The Python/C API

Version 3.14.0rc3

Guido van Rossum and the Python development team

octobre 01, 2025

Python Software Foundation
Email : docs@python.org

Table des matières

1 Introduction **3**
- 1.1 Language version compatibility . . . 3
- 1.2 Coding standards . . . 3
- 1.3 Include Files . . . 3
- 1.4 Useful macros . . . 4
- 1.5 Objects, Types and Reference Counts . . . 6
  - 1.5.1 Compteurs de références . . . 6
  - 1.5.2 Types . . . 10
- 1.6 Exceptions . . . 10
- 1.7 Embarquer Python . . . 12
- 1.8 Debugging Builds . . . 12
- 1.9 Recommended third party tools . . . 13

2 Stabilité de l'API C **15**
- 2.1 Unstable C API . . . 15
- 2.2 ABI stable . . . 15
  - 2.2.1 Limited C API . . . 16
  - 2.2.2 Stable ABI . . . 16
  - 2.2.3 Porté de l'API restreinte et performance . . . 16
  - 2.2.4 Inconvénients de l'API restreinte . . . 17
- 2.3 Considérations relatives aux plateformes . . . 17
- 2.4 Contenu de l'API restreinte . . . 17

3 The Very High Level Layer **35**

4 Reference Counting **41**

5 Gestion des exceptions **45**
- 5.1 Printing and clearing . . . 45
- 5.2 Lever des exceptions . . . 46
- 5.3 Issuing warnings . . . 49
- 5.4 Querying the error indicator . . . 49
- 5.5 Traitement des signaux . . . 53
- 5.6 Exception Classes . . . 54
- 5.7 Objets exception . . . 54
- 5.8 Objets exception Unicode . . . 55
- 5.9 Contrôle de la récursion . . . 56
- 5.10 Exception and warning types . . . 57
  - 5.10.1 Exception types . . . 57
  - 5.10.2 OSError aliases . . . 62
  - 5.10.3 Warning types . . . 63

6 Defining extension modules **65**
6.1 Multiple module instances . . . 65
6.2 Initialization function . . . 66
6.3 Multi-phase initialization . . . 67
6.4 Legacy single-phase initialization . . . 67

7 Utilitaires **69**
7.1 Operating System Utilities . . . 69
7.2 System Functions . . . 72
7.3 Process Control . . . 74
7.4 Importer des modules . . . 74
7.5 Data marshalling support . . . 78
7.6 Analyse des arguments et construction des valeurs . . . 79
7.6.1 Analyse des arguments . . . 79
7.6.2 Construction des valeurs . . . 85
7.7 Conversion et formatage de chaînes . . . 87
7.8 PyHash API . . . 89
7.9 Réflexion . . . 91
7.10 Codec registry and support functions . . . 92
7.10.1 Codec lookup API . . . 92
7.10.2 Registry API for Unicode encoding error handlers . . . 93
7.11 PyTime C API . . . 93
7.11.1 Types . . . 94
7.11.2 Clock Functions . . . 94
7.11.3 Raw Clock Functions . . . 94
7.11.4 Conversion functions . . . 95
7.12 Support for Perf Maps . . . 95

8 Couche d'abstraction des objets **97**
8.1 Protocole Objet . . . 97
8.2 Call Protocol . . . 107
8.2.1 The *tp_call* Protocol . . . 107
8.2.2 The Vectorcall Protocol . . . 107
8.2.3 Object Calling API . . . 109
8.2.4 Call Support API . . . 111
8.3 Number Protocol . . . 111
8.4 Sequence Protocol . . . 114
8.5 Protocole de correspondance . . . 116
8.6 Protocole d'itération . . . 118
8.7 Protocole tampon . . . 119
8.7.1 La structure *buffer* . . . 119
8.7.2 Buffer request types . . . 121
8.7.3 Complex arrays . . . 123
8.7.4 Fonctions relatives aux tampons . . . 124

9 Couche des objets concrets **127**
9.1 Objets fondamentaux . . . 127
9.1.1 Objets type . . . 127
9.1.2 L'objet `None` . . . 134
9.2 Objets numériques . . . 134
9.2.1 Objets *Integer* . . . 134
9.2.2 Les objets booléens . . . 144
9.2.3 Floating-Point Objects . . . 144
9.2.4 Objets représentant des nombres complexes . . . 146
9.3 Objets séquences . . . 148
9.3.1 Objets *bytes* . . . 148
9.3.2 Objets tableau d'octets . . . 150
9.3.3 Objets Unicode et Codecs . . . 151
9.3.4 Tuple Objects . . . 172

9.3.5 Struct Sequence Objects . . . 173
9.3.6 List Objects . . . 174
9.4 Objets conteneurs . . . 176
9.4.1 Objets dictionnaires . . . 176
9.4.2 Set Objects . . . 181
9.5 Objets fonctions . . . 183
9.5.1 Objets fonction . . . 183
9.5.2 Instance Method Objects . . . 185
9.5.3 Objets méthode . . . 185
9.5.4 Objets Cellules . . . 186
9.5.5 Objets code . . . 187
9.5.6 Code Object Flags . . . 189
9.5.7 Extra information . . . 191
9.6 Autres objets . . . 192
9.6.1 Objets fichiers . . . 192
9.6.2 Module Objects . . . 193
9.6.3 Module definitions . . . 194
9.6.4 Creating extension modules dynamically . . . 197
9.6.5 Support functions . . . 198
9.6.6 Itérateurs . . . 201
9.6.7 Les descripteurs . . . 201
9.6.8 Slice Objects . . . 202
9.6.9 Objets de type MemoryView . . . 203
9.6.10 Objets à références faibles . . . 204
9.6.11 Capsules . . . 205
9.6.12 Objets décrivant les *frames* . . . 207
9.6.13 Objets générateur . . . 209
9.6.14 Objets coroutines . . . 210
9.6.15 Context Variables Objects . . . 210
9.6.16 Objets DateTime . . . 212
9.6.17 Objects for Type Hinting . . . 216

10 Initialization, Finalization, and Threads **217**
10.1 Before Python Initialization . . . 217
10.2 Global configuration variables . . . 218
10.3 Initializing and finalizing the interpreter . . . 221
10.4 Process-wide parameters . . . 223
10.5 Thread State and the Global Interpreter Lock . . . 227
10.5.1 Detaching the thread state from extension code . . . 227
10.5.2 Non-Python created threads . . . 228
10.5.3 Supporting subinterpreters in non-Python threads . . . 228
10.5.4 Cautions about fork() . . . 229
10.5.5 Cautions regarding runtime finalization . . . 229
10.5.6 High-level API . . . 229
10.5.7 Low-level API . . . 232
10.6 Sub-interpreter support . . . 235
10.6.1 A Per-Interpreter GIL . . . 237
10.6.2 Bugs and caveats . . . 238
10.7 Asynchronous Notifications . . . 238
10.8 Profiling and Tracing . . . 239
10.9 Reference tracing . . . 240
10.10 Support avancé du débogueur . . . 241
10.11 Thread Local Storage Support . . . 241
10.11.1 Thread Specific Storage (TSS) API . . . 242
10.11.2 Thread Local Storage (TLS) API . . . 243
10.12 Primitives de synchronisation . . . 243
10.12.1 Python Critical Section API . . . 244

11 Python Initialization Configuration **247**
11.1 PyInitConfig C API . . . 247
11.1.1 Example . . . 247
11.1.2 Create Config . . . 248
11.1.3 Error Handling . . . 248
11.1.4 Get Options . . . 249
11.1.5 Set Options . . . 249
11.1.6 Module . . . 249
11.1.7 Initialize Python . . . 250
11.2 Configuration Options . . . 250
11.3 Runtime Python configuration API . . . 251
11.4 PyConfig C API . . . 252
11.4.1 Example . . . 253
11.4.2 PyWideStringList . . . 253
11.4.3 PyStatus . . . 254
11.4.4 PyPreConfig . . . 255
11.4.5 Preinitialize Python with PyPreConfig . . . 256
11.4.6 PyConfig . . . 257
11.4.7 Initialization with PyConfig . . . 268
11.4.8 Isolated Configuration . . . 270
11.4.9 Python Configuration . . . 270
11.4.10 Python Path Configuration . . . 270
11.5 Py_GetArgcArgv() . . . 271
11.6 Delaying main module execution . . . 272

12 Memory Management **273**
12.1 Aperçu . . . 273
12.2 Allocator Domains . . . 274
12.3 Raw Memory Interface . . . 274
12.4 Memory Interface . . . 275
12.5 Object allocators . . . 276
12.6 Default Memory Allocators . . . 278
12.7 Customize Memory Allocators . . . 278
12.8 Debug hooks on the Python memory allocators . . . 279
12.9 The pymalloc allocator . . . 281
12.9.1 Customize pymalloc Arena Allocator . . . 281
12.10 The mimalloc allocator . . . 281
12.11 tracemalloc C API . . . 281
12.12 Exemples . . . 282

13 Implémentation d'objets **283**
13.1 Allouer des objets dans le tas . . . 283
13.2 Object Life Cycle . . . 285
13.2.1 Life Events . . . 285
13.2.2 Cyclic Isolate Destruction . . . 287
13.2.3 Functions . . . 288
13.3 Common Object Structures . . . 289
13.3.1 Base object types and macros . . . 289
13.3.2 Implementing functions and methods . . . 291
13.3.3 Accessing attributes of extension types . . . 294
13.4 Type Object Structures . . . 297
13.4.1 Quick Reference . . . 298
13.4.2 PyTypeObject Definition . . . 302
13.4.3 PyObject Slots . . . 303
13.4.4 PyVarObject Slots . . . 304
13.4.5 PyTypeObject Slots . . . 304
13.4.6 Static Types . . . 327
13.4.7 Heap Types . . . 327

13.4.8 Number Object Structures . . . 327
13.4.9 Mapping Object Structures . . . 330
13.4.10 Sequence Object Structures . . . 330
13.4.11 Buffer Object Structures . . . 331
13.4.12 Async Object Structures . . . 332
13.4.13 Slot Type typedefs . . . 332
13.4.14 Exemples . . . 334
13.5 Supporting Cyclic Garbage Collection . . . 336
13.5.1 Controlling the Garbage Collector State . . . 339
13.5.2 Querying Garbage Collector State . . . 340

14 Version des API et ABI **341**
14.1 Build-time version constants . . . 341
14.2 Run-time version . . . 341
14.3 Bit-packing macros . . . 342

15 Monitoring C API **343**

16 Generating Execution Events **345**
16.1 Managing the Monitoring State . . . 346

A Glossaire **351**

B About this documentation **369**
B.1 Contributors to the Python documentation . . . 369

C Histoire et licence **371**
C.1 Histoire du logiciel . . . 371
C.2 Conditions générales pour accéder à, ou utiliser, Python . . . 372
C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2 . . . 372
C.2.2 LICENCE D'UTILISATION BEOPEN.COM POUR PYTHON 2.0 . . . 373
C.2.3 LICENCE D'UTILISATION CNRI POUR PYTHON 1.6.1 . . . 373
C.2.4 LICENCE D'UTILISATION CWI POUR PYTHON 0.9.0 à 1.2 . . . 375
C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION . . . 375
C.3 Licences et remerciements pour les logiciels tiers . . . 375
C.3.1 Mersenne twister . . . 375
C.3.2 Interfaces de connexion (*sockets*) . . . 376
C.3.3 Interfaces de connexion asynchrones . . . 377
C.3.4 Gestion de témoin (*cookie*) . . . 377
C.3.5 Traçage d'exécution . . . 378
C.3.6 Les fonctions UUencode et UUdecode . . . 378
C.3.7 Appel de procédures distantes en XML (*RPC*, pour *Remote Procedure Call*) . . . 379
C.3.8 test_epoll . . . 379
C.3.9 Select kqueue . . . 380
C.3.10 SipHash24 . . . 380
C.3.11 *strtod* et *dtoa* . . . 381
C.3.12 OpenSSL . . . 381
C.3.13 expat . . . 384
C.3.14 libffi . . . 385
C.3.15 zlib . . . 385
C.3.16 cfuhash . . . 386
C.3.17 libmpdec . . . 387
C.3.18 Ensemble de tests C14N du W3C . . . 387
C.3.19 mimalloc . . . 388
C.3.20 *asyncio* . . . 388
C.3.21 Global Unbounded Sequences (GUS) . . . 389
C.3.22 Zstandard bindings . . . 389

D Copyright **391**

Bibliographie **393**

Index **395**

C'est la documentation de l'API utilisée par les développeurs C et C++ écrivant des modules d'extension ou intégrant Python. Elle va de pair avec extending-index, qui décrit les principes généraux de l'écriture d'extensions, mais ne rentre pas dans les détails de chaque fonction de l'API.

