PyModule_Create(&sublistmodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&SubListType);
    if (PyModule_AddObject(m, "SubList", (PyObject *) &SubListType) < 0) {
        Py_DECREF(&SubListType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}

```

As you can see, the source code closely resembles the Custom examples in previous sections. We will break down the main differences between them.

```
typedef struct {
    PyListObject list;
    int state;
} SubListObject;
```

The primary difference for derived type objects is that the base type's object structure must be the first value. The base type will already include the `PyObject_HEAD()` at the beginning of its structure.

When a Python object is a `SubList` instance, its `PyObject *` pointer can be safely cast to both `PyListObject *` and `SubListObject *`:

```
static int
SubList_init(SubListObject *self, PyObject *args, PyObject *kwds)
{
    if (PyList_Type.tp_init((PyObject *) self, args, kwds) < 0)
        return -1;
    self->state = 0;
    return 0;
}
```

We see above how to call through to the `__init__` method of the base type.

This pattern is important when writing a type with custom `tp_new` and `tp_dealloc` members. The `tp_new` handler should not actually create the memory for the object with its `tp_alloc`, but let the base class handle it by calling its own `tp_new`.

The `PyTypeObject` struct supports a `tp_base` specifying the type's concrete base class. Due to cross-platform compiler issues, you can't fill that field directly with a reference to `PyList_Type`; it should be done later in the module initialization function:

```
PyMODINIT_FUNC
PyInit_sublist(void)
{
    PyObject* m;
    SubListType.tp_base = &PyList_Type;
    if (PyType_Ready(&SubListType) < 0)
        return NULL;

    m = PyModule_Create(&sublistmodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&SubListType);
    if (PyModule_AddObject(m, "SubList", (PyObject *) &SubListType) < 0) {
        Py_DECREF(&SubListType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

Before calling `PyType_Ready()`, the type structure must have the `tp_base` slot filled in. When we are deriving an existing type, it is not necessary to fill out the `tp_alloc` slot with `PyType_GenericNew()` – the allocation function from the base type will be inherited.

After that, calling `PyType_Ready()` and adding the type object to the module is the same as with the basic `Custom` examples.

2.3 Defining Extension Types: Assorted Topics

This section aims to give a quick fly-by on the various type methods you can implement and what they do.

Here is the definition of `PyObject`, with some fields only used in debug builds omitted:

```
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 */
    getiterfunc tp_iter;
```

(continua na próxima página)

(continuação da página anterior)

```

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;

```

Now that's a *lot* of methods. Don't worry too much though – if you have a type you want to define, the chances are very good that you will only implement a handful of these.

As you probably expect by now, we're going to go over this and give more information about the various handlers. We won't go in the order they are defined in the structure, because there is a lot of historical baggage that impacts the ordering of the fields. It's often easiest to find an example that includes the fields you need and then change the values to suit your new type.

```

const char *tp_name; /* For printing */

```

The name of the type – as mentioned in the previous chapter, this will appear in various places, almost entirely for diagnostic purposes. Try to choose something that will be helpful in such a situation!

```

Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

```

These fields tell the runtime how much memory to allocate when new objects of this type are created. Python has some built-in support for variable length structures (think: strings, tuples) which is where the `tp_itemsize` field comes in. This will be dealt with later.

```

const char *tp_doc;

```

Here you can put a string (or its address) that you want returned when the Python script references `obj.__doc__` to retrieve the doc string.

Now we come to the basic type methods – the ones most extension types will implement.

2.3.1 Finalization and De-allocation

```
destructor tp_dealloc;
```

This function is called when the reference count of the instance of your type is reduced to zero and the Python interpreter wants to reclaim it. If your type has memory to free or other clean-up to perform, you can put it here. The object itself needs to be freed here as well. Here is an example of this function:

```
static void
newdatatype_dealloc(newdatatypeobject *obj)
{
    free(obj->obj_UnderlyingDatatypePtr);
    Py_TYPE(obj)->tp_free((PyObject *)obj);
}
```

If your type supports garbage collection, the destructor should call `PyObject_GC_UnTrack()` before clearing any member fields:

```
static void
newdatatype_dealloc(newdatatypeobject *obj)
{
    PyObject_GC_UnTrack(obj);
    Py_CLEAR(obj->other_obj);
    ...
    Py_TYPE(obj)->tp_free((PyObject *)obj);
}
```

One important requirement of the deallocator function is that it leaves any pending exceptions alone. This is important since deallocators are frequently called as the interpreter unwinds the Python stack; when the stack is unwound due to an exception (rather than normal returns), nothing is done to protect the deallocators from seeing that an exception has already been set. Any actions which a deallocator performs which may cause additional Python code to be executed may detect that an exception has been set. This can lead to misleading errors from the interpreter. The proper way to protect against this is to save a pending exception before performing the unsafe action, and restoring it when done. This can be done using the `PyErr_Fetch()` and `PyErr_Restore()` functions:

```
static void
my_dealloc(PyObject *obj)
{
    PyObject *self = (PyObject *) obj;
    PyObject *cbresult;

    if (self->my_callback != NULL) {
        PyObject *err_type, *err_value, *err_traceback;

        /* This saves the current exception state */
        PyErr_Fetch(&err_type, &err_value, &err_traceback);

        cbresult = PyObject_CallNoArgs(self->my_callback);
        if (cbresult == NULL)
            PyErr_WriteUnraisable(self->my_callback);
        else
            Py_DECREF(cbresult);

        /* This restores the saved exception state */
        PyErr_Restore(err_type, err_value, err_traceback);

        Py_DECREF(self->my_callback);
    }
}
```

(continua na próxima página)

(continuação da página anterior)

```

    }
    Py_TYPE(obj) -> tp_free (PyObject*) self);
}

```

Nota: There are limitations to what you can safely do in a deallocator function. First, if your type supports garbage collection (using `tp_traverse` and/or `tp_clear`), some of the object's members can have been cleared or finalized by the time `tp_dealloc` is called. Second, in `tp_dealloc`, your object is in an unstable state: its reference count is equal to zero. Any call to a non-trivial object or API (as in the example above) might end up calling `tp_dealloc` again, causing a double free and a crash.

Starting with Python 3.4, it is recommended not to put any complex finalization code in `tp_dealloc`, and instead use the new `tp_finalize` type method.

Ver também:

[PEP 442](#) explains the new finalization scheme.

2.3.2 Object Presentation

In Python, there are two ways to generate a textual representation of an object: the `repr()` function, and the `str()` function. (The `print()` function just calls `str()`.) These handlers are both optional.

```

reprfunc tp_repr;
reprfunc tp_str;

```

The `tp_repr` handler should return a string object containing a representation of the instance for which it is called. Here is a simple example:

```

static PyObject *
newdatatype_repr(newdatatypeobject * obj)
{
    return PyUnicode_FromFormat("Repr-ified_newdatatype{{size:%d}}",
                                obj->obj_UnderlyingDatatypePtr->size);
}

```

If no `tp_repr` handler is specified, the interpreter will supply a representation that uses the type's `tp_name` and a uniquely-identifying value for the object.

The `tp_str` handler is to `str()` what the `tp_repr` handler described above is to `repr()`; that is, it is called when Python code calls `str()` on an instance of your object. Its implementation is very similar to the `tp_repr` function, but the resulting string is intended for human consumption. If `tp_str` is not specified, the `tp_repr` handler is used instead.

Here is a simple example:

```

static PyObject *
newdatatype_str(newdatatypeobject * obj)
{
    return PyUnicode_FromFormat("Stringified_newdatatype{{size:%d}}",
                                obj->obj_UnderlyingDatatypePtr->size);
}

```


2.3.3 Attribute Management

For every object which can support attributes, the corresponding type must provide the functions that control how the attributes are resolved. There needs to be a function which can retrieve attributes (if any are defined), and another to set attributes (if setting attributes is allowed). Removing an attribute is a special case, for which the new value passed to the handler is `NULL`.