CHAPITRE 1

Introduction

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

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

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

1.1 Language version compatibility

Python's C API is compatible with C11 and C++11 versions of C and C++.

This is a lower limit : the C API does not require features from later C/C++ versions. You do *not* need to enable your compiler's "c11 mode".

1.2 Coding standards

If you're writing C code for inclusion in CPython, you **must** follow the guidelines and standards defined in **PEP 7**. These guidelines apply regardless of the version of Python you are contributing to. Following these conventions is not necessary for your own third party extension modules, unless you eventually expect to contribute them to Python.

1.3 Include Files

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

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

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

> **Note**
>
> Python pouvant définir certaines définitions pré-processeur qui affectent les têtes standard sur certains systèmes, vous *devez* inclure `Python.h` avant les en-têtes standards.
>
> It is recommended to always define `PY_SSIZE_T_CLEAN` before including `Python.h`. See *Analyse des arguments et construction des valeurs* for a description of this macro.

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

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

The header files are typically installed with Python. On Unix, these are located in the directories *prefix*`/include/pythonversion/` and *exec_prefix*`/include/pythonversion/`, where `prefix` and `exec_prefix` are defined by the corresponding parameters to Python's **`configure`** script and *version* is `'%d.%d' % sys.version_info[:2]`. On Windows, the headers are installed in *prefix*`/include`, where `prefix` is the installation directory specified to the installer.

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

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

1.4 Useful macros

Several useful macros are defined in the Python header files. Many are defined closer to where they are useful (for example, *Py_RETURN_NONE*, *PyMODINIT_FUNC*). Others of a more general utility are defined here. This is not necessarily a complete listing.

Py_ABS(x)

Return the absolute value of `x`.

Ajouté dans la version 3.3.

Py_ALWAYS_INLINE

Ask the compiler to always inline a static inline function. The compiler can ignore it and decide to not inline the function.

It can be used to inline performance critical static inline functions when building Python in debug mode with function inlining disabled. For example, MSC disables function inlining when building in debug mode.

Marking blindly a static inline function with Py_ALWAYS_INLINE can result in worse performances (due to increased code size for example). The compiler is usually smarter than the developer for the cost/benefit analysis.

If Python is built in debug mode (if the *Py_DEBUG* macro is defined), the *Py_ALWAYS_INLINE* macro does nothing.

It must be specified before the function return type. Usage :

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

Ajouté dans la version 3.11.

Py_CHARMASK(c)

Argument must be a character or an integer in the range [-128, 127] or [0, 255]. This macro returns `c` cast to an `unsigned char`.

Py_DEPRECATED(version)

Use this for deprecated declarations. The macro must be placed before the symbol name.

Exemple :

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

Modifié dans la version 3.8 : MSVC support was added.

Py_GETENV(s)

Like `getenv(s)`, but returns `NULL` if `-E` was passed on the command line (see *PyConfig.use_environment*).

Py_MAX(x, y)

Return the maximum value between `x` and `y`.

Ajouté dans la version 3.3.

Py_MEMBER_SIZE(type, member)

Return the size of a structure (`type`) `member` in bytes.

Ajouté dans la version 3.6.

Py_MIN(x, y)

Return the minimum value between `x` and `y`.

Ajouté dans la version 3.3.

Py_NO_INLINE

Disable inlining on a function. For example, it reduces the C stack consumption : useful on LTO+PGO builds which heavily inline code (see bpo-33720).

Utilisation :

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

Ajouté dans la version 3.11.

Py_STRINGIFY(x)

Convert `x` to a C string. E.g. `Py_STRINGIFY(123)` returns `"123"`.

Ajouté dans la version 3.4.

Py_UNREACHABLE()

Use this when you have a code path that cannot be reached by design. For example, in the `default:` clause in a `switch` statement for which all possible values are covered in `case` statements. Use this in places where you might be tempted to put an `assert(0)` or `abort()` call.

In release mode, the macro helps the compiler to optimize the code, and avoids a warning about unreachable code. For example, the macro is implemented with `__builtin_unreachable()` on GCC in release mode.

A use for `Py_UNREACHABLE()` is following a call a function that never returns but that is not declared `_Py_NO_RETURN`.

If a code path is very unlikely code but can be reached under exceptional case, this macro must not be used. For example, under low memory condition or if a system call returns a value out of the expected range. In this

case, it's better to report the error to the caller. If the error cannot be reported to caller, *Py_FatalError()* can be used.

Ajouté dans la version 3.7.

Py_UNUSED(arg)

Use this for unused arguments in a function definition to silence compiler warnings. Example : `int func(int a, int Py_UNUSED(b)) { return a; }`.

Ajouté dans la version 3.4.

PyDoc_STRVAR(name, str)

Creates a variable with name `name` that can be used in docstrings. If Python is built without docstrings, the value will be empty.

Use *PyDoc_STRVAR* for docstrings to support building Python without docstrings, as specified in **PEP 7**.

Exemple :

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

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

PyDoc_STR(str)

Creates a docstring for the given input string or an empty string if docstrings are disabled.

Use *PyDoc_STR* in specifying docstrings to support building Python without docstrings, as specified in **PEP 7**.

Exemple :

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

1.5 Objects, Types and Reference Counts

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

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

1.5.1 Compteurs de références

The reference count is important because today's computers have a finite (and often severely limited) memory size ; it counts how many different places there are that have a *strong reference* to an object. Such a place could be another object, or a global (or static) C variable, or a local variable in some C function. When the last *strong reference* to an

object is released (i.e. its reference count becomes zero), the object is deallocated. If it contains references to other objects, those references are released. Those other objects may be deallocated in turn, if there are no more references to them, and so on. (There's an obvious problem with objects that reference each other here ; for now, the solution is "don't do that.")

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

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

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

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

Reference Count Details

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

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

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

```
PyObject *t;

t = PyTuple_New(3);
PyTuple_SetItem(t, 0, PyLong_FromLong(1L));
```

(suite sur la page suivante)

(suite de la page précédente)

```
PyTuple_SetItem(t, 1, PyLong_FromLong(2L));
PyTuple_SetItem(t, 2, PyUnicode_FromString("three"));
```

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

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

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

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

```
PyObject *tuple, *list;

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

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

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

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

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

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

item from the same list using *PySequence_GetItem()* (which happens to take exactly the same arguments), you do own a reference to the returned object.

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

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

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

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

1.5.2 Types

There are few other data types that play a significant role in the Python/C API ; most are simple C types such as `int`, `long`, `double` and `char*`. A few structure types are used to describe static tables used to list the functions exported by a module or the data attributes of a new object type, and another is used to describe the value of a complex number. These will be discussed together with the functions that use them.

type **`Py_ssize_t`**

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

1.6 Exceptions

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

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

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

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

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

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

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

```
def incr_item(dict, key):
    try:
        item = dict[key]
```

(suite sur la page suivante)

(suite de la page précédente)

```
    except KeyError:
        item = 0
    dict[key] = item + 1
```

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

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

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

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

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

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

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

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

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

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

1.7 Embarquer Python

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

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

Py_Initialize() does not set the "script argument list" (`sys.argv`). If this variable is needed by Python code that will be executed later, setting *PyConfig.argv* and *PyConfig.parse_argv* must be set : see *Python Initialization Configuration*.

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

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

The embedding application can steer the search by setting *PyConfig.program_name* *before* calling *Py_InitializeFromConfig()*. Note that `PYTHONHOME` still overrides this and `PYTHONPATH` is still inserted in front of the standard path. An application that requires total control has to provide its own implementation of *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()*, and *Py_GetProgramFullPath()* (all defined in `Modules/getpath.c`).

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

1.8 Debugging Builds

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

A full list of the various types of debugging builds is in the file `Misc/SpecialBuilds.txt` in the Python source distribution. Builds are available that support tracing of reference counts, debugging the memory allocator, or low-level profiling of the main interpreter loop. Only the most frequently used builds will be described in the remainder of this section.

Py_DEBUG

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

In addition to the reference count debugging described below, extra checks are performed, see Python Debug Build.

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

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

1.9 Recommended third party tools

The following third party tools offer both simpler and more sophisticated approaches to creating C, C++ and Rust extensions for Python :

- Cython
- cffi
- HPy
- nanobind (C++)
- Numba
- pybind11 (C++)
- PyO3 (Rust)
- SWIG

Using tools such as these can help avoid writing code that is tightly bound to a particular version of CPython, avoid reference counting errors, and focus more on your own code than on using the CPython API. In general, new versions of Python can be supported by updating the tool, and your code will often use newer and more efficient APIs automatically. Some tools also support compiling for other implementations of Python from a single set of sources.

These projects are not supported by the same people who maintain Python, and issues need to be raised with the projects directly. Remember to check that the project is still maintained and supported, as the list above may become outdated.

Voir aussi

Python Packaging User Guide : Binary Extensions
The Python Packaging User Guide not only covers several available tools that simplify the creation of binary extensions, but also discusses the various reasons why creating an extension module may be desirable in the first place.

CHAPITRE 2

Stabilité de l'API C

Unless documented otherwise, Python's C API is covered by the Backwards Compatibility Policy, **PEP 387**. Most changes to it are source-compatible (typically by only adding new API). Changing existing API or removing API is only done after a deprecation period or to fix serious issues.

CPython's Application Binary Interface (ABI) is forward- and backwards-compatible across a minor release (if these are compiled the same way ; see *Considérations relatives aux plateformes* below). So, code compiled for Python 3.10.0 will work on 3.10.8 and vice versa, but will need to be compiled separately for 3.9.x and 3.11.x.

There are two tiers of C API with different stability expectations :

- *Unstable API*, may change in minor versions without a deprecation period. It is marked by the `PyUnstable` prefix in names.
- *Limited API*, is compatible across several minor releases. When `Py_LIMITED_API` is defined, only this subset is exposed from `Python.h`.

These are discussed in more detail below.

Names prefixed by an underscore, such as `_Py_InternalState`, are private API that can change without notice even in patch releases. If you need to use this API, consider reaching out to CPython developers to discuss adding public API for your use case.

2.1 Unstable C API

Any API named with the `PyUnstable` prefix exposes CPython implementation details, and may change in every minor release (e.g. from 3.9 to 3.10) without any deprecation warnings. However, it will not change in a bugfix release (e.g. from 3.10.0 to 3.10.1).

It is generally intended for specialized, low-level tools like debuggers.

Projects that use this API are expected to follow CPython development and spend extra effort adjusting to changes.

2.2 ABI stable

For simplicity, this document talks about *extensions*, but the Limited API and Stable ABI work the same way for all uses of the API – for example, embedding Python.

2.2.1 Limited C API

Python 3.2 introduced the *Limited API*, a subset of Python's C API. Extensions that only use the Limited API can be compiled once and be loaded on multiple versions of Python. Contents of the Limited API are *listed below*.

Py_LIMITED_API

Définissez cette macro avant d'inclure `Python.h` pour n'inclure que l'API restreinte et indiquer sa version.

Define `Py_LIMITED_API` to the value of *PY_VERSION_HEX* corresponding to the lowest Python version your extension supports. The extension will be ABI-compatible with all Python 3 releases from the specified one onward, and can use Limited API introduced up to that version.

Il est recommandé de renseigner une version mineure minimale (par exemple `0x030A0000` pour Python 3.10) plutôt que d'utiliser directement la macro `PY_VERSION_HEX` pour ne pas dépendre de la version de Python utilisée lors de la compilation.

Vous pouvez aussi définir `Py_LIMITED_API` à `3`. Cette valeur est équivalente à `0x03020000` correspondant à Python 3.2, la première version à supporter l'API restreinte.

2.2.2 Stable ABI

To enable this, Python provides a *Stable ABI* : a set of symbols that will remain ABI-compatible across Python 3.x versions.

Note

The Stable ABI prevents ABI issues, like linker errors due to missing symbols or data corruption due to changes in structure layouts or function signatures. However, other changes in Python can change the *behavior* of extensions. See Python's Backwards Compatibility Policy (**PEP 387**) for details.

The Stable ABI contains symbols exposed in the *Limited API*, but also other ones – for example, functions necessary to support older versions of the Limited API.

Sur Windows les extensions qui utilisent l'ABI stable doivent être liées avec `python3.dll` et non pas avec une bibliothèque spécifique à une version comme `python39.dll`.

Sur certaines plateformes Python essaiera de charger une bibliothèque partagée dont le nom contient `abi3` (par exemple `mymodule.abi3.so`). Il ne vérifie pas si ces extensions respectent l'ABI stable. L'utilisateur (ou ses outils d'administration) doit s'assurer que les extensions compilées avec une version donnée de l'API restreinte, par exemple 3.10+, ne sont pas utilisées avec des versions plus anciennes de Python.

Toutes les fonctions de l'ABI stable sont présentes dans la bibliothèque dynamique de Python en tant que fonction, et pas uniquement comme macro. Elles peuvent donc être utilisées avec des langages qui n'utilisent pas le pré-processeur C.

2.2.3 Porté de l'API restreinte et performance

L'objectif de l'API restreinte est de permettre l'ensemble des opérations possibles avec l'API C étendue, mais peut avoir un impact sur les performances.

Par exemple la fonction *PyList_GetItem()* est disponible, mais la macro « dangereuse » *PyList_GET_ITEM()* ne l'est pas. Cette macro peut être plus rapide car elle dépend de détails d'implémentation spécifique à l'objet `list`.

Si `Py_LIMITED_API` n'est pas défini certaines fonctions de l'API C seront remplacées par des macros ou une version en-ligne. Définir `Py_LIMITED_API` désactive cette optimisation, permettant de se garantir contre les évolutions des structures de données utilisées par Python, et peut réduire les performances.

En omettant la définition de `Py_LIMITED_API` il est possible de compiler une extension utilisant l'API restreinte avec une version spécifique de l'ABI. Les performances seront meilleures pour cette version de Python, mais la compatibilité sera réduite. Compiler en définissant `Py_LIMITED_API` produira une extension qui peut être utilisée quand une variante spécifique à une version n'est pas disponible — par exemple pour une version alpha de Python.

2.2.4 Inconvénients de l'API restreinte

Note that compiling with `Py_LIMITED_API` is *not* a complete guarantee that code conforms to the *Limited API* or the *Stable ABI*. `Py_LIMITED_API` only covers definitions, but an API also includes other issues, such as expected semantics.

Une des limitations est que `Py_LIMITED_API` ne protège pas contre l'appel d'une fonction avec des arguments qui sont invalides pour une version de Python plus ancienne. Par exemple considérons une fonction qui accepte `NULL` comme argument à partir de 3.9. `NULL` permet maintenant d'utiliser le comportement par défaut, mais Python 3.8 essayera d'accéder à l'objet pointé et dé-référencera `NULL`, causant un crash. Des problèmes similaires peuvent se produire avec les attributs des structures.

Un autre problème est que certains attributs ne sont pas encore cachés lorsque `Py_LIMITED_API` est défini, même s'ils ne font pas partie de l'API restreinte.

Pour ces raisons il est recommandé de tester une extension avec *l'ensemble des versions mineures* supportées de Python, et généralement de la compiler avec la plus *ancienne* de ces versions.

Il est aussi recommandé de vérifier la documentation de toutes les API utilisées pour vérifier qu'elles fassent bien partie de l'API restreinte. Même lorsque `Py_LIMITED_API` est défini quelques fonctions privées peuvent être exposées aux utilisateurs pour des raisons techniques, ou par erreur.

Notez aussi que l'API restreinte n'est pas forcément stable : compiler avec Python 3.8 en définissant `Py_LIMITED_API` garanti que l'extension fonctionnera avec Python 3.12, mais pas qu'elle pourra *être compilée* avec Python 3.12. En particulier certaines parties de l'API restreinte peuvent être dépréciées et retirées tant que l'ABI stable n'est pas modifiée.

2.3 Considérations relatives aux plateformes

ABI stability depends not only on Python, but also on the compiler used, lower-level libraries and compiler options. For the purposes of the *Stable ABI*, these details define a "platform". They usually depend on the OS type and processor architecture

Les distributeurs de Python doivent s'assurer que toutes les versions de Python pour une plateforme donnée sont compilées de manière à ne pas rompre la compatibilité de l'ABI stable. C'est le cas des versions produites pour Windows et macOS de `python.org` et de la plupart des distributions tierces.

2.4 Contenu de l'API restreinte

Currently, the *Limited API* includes the following items :

- *PY_VECTORCALL_ARGUMENTS_OFFSET*
- *PyAIter_Check()*
- *PyArg_Parse()*
- *PyArg_ParseTuple()*
- *PyArg_ParseTupleAndKeywords()*
- *PyArg_UnpackTuple()*
- *PyArg_VaParse()*
- *PyArg_VaParseTupleAndKeywords()*
- *PyArg_ValidateKeywordArguments()*
- *PyBaseObject_Type*
- *PyBool_FromLong()*
- *PyBool_Type*
- *PyBuffer_FillContiguousStrides()*
- *PyBuffer_FillInfo()*
- *PyBuffer_FromContiguous()*
- *PyBuffer_GetPointer()*
- *PyBuffer_IsContiguous()*
- *PyBuffer_Release()*

— *PyBuffer_SizeFromFormat()*
— *PyBuffer_ToContiguous()*
— PyByteArrayIter_Type
— *PyByteArray_AsString()*
— *PyByteArray_Concat()*
— *PyByteArray_FromObject()*
— *PyByteArray_FromStringAndSize()*
— *PyByteArray_Resize()*
— *PyByteArray_Size()*
— *PyByteArray_Type*
— PyBytesIter_Type
— *PyBytes_AsString()*
— *PyBytes_AsStringAndSize()*
— *PyBytes_Concat()*
— *PyBytes_ConcatAndDel()*
— PyBytes_DecodeEscape()
— *PyBytes_FromFormat()*
— *PyBytes_FromFormatV()*
— *PyBytes_FromObject()*
— *PyBytes_FromString()*
— *PyBytes_FromStringAndSize()*
— PyBytes_Repr()
— *PyBytes_Size()*
— *PyBytes_Type*
— *PyCFunction*
— *PyCFunctionFast*
— *PyCFunctionFastWithKeywords*
— *PyCFunctionWithKeywords*
— PyCFunction_GetFlags()
— PyCFunction_GetFunction()
— PyCFunction_GetSelf()
— *PyCFunction_New()*
— *PyCFunction_NewEx()*
— PyCFunction_Type
— *PyCMethod_New()*
— *PyCallIter_New()*
— *PyCallIter_Type*
— *PyCallable_Check()*
— *PyCapsule_Destructor*
— *PyCapsule_GetContext()*
— *PyCapsule_GetDestructor()*
— *PyCapsule_GetName()*
— *PyCapsule_GetPointer()*
— *PyCapsule_Import()*
— *PyCapsule_IsValid()*
— *PyCapsule_New()*
— *PyCapsule_SetContext()*
— *PyCapsule_SetDestructor()*
— *PyCapsule_SetName()*
— *PyCapsule_SetPointer()*
— PyCapsule_Type
— PyClassMethodDescr_Type
— *PyCodec_BackslashReplaceErrors()*
— *PyCodec_Decode()*
— *PyCodec_Decoder()*
— *PyCodec_Encode()*
— *PyCodec_Encoder()*
— *PyCodec_IgnoreErrors()*

— *PyCodec_IncrementalDecoder()*
— *PyCodec_IncrementalEncoder()*
— *PyCodec_KnownEncoding()*
— *PyCodec_LookupError()*
— *PyCodec_NameReplaceErrors()*
— *PyCodec_Register()*
— *PyCodec_RegisterError()*
— *PyCodec_ReplaceErrors()*
— *PyCodec_StreamReader()*
— *PyCodec_StreamWriter()*
— *PyCodec_StrictErrors()*
— *PyCodec_Unregister()*
— *PyCodec_XMLCharRefReplaceErrors()*
— *PyComplex_FromDoubles()*
— *PyComplex_ImagAsDouble()*
— *PyComplex_RealAsDouble()*
— *PyComplex_Type*
— *PyDescr_NewClassMethod()*
— *PyDescr_NewGetSet()*
— *PyDescr_NewMember()*
— *PyDescr_NewMethod()*
— PyDictItems_Type
— PyDictIterItem_Type