Python supports two pairs of attribute handlers; a type that supports attributes only needs to implement the functions for one pair. The difference is that one pair takes the name of the attribute as a `char*`, while the other accepts a `PyObject*`. Each type can use whichever pair makes more sense for the implementation's convenience.

```
getattrofunc  tp_getattr;          /* char * version */
setattrofunc  tp_setattr;
/* ... */
getattrofunc  tp_getattro;        /* PyObject * version */
setattrofunc  tp_setattro;
```

If accessing attributes of an object is always a simple operation (this will be explained shortly), there are generic implementations which can be used to provide the `PyObject*` version of the attribute management functions. The actual need for type-specific attribute handlers almost completely disappeared starting with Python 2.2, though there are many examples which have not been updated to use some of the new generic mechanism that is available.

Generic Attribute Management

Most extension types only use *simple* attributes. So, what makes the attributes simple? There are only a couple of conditions that must be met:

1. The name of the attributes must be known when `PyType_Ready()` is called.
2. No special processing is needed to record that an attribute was looked up or set, nor do actions need to be taken based on the value.

Note that this list does not place any restrictions on the values of the attributes, when the values are computed, or how relevant data is stored.

When `PyType_Ready()` is called, it uses three tables referenced by the type object to create *descriptors* which are placed in the dictionary of the type object. Each descriptor controls access to one attribute of the instance object. Each of the tables is optional; if all three are `NULL`, instances of the type will only have attributes that are inherited from their base type, and should leave the `tp_getattro` and `tp_setattro` fields `NULL` as well, allowing the base type to handle attributes.

The tables are declared as three fields of the type object:

```
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
```

If `tp_methods` is not `NULL`, it must refer to an array of `PyMethodDef` structures. Each entry in the table is an instance of this structure:

```
typedef struct PyMethodDef {
    const char *ml_name;          /* method name */
    PyCFunction ml_meth;          /* implementation function */
    int ml_flags;                 /* flags */
    const char *ml_doc;           /* docstring */
} PyMethodDef;
```

One entry should be defined for each method provided by the type; no entries are needed for methods inherited from a base type. One additional entry is needed at the end; it is a sentinel that marks the end of the array. The `ml_name` field of the sentinel must be `NULL`.

The second table is used to define attributes which map directly to data stored in the instance. A variety of primitive C types are supported, and access may be read-only or read-write. The structures in the table are defined as:

```
typedef struct PyMemberDef {
    const char *name;
    int type;
    int offset;
    int flags;
    const char *doc;
} PyMemberDef;
```

For each entry in the table, a *descriptor* will be constructed and added to the type which will be able to extract a value from the instance structure. The `type` field should contain one of the type codes defined in the `structmember.h` header; the value will be used to determine how to convert Python values to and from C values. The `flags` field is used to store flags which control how the attribute can be accessed.

The following flag constants are defined in `structmember.h`; they may be combined using bitwise-OR.

Constante	Significado
READONLY	Never writable.
READ_RESTRICTED	Not readable in restricted mode.
WRITE_RESTRICTED	Not writable in restricted mode.
RESTRICTED	Not readable or writable in restricted mode.

An interesting advantage of using the `tp_members` table to build descriptors that are used at runtime is that any attribute defined this way can have an associated doc string simply by providing the text in the table. An application can use the introspection API to retrieve the descriptor from the class object, and get the doc string using its `__doc__` attribute.

As with the `tp_methods` table, a sentinel entry with a `name` value of `NULL` is required.

Type-specific Attribute Management

For simplicity, only the `char*` version will be demonstrated here; the type of the `name` parameter is the only difference between the `char*` and `PyObject*` flavors of the interface. This example effectively does the same thing as the generic example above, but does not use the generic support added in Python 2.2. It explains how the handler functions are called, so that if you do need to extend their functionality, you'll understand what needs to be done.

The `tp_getattr` handler is called when the object requires an attribute look-up. It is called in the same situations where the `__getattr__()` method of a class would be called.

Aqui está um exemplo:

```
static PyObject *
newdatatype_getattr(newdatatypeobject *obj, char *name)
{
    if (strcmp(name, "data") == 0)
    {
        return PyLong_FromLong(obj->data);
    }

    PyErr_Format(PyExc_AttributeError,
        "%50s' object has no attribute '%.400s'",

```

(continua na próxima página)

(continuação da página anterior)

```

        tp->tp_name, name);
    return NULL;
}
    
```

The `tp_setattr` handler is called when the `__setattr__()` or `__delattr__()` method of a class instance would be called. When an attribute should be deleted, the third parameter will be `NULL`. Here is an example that simply raises an exception; if this were really all you wanted, the `tp_setattr` handler should be set to `NULL`.

```

static int
newdatatype_setattr(newdatatypeobject *obj, char *name, PyObject *v)
{
    PyErr_Format(PyExc_RuntimeError, "Read-only attribute: %s", name);
    return -1;
}
    
```

2.3.4 Object Comparison

```
richcmpfunc tp_richcompare;
```

The `tp_richcompare` handler is called when comparisons are needed. It is analogous to the rich comparison methods, like `__lt__()`, and also called by `PyObject_RichCompare()` and `PyObject_RichCompareBool()`.

This function is called with two Python objects and the operator as arguments, where the operator is one of `Py_EQ`, `Py_NE`, `Py_LE`, `Py_GE`, `Py_LT` or `Py_GT`. It should compare the two objects with respect to the specified operator and return `Py_True` or `Py_False` if the comparison is successful, `Py_NotImplemented` to indicate that comparison is not implemented and the other object's comparison method should be tried, or `NULL` if an exception was set.

Here is a sample implementation, for a datatype that is considered equal if the size of an internal pointer is equal:

```

static PyObject *
newdatatype_richcmp(PyObject *obj1, PyObject *obj2, int op)
{
    PyObject *result;
    int c, size1, size2;

    /* code to make sure that both arguments are of type
       newdatatype omitted */

    size1 = obj1->obj_UnderlyingDatatypePtr->size;
    size2 = obj2->obj_UnderlyingDatatypePtr->size;

    switch (op) {
        case Py_LT: c = size1 < size2; break;
        case Py_LE: c = size1 <= size2; break;
        case Py_EQ: c = size1 == size2; break;
        case Py_NE: c = size1 != size2; break;
        case Py_GT: c = size1 > size2; break;
        case Py_GE: c = size1 >= size2; break;
    }
    result = c ? Py_True : Py_False;
    Py_INCREF(result);
    return result;
}
    
```

2.3.5 Abstract Protocol Support

Python supports a variety of *abstract* ‘protocols;’ the specific interfaces provided to use these interfaces are documented in abstract.

A number of these abstract interfaces were defined early in the development of the Python implementation. In particular, the number, mapping, and sequence protocols have been part of Python since the beginning. Other protocols have been added over time. For protocols which depend on several handler routines from the type implementation, the older protocols have been defined as optional blocks of handlers referenced by the type object. For newer protocols there are additional slots in the main type object, with a flag bit being set to indicate that the slots are present and should be checked by the interpreter. (The flag bit does not indicate that the slot values are non-NULL. The flag may be set to indicate the presence of a slot, but a slot may still be unfilled.)

```
PyNumberMethods    *tp_as_number;
PySequenceMethods  *tp_as_sequence;
PyMappingMethods    *tp_as_mapping;
```

If you wish your object to be able to act like a number, a sequence, or a mapping object, then you place the address of a structure that implements the C type `PyNumberMethods`, `PySequenceMethods`, or `PyMappingMethods`, respectively. It is up to you to fill in this structure with appropriate values. You can find examples of the use of each of these in the `Objects` directory of the Python source distribution.

```
hashfunc tp_hash;
```

This function, if you choose to provide it, should return a hash number for an instance of your data type. Here is a simple example:

```
static Py_hash_t
newdatatype_hash(newdatatypeobject *obj)
{
    Py_hash_t result;
    result = obj->some_size + 32767 * obj->some_number;
    if (result == -1)
        result = -2;
    return result;
}
```

`Py_hash_t` is a signed integer type with a platform-varying width. Returning `-1` from `tp_hash` indicates an error, which is why you should be careful to avoid returning it when hash computation is successful, as seen above.

```
ternaryfunc tp_call;
```

This function is called when an instance of your data type is “called”, for example, if `obj1` is an instance of your data type and the Python script contains `obj1('hello')`, the `tp_call` handler is invoked.

This function takes three arguments:

1. *self* is the instance of the data type which is the subject of the call. If the call is `obj1('hello')`, then *self* is `obj1`.
2. *args* is a tuple containing the arguments to the call. You can use `PyArg_ParseTuple()` to extract the arguments.
3. *kws* is a dictionary of keyword arguments that were passed. If this is non-NULL and you support keyword arguments, use `PyArg_ParseTupleAndKeywords()` to extract the arguments. If you do not want to support keyword arguments and this is non-NULL, raise a `TypeError` with a message saying that keyword arguments are not supported.

Here is a toy `tp_call` implementation:

```
static PyObject *
newdatatype_call(newdatatypeobject *self, PyObject *args, PyObject *kwargs)
{
    PyObject *result;
    const char *arg1;
    const char *arg2;
    const char *arg3;

    if (!PyArg_ParseTuple(args, "sss:call", &arg1, &arg2, &arg3)) {
        return NULL;
    }
    result = PyUnicode_FromFormat(
        "Returning -- value: [%d] arg1: [%s] arg2: [%s] arg3: [%s]\n",
        obj->obj_UnderlyingDatatypePtr->size,
        arg1, arg2, arg3);
    return result;
}
```

```
/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;
```

These functions provide support for the iterator protocol. Both handlers take exactly one parameter, the instance for which they are being called, and return a new reference. In the case of an error, they should set an exception and return NULL. `tp_iter` corresponds to the Python `__iter__()` method, while `tp_iternext` corresponds to the Python `__next__()` method.

Any *iterable* object must implement the `tp_iter` handler, which must return an *iterator* object. Here the same guidelines apply as for Python classes:

- For collections (such as lists and tuples) which can support multiple independent iterators, a new iterator should be created and returned by each call to `tp_iter`.
- Objects which can only be iterated over once (usually due to side effects of iteration, such as file objects) can implement `tp_iter` by returning a new reference to themselves – and should also therefore implement the `tp_iternext` handler.

Any *iterator* object should implement both `tp_iter` and `tp_iternext`. An iterator's `tp_iter` handler should return a new reference to the iterator. Its `tp_iternext` handler should return a new reference to the next object in the iteration, if there is one. If the iteration has reached the end, `tp_iternext` may return NULL without setting an exception, or it may set `StopIteration` *in addition* to returning NULL; avoiding the exception can yield slightly better performance. If an actual error occurs, `tp_iternext` should always set an exception and return NULL.

2.3.6 Weak Reference Support

One of the goals of Python's weak reference implementation is to allow any type to participate in the weak reference mechanism without incurring the overhead on performance-critical objects (such as numbers).

Ver também:

Documentação do módulo `weakref`.

For an object to be weakly referencable, the extension type must do two things:

1. Include a `PyObject*` field in the C object structure dedicated to the weak reference mechanism. The object's constructor should leave it NULL (which is automatic when using the default `tp_alloc`).

2. Set the `tp_weaklistoffset` type member to the offset of the aforementioned field in the C object structure, so that the interpreter knows how to access and modify that field.

Concretely, here is how a trivial object structure would be augmented with the required field:

```
typedef struct {
    PyObject_HEAD
    PyObject *weakreflist; /* List of weak references */
} TrivialObject;
```

And the corresponding member in the statically-declared type object:

```
static PyTypeObject TrivialType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    /* ... other members omitted for brevity ... */
    .tp_weaklistoffset = offsetof(TrivialObject, weakreflist),
};
```

The only further addition is that `tp_dealloc` needs to clear any weak references (by calling `PyObject_ClearWeakRefs()`) if the field is non-NULL:

```
static void
Trivial_dealloc(TrivialObject *self)
{
    /* Clear weakrefs first before calling any destructors */
    if (self->weakreflist != NULL)
        PyObject_ClearWeakRefs((PyObject *) self);
    /* ... remainder of destruction code omitted for brevity ... */
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

2.3.7 More Suggestions

In order to learn how to implement any specific method for your new data type, get the *CPython* source code. Go to the `Objects` directory, then search the C source files for `tp_` plus the function you want (for example, `tp_richcompare`). You will find examples of the function you want to implement.

When you need to verify that an object is a concrete instance of the type you are implementing, use the `PyObject_TypeCheck()` function. A sample of its use might be something like the following:

```
if (!PyObject_TypeCheck(some_object, &MyType)) {
    PyErr_SetString(PyExc_TypeError, "arg #1 not a mything");
    return NULL;
}
```

Ver também:

Download CPython source releases. <https://www.python.org/downloads/source/>

The CPython project on GitHub, where the CPython source code is developed. <https://github.com/python/cpython>

2.4 Construindo extensões C e C++

Uma extensão C para CPython é uma biblioteca compartilhada (por exemplo, um arquivo `.so` no Linux, `.pyd` no Windows), que exporta uma *função de inicialização*.

Para ser importável, a biblioteca compartilhada deve estar disponível em `PYTHONPATH`, e deve ser nomeada após o nome do módulo, com uma extensão apropriada. Ao usar `distutils`, o nome do arquivo correto é gerado automaticamente.

A função de inicialização tem a assinatura:

`PyObject* PyInit_modulename (void)`

Ela retorna um módulo totalmente inicializado ou uma instância de `PyModuleDef`. Veja `initializing-modules` para detalhes.

Para módulos com nomes somente ASCII, a função deve ser nomeada `PyInit_<nomemódulo>`, com `<nomemódulo>` substituído pelo nome do módulo. Ao usar `multi-phase-initialization`, nomes de módulos não ASCII são permitidos. Neste caso, o nome da função de inicialização é `PyInitU_<nomemódulo>`, com `<nomemódulo>` codificado usando a codificação *punycode* do Python com hífenes substituídos por sublinhados. Em Python:

```
def initfunc_name(name):
    try:
        suffix = b'_' + name.encode('ascii')
    except UnicodeEncodeError:
        suffix = b'U_' + name.encode('punycode').replace(b'-', b'_')
    return b'PyInit' + suffix
```

É possível exportar vários módulos de uma única biblioteca compartilhada, definindo várias funções de inicialização. No entanto, importá-los requer o uso de links simbólicos ou um importador personalizado, porque por padrão apenas a função correspondente ao nome do arquivo é encontrada. Veja a seção “*Multiple modules in one library*” na [PEP 489](#) para detalhes.

2.4.1 Construindo extensões C e C++ com distutils

Módulos de extensão podem ser construídos usando `distutils`, que está incluído no Python. Visto que `distutils` também suporta a criação de pacotes binários, os usuários não precisam necessariamente de um compilador e `distutils` para instalar a extensão.

Um pacote `distutils` contém um script guia `setup.py`. Este é um arquivo Python simples, que, no caso mais simples, pode ter a seguinte aparência:

```
from distutils.core import setup, Extension

module1 = Extension('demo',
                    sources = ['demo.c'])

setup (name = 'PackageName',
      version = '1.0',
      description = 'This is a demo package',
      ext_modules = [module1])
```

Com este `setup.py` e um arquivo `demo.c`, executando

```
python setup.py build
```

vai compilar `demo.c`, e produzir um módulo de extensão chamado `demo` no diretório `build`. Dependendo do sistema, o arquivo do módulo vai terminar em um subdiretório `build/lib.system`, e pode ter um nome como `demo.so` ou `demo.pyd`.

No `setup.py`, toda a execução é realizada chamando a função `setup`. Isso leva um número variável de argumentos nomeados, dos quais o exemplo acima usa apenas um subconjunto. Especificamente, o exemplo especifica meta-informação para construir pacotes e especifica o conteúdo do pacote. Normalmente, um pacote conterá módulos adicionais, como módulos-fonte Python, documentação, subpacotes etc. Consulte a documentação do `distutils` em `distutils-index` para aprender mais sobre os recursos do `distutils`; esta seção explica apenas a construção de módulos de extensão.

É comum pré-computar argumentos para `setup()`, para melhor estruturar o script guia. No exemplo acima, o argumento `ext_modules` para `setup()` é uma lista de módulos de extensão, cada um dos quais é uma instância de `Extension`. No exemplo, a instância define uma extensão chamada `demo` que é construída compilando um único arquivo fonte, `demo.c`.

Em muitos casos, construir uma extensão é mais complexo, uma vez que definições de pré-processador adicionais e bibliotecas podem ser necessárias. Isso é demonstrado no exemplo abaixo.