— PyDictIterKey_Type
— PyDictIterValue_Type
— PyDictKeys_Type
— *PyDictProxy_New()*
— PyDictProxy_Type
— PyDictRevIterItem_Type
— PyDictRevIterKey_Type
— PyDictRevIterValue_Type
— PyDictValues_Type
— *PyDict_Clear()*
— *PyDict_Contains()*
— *PyDict_Copy()*
— *PyDict_DelItem()*
— *PyDict_DelItemString()*
— *PyDict_GetItem()*
— *PyDict_GetItemRef()*
— *PyDict_GetItemString()*
— *PyDict_GetItemStringRef()*
— *PyDict_GetItemWithError()*
— *PyDict_Items()*
— *PyDict_Keys()*
— *PyDict_Merge()*
— *PyDict_MergeFromSeq2()*
— *PyDict_New()*
— *PyDict_Next()*
— *PyDict_SetItem()*
— *PyDict_SetItemString()*
— *PyDict_Size()*
— *PyDict_Type*
— *PyDict_Update()*
— *PyDict_Values()*
— *PyEllipsis_Type*
— PyEnum_Type
— *PyErr_BadArgument()*
— *PyErr_BadInternalCall()*

— *PyErr_CheckSignals()*
— *PyErr_Clear()*
— PyErr_Display()
— *PyErr_DisplayException()*
— *PyErr_ExceptionMatches()*
— *PyErr_Fetch()*
— *PyErr_Format()*
— *PyErr_FormatV()*
— *PyErr_GetExcInfo()*
— *PyErr_GetHandledException()*
— *PyErr_GetRaisedException()*
— *PyErr_GivenExceptionMatches()*
— *PyErr_NewException()*
— *PyErr_NewExceptionWithDoc()*
— *PyErr_NoMemory()*
— *PyErr_NormalizeException()*
— *PyErr_Occurred()*
— *PyErr_Print()*
— *PyErr_PrintEx()*
— PyErr_ProgramText()
— *PyErr_ResourceWarning()*
— *PyErr_Restore()*
— *PyErr_SetExcFromWindowsErr()*
— *PyErr_SetExcFromWindowsErrWithFilename()*
— *PyErr_SetExcFromWindowsErrWithFilenameObject()*
— *PyErr_SetExcFromWindowsErrWithFilenameObjects()*
— *PyErr_SetExcInfo()*
— *PyErr_SetFromErrno()*
— *PyErr_SetFromErrnoWithFilename()*
— *PyErr_SetFromErrnoWithFilenameObject()*
— *PyErr_SetFromErrnoWithFilenameObjects()*
— *PyErr_SetFromWindowsErr()*
— *PyErr_SetFromWindowsErrWithFilename()*
— *PyErr_SetHandledException()*
— *PyErr_SetImportError()*
— *PyErr_SetImportErrorSubclass()*
— *PyErr_SetInterrupt()*
— *PyErr_SetInterruptEx()*
— *PyErr_SetNone()*
— *PyErr_SetObject()*
— *PyErr_SetRaisedException()*
— *PyErr_SetString()*
— *PyErr_SyntaxLocation()*
— *PyErr_SyntaxLocationEx()*
— *PyErr_WarnEx()*
— *PyErr_WarnExplicit()*
— *PyErr_WarnFormat()*
— *PyErr_WriteUnraisable()*
— *PyEval_AcquireThread()*
— *PyEval_EvalCode()*
— *PyEval_EvalCodeEx()*
— *PyEval_EvalFrame()*
— *PyEval_EvalFrameEx()*
— *PyEval_GetBuiltins()*
— *PyEval_GetFrame()*
— *PyEval_GetFrameBuiltins()*
— *PyEval_GetFrameGlobals()*
— *PyEval_GetFrameLocals()*

— *PyEval_GetFuncDesc()*
— *PyEval_GetFuncName()*
— *PyEval_GetGlobals()*
— *PyEval_GetLocals()*
— *PyEval_InitThreads()*
— *PyEval_ReleaseThread()*
— *PyEval_RestoreThread()*
— *PyEval_SaveThread()*
— *PyExc_ArithmeticError*
— *PyExc_AssertionError*
— *PyExc_AttributeError*
— *PyExc_BaseException*
— *PyExc_BaseExceptionGroup*
— *PyExc_BlockingIOError*
— *PyExc_BrokenPipeError*
— *PyExc_BufferError*
— *PyExc_BytesWarning*
— *PyExc_ChildProcessError*
— *PyExc_ConnectionAbortedError*
— *PyExc_ConnectionError*
— *PyExc_ConnectionRefusedError*
— *PyExc_ConnectionResetError*
— *PyExc_DeprecationWarning*
— *PyExc_EOFError*
— *PyExc_EncodingWarning*
— *PyExc_EnvironmentError*
— *PyExc_Exception*
— *PyExc_FileExistsError*
— *PyExc_FileNotFoundError*
— *PyExc_FloatingPointError*
— *PyExc_FutureWarning*
— *PyExc_GeneratorExit*
— *PyExc_IOError*
— *PyExc_ImportError*
— *PyExc_ImportWarning*
— *PyExc_IndentationError*
— *PyExc_IndexError*
— *PyExc_InterruptedError*
— *PyExc_IsADirectoryError*
— *PyExc_KeyError*
— *PyExc_KeyboardInterrupt*
— *PyExc_LookupError*
— *PyExc_MemoryError*
— *PyExc_ModuleNotFoundError*
— *PyExc_NameError*
— *PyExc_NotADirectoryError*
— *PyExc_NotImplementedError*
— *PyExc_OSError*
— *PyExc_OverflowError*
— *PyExc_PendingDeprecationWarning*
— *PyExc_PermissionError*
— *PyExc_ProcessLookupError*
— *PyExc_RecursionError*
— *PyExc_ReferenceError*
— *PyExc_ResourceWarning*
— *PyExc_RuntimeError*
— *PyExc_RuntimeWarning*
— *PyExc_StopAsyncIteration*

— *PyExc_StopIteration*
— *PyExc_SyntaxError*
— *PyExc_SyntaxWarning*
— *PyExc_SystemError*
— *PyExc_SystemExit*
— *PyExc_TabError*
— *PyExc_TimeoutError*
— *PyExc_TypeError*
— *PyExc_UnboundLocalError*
— *PyExc_UnicodeDecodeError*
— *PyExc_UnicodeEncodeError*
— *PyExc_UnicodeError*
— *PyExc_UnicodeTranslateError*
— *PyExc_UnicodeWarning*
— *PyExc_UserWarning*
— *PyExc_ValueError*
— *PyExc_Warning*
— *PyExc_WindowsError*
— *PyExc_ZeroDivisionError*
— *PyExceptionClass_Name()*
— *PyException_GetArgs()*
— *PyException_GetCause()*
— *PyException_GetContext()*
— *PyException_GetTraceback()*
— *PyException_SetArgs()*
— *PyException_SetCause()*
— *PyException_SetContext()*
— *PyException_SetTraceback()*
— *PyFile_FromFd()*
— *PyFile_GetLine()*
— *PyFile_WriteObject()*
— *PyFile_WriteString()*
— PyFilter_Type
— *PyFloat_AsDouble()*
— *PyFloat_FromDouble()*
— *PyFloat_FromString()*
— *PyFloat_GetInfo()*
— *PyFloat_GetMax()*
— *PyFloat_GetMin()*
— *PyFloat_Type*
— *PyFrameObject*
— *PyFrame_GetCode()*
— *PyFrame_GetLineNumber()*
— *PyFrozenSet_New()*
— *PyFrozenSet_Type*
— *PyGC_Collect()*
— *PyGC_Disable()*
— *PyGC_Enable()*
— *PyGC_IsEnabled()*
— *PyGILState_Ensure()*
— *PyGILState_GetThisThreadState()*
— *PyGILState_Release()*
— PyGILState_STATE
— *PyGetSetDef*
— PyGetSetDescr_Type
— *PyImport_AddModule()*
— *PyImport_AddModuleObject()*
— *PyImport_AddModuleRef()*

— *PyImport_AppendInittab()*
— *PyImport_ExecCodeModule()*
— *PyImport_ExecCodeModuleEx()*
— *PyImport_ExecCodeModuleObject()*
— *PyImport_ExecCodeModuleWithPathnames()*
— *PyImport_GetImporter()*
— *PyImport_GetMagicNumber()*
— *PyImport_GetMagicTag()*
— *PyImport_GetModule()*
— *PyImport_GetModuleDict()*
— *PyImport_Import()*
— *PyImport_ImportFrozenModule()*
— *PyImport_ImportFrozenModuleObject()*
— *PyImport_ImportModule()*
— *PyImport_ImportModuleLevel()*
— *PyImport_ImportModuleLevelObject()*
— *PyImport_ImportModuleNoBlock()*
— *PyImport_ReloadModule()*
— *PyIndex_Check()*
— *PyInterpreterState*
— *PyInterpreterState_Clear()*
— *PyInterpreterState_Delete()*
— *PyInterpreterState_Get()*
— *PyInterpreterState_GetDict()*
— *PyInterpreterState_GetID()*
— *PyInterpreterState_New()*
— *PyIter_Check()*
— *PyIter_Next()*
— *PyIter_NextItem()*
— *PyIter_Send()*
— PyListIter_Type
— PyListRevIter_Type
— *PyList_Append()*
— *PyList_AsTuple()*
— *PyList_GetItem()*
— *PyList_GetItemRef()*
— *PyList_GetSlice()*
— *PyList_Insert()*
— *PyList_New()*
— *PyList_Reverse()*
— *PyList_SetItem()*
— *PyList_SetSlice()*
— *PyList_Size()*
— *PyList_Sort()*
— *PyList_Type*
— *PyLongObject*
— PyLongRangeIter_Type
— *PyLong_AsDouble()*
— *PyLong_AsInt()*
— *PyLong_AsInt32()*
— *PyLong_AsInt64()*
— *PyLong_AsLong()*
— *PyLong_AsLongAndOverflow()*
— *PyLong_AsLongLong()*
— *PyLong_AsLongLongAndOverflow()*
— *PyLong_AsNativeBytes()*
— *PyLong_AsSize_t()*
— *PyLong_AsSsize_t()*

— *PyLong_AsUInt32()*
— *PyLong_AsUInt64()*
— *PyLong_AsUnsignedLong()*
— *PyLong_AsUnsignedLongLong()*
— *PyLong_AsUnsignedLongLongMask()*
— *PyLong_AsUnsignedLongMask()*
— *PyLong_AsVoidPtr()*
— *PyLong_FromDouble()*
— *PyLong_FromInt32()*
— *PyLong_FromInt64()*
— *PyLong_FromLong()*
— *PyLong_FromLongLong()*
— *PyLong_FromNativeBytes()*
— *PyLong_FromSize_t()*
— *PyLong_FromSsize_t()*
— *PyLong_FromString()*
— *PyLong_FromUInt32()*
— *PyLong_FromUInt64()*
— *PyLong_FromUnsignedLong()*
— *PyLong_FromUnsignedLongLong()*
— *PyLong_FromUnsignedNativeBytes()*
— *PyLong_FromVoidPtr()*
— *PyLong_GetInfo()*
— *PyLong_Type*
— PyMap_Type
— *PyMapping_Check()*
— *PyMapping_GetItemString()*
— *PyMapping_GetOptionalItem()*
— *PyMapping_GetOptionalItemString()*
— *PyMapping_HasKey()*
— *PyMapping_HasKeyString()*
— *PyMapping_HasKeyStringWithError()*
— *PyMapping_HasKeyWithError()*
— *PyMapping_Items()*
— *PyMapping_Keys()*
— *PyMapping_Length()*
— *PyMapping_SetItemString()*
— *PyMapping_Size()*
— *PyMapping_Values()*
— *PyMem_Calloc()*
— *PyMem_Free()*
— *PyMem_Malloc()*
— *PyMem_RawCalloc()*
— *PyMem_RawFree()*
— *PyMem_RawMalloc()*
— *PyMem_RawRealloc()*
— *PyMem_Realloc()*
— *PyMemberDef*
— PyMemberDescr_Type
— *PyMember_GetOne()*
— *PyMember_SetOne()*
— *PyMemoryView_FromBuffer()*
— *PyMemoryView_FromMemory()*
— *PyMemoryView_FromObject()*
— *PyMemoryView_GetContiguous()*
— PyMemoryView_Type
— *PyMethodDef*
— PyMethodDescr_Type

— *PyModuleDef*
— PyModuleDef_Base
— *PyModuleDef_Init()*
— PyModuleDef_Type
— *PyModule_Add()*
— *PyModule_AddFunctions()*
— *PyModule_AddIntConstant()*
— *PyModule_AddObject()*
— *PyModule_AddObjectRef()*
— *PyModule_AddStringConstant()*
— *PyModule_AddType()*
— *PyModule_Create2()*
— *PyModule_ExecDef()*
— *PyModule_FromDefAndSpec2()*
— *PyModule_GetDef()*
— *PyModule_GetDict()*
— *PyModule_GetFilename()*
— *PyModule_GetFilenameObject()*
— *PyModule_GetName()*
— *PyModule_GetNameObject()*
— *PyModule_GetState()*
— *PyModule_New()*
— *PyModule_NewObject()*
— *PyModule_SetDocString()*
— *PyModule_Type*
— *PyNumber_Absolute()*
— *PyNumber_Add()*
— *PyNumber_And()*
— *PyNumber_AsSsize_t()*
— *PyNumber_Check()*
— *PyNumber_Divmod()*
— *PyNumber_Float()*
— *PyNumber_FloorDivide()*
— *PyNumber_InPlaceAdd()*
— *PyNumber_InPlaceAnd()*
— *PyNumber_InPlaceFloorDivide()*
— *PyNumber_InPlaceLshift()*
— *PyNumber_InPlaceMatrixMultiply()*
— *PyNumber_InPlaceMultiply()*
— *PyNumber_InPlaceOr()*
— *PyNumber_InPlacePower()*
— *PyNumber_InPlaceRemainder()*
— *PyNumber_InPlaceRshift()*
— *PyNumber_InPlaceSubtract()*
— *PyNumber_InPlaceTrueDivide()*
— *PyNumber_InPlaceXor()*
— *PyNumber_Index()*
— *PyNumber_Invert()*
— *PyNumber_Long()*
— *PyNumber_Lshift()*
— *PyNumber_MatrixMultiply()*
— *PyNumber_Multiply()*
— *PyNumber_Negative()*
— *PyNumber_Or()*
— *PyNumber_Positive()*
— *PyNumber_Power()*
— *PyNumber_Remainder()*
— *PyNumber_Rshift()*

— *PyNumber_Subtract()*
— *PyNumber_ToBase()*
— *PyNumber_TrueDivide()*
— *PyNumber_Xor()*
— *PyOS_AfterFork()*
— *PyOS_AfterFork_Child()*
— *PyOS_AfterFork_Parent()*
— *PyOS_BeforeFork()*
— *PyOS_CheckStack()*
— *PyOS_FSPath()*
— *PyOS_InputHook*
— PyOS_InterruptOccurred()
— *PyOS_double_to_string()*
— *PyOS_getsig()*
— PyOS_mystricmp()
— PyOS_mystrnicmp()
— *PyOS_setsig()*
— *PyOS_sighandler_t*
— *PyOS_snprintf()*
— *PyOS_string_to_double()*
— *PyOS_strtol()*
— *PyOS_strtoul()*
— *PyOS_vsnprintf()*
— *PyObject*
— *PyObject.ob_refcnt*
— *PyObject.ob_type*
— *PyObject_ASCII()*
— *PyObject_AsFileDescriptor()*
— *PyObject_Bytes()*
— *PyObject_Call()*
— *PyObject_CallFunction()*
— *PyObject_CallFunctionObjArgs()*
— *PyObject_CallMethod()*
— *PyObject_CallMethodObjArgs()*
— *PyObject_CallNoArgs()*
— *PyObject_CallObject()*
— *PyObject_Calloc()*
— *PyObject_CheckBuffer()*
— *PyObject_ClearWeakRefs()*
— *PyObject_CopyData()*
— *PyObject_DelAttr()*
— *PyObject_DelAttrString()*
— *PyObject_DelItem()*
— *PyObject_DelItemString()*
— *PyObject_Dir()*
— *PyObject_Format()*
— *PyObject_Free()*
— *PyObject_GC_Del()*
— *PyObject_GC_IsFinalized()*
— *PyObject_GC_IsTracked()*
— *PyObject_GC_Track()*
— *PyObject_GC_UnTrack()*
— *PyObject_GenericGetAttr()*
— *PyObject_GenericGetDict()*
— *PyObject_GenericSetAttr()*
— *PyObject_GenericSetDict()*
— *PyObject_GetAIter()*
— *PyObject_GetAttr()*

— *PyObject_GetAttrString()*
— *PyObject_GetBuffer()*
— *PyObject_GetItem()*
— *PyObject_GetIter()*
— *PyObject_GetOptionalAttr()*
— *PyObject_GetOptionalAttrString()*
— *PyObject_GetTypeData()*
— *PyObject_HasAttr()*
— *PyObject_HasAttrString()*
— *PyObject_HasAttrStringWithError()*
— *PyObject_HasAttrWithError()*
— *PyObject_Hash()*
— *PyObject_HashNotImplemented()*
— *PyObject_Init()*
— *PyObject_InitVar()*
— *PyObject_IsInstance()*
— *PyObject_IsSubclass()*
— *PyObject_IsTrue()*
— *PyObject_Length()*
— *PyObject_Malloc()*
— *PyObject_Not()*
— *PyObject_Realloc()*
— *PyObject_Repr()*
— *PyObject_RichCompare()*
— *PyObject_RichCompareBool()*
— *PyObject_SelfIter()*
— *PyObject_SetAttr()*
— *PyObject_SetAttrString()*
— *PyObject_SetItem()*
— *PyObject_Size()*
— *PyObject_Str()*
— *PyObject_Type()*
— *PyObject_Vectorcall()*
— *PyObject_VectorcallMethod()*
— *PyProperty_Type*
— PyRangeIter_Type
— PyRange_Type
— PyReversed_Type
— *PySeqIter_New()*
— *PySeqIter_Type*
— *PySequence_Check()*
— *PySequence_Concat()*
— *PySequence_Contains()*
— *PySequence_Count()*
— *PySequence_DelItem()*
— *PySequence_DelSlice()*
— *PySequence_Fast()*
— *PySequence_GetItem()*
— *PySequence_GetSlice()*
— *PySequence_In()*
— *PySequence_InPlaceConcat()*
— *PySequence_InPlaceRepeat()*
— *PySequence_Index()*
— *PySequence_Length()*
— *PySequence_List()*
— *PySequence_Repeat()*
— *PySequence_SetItem()*
— *PySequence_SetSlice()*

— *PySequence_Size()*
— *PySequence_Tuple()*
— PySetIter_Type
— *PySet_Add()*
— *PySet_Clear()*
— *PySet_Contains()*
— *PySet_Discard()*
— *PySet_New()*
— *PySet_Pop()*
— *PySet_Size()*
— *PySet_Type*
— *PySlice_AdjustIndices()*
— *PySlice_GetIndices()*
— *PySlice_GetIndicesEx()*
— *PySlice_New()*
— *PySlice_Type*
— *PySlice_Unpack()*
— *PyState_AddModule()*
— *PyState_FindModule()*
— *PyState_RemoveModule()*
— *PyStructSequence_Desc*
— *PyStructSequence_Field*
— *PyStructSequence_GetItem()*
— *PyStructSequence_New()*
— *PyStructSequence_NewType()*
— *PyStructSequence_SetItem()*
— *PyStructSequence_UnnamedField*
— PySuper_Type
— *PySys_Audit()*
— *PySys_AuditTuple()*
— *PySys_FormatStderr()*
— *PySys_FormatStdout()*
— *PySys_GetObject()*
— *PySys_GetXOptions()*
— *PySys_ResetWarnOptions()*
— *PySys_SetArgv()*
— *PySys_SetArgvEx()*
— *PySys_SetObject()*
— *PySys_WriteStderr()*
— *PySys_WriteStdout()*
— *PyThreadState*
— *PyThreadState_Clear()*
— *PyThreadState_Delete()*
— *PyThreadState_Get()*
— *PyThreadState_GetDict()*
— *PyThreadState_GetFrame()*
— *PyThreadState_GetID()*
— *PyThreadState_GetInterpreter()*
— *PyThreadState_New()*
— *PyThreadState_SetAsyncExc()*
— *PyThreadState_Swap()*
— PyThread_GetInfo()
— *PyThread_ReInitTLS()*
— PyThread_acquire_lock()
— PyThread_acquire_lock_timed()
— PyThread_allocate_lock()
— *PyThread_create_key()*
— *PyThread_delete_key()*

— *PyThread_delete_key_value()*
— PyThread_exit_thread()
— PyThread_free_lock()
— *PyThread_get_key_value()*
— PyThread_get_stacksize()
— PyThread_get_thread_ident()
— PyThread_get_thread_native_id()
— PyThread_init_thread()
— PyThread_release_lock()
— *PyThread_set_key_value()*
— PyThread_set_stacksize()
— PyThread_start_new_thread()
— *PyThread_tss_alloc()*
— *PyThread_tss_create()*
— *PyThread_tss_delete()*
— *PyThread_tss_free()*
— *PyThread_tss_get()*