```
from distutils.core import setup, Extension

module1 = Extension('demo',
                    define_macros = [('MAJOR_VERSION', '1'),
                                     ('MINOR_VERSION', '0')],
                    include_dirs = ['/usr/local/include'],
                    libraries = ['tcl83'],
                    library_dirs = ['/usr/local/lib'],
                    sources = ['demo.c'])

setup (name = 'PackageName',
      version = '1.0',
      description = 'This is a demo package',
      author = 'Martin v. Loewis',
      author_email = 'martin@v.loewis.de',
      url = 'https://docs.python.org/extending/building',
      long_description = '''
This is really just a demo package.
''',
      ext_modules = [module1])
```

Neste exemplo, `setup()` é chamado com meta-informação adicional, o que é recomendado quando os pacotes de distribuição precisam ser construídos. Para a extensão em si, especifica definições de pré-processador, diretórios de inclusão, diretórios de biblioteca e bibliotecas. Dependendo do compilador, `distutils` passa essas informações de maneiras diferentes para o compilador. Por exemplo, no Unix, isso pode resultar nos comandos de compilação

```
gcc -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -fPIC -DMAJOR_VERSION=1 -DMINOR_
↪VERSION=0 -I/usr/local/include -I/usr/local/include/python2.2 -c demo.c -o build/
↪temp.linux-i686-2.2/demo.o

gcc -shared build/temp.linux-i686-2.2/demo.o -L/usr/local/lib -ltcl83 -o build/lib.
↪linux-i686-2.2/demo.so
```

Essas linhas são apenas para fins de demonstração; Os usuários de `distutils` devem confiar que `distutils` acertará as invocações.

2.4.2 Distribuindo seus módulos de extensão

Quando uma extensão é construída com sucesso, há três maneiras de usá-la.

Os usuários finais normalmente desejam instalar o módulo, eles fazem isso executando

```
python setup.py install
```

Os mantenedores do módulo devem produzir pacotes fonte; para fazer isso, eles executam:

```
python setup.py sdist
```

Em alguns casos, arquivos adicionais precisam ser incluídos em uma distribuição fonte; isso é feito através de um arquivo `MANIFEST.in`; veja `manifest` para detalhes.

Se a distribuição fonte foi construída com sucesso, os mantenedores também podem criar distribuições binárias. Dependendo da plataforma, um dos seguintes comandos pode ser usado para fazer isso.

```
python setup.py bdist_wininst
python setup.py bdist_rpm
python setup.py bdist_dumb
```

2.5 Construindo Extensões C e C++ no Windows

Este capítulo explica brevemente como criar um módulo de extensão do Windows para Python usando o Microsoft Visual C++ e segue com informações mais detalhadas sobre como ele funciona. O material explicativo é útil para o programador do Windows aprender a construir extensões Python e o programador Unix interessado em produzir software que possa ser construído com sucesso no Unix e no Windows.

Os autores de módulos são encorajados a usar a abordagem `distutils` para construir módulos de extensão, em vez daquele descrito nesta seção. Você ainda precisará do compilador C que foi usado para construir o Python; normalmente o Microsoft Visual C++.

Nota: Este capítulo menciona vários nomes de arquivos que incluem um número de versão do Python codificado. Esses nomes de arquivos são representados com o número da versão mostrado como `XY`; na prática, '`X`' será o número da versão principal e '`Y`' será o número da versão secundária da versão do Python com a qual você está trabalhando. Por exemplo, se você estiver usando o Python 2.2.1, `XY` será `22`.

2.5.1 Uma abordagem de livro de receitas

Existem duas abordagens para construir módulos de extensão no Windows, assim como no Unix: use o pacote `distutils` para controlar o processo de construção ou faça as coisas manualmente. A abordagem `distutils` funciona bem para a maioria das extensões; documentação sobre o uso de `distutils` para construir e empacotar módulos de extensão está disponível em `distutils-index`. Se você achar que realmente precisa fazer as coisas manualmente, pode ser instrutivo estudar o arquivo do projeto para o módulo de biblioteca padrão `winsound`.

2.5.2 Diferenças entre o Unix e o Windows

O Unix e o Windows usam paradigmas completamente diferentes para o carregamento do código em tempo de execução. Antes de tentar construir um módulo que possa ser carregado dinamicamente, esteja ciente de como o seu sistema funciona.

No Unix, um arquivo de objeto compartilhado (.so) contém código a ser usado pelo programa e também os nomes de funções e dados que ele espera encontrar no programa. Quando o arquivo é associado ao programa, todas as referências a essas funções e dados no código do arquivo são alteradas para apontar para os locais reais no programa em que as funções e os dados são colocados na memória. Isso é basicamente uma operação de vinculação.

No Windows, um arquivo de biblioteca de vínculo dinâmico (.dll) não possui referências pendentes. Em vez disso, um acesso a funções ou dados passa por uma tabela de pesquisa. Portanto, o código DLL não precisa ser corrigido no tempo de execução para se referir à memória do programa; em vez disso, o código já usa a tabela de pesquisa da DLL e a tabela de pesquisa é modificada em tempo de execução para apontar para as funções e dados.

No Unix, existe apenas um tipo de arquivo de biblioteca (.a) que contém código de vários arquivos de objetos (.o). Durante a etapa da vinculação para criar um arquivo de objeto compartilhado (.so), o vinculador pode achar que não sabe onde um identificador está definido. O vinculador procurará nos arquivos de objeto nas bibliotecas; se encontrar, incluirá todo o código desse arquivo de objeto.

No Windows, existem dois tipos de biblioteca, uma biblioteca estática e uma biblioteca de importação (ambas chamadas .lib). Uma biblioteca estática é como um arquivo Unix .a; contém código a ser incluído conforme necessário. Uma biblioteca de importação é basicamente usada apenas para garantir ao vinculador que um determinado identificador é legal e estará presente no programa quando a DLL for carregada. Portanto, o vinculador usa as informações da biblioteca de importação para construir a tabela de pesquisa para o uso de identificadores que não estão incluídos na DLL. Quando uma aplicação ou uma DLL é vinculado, pode ser gerada uma biblioteca de importação, que precisará ser usada para todas as DLLs futuras que dependem dos símbolos na aplicação ou DLL.

Suponha que você esteja construindo dois módulos de carregamento dinâmico, B e C, que devem compartilhar outro bloco de código A. No Unix, você *não* passaria A.a ao ligador para B.so e C.so; isso faria com que fosse incluído duas vezes, para que B e C tivessem sua própria cópia. No Windows, a construção A.dll também construirá A.lib. Você *passa* A.lib ao ligador para B e C. A.lib não contém código; apenas contém informações que serão usadas em tempo de execução para acessar o código de A.

No Windows, usar uma biblioteca de importação é como usar `import spam`; fornece acesso aos nomes de spam, mas não cria uma cópia separada. No Unix, vincular a uma biblioteca é mais como `from spam import *`; ele cria uma cópia separada.

2.5.3 Usando DLLs na prática

O Python para Windows é criado no Microsoft Visual C++; o uso de outros compiladores pode ou não funcionar. O restante desta seção é específico do MSVC++.

Ao criar DLLs no Windows, você deve passar `pythonXY.lib` para o ligador. Para construir duas DLLs, `spam` e `ni` (que usa funções C encontradas em `spam`), você pode usar estes comandos:

```
cl /LD /I/python/include spam.c ../libs/pythonXY.lib
cl /LD /I/python/include ni.c spam.lib ../libs/pythonXY.lib
```

O primeiro comando criou três arquivos: `spam.obj`, `spam.dll` e `spam.lib`. O `spam.dll` não contém nenhuma função Python (como `PyArg_ParseTuple()`), mas sabe como encontrar o código Python graças a `pythonXY.lib`.

O segundo comando criou `ni.dll` (e `.obj` e `.lib`), que sabe como encontrar as funções necessárias do `spam` e também do executável do Python.

Nem todo identificador é exportado para a tabela de pesquisa. Se você deseja que outros módulos (incluindo Python) possam ver seus identificadores, é necessário dizer `_declspec(dllexport)`, como em `void _declspec(dllexport) initspam(void)` ou `PyObject _declspec(dllexport) *NiGetSpamData(void)`.

O Developer Studio jogará muitas bibliotecas de importação que você realmente não precisa, adicionando cerca de 100K ao seu executável. Para se livrar delas, use a caixa de diálogo Project Settings, aba Link, para especificar *ignore default libraries*. Adicione o `msvcrtxx.lib` correto à lista de bibliotecas.

Incorporando o tempo de execução do CPython em uma aplicação maior

Às vezes, em vez de criar uma extensão que é executada dentro do interpretador Python como a aplicação principal, é desejável incorporar o tempo de execução do CPython em uma aplicação maior. Esta seção aborda alguns dos detalhes envolvidos para fazer isso com êxito.

3.1 Incorporando o Python numa Outra Aplicação

Os capítulos anteriores discutiram como estender o Python, ou seja, como expandir a funcionalidade do Python anexando uma biblioteca de funções em C a ele. Também é possível fazer o inverso: enriquecer sua aplicação em C/C++ incorporando o Python nela. A incorporação fornece à sua aplicação a capacidade de implementar parte da funcionalidade da aplicação em Python em vez de C ou C++. Isso pode ser usado para diversos propósitos; um exemplo seria permitir que os usuários personalizem a aplicação de acordo com suas necessidades escrevendo alguns scripts em Python. Você também pode usá-la se parte da funcionalidade puder ser escrita em Python mais facilmente.

Incorporar o Python é semelhante a estendê-lo, mas não exatamente. A diferença é que, ao estender o Python, o programa principal da aplicação ainda é o interpretador Python, enquanto que, se você incorporar o Python, o programa principal pode não ter nada a ver com o Python — em vez disso, algumas partes da aplicação chamam ocasionalmente o interpretador Python para executar algum código Python.

So if you are embedding Python, you are providing your own main program. One of the things this main program has to do is initialize the Python interpreter. At the very least, you have to call the function `Py_Initialize()`. There are optional calls to pass command line arguments to Python. Then later you can call the interpreter from any part of the application.

There are several different ways to call the interpreter: you can pass a string containing Python statements to `PyRun_SimpleString()`, or you can pass a stdio file pointer and a file name (for identification in error messages only) to `PyRun_SimpleFile()`. You can also call the lower-level operations described in the previous chapters to construct and use Python objects.

Ver também:

c-api-index The details of Python's C interface are given in this manual. A great deal of necessary information can be found here.

3.1.1 Very High Level Embedding

The simplest form of embedding Python is the use of the very high level interface. This interface is intended to execute a Python script without needing to interact with the application directly. This can for example be used to perform some operation on a file.

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

int
main(int argc, char *argv[])
{
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }
    Py_SetProgramName(program); /* optional but recommended */
    Py_Initialize();
    PyRun_SimpleString("from time import time,ctime\n"
                      "print('Today is', ctime(time()))\n");
    if (Py_FinalizeEx() < 0) {
        exit(120);
    }
    PyMem_RawFree(program);
    return 0;
}
```

The `Py_SetProgramName()` function should be called before `Py_Initialize()` to inform the interpreter about paths to Python run-time libraries. Next, the Python interpreter is initialized with `Py_Initialize()`, followed by the execution of a hard-coded Python script that prints the date and time. Afterwards, the `Py_FinalizeEx()` call shuts the interpreter down, followed by the end of the program. In a real program, you may want to get the Python script from another source, perhaps a text-editor routine, a file, or a database. Getting the Python code from a file can better be done by using the `PyRun_SimpleFile()` function, which saves you the trouble of allocating memory space and loading the file contents.

3.1.2 Beyond Very High Level Embedding: An overview

The high level interface gives you the ability to execute arbitrary pieces of Python code from your application, but exchanging data values is quite cumbersome to say the least. If you want that, you should use lower level calls. At the cost of having to write more C code, you can achieve almost anything.

It should be noted that extending Python and embedding Python is quite the same activity, despite the different intent. Most topics discussed in the previous chapters are still valid. To show this, consider what the extension code from Python to C really does:

1. Convert data values from Python to C,
2. Perform a function call to a C routine using the converted values, and
3. Convert the data values from the call from C to Python.

When embedding Python, the interface code does:

1. Convert data values from C to Python,
2. Perform a function call to a Python interface routine using the converted values, and
3. Convert the data values from the call from Python to C.

As you can see, the data conversion steps are simply swapped to accommodate the different direction of the cross-language transfer. The only difference is the routine that you call between both data conversions. When extending, you call a C routine, when embedding, you call a Python routine.

This chapter will not discuss how to convert data from Python to C and vice versa. Also, proper use of references and dealing with errors is assumed to be understood. Since these aspects do not differ from extending the interpreter, you can refer to earlier chapters for the required information.

3.1.3 Pure Embedding

The first program aims to execute a function in a Python script. Like in the section about the very high level interface, the Python interpreter does not directly interact with the application (but that will change in the next section).

The code to run a function defined in a Python script is:

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

int
main(int argc, char *argv[])
{
    PyObject *pName, *pModule, *pFunc;
    PyObject *pArgs, *pValue;
    int i;

    if (argc < 3) {
        fprintf(stderr, "Usage: call pythonfile funcname [args]\n");
        return 1;
    }

    Py_Initialize();
    pName = PyUnicode_DecodeFSDefault(argv[1]);
    /* Error checking of pName left out */

    pModule = PyImport_Import(pName);
    Py_DECREF(pName);

    if (pModule != NULL) {
        pFunc = PyObject_GetAttrString(pModule, argv[2]);
        /* pFunc is a new reference */

        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyLong_FromLong(atoi(argv[i + 3]));
                if (!pValue) {
                    Py_DECREF(pArgs);
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert argument\n");
                    return 1;
                }
                /* pValue reference stolen here: */
                PyTuple_SetItem(pArgs, i, pValue);
            }
            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);
            if (pValue != NULL) {
```

(continua na próxima página)

(continuação da página anterior)

```

        printf("Result of call: %ld\n", PyLong_AsLong(pValue));
        Py_DECREF(pValue);
    }
    else {
        Py_DECREF(pFunc);
        Py_DECREF(pModule);
        PyErr_Print();
        fprintf(stderr, "Call failed\n");
        return 1;
    }
}
else {
    if (PyErr_Occurred())
        PyErr_Print();
    fprintf(stderr, "Cannot find function \"%s\"\n", argv[2]);
}
Py_XDECREF(pFunc);
Py_DECREF(pModule);
}
else {
    PyErr_Print();
    fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
    return 1;
}
if (Py_FinalizeEx() < 0) {
    return 120;
}
return 0;
}

```

This code loads a Python script using `argv[1]`, and calls the function named in `argv[2]`. Its integer arguments are the other values of the `argv` array. If you *compile and link* this program (let's call the finished executable **call**), and use it to execute a Python script, such as:

```

def multiply(a,b):
    print("Will compute", a, "times", b)
    c = 0
    for i in range(0, a):
        c = c + b
    return c

```

then the result should be:

```

$ call multiply multiply 3 2
Will compute 3 times 2
Result of call: 6

```

Although the program is quite large for its functionality, most of the code is for data conversion between Python and C, and for error reporting. The interesting part with respect to embedding Python starts with

```

Py_Initialize();
pName = PyUnicode_DecodeFSDefault(argv[1]);
/* Error checking of pName left out */
pModule = PyImport_Import(pName);

```

After initializing the interpreter, the script is loaded using `PyImport_Import()`. This routine needs a Python string as its argument, which is constructed using the `PyUnicode_FromString()` data conversion routine.