— *PyThread_tss_is_created()*
— *PyThread_tss_set()*
— PyTraceBack_Here()
— PyTraceBack_Print()
— PyTraceBack_Type
— PyTupleIter_Type
— *PyTuple_GetItem()*
— *PyTuple_GetSlice()*
— *PyTuple_New()*
— *PyTuple_Pack()*
— *PyTuple_SetItem()*
— *PyTuple_Size()*
— *PyTuple_Type*
— *PyTypeObject*
— *PyType_ClearCache()*
— *PyType_Freeze()*
— *PyType_FromMetaclass()*
— *PyType_FromModuleAndSpec()*
— *PyType_FromSpec()*
— *PyType_FromSpecWithBases()*
— *PyType_GenericAlloc()*
— *PyType_GenericNew()*
— *PyType_GetBaseByToken()*
— *PyType_GetFlags()*
— *PyType_GetFullyQualifiedName()*
— *PyType_GetModule()*
— *PyType_GetModuleByDef()*
— *PyType_GetModuleName()*
— *PyType_GetModuleState()*
— *PyType_GetName()*
— *PyType_GetQualName()*
— *PyType_GetSlot()*
— *PyType_GetTypeDataSize()*
— *PyType_IsSubtype()*
— *PyType_Modified()*
— *PyType_Ready()*
— *PyType_Slot*
— *PyType_Spec*
— *PyType_Type*
— *PyUnicodeDecodeError_Create()*
— *PyUnicodeDecodeError_GetEncoding()*

— *PyUnicodeDecodeError_GetEnd()*
— *PyUnicodeDecodeError_GetObject()*
— *PyUnicodeDecodeError_GetReason()*
— *PyUnicodeDecodeError_GetStart()*
— *PyUnicodeDecodeError_SetEnd()*
— *PyUnicodeDecodeError_SetReason()*
— *PyUnicodeDecodeError_SetStart()*
— *PyUnicodeEncodeError_GetEncoding()*
— *PyUnicodeEncodeError_GetEnd()*
— *PyUnicodeEncodeError_GetObject()*
— *PyUnicodeEncodeError_GetReason()*
— *PyUnicodeEncodeError_GetStart()*
— *PyUnicodeEncodeError_SetEnd()*
— *PyUnicodeEncodeError_SetReason()*
— *PyUnicodeEncodeError_SetStart()*
— *PyUnicodeIter_Type*
— *PyUnicodeTranslateError_GetEnd()*
— *PyUnicodeTranslateError_GetObject()*
— *PyUnicodeTranslateError_GetReason()*
— *PyUnicodeTranslateError_GetStart()*
— *PyUnicodeTranslateError_SetEnd()*
— *PyUnicodeTranslateError_SetReason()*
— *PyUnicodeTranslateError_SetStart()*
— *PyUnicode_Append()*
— *PyUnicode_AppendAndDel()*
— *PyUnicode_AsASCIIString()*
— *PyUnicode_AsCharmapString()*
— PyUnicode_AsDecodedObject()
— PyUnicode_AsDecodedUnicode()
— PyUnicode_AsEncodedObject()
— *PyUnicode_AsEncodedString()*
— PyUnicode_AsEncodedUnicode()
— *PyUnicode_AsLatin1String()*
— *PyUnicode_AsMBCSString()*
— *PyUnicode_AsRawUnicodeEscapeString()*
— *PyUnicode_AsUCS4()*
— *PyUnicode_AsUCS4Copy()*
— *PyUnicode_AsUTF16String()*
— *PyUnicode_AsUTF32String()*
— *PyUnicode_AsUTF8AndSize()*
— *PyUnicode_AsUTF8String()*
— *PyUnicode_AsUnicodeEscapeString()*
— *PyUnicode_AsWideChar()*
— *PyUnicode_AsWideCharString()*
— *PyUnicode_BuildEncodingMap()*
— *PyUnicode_Compare()*
— *PyUnicode_CompareWithASCIIString()*
— *PyUnicode_Concat()*
— *PyUnicode_Contains()*
— *PyUnicode_Count()*
— *PyUnicode_Decode()*
— *PyUnicode_DecodeASCII()*
— *PyUnicode_DecodeCharmap()*
— *PyUnicode_DecodeCodePageStateful()*
— *PyUnicode_DecodeFSDefault()*
— *PyUnicode_DecodeFSDefaultAndSize()*
— *PyUnicode_DecodeLatin1()*
— *PyUnicode_DecodeLocale()*

— *PyUnicode_DecodeLocaleAndSize()*
— *PyUnicode_DecodeMBCS()*
— *PyUnicode_DecodeMBCSStateful()*
— *PyUnicode_DecodeRawUnicodeEscape()*
— *PyUnicode_DecodeUTF16()*
— *PyUnicode_DecodeUTF16Stateful()*
— *PyUnicode_DecodeUTF32()*
— *PyUnicode_DecodeUTF32Stateful()*
— *PyUnicode_DecodeUTF7()*
— *PyUnicode_DecodeUTF7Stateful()*
— *PyUnicode_DecodeUTF8()*
— *PyUnicode_DecodeUTF8Stateful()*
— *PyUnicode_DecodeUnicodeEscape()*
— *PyUnicode_EncodeCodePage()*
— *PyUnicode_EncodeFSDefault()*
— *PyUnicode_EncodeLocale()*
— *PyUnicode_Equal()*
— *PyUnicode_EqualToUTF8()*
— *PyUnicode_EqualToUTF8AndSize()*
— *PyUnicode_FSConverter()*
— *PyUnicode_FSDecoder()*
— *PyUnicode_Find()*
— *PyUnicode_FindChar()*
— *PyUnicode_Format()*
— *PyUnicode_FromEncodedObject()*
— *PyUnicode_FromFormat()*
— *PyUnicode_FromFormatV()*
— *PyUnicode_FromObject()*
— *PyUnicode_FromOrdinal()*
— *PyUnicode_FromString()*
— *PyUnicode_FromStringAndSize()*
— *PyUnicode_FromWideChar()*
— *PyUnicode_GetDefaultEncoding()*
— *PyUnicode_GetLength()*
— *PyUnicode_InternFromString()*
— *PyUnicode_InternInPlace()*
— *PyUnicode_IsIdentifier()*
— *PyUnicode_Join()*
— *PyUnicode_Partition()*
— *PyUnicode_RPartition()*
— *PyUnicode_RSplit()*
— *PyUnicode_ReadChar()*
— *PyUnicode_Replace()*
— *PyUnicode_Resize()*
— *PyUnicode_RichCompare()*
— *PyUnicode_Split()*
— *PyUnicode_Splitlines()*
— *PyUnicode_Substring()*
— *PyUnicode_Tailmatch()*
— *PyUnicode_Translate()*
— *PyUnicode_Type*
— *PyUnicode_WriteChar()*
— *PyVarObject*
— PyVarObject.ob_base
— *PyVarObject.ob_size*
— *PyVectorcall_Call()*
— *PyVectorcall_NARGS()*
— PyWeakReference

— *PyWeakref_GetObject()*
— *PyWeakref_GetRef()*
— *PyWeakref_NewProxy()*
— *PyWeakref_NewRef()*
— PyWrapperDescr_Type
— *PyWrapper_New()*
— PyZip_Type
— *Py_AddPendingCall()*
— *Py_AtExit()*
— *Py_BEGIN_ALLOW_THREADS*
— *Py_BLOCK_THREADS*
— *Py_BuildValue()*
— *Py_BytesMain()*
— *Py_CompileString()*
— *Py_DecRef()*
— *Py_DecodeLocale()*
— *Py_END_ALLOW_THREADS*
— *Py_EncodeLocale()*
— *Py_EndInterpreter()*
— *Py_EnterRecursiveCall()*
— *Py_Exit()*
— *Py_FatalError()*
— Py_FileSystemDefaultEncodeErrors
— Py_FileSystemDefaultEncoding
— *Py_Finalize()*
— *Py_FinalizeEx()*
— *Py_GenericAlias()*
— *Py_GenericAliasType*
— *Py_GetBuildInfo()*
— *Py_GetCompiler()*
— *Py_GetConstant()*
— *Py_GetConstantBorrowed()*
— *Py_GetCopyright()*
— *Py_GetExecPrefix()*
— *Py_GetPath()*
— *Py_GetPlatform()*
— *Py_GetPrefix()*
— *Py_GetProgramFullPath()*
— *Py_GetProgramName()*
— *Py_GetPythonHome()*
— Py_GetRecursionLimit()
— *Py_GetVersion()*
— Py_HasFileSystemDefaultEncoding
— *Py_IncRef()*
— *Py_Initialize()*
— *Py_InitializeEx()*
— *Py_Is()*
— *Py_IsFalse()*
— *Py_IsFinalizing()*
— *Py_IsInitialized()*
— *Py_IsNone()*
— *Py_IsTrue()*
— *Py_LeaveRecursiveCall()*
— *Py_Main()*
— Py_MakePendingCalls()
— *Py_NewInterpreter()*
— *Py_NewRef()*
— *Py_PACK_FULL_VERSION()*

— *Py_PACK_VERSION()*
— *Py_REFCNT()*
— *Py_ReprEnter()*
— *Py_ReprLeave()*
— *Py_SetProgramName()*
— *Py_SetPythonHome()*
— Py_SetRecursionLimit()
— *Py_TYPE()*
— *Py_UCS4*
— *Py_UNBLOCK_THREADS*
— Py_UTF8Mode
— *Py_VaBuildValue()*
— *Py_Version*
— *Py_XNewRef()*
— *Py_buffer*
— Py_intptr_t
— *Py_ssize_t*
— Py_uintptr_t
— *allocfunc*
— *binaryfunc*
— *descrgetfunc*
— *descrsetfunc*
— *destructor*
— *getattrfunc*
— *getattrofunc*
— *getbufferproc*
— *getiterfunc*
— *getter*
— *hashfunc*
— *initproc*
— *inquiry*
— *iternextfunc*
— *lenfunc*
— *newfunc*
— *objobjargproc*
— *objobjproc*
— *releasebufferproc*
— *reprfunc*
— *richcmpfunc*
— *setattrfunc*
— *setattrofunc*
— *setter*
— *ssizeargfunc*
— *ssizeobjargproc*
— ssizessizeargfunc
— ssizessizeobjargproc
— symtable
— *ternaryfunc*
— *traverseproc*
— *unaryfunc*
— *vectorcallfunc*
— *visitproc*

CHAPITRE 3

The Very High Level Layer

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Note

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

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

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

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

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

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

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

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

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

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

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

Part of the Stable ABI. Can be set to point to a function with the prototype `int func(void)`. The function will be called when Python's interpreter prompt is about to become idle and wait for user input from the terminal. The return value is ignored. Overriding this hook can be used to integrate the interpreter's prompt with other event loops, as done in the `Modules/_tkinter.c` in the Python source code.

Modifié dans la version 3.12 : This function is only called from the *main interpreter*.

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

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

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

Modifié dans la version 3.4 : The result must be allocated by *PyMem_RawMalloc()* or *PyMem_RawRealloc()*, instead of being allocated by *PyMem_Malloc()* or *PyMem_Realloc()*.

Modifié dans la version 3.12 : This function is only called from the *main interpreter*.

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

Valeur de retour : nouvelle référence. This is a simplified interface to `PyRun_StringFlags()` below, leaving *flags* set to `NULL`.

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

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

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

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

Valeur de retour : nouvelle référence. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *closeit* set to `0` and *flags* set to `NULL`.

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

Valeur de retour : nouvelle référence. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *flags* set to `NULL`.

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

Valeur de retour : nouvelle référence. This is a simplified interface to `PyRun_FileExFlags()` below, leaving *closeit* set to `0`.

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

Valeur de retour : nouvelle référence. Similar to `PyRun_StringFlags()`, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file, it is decoded from the *filesystem encoding and error handler*. If *closeit* is true, the file is closed before `PyRun_FileExFlags()` returns.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. This is a simplified interface to `Py_CompileStringFlags()` below, leaving *flags* set to `NULL`.

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

Valeur de retour : nouvelle référence. This is a simplified interface to `Py_CompileStringExFlags()` below, with *optimize* set to `-1`.

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

Valeur de retour : nouvelle référence. Parse and compile the Python source code in *str*, returning the resulting code object. The start token is given by *start* ; this can be used to constrain the code which can be compiled and should be `Py_eval_input`, `Py_file_input`, or `Py_single_input`. The filename specified by *filename* is used to construct the code object and may appear in tracebacks or `SyntaxError` exception messages. This returns `NULL` if the code cannot be parsed or compiled.

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

Ajouté dans la version 3.4.

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

Valeur de retour : nouvelle référence. Like `Py_CompileStringObject()`, but *filename* is a byte string decoded from the *filesystem encoding and error handler*.

Ajouté dans la version 3.2.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. This is a simplified interface to *PyEval_EvalCodeEx()*, with just the code object, and global and local variables. The other arguments are set to NULL.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Evaluate a precompiled code object, given a particular environment for its evaluation. This environment consists of a dictionary of global variables, a mapping object of local variables, arrays of arguments, keywords and defaults, a dictionary of default values for *keyword-only* arguments and a closure tuple of cells.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Evaluate an execution frame. This is a simplified interface to *PyEval_EvalFrameEx()*, for backward compatibility.

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

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

Modifié dans la version 3.4 : Cette fonction inclut maintenant une assertion de débogage afin d'assurer qu'elle ne passe pas sous silence une exception active.

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

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

int **Py_eval_input**

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

int **Py_file_input**

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

int **Py_single_input**

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

struct **PyCompilerFlags**

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

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

int **cf_flags**

Compiler flags.

int **cf_feature_version**

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

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

Modifié dans la version 3.8 : Added *cf_feature_version* field.

The available compiler flags are accessible as macros :

PyCF_ALLOW_TOP_LEVEL_AWAIT

PyCF_ONLY_AST

PyCF_OPTIMIZED_AST

PyCF_TYPE_COMMENTS

See compiler flags in documentation of the `ast` Python module, which exports these constants under the same names.

The "`PyCF`" flags above can be combined with "`CO_FUTURE`" flags such as *`CO_FUTURE_ANNOTATIONS`* to enable features normally selectable using future statements. See *Code Object Flags* for a complete list.

CHAPITRE 4

Reference Counting

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

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

Part of the Stable ABI *Depuis la version 3.14.* Get the reference count of the Python object *o*.

Note that the returned value may not actually reflect how many references to the object are actually held. For example, some objects are *immortal* and have a very high refcount that does not reflect the actual number of references. Consequently, do not rely on the returned value to be accurate, other than a value of 0 or 1.

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

> **Note**
>
> On *free threaded* builds of Python, returning 1 isn't sufficient to determine if it's safe to treat *o* as having no access by other threads. Use `PyUnstable_Object_IsUniquelyReferenced()` for that instead.
>
> See also the function `PyUnstable_Object_IsUniqueReferencedTemporary()`.

Modifié dans la version 3.10 : `Py_REFCNT()` is changed to the inline static function.

Modifié dans la version 3.11 : The parameter type is no longer `const PyObject*`.

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

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

On Python build with Free Threading, if *refcnt* is larger than `UINT32_MAX`, the object is made *immortal*.

This function has no effect on *immortal* objects.

Ajouté dans la version 3.9.

Modifié dans la version 3.12 : Immortal objects are not modified.

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

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

This function has no effect on *immortal* objects.

Cette fonction est souvent utilisée pour convertir une *référence empruntée* en une *référence forte* *sur place*. La fonction `Py_NewRef()` peut être utilisée pour créer une nouvelle *référence forte*.

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

L'objet ne doit pas être `NULL`, la fonction *Py_XINCREF()* doit être utilisée s'il est possible qu'il soit `NULL`.

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

Modifié dans la version 3.12 : Immortal objects are not modified.

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

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

Voir aussi *Py_XNewRef()*.

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

Part of the Stable ABI *Depuis la version 3.10.* Create a new *strong reference* to an object : call *Py_INCREF()* on *o* and return the object *o*.

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

L'objet *o* ne doit pas être `NULL` et la fonction *Py_XNewRef()* doit être utilisée si *o* peut être `NULL`.

Par exemple :

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

peut s'écrire :

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

Voir aussi *Py_INCREF()*.

Ajouté dans la version 3.10.

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

Part of the Stable ABI *Depuis la version 3.10.* Semblable à *Py_NewRef()* mais l'objet *o* peut être `NULL`.

Cette fonction renvoie `NULL` si l'objet *o* est `NULL`.

Ajouté dans la version 3.10.

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

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

This function has no effect on *immortal* objects.

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

Cette fonction est généralement utilisée pour supprimer une *référence forte* avant qu'elle ne soit plus accessible.

L'objet en argument ne doit pas être `NULL`. *Py_XDECREF()* doit être utilisée si l'objet peut être `NULL`.

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

Avertissement

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

Modifié dans la version 3.12 : Immortal objects are not modified.

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

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

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

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

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

Modifié dans la version 3.12 : The macro argument is now only evaluated once. If the argument has side effects, these are no longer duplicated.

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

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

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

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

Py_SETREF(dst, src)

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

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

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

The safe way is :

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

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

Ajouté dans la version 3.6.

Modifié dans la version 3.12 : The macro arguments are now only evaluated once. If an argument has side effects, these are no longer duplicated.

Py_XSETREF(dst, src)

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

Ajouté dans la version 3.6.

Modifié dans la version 3.12 : The macro arguments are now only evaluated once. If an argument has side effects, these are no longer duplicated.

CHAPITRE 5

Gestion des exceptions

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

Concretely, the error indicator consists of three object pointers : the exception's type, the exception's value, and the traceback object. Any of those pointers can be `NULL` if non-set (although some combinations are forbidden, for example you can't have a non-`NULL` traceback if the exception type is `NULL`).

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

Note

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

5.1 Printing and clearing

void **`PyErr_Clear`**()

Part of the Stable ABI. Clear the error indicator. If the error indicator is not set, there is no effect.

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

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

Call this function **only** when the error indicator is set. Otherwise it will cause a fatal error !

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

Modifié dans la version 3.12 : The setting of `sys.last_exc` was added.

void **PyErr_Print**()

Part of the Stable ABI. Alias for `PyErr_PrintEx(1)`.

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

Part of the Stable ABI. Call `sys.unraisablehook()` using the current exception and *obj* argument.

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

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

An exception must be set when calling this function.

Modifié dans la version 3.4 : Print a traceback. Print only traceback if *obj* is `NULL`.

Modifié dans la version 3.8 : Use `sys.unraisablehook()`.

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

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

Ajouté dans la version 3.13.

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

Part of the Stable ABI *Depuis la version 3.12.* Print the standard traceback display of `exc` to `sys.stderr`, including chained exceptions and notes.

Ajouté dans la version 3.12.

5.2 Lever des exceptions

These functions help you set the current thread's error indicator. For convenience, some of these functions will always return a `NULL` pointer for use in a `return` statement.

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

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

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

Part of the Stable ABI. This function is similar to *PyErr_SetString()* but lets you specify an arbitrary Python object for the "value" of the exception.

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

Valeur de retour : toujours NULL. Part of the Stable ABI. This function sets the error indicator and returns `NULL`. *exception* should be a Python exception class. The *format* and subsequent parameters help format the error message; they have the same meaning and values as in *PyUnicode_FromFormat()*. *format* is an ASCII-encoded string.

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

Valeur de retour : toujours NULL. Part of the Stable ABI *Depuis la version 3.5.* Same as *PyErr_Format()*, but taking a `va_list` argument rather than a variable number of arguments.

Ajouté dans la version 3.5.

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

Part of the Stable ABI. This is a shorthand for `PyErr_SetObject(type, Py_None)`.

int **PyErr_BadArgument**()

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

PyObject ***PyErr_NoMemory**()

Valeur de retour : toujours NULL. Part of the Stable ABI. This is a shorthand for `PyErr_SetNone(PyExc_MemoryError)`; it returns `NULL` so an object allocation function can write `return PyErr_NoMemory();` when it runs out of memory.

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

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

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

Valeur de retour : toujours NULL. Part of the Stable ABI. Similar to *`PyErr_SetFromErrno()`*, with the additional behavior that if *filenameObject* is not `NULL`, it is passed to the constructor of *type* as a third parameter. In the case of `OSError` exception, this is used to define the `filename` attribute of the exception instance.

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

Valeur de retour : toujours NULL. Part of the Stable ABI *Depuis la version 3.7.* Similar to *`PyErr_SetFromErrnoWithFilenameObject()`*, but takes a second filename object, for raising errors when a function that takes two filenames fails.

Ajouté dans la version 3.4.

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

Valeur de retour : toujours NULL. Part of the Stable ABI. Similar to *`PyErr_SetFromErrnoWithFilenameObject()`*, but the filename is given as a C string. *filename* is decoded from the *filesystem encoding and error handler*.

PyObject ***PyErr_SetFromWindowsErr**(int ierr)

Valeur de retour : toujours NULL. Part of the Stable ABI *on Windows Depuis la version 3.7.* This is a convenience function to raise `OSError`. If called with *ierr* of `0`, the error code returned by a call to `GetLastError()` is used instead. It calls the Win32 function `FormatMessage()` to retrieve the Windows description of error code given by *ierr* or `GetLastError()`, then it constructs a `OSError` object with the `winerror` attribute set to the error code, the `strerror` attribute set to the corresponding error message (gotten from `FormatMessage()`), and then calls `PyErr_SetObject(PyExc_OSError, object)`. This function always returns `NULL`.

Availability : Windows.

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

Valeur de retour : toujours NULL. Part of the Stable ABI *on Windows Depuis la version 3.7.* Similar to *`PyErr_SetFromWindowsErr()`*, with an additional parameter specifying the exception type to be raised.

Availability : Windows.

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

Valeur de retour : toujours NULL. Part of the Stable ABI *on Windows Depuis la version 3.7.* Similar to *`PyErr_SetFromWindowsErr()`*, with the additional behavior that if *filename* is not `NULL`, it is decoded from the filesystem encoding (`os.fsdecode()`) and passed to the constructor of `OSError` as a third parameter to be used to define the `filename` attribute of the exception instance.

Availability : Windows.

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

Valeur de retour : toujours NULL. Part of the Stable ABI *on Windows Depuis la version 3.7.* Similar to `PyErr_SetExcFromWindowsErr()`, with the additional behavior that if *filename* is not `NULL`, it is passed to the constructor of `OSError` as a third parameter to be used to define the `filename` attribute of the exception instance.

Availability : Windows.

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

Valeur de retour : toujours NULL. Part of the Stable ABI *on Windows Depuis la version 3.7.* Similar to `PyErr_SetExcFromWindowsErrWithFilenameObject()`, but accepts a second filename object.

Availability : Windows.

Ajouté dans la version 3.4.

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

Valeur de retour : toujours NULL. Part of the Stable ABI *on Windows Depuis la version 3.7.* Similar to `PyErr_SetFromWindowsErrWithFilename()`, with an additional parameter specifying the exception type to be raised.

Availability : Windows.

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

Valeur de retour : toujours NULL. Part of the Stable ABI *Depuis la version 3.7.* This is a convenience function to raise `ImportError`. *msg* will be set as the exception's message string. *name* and *path*, both of which can be `NULL`, will be set as the `ImportError`'s respective `name` and `path` attributes.

Ajouté dans la version 3.3.

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

Valeur de retour : toujours NULL. Part of the Stable ABI *Depuis la version 3.6.* Much like `PyErr_SetImportError()` but this function allows for specifying a subclass of `ImportError` to raise.

Ajouté dans la version 3.6.

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

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

Ajouté dans la version 3.4.

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

Part of the Stable ABI *Depuis la version 3.7.* Like `PyErr_SyntaxLocationObject()`, but *filename* is a byte string decoded from the *filesystem encoding and error handler*.

Ajouté dans la version 3.2.

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

Part of the Stable ABI. Like `PyErr_SyntaxLocationEx()`, but the *col_offset* parameter is omitted.

void **PyErr_BadInternalCall**()

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

5.3 Issuing warnings

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

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

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

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

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

int **PyErr_WarnExplicitObject**(*PyObject* *category, *PyObject* *message, *PyObject* *filename, int lineno, *PyObject* *module, *PyObject* *registry)

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

Ajouté dans la version 3.4.

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

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

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

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

Ajouté dans la version 3.2.

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

Part of the Stable ABI *Depuis la version 3.6.* Function similar to `PyErr_WarnFormat()`, but *category* is `ResourceWarning` and it passes *source* to `warnings.WarningMessage`.

Ajouté dans la version 3.6.

5.4 Querying the error indicator

PyObject ***PyErr_Occurred**()

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

The caller must have an *attached thread state*.

Note

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

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

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

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

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

PyObject ***PyErr_GetRaisedException**(void)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.12.* Return the exception currently being raised, clearing the error indicator at the same time. Return `NULL` if the error indicator is not set.

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

Par exemple :

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

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

   PyErr_SetRaisedException(exc);
}
```