```
pFunc = PyObject_GetAttrString(pModule, argv[2]);
/* pFunc is a new reference */

if (pFunc && PyCallable_Check(pFunc)) {
    ...
}
Py_XDECREF(pFunc);
```

Once the script is loaded, the name we're looking for is retrieved using `PyObject_GetAttrString()`. If the name exists, and the object returned is callable, you can safely assume that it is a function. The program then proceeds by constructing a tuple of arguments as normal. The call to the Python function is then made with:

```
pValue = PyObject_CallObject(pFunc, pArgs);
```

Upon return of the function, `pValue` is either `NULL` or it contains a reference to the return value of the function. Be sure to release the reference after examining the value.

3.1.4 Extending Embedded Python

Until now, the embedded Python interpreter had no access to functionality from the application itself. The Python API allows this by extending the embedded interpreter. That is, the embedded interpreter gets extended with routines provided by the application. While it sounds complex, it is not so bad. Simply forget for a while that the application starts the Python interpreter. Instead, consider the application to be a set of subroutines, and write some glue code that gives Python access to those routines, just like you would write a normal Python extension. For example:

```
static int numargs=0;

/* Return the number of arguments of the application command line */
static PyObject*
emb_numargs(PyObject *self, PyObject *args)
{
    if(!PyArg_ParseTuple(args, ":numargs"))
        return NULL;
    return PyLong_FromLong(numargs);
}

static PyMethodDef EmbMethods[] = {
    {"numargs", emb_numargs, METH_VARARGS,
     "Return the number of arguments received by the process."},
    {NULL, NULL, 0, NULL}
};

static PyModuleDef EmbModule = {
    PyModuleDef_HEAD_INIT, "emb", NULL, -1, EmbMethods,
    NULL, NULL, NULL, NULL
};

static PyObject*
PyInit_emb(void)
{
    return PyModule_Create(&EmbModule);
}
```

Insert the above code just above the `main()` function. Also, insert the following two statements before the call to `Py_Initialize()`:

```
numargs = argc;
PyImport_AppendInittab("emb", &PyInit_emb);
```

These two lines initialize the `numargs` variable, and make the `emb.numargs()` function accessible to the embedded Python interpreter. With these extensions, the Python script can do things like

```
import emb
print("Number of arguments", emb.numargs())
```

In a real application, the methods will expose an API of the application to Python.

3.1.5 Embedding Python in C++

It is also possible to embed Python in a C++ program; precisely how this is done will depend on the details of the C++ system used; in general you will need to write the main program in C++, and use the C++ compiler to compile and link your program. There is no need to recompile Python itself using C++.

3.1.6 Compiling and Linking under Unix-like systems

It is not necessarily trivial to find the right flags to pass to your compiler (and linker) in order to embed the Python interpreter into your application, particularly because Python needs to load library modules implemented as C dynamic extensions (`.so` files) linked against it.

To find out the required compiler and linker flags, you can execute the `pythonX.Y-config` script which is generated as part of the installation process (a `python3-config` script may also be available). This script has several options, of which the following will be directly useful to you:

- `pythonX.Y-config --cflags` will give you the recommended flags when compiling:

```
$ /opt/bin/python3.4-config --cflags
-I/opt/include/python3.4m -I/opt/include/python3.4m -DNDEBUG -g -fwrapv -O3 -Wall
-Wstrict-prototypes
```

- `pythonX.Y-config --ldflags` will give you the recommended flags when linking:

```
$ /opt/bin/python3.4-config --ldflags
-L/opt/lib/python3.4/config-3.4m -lpthread -ldl -lutil -lm -lpython3.4m -Xlinker -
-export-dynamic
```

Nota: To avoid confusion between several Python installations (and especially between the system Python and your own compiled Python), it is recommended that you use the absolute path to `pythonX.Y-config`, as in the above example.

If this procedure doesn't work for you (it is not guaranteed to work for all Unix-like platforms; however, we welcome bug reports) you will have to read your system's documentation about dynamic linking and/or examine Python's `Makefile` (use `sysconfig.get_makefile_filename()` to find its location) and compilation options. In this case, the `sysconfig` module is a useful tool to programmatically extract the configuration values that you will want to combine together. For example:

```
>>> import sysconfig
>>> sysconfig.get_config_var('LIBS')
'-lpthread -ldl -lutil'
>>> sysconfig.get_config_var('LINKFORSHARED')
'-Xlinker -export-dynamic'
```

>>> 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 a [PEP 492](#).

BDFL Abreviação da expressão da língua inglesa “Benevolent Dictator for Life” (em português, “Ditador Benevolente Vitalício”), referindo-se a [Guido van Rossum](#), criador do Python.

arquivo binário Um *objeto arquivo* capaz de ler e gravar em *objetos 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 `bufferobjects` 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.

classe Um modelo para criação de objetos definidos pelo usuário. Definições de classe normalmente contém definições de métodos que operam sobre instâncias da classe.

variável de classe Uma variável definida em uma classe e destinada a ser modificada apenas no nível da classe (ou seja, não em uma instância da classe).

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 no Python presume a existência de chaves ou atributos válidos e captura exceções caso essa premissa se prove falsa. Este estilo limpo e rápido se caracteriza pela presença de várias instruções `try` e `except`. A técnica diverge do estilo *LBYL*, comum em outras linguagens como C, por exemplo.

expressão Uma parte da sintaxe que pode ser avaliada para algum valor. Em outras palavras, uma expressão é a acumulação de elementos de expressão como literais, nomes, atributos de acesso, operadores ou chamadas de funções, todos os quais retornam um valor. Em contraste com muitas outras linguagens, nem todas as construções de linguagem são expressões. Também existem *instruções*, as quais não podem ser usadas como expressões, como, por exemplo, `while`. Atribuições também são instruções, não expressões.

módulo de extensão Um módulo escrito em C ou C++, usando a API C do Python para interagir tanto com código de usuário quanto do núcleo.

f-string Literais string prefixadas com `'f'` ou `'F'` são conhecidas como “f-strings” que é uma abreviação de formatted string literals. Veja também [PEP 498](#).

objeto arquivo Um objeto que expõe uma API orientada a arquivos (com métodos tais como `read()` ou `write()`) para um recurso subjacente. Dependendo da maneira como foi criado, um objeto arquivo pode mediar o acesso a um arquivo real no disco ou outro tipo de dispositivo de armazenamento ou de comunicação (por exemplo a entrada/saída padrão, buffers em memória, soquetes, pipes, etc.). Objetos arquivo também são chamados de *objetos arquivo ou similares* ou *fluxos*.

Atualmente há três categorias de objetos arquivo: *arquivos binários brutos*, *arquivos binários em buffer* e *arquivos textos*. Suas interfaces estão definidas no módulo `io`. A forma canônica para criar um objeto arquivo é usando a função `open()`.

objeto arquivo ou similar Um sinônimo do termo *objeto arquivo*.

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 instrução `future`, `from __future__ import <feature>`, direciona o compilador a compilar o módulo atual usando sintaxe ou semântica que será padrão em uma versão futura de Python. O módulo

`__future__` documenta os possíveis valores de *feature*. Importando esse módulo e avaliando suas variáveis, você pode ver quando um novo recurso foi inicialmente adicionado à linguagem e quando será (ou se já é) o padrão:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

coleta de lixo Também conhecido como *garbage collection*, é o processo de liberar a memória quando ela não é mais utilizada. Python executa a liberação da memória através da contagem de referências e um coletor de lixo cíclico que é capaz de detectar e interromper referências cíclicas. O coletor de lixo pode ser controlado usando o módulo `gc`.

gerador Uma função que retorna um *iterador gerador*. É parecida com uma função normal, exceto pelo fato de conter expressões `yield` para produzir uma série de valores que podem ser usados em um laço “for” ou que podem ser obtidos um de cada vez com a função `next()`.

Normalmente refere-se a uma função geradora, mas pode referir-se a um *iterador gerador* em alguns contextos. Em alguns casos onde o significado desejado não está claro, usar o termo completo evita ambiguidade.

iterador gerador Um objeto criado por uma função *geradora*.

Cada `yield` suspende temporariamente o processamento, memorizando o estado da execução 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 Um *tipo* que pode ser parametrizado; tipicamente uma classe contêiner tal como `list` ou `dict`. Usado para *dicas de tipo* e *anotações*.

Para mais detalhes, veja tipo apelido genérico, **PEP 483**, **PEP 484**, **PEP 585**, e o módulo `typing`.

GIL Veja *trava global do interpretador*.

trava global do interpretador O mecanismo utilizado pelo interpretador *CPython* para garantir que apenas uma thread execute o *bytecode* Python por vez. Isto simplifica a implementação do CPython ao fazer com que o modelo de objetos (incluindo tipos embutidos críticos como o `dict`) ganhem segurança implícita contra acesso concorrente. Travar todo o interpretador facilita que o interpretador em si seja multitarefa, às custas de muito do paralelismo já provido por máquinas multiprocessador.

No entanto, alguns módulos de extensão, tanto da biblioteca padrão quanto de terceiros, são desenvolvidos de forma a liberar 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.

iterável Um objeto capaz de retornar seus membros um de cada vez. Exemplos de iteráveis incluem todos os tipos de sequência (tais como `list`, `str` e `tuple`) e alguns tipos 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` 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 no Python aceitam funções chave para controlar como os elementos são ordenados ou agrupados. Algumas delas incluem `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` e `itertools.groupby()`.

Há várias maneiras de se criar funções chave. Por exemplo, o método `str.lower()` pode servir como uma função chave para ordenações insensíveis à caixa. Alternativamente, uma função chave ad-hoc pode ser construída a partir de uma expressão `lambda`, como `lambda r: (r[0], r[2])`. Além disso, 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 travas 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*.

spec de módulo Um espaço de nomes que contém as informações relacionadas à importação usadas para carregar um módulo. Uma instância de `importlib.machinery.ModuleSpec`.

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 `tuple` 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.

objeto 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 *somentepos1* e *somentepos2* 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 *somente_nom1* and *somente_nom2* 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 também o termo *argumento* no glossário, a pergunta do FAQ sobre a diferença entre argumentos e parâmetros, a classe `inspect.Parameter`, a seção *function* e a [PEP 362](#).

entrada de caminho Um local único no *caminho de importação* que o *localizador baseado no caminho* consulta para encontrar módulos a serem importados.

localizador de entrada de caminho Um *localizador* retornado por um chamável em `sys.path_hooks` (ou seja, um *gancho de entrada de caminho*) que sabe como localizar os módulos *entrada de caminho*.

Veja `importlib.abc.PathEntryFinder` para os métodos que localizadores de entrada de caminho implementam.

gancho de entrada de caminho Um chamável na lista `sys.path_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* padrão que procura por um *caminho de importação* de módulos.

objeto caminho ou similar Um objeto representando um caminho de sistema de arquivos. Um objeto caminho ou similar é ou um objeto `str` ou `bytes` representando um caminho, ou um objeto implementando o protocolo `os.PathLike`. Um objeto que suporta o protocolo `os.PathLike` pode ser convertido para um arquivo de caminho do sistema `str` ou `bytes`, através da chamada da função `os.fspath()`; `os.fsdecode()` e `os.fsencode()` podem ser usadas para garantir um `str` ou `bytes` como resultado, respectivamente. Introduzido na [PEP 519](#).

PEP Proposta de melhoria do Python. Uma PEP é um documento de design que fornece informação para a comunidade Python, ou descreve uma nova funcionalidade para o Python ou seus predecessores ou ambientes. PEPs devem prover uma especificação técnica concisa e um racional para funcionalidades propostas.

PEPs têm a intenção de ser os mecanismos primários para propor novas funcionalidades significativas, para coletar opiniões da comunidade sobre um problema, e para documentar as decisões de design que foram adicionadas ao Python. O autor da PEP é responsável por construir um consenso dentro da comunidade e documentar opiniões dissidentes.

Veja [PEP 1](#).

porção Um conjunto de arquivos em um único diretório (possivelmente armazenado em um arquivo zip) que contribuem para um pacote de espaço de nomes, conforme definido em [PEP 420](#).

argumento posicional Veja *argumento*.

API provisória Uma API provisória é uma API que foi deliberadamente excluída das bibliotecas padrões com compatibilidade retroativa garantida. Enquanto mudanças maiores para tais interfaces não são esperadas, contanto que elas sejam marcadas como provisórias, mudanças retroativas incompatíveis (até e incluindo a remoção da interface) podem ocorrer se consideradas necessárias pelos desenvolvedores principais. Tais mudanças não serão feitas gratuitamente – elas irão ocorrer apenas se sérias falhas fundamentais forem descobertas, que foram esquecidas anteriormente a inclusão da API.

Mesmo para APIs provisórias, mudanças retroativas incompatíveis são vistas como uma “solução em último caso” - cada tentativa ainda será feita para encontrar uma resolução retroativa compatível para quaisquer problemas encontrados.

Esse processo permite que a biblioteca padrão continue a evoluir com o passar do tempo, sem se prender em erros de design problemáticos por períodos de tempo prolongados. Veja [PEP 411](#) para mais detalhes.

pacote provisório Veja *API provisória*.

Python 3000 Apelido para a linha de lançamento da versão do Python 3.x (cunhada há muito tempo, quando o lançamento da versão 3 era algo em um futuro muito distante.) Esse termo possui a seguinte abreviação: “Py3k”.

Pythônico Uma ideia ou um pedaço de código que segue de perto as formas de escritas mais comuns da linguagem Python, ao invés de implementar códigos usando conceitos comuns a outras linguagens. Por exemplo, um formato comum em Python é fazer um laço sobre todos os elementos de uma iterável usando a instrução `for`. Muitas outras linguagens não têm esse tipo de construção, então as pessoas que não estão familiarizadas com o Python usam um contador numérico:

```
for i in range(len(food)):
    print(food[i])
```

Ao contrário do método mais limpo, 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.

sequência Um *iterável* com suporte para acesso eficiente a seus elementos através de índices inteiros via método especial `__getitem__()` e que define o método `__len__()` que devolve o tamanho da sequência. Alguns tipos de sequência embutidos são: `list`, `str`, `tuple`, e `bytes`. Note que `dict` também tem suporte para `__getitem__()` e `__len__()`, mas é considerado um 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 [compreensões](#).

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 Uma string em Python é uma sequência de pontos de código Unicode (no intervalo U+0000–U+10FFFF). Para armazenar ou transferir uma string, ela precisa ser serializada como uma sequência de bytes.

A serialização de uma string em uma sequência de bytes é conhecida como “codificação” e a recriação da string a partir de uma sequência de bytes é conhecida como “decodificação”.

Há uma variedade de diferentes serializações de texto codecs, que são coletivamente chamadas de “codificações de texto”.

arquivo texto Um *objeto arquivo* apto a ler e escrever objetos `str`. Geralmente, um arquivo texto, na verdade, acessa um fluxo de dados de bytes e captura o *codificador de texto* automaticamente. Exemplos de arquivos texto são: arquivos abertos em modo texto (`'r'` or `'w'`), `sys.stdin`, `sys.stdout`, e instâncias de `io.StringIO`.

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 apóstrofes (`'''`). Enquanto elas não fornecem nenhuma funcionalidade não disponível com strings de aspas simples, elas são úteis para inúmeras razões. Elas permitem que você inclua aspas simples e duplas não 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)`.

apelido de tipo Um sinônimo para um tipo, criado através da atribuição do tipo para um identificador.

Apelidos de tipo são úteis para simplificar *dicas de tipo*. Por exemplo:

```
def remove_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, pelo seu projeto de referência alternativa em Python, do qual [Sphinx](#) pegou muitas boas ideias.