Voir aussi

PyErr_GetHandledException(), to save the exception currently being handled.

Ajouté dans la version 3.12.

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

Part of the Stable ABI *Depuis la version 3.12.* Set *exc* as the exception currently being raised, clearing the existing exception if one is set.

Avertissement

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

Ajouté dans la version 3.12.

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

Part of the Stable ABI. Obsolète depuis la version 3.12 : Use *PyErr_GetRaisedException()* instead.

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

Note

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

Par exemple :

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

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

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

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

Part of the Stable ABI. Obsolète depuis la version 3.12 : Use `PyErr_SetRaisedException()` instead.

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

Note

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

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

Part of the Stable ABI. Obsolète depuis la version 3.12 : Use `PyErr_GetRaisedException()` instead, to avoid any possible de-normalization.

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

Note

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

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

PyObject ***PyErr_GetHandledException**(void)

Part of the Stable ABI *Depuis la version 3.11.* Retrieve the active exception instance, as would be returned by `sys.exception()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. Returns a new reference to the exception or `NULL`. Does not modify the interpreter's exception state.

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

Ajouté dans la version 3.11.

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

Part of the Stable ABI *Depuis la version 3.11.* Set the active exception, as known from `sys.exception()`. This refers to an exception that was *already caught*, not to an exception that was freshly raised. To clear the exception state, pass `NULL`.

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

Ajouté dans la version 3.11.

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

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

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

Ajouté dans la version 3.3.

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

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

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

Ajouté dans la version 3.3.

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

5.5 Traitement des signaux

int **PyErr_CheckSignals**()

Part of the Stable ABI. This function interacts with Python's signal handling.

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

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

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

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

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

void **PyErr_SetInterrupt**()

Part of the Stable ABI. Simulate the effect of a `SIGINT` signal arriving. This is equivalent to `PyErr_SetInterruptEx(SIGINT)`.

> **Note**
>
> This function is async-signal-safe. It can be called without an *attached thread state* and from a C signal handler.

int **PyErr_SetInterruptEx**(int signum)

Part of the Stable ABI *Depuis la version 3.10.* Simulate the effect of a signal arriving. The next time *PyErr_CheckSignals()* is called, the Python signal handler for the given signal number will be called.

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

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

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

> **Note**
>
> This function is async-signal-safe. It can be called without an *attached thread state* and from a C signal handler.

Ajouté dans la version 3.10.

int **PySignal_SetWakeupFd**(int fd)

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

The value `-1` disables the feature; this is the initial state. This is equivalent to `signal.set_wakeup_fd()` in Python, but without any error checking. *fd* should be a valid file descriptor. The function should only be

called from the main thread.

Modifié dans la version 3.5 : On Windows, the function now also supports socket handles.

5.6 Exception Classes

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. This utility function creates and returns a new exception class. The *name* argument must be the name of the new exception, a C string of the form `module.classname`. The *base* and *dict* arguments are normally `NULL`. This creates a class object derived from `Exception` (accessible in C as `PyExc_Exception`).

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

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Same as `PyErr_NewException()`, except that the new exception class can easily be given a docstring : If *doc* is non-`NULL`, it will be used as the docstring for the exception class.

Ajouté dans la version 3.2.

int **PyExceptionClass_Check**(*PyObject* *ob)

Return non-zero if *ob* is an exception class, zero otherwise. This function always succeeds.

const char ***PyExceptionClass_Name**(*PyObject* *ob)

Part of the Stable ABI *Depuis la version 3.8.* Return `tp_name` of the exception class *ob*.

5.7 Objets exception

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the traceback associated with the exception as a new reference, as accessible from Python through the `__traceback__` attribute. If there is no traceback associated, this returns `NULL`.

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

Part of the Stable ABI. Set the traceback associated with the exception to *tb*. Use `Py_None` to clear it.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the context (another exception instance during whose handling *ex* was raised) associated with the exception as a new reference, as accessible from Python through the `__context__` attribute. If there is no context associated, this returns `NULL`.

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

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

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the cause (either an exception instance, or `None`, set by `raise ... from ...`) associated with the exception as a new reference, as accessible from Python through the `__cause__` attribute.

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

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

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

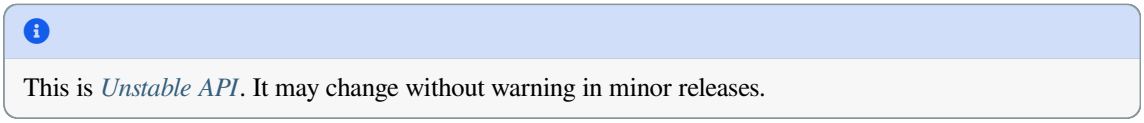
PyObject ***PyException_GetArgs**(*PyObject* *ex)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.12.* Return `args` of exception *ex*.

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

Part of the Stable ABI *Depuis la version 3.12.* Set `args` of exception *ex* to *args*.

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



This is *Unstable API*. It may change without warning in minor releases.

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

Ajouté dans la version 3.12.

5.8 Objets exception Unicode

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

PyObject ***PyUnicodeDecodeError_Create**(const char *encoding, const char *object, *Py_ssize_t* length, *Py_ssize_t* start, *Py_ssize_t* end, const char *reason)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a `UnicodeDecodeError` object with the attributes *encoding*, *object*, *length*, *start*, *end* and *reason*. *encoding* and *reason* are UTF-8 encoded strings.

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

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the *encoding* attribute of the given exception object.

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

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

PyObject ***PyUnicodeTranslateError_GetObject**(*PyObject* *exc)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the *object* attribute of the given exception object.

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

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

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

Part of the Stable ABI. Get the *start* attribute of the given exception object and place it into **start*. *start* must not be `NULL`. Return `0` on success, `-1` on failure.

If the `UnicodeError.object` is an empty sequence, the resulting *start* is `0`. Otherwise, it is clipped to `[0, len(object) - 1]`.

Voir aussi

`UnicodeError.start`

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

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

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

Part of the Stable ABI. Set the *start* attribute of the given exception object to *start*. Return `0` on success, `-1` on failure.

> **Note**
>
> While passing a negative *start* does not raise an exception, the corresponding getters will not consider it as a relative offset.

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

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

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

Part of the Stable ABI. Get the *end* attribute of the given exception object and place it into **end*. *end* must not be `NULL`. Return `0` on success, `-1` on failure.

If the `UnicodeError.object` is an empty sequence, the resulting *end* is `0`. Otherwise, it is clipped to `[1, len(object)]`.

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

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

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

Part of the Stable ABI. Set the *end* attribute of the given exception object to *end*. Return `0` on success, `-1` on failure.

> **Voir aussi**
>
> `UnicodeError.end`

PyObject ***PyUnicodeDecodeError_GetReason**(*PyObject* *exc)

PyObject ***PyUnicodeEncodeError_GetReason**(*PyObject* *exc)

PyObject ***PyUnicodeTranslateError_GetReason**(*PyObject* *exc)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the *reason* attribute of the given exception object.

int **PyUnicodeDecodeError_SetReason**(*PyObject* *exc, const char *reason)

int **PyUnicodeEncodeError_SetReason**(*PyObject* *exc, const char *reason)

int **PyUnicodeTranslateError_SetReason**(*PyObject* *exc, const char *reason)

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

5.9 Contrôle de la récursion

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

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

Part of the Stable ABI *Depuis la version 3.9.* Marks a point where a recursive C-level call is about to be performed.