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 3.9.21

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),
→and
the Individual or Organization ("Licensee") accessing and otherwise using
→Python
3.9.21 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 3.9.21 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 3.9.21 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 3.9.21 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
3.9.21.
4. PSF is making Python 3.9.21 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 3.9.21 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.9.21 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.9.21, 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 3.9.21, 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 3.9.21

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

Os módulos `hashlib`, `posix`, `ssl`, `crypt` usam a biblioteca OpenSSL para desempenho adicional se forem disponibilizados pelo sistema operacional. Além disso, os instaladores do Windows e do Mac OS X para Python podem incluir uma cópia das bibliotecas do OpenSSL, portanto incluímos uma cópia da licença do OpenSSL aqui:

```

LICENSE ISSUES
=====

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
-----

/* =====
 * Copyright (c) 1998-2008 The OpenSSL Project. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in

```

(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 construção:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly          Mark Adler
jloup@gzip.org            madler@alumni.caltech.edu
```

C.3.16 cfuhash

A implementação da tabela de hash usada pelo `tracemalloc` é baseada no projeto `cfuhash`:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of source code must retain the above copyright
  notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above
  copyright notice, this list of conditions and the following
  disclaimer in the documentation and/or other materials provided
  with the distribution.

* Neither the name of the author nor the names of its
  contributors may be used to endorse or promote products derived
  from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
```

(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. Todos os direitos reservados.

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

..., [71](#)
2to3, [71](#)
>>>, [71](#)
__future__, [75](#)
__slots__, [82](#)

A

aguardável, [72](#)
ambiente virtual, [84](#)
anotação, [71](#)
anotação de função, [75](#)
anotação de variável, [84](#)
apelido de tipo, [83](#)
API provisória, [81](#)
argumento, [71](#)
argumento nomeado, [78](#)
argumento posicional, [81](#)
arquivo binário, [72](#)
arquivo texto, [83](#)
aspas triplas, [83](#)
atributo, [72](#)

B

BDFL, [72](#)
bytecode, [73](#)

C

caminho de importação, [77](#)
carregador, [78](#)
C-contiguous, [73](#)
classe, [73](#)
classe base abstrata, [71](#)
classe estilo novo, [80](#)
codificador de texto, [83](#)
coerção, [73](#)
coleta de lixo, [76](#)
compreensão de conjunto, [83](#)
compreensão de dicionário, [74](#)

compreensão de lista, [78](#)
contagem de referências, [82](#)
contíguo, [73](#)
corrotina, [74](#)
CPython, [74](#)

D

deallocation, object, [51](#)
decorador, [74](#)
descritor, [74](#)
desligamento do interpretador, [77](#)
despacho único, [83](#)
dica de tipo, [84](#)
dicionário, [74](#)
divisão pelo piso, [75](#)
docstring, [74](#)

E

EAFP, [75](#)
entrada de caminho, [81](#)
escopo aninhado, [80](#)
espaço de nomes, [79](#)
expressão, [75](#)
expressão geradora, [76](#)

F

f-string, [75](#)
fatia, [83](#)
finalization, of objects, [51](#)
Fortran contiguous, [73](#)
função, [75](#)
função chave, [78](#)
função de corrotina, [74](#)
função de retorno, [73](#)
função genérica, [76](#)
função interna
 repr, [52](#)

G

gancho de entrada de caminho, [81](#)

generator, [76](#)
 generator expression, [76](#)
 gerador, [76](#)
 gerador assíncrono, [72](#)
 gerenciador de contexto, [73](#)
 gerenciador de contexto assíncrono, [72](#)
 GIL, [76](#)

H

hasheável, [77](#)

I

IDLE, [77](#)
 importação, [77](#)
 importador, [77](#)
 imutável, [77](#)
 instrução, [83](#)
 interativo, [77](#)
 interpretado, [77](#)
 iterador, [78](#)
 iterador assíncrono, [72](#)
 iterador gerador, [76](#)
 iterador gerador assíncrono, [72](#)
 iterável, [77](#)
 iterável assíncrono, [72](#)

L

lambda, [78](#)
 LBYL, [78](#)
 lista, [78](#)
 localizador, [75](#)
 localizador baseado no caminho, [81](#)
 localizador de entrada de caminho, [81](#)
 localizador de metacaminho, [79](#)

M

magic
 method, [78](#)
 mapeamento, [79](#)
 máquina virtual, [84](#)
 metaclasses, [79](#)
 method
 magic, [78](#)
 special, [83](#)
 método, [79](#)
 método especial, [83](#)
 método mágico, [78](#)
 módulo, [79](#)
 módulo de extensão, [75](#)
 MRO, [79](#)
 mutável, [79](#)

N

nome qualificado, [82](#)

novas linhas universais, [84](#)
 número complexo, [73](#)

O

object
 deallocation, [51](#)
 finalization, [51](#)
 objeto, [80](#)
 objeto arquivo, [75](#)
 objeto arquivo ou similar, [75](#)
 objeto byte ou similar, [73](#)
 objeto caminho ou similar, [81](#)
 ordem de resolução de métodos, [79](#)

P

pacote, [80](#)
 pacote de espaço de nomes, [80](#)
 pacote provisório, [81](#)
 pacote regular, [82](#)
 parâmetro, [80](#)
 PEP, [81](#)
 Philbrick, Geoff, [15](#)
 porção, [81](#)
 Propostas Estendidas Python
 PEP 1, [81](#)
 PEP 238, [75](#)
 PEP 278, [84](#)
 PEP 302, [75](#), [78](#)
 PEP 343, [73](#)
 PEP 362, [72](#), [81](#)
 PEP 411, [81](#)
 PEP 420, [75](#), [80](#), [81](#)
 PEP 442, [52](#)
 PEP 443, [76](#)
 PEP 451, [75](#)
 PEP 483, [76](#)
 PEP 484, [71](#), [75](#), [76](#), [83](#), [84](#)
 PEP 489, [11](#), [59](#)
 PEP 492, [72](#), [74](#)
 PEP 498, [75](#)
 PEP 519, [81](#)
 PEP 525, [72](#)
 PEP 526, [71](#), [84](#)
 PEP 585, [76](#)
 PEP 3116, [84](#)
 PEP 3155, [82](#)
 PyArg_ParseTuple(), [14](#)
 PyArg_ParseTupleAndKeywords(), [15](#)
 pyc baseado em hash, [77](#)
 PyErr_Fetch(), [51](#)
 PyErr_Restore(), [51](#)
 PyInit_modulename (*função C*), [59](#)
 PyObject_CallObject(), [12](#)
 Python 3000, [81](#)

Pythônico, [82](#)
 PYTHONPATH, [59](#)

R

READ_RESTRICTED, [54](#)
 READONLY, [54](#)
 repr
 função interna, [52](#)
 RESTRICTED, [54](#)

S

sequência, [82](#)
 spec de módulo, [79](#)
 special
 method, [83](#)
 string
 object representation, [52](#)

T

tipagem pato, [74](#)
 tipo, [83](#)
 tipo genérico, [76](#)
 trava global do interpretador, [76](#)
 tupla nomeada, [79](#)

V

váriavel de ambiente
 PYTHONPATH, [59](#)
 variável de classe, [73](#)
 variável de contexto, [73](#)
 visão de dicionário, [74](#)

W

WRITE_RESTRICTED, [54](#)

Z

Zen do Python, [84](#)