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

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

Modifié dans la version 3.9 : This function is now also available in the *limited API*.

void **Py_LeaveRecursiveCall**(void)

Part of the Stable ABI *Depuis la version 3.9.* Ends a `Py_EnterRecursiveCall()`. Must be called once for each *successful* invocation of `Py_EnterRecursiveCall()`.

Modifié dans la version 3.9 : This function is now also available in the *limited API*.

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

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

Part of the Stable ABI. Called at the beginning of the `tp_repr` implementation to detect cycles.

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

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

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

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

Part of the Stable ABI. Ends a `Py_ReprEnter()`. Must be called once for each invocation of `Py_ReprEnter()` that returns zero.

5.10 Exception and warning types

All standard Python exceptions and warning categories are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type `PyObject*`; they are all class objects.

For completeness, here are all the variables :

5.10.1 Exception types

C name	Python name
PyObject ***PyExc_BaseException** *Part of the* Stable ABI.	`BaseException`
PyObject ***PyExc_BaseExceptionGroup** *Part of the* Stable ABI *Depuis la version 3.11.*	`BaseExceptionGroup`
PyObject ***PyExc_Exception** *Part of the* Stable ABI.	`Exception`
PyObject ***PyExc_ArithmeticError** *Part of the* Stable ABI.	`ArithmeticError`
PyObject ***PyExc_AssertionError** *Part of the* Stable ABI.	`AssertionError`

suite sur la page suivante

Tableau 1 – suite de la page précédente

C name	Python name
PyObject ***PyExc_AttributeError** *Part of the* Stable ABI.	`AttributeError`
PyObject ***PyExc_BlockingIOError** *Part of the* Stable ABI *Depuis la version 3.7.*	`BlockingIOError`
PyObject ***PyExc_BrokenPipeError** *Part of the* Stable ABI *Depuis la version 3.7.*	`BrokenPipeError`
PyObject ***PyExc_BufferError** *Part of the* Stable ABI.	`BufferError`
PyObject ***PyExc_ChildProcessError** *Part of the* Stable ABI *Depuis la version 3.7.*	`ChildProcessError`
PyObject ***PyExc_ConnectionAbortedError** *Part of the* Stable ABI *Depuis la version 3.7.*	`ConnectionAbortedError`
PyObject ***PyExc_ConnectionError** *Part of the* Stable ABI *Depuis la version 3.7.*	`ConnectionError`
PyObject ***PyExc_ConnectionRefusedError** *Part of the* Stable ABI *Depuis la version 3.7.*	`ConnectionRefusedError`
PyObject ***PyExc_ConnectionResetError** *Part of the* Stable ABI *Depuis la version 3.7.*	`ConnectionResetError`
PyObject ***PyExc_EOFError** *Part of the* Stable ABI.	`EOFError`
PyObject ***PyExc_FileExistsError** *Part of the* Stable ABI *Depuis la version 3.7.*	`FileExistsError`
PyObject ***PyExc_FileNotFoundError** *Part of the* Stable ABI *Depuis la version 3.7.*	`FileNotFoundError`
PyObject ***PyExc_FloatingPointError** *Part of the* Stable ABI.	`FloatingPointError`

suite sur la page suivante

Tableau 1 – suite de la page précédente

C name	Python name
PyObject ***PyExc_GeneratorExit** *Part of the* Stable ABI.	`GeneratorExit`
PyObject ***PyExc_ImportError** *Part of the* Stable ABI.	`ImportError`
PyObject ***PyExc_IndentationError** *Part of the* Stable ABI.	`IndentationError`
PyObject ***PyExc_IndexError** *Part of the* Stable ABI.	`IndexError`
PyObject ***PyExc_InterruptedError** *Part of the* Stable ABI *Depuis la version 3.7.*	`InterruptedError`
PyObject ***PyExc_IsADirectoryError** *Part of the* Stable ABI *Depuis la version 3.7.*	`IsADirectoryError`
PyObject ***PyExc_KeyError** *Part of the* Stable ABI.	`KeyError`
PyObject ***PyExc_KeyboardInterrupt** *Part of the* Stable ABI.	`KeyboardInterrupt`
PyObject ***PyExc_LookupError** *Part of the* Stable ABI.	`LookupError`
PyObject ***PyExc_MemoryError** *Part of the* Stable ABI.	`MemoryError`
PyObject ***PyExc_ModuleNotFoundError** *Part of the* Stable ABI *Depuis la version 3.6.*	`ModuleNotFoundError`
PyObject ***PyExc_NameError** *Part of the* Stable ABI.	`NameError`
PyObject ***PyExc_NotADirectoryError** *Part of the* Stable ABI *Depuis la version 3.7.*	`NotADirectoryError`

suite sur la page suivante

Tableau 1 – suite de la page précédente

C name	Python name
PyObject ***PyExc_NotImplementedError** *Part of the* Stable ABI.	`NotImplementedError`
PyObject ***PyExc_OSError** *Part of the* Stable ABI.	`OSError`
PyObject ***PyExc_OverflowError** *Part of the* Stable ABI.	`OverflowError`
PyObject ***PyExc_PermissionError** *Part of the* Stable ABI *Depuis la version 3.7.*	`PermissionError`
PyObject ***PyExc_ProcessLookupError** *Part of the* Stable ABI *Depuis la version 3.7.*	`ProcessLookupError`
PyObject ***PyExc_PythonFinalizationError**	`PythonFinalizationError`
PyObject ***PyExc_RecursionError** *Part of the* Stable ABI *Depuis la version 3.7.*	`RecursionError`
PyObject ***PyExc_ReferenceError** *Part of the* Stable ABI.	`ReferenceError`
PyObject ***PyExc_RuntimeError** *Part of the* Stable ABI.	`RuntimeError`
PyObject ***PyExc_StopAsyncIteration** *Part of the* Stable ABI *Depuis la version 3.7.*	`StopAsyncIteration`
PyObject ***PyExc_StopIteration** *Part of the* Stable ABI.	`StopIteration`
PyObject ***PyExc_SyntaxError** *Part of the* Stable ABI.	`SyntaxError`
PyObject ***PyExc_SystemError** *Part of the* Stable ABI.	`SystemError`
PyObject ***PyExc_SystemExit** *Part of the* Stable ABI.	`SystemExit`

suite sur la page suivante

Tableau 1 – suite de la page précédente

C name	Python name
PyObject ***PyExc_TabError** *Part of the* Stable ABI.	`TabError`
PyObject ***PyExc_TimeoutError** *Part of the* Stable ABI *Depuis la version 3.7.*	`TimeoutError`
PyObject ***PyExc_TypeError** *Part of the* Stable ABI.	`TypeError`
PyObject ***PyExc_UnboundLocalError** *Part of the* Stable ABI.	`UnboundLocalError`
PyObject ***PyExc_UnicodeDecodeError** *Part of the* Stable ABI.	`UnicodeDecodeError`
PyObject ***PyExc_UnicodeEncodeError** *Part of the* Stable ABI.	`UnicodeEncodeError`
PyObject ***PyExc_UnicodeError** *Part of the* Stable ABI.	`UnicodeError`
PyObject ***PyExc_UnicodeTranslateError** *Part of the* Stable ABI.	`UnicodeTranslateError`
PyObject ***PyExc_ValueError** *Part of the* Stable ABI.	`ValueError`
PyObject ***PyExc_ZeroDivisionError** *Part of the* Stable ABI.	`ZeroDivisionError`

Ajouté dans la version 3.3 : *PyExc_BlockingIOError*, *PyExc_BrokenPipeError*, *PyExc_ChildProcessError*, *PyExc_ConnectionError*, *PyExc_ConnectionAbortedError*, *PyExc_ConnectionRefusedError*, *PyExc_ConnectionResetError*, *PyExc_FileExistsError*, *PyExc_FileNotFoundError*, *PyExc_InterruptedError*, *PyExc_IsADirectoryError*, *PyExc_NotADirectoryError*, *PyExc_PermissionError*, *PyExc_ProcessLookupError* and *PyExc_TimeoutError* were introduced following **PEP 3151**.

Ajouté dans la version 3.5 : *PyExc_StopAsyncIteration* et *PyExc_RecursionError*.

Ajouté dans la version 3.6 : *PyExc_ModuleNotFoundError*.

Ajouté dans la version 3.11 : *PyExc_BaseExceptionGroup*.

5.10.2 OSError aliases

The following are a compatibility aliases to *PyExc_OSError*.

Modifié dans la version 3.3 : These aliases used to be separate exception types.

C name	Python name	Notes
PyObject ***PyExc_EnvironmentError** *Part of the* Stable ABI.	`OSError`	
PyObject ***PyExc_IOError** *Part of the* Stable ABI.	`OSError`	
PyObject ***PyExc_WindowsError** *Part of the* Stable ABI *on Windows Depuis la version 3.7.*	`OSError`	[win]

Notes :

5.10.3 Warning types

C name	Python name
PyObject ***PyExc_Warning** *Part of the* Stable ABI.	`Warning`
PyObject ***PyExc_BytesWarning** *Part of the* Stable ABI.	`BytesWarning`
PyObject ***PyExc_DeprecationWarning** *Part of the* Stable ABI.	`DeprecationWarning`
PyObject ***PyExc_EncodingWarning** *Part of the* Stable ABI *Depuis la version 3.10.*	`EncodingWarning`
PyObject ***PyExc_FutureWarning** *Part of the* Stable ABI.	`FutureWarning`
PyObject ***PyExc_ImportWarning** *Part of the* Stable ABI.	`ImportWarning`
PyObject ***PyExc_PendingDeprecationWarning** *Part of the* Stable ABI.	`PendingDeprecationWarning`
PyObject ***PyExc_ResourceWarning** *Part of the* Stable ABI *Depuis la version 3.7.*	`ResourceWarning`
PyObject ***PyExc_RuntimeWarning** *Part of the* Stable ABI.	`RuntimeWarning`
PyObject ***PyExc_SyntaxWarning** *Part of the* Stable ABI.	`SyntaxWarning`
PyObject ***PyExc_UnicodeWarning** *Part of the* Stable ABI.	`UnicodeWarning`
PyObject ***PyExc_UserWarning** *Part of the* Stable ABI.	`UserWarning`

Ajouté dans la version 3.2 : *PyExc_ResourceWarning*.

Ajouté dans la version 3.10 : *PyExc_EncodingWarning*.

CHAPITRE 6

Defining extension modules

A C extension for CPython is a shared library (for example, a `.so` file on Linux, `.pyd` DLL on Windows), which is loadable into the Python process (for example, it is compiled with compatible compiler settings), and which exports an *initialization function*.

To be importable by default (that is, by `importlib.machinery.ExtensionFileLoader`), the shared library must be available on `sys.path`, and must be named after the module name plus an extension listed in `importlib.machinery.EXTENSION_SUFFIXES`.

Note

Building, packaging and distributing extension modules is best done with third-party tools, and is out of scope of this document. One suitable tool is Setuptools, whose documentation can be found at https://setuptools.pypa.io/en/latest/setuptools.html.

Normally, the initialization function returns a module definition initialized using *`PyModuleDef_Init()`*. This allows splitting the creation process into several phases :

- Before any substantial code is executed, Python can determine which capabilities the module supports, and it can adjust the environment or refuse loading an incompatible extension.
- By default, Python itself creates the module object -- that is, it does the equivalent of `object.__new__()` for classes. It also sets initial attributes like `__package__` and `__loader__`.
- Afterwards, the module object is initialized using extension-specific code -- the equivalent of `__init__()` on classes.

This is called *multi-phase initialization* to distinguish it from the legacy (but still supported) *single-phase initialization* scheme, where the initialization function returns a fully constructed module. See the *single-phase-initialization section below* for details.

Modifié dans la version 3.5 : Added support for multi-phase initialization (**PEP 489**).

6.1 Multiple module instances

By default, extension modules are not singletons. For example, if the `sys.modules` entry is removed and the module is re-imported, a new module object is created, and typically populated with fresh method and type objects. The old module is subject to normal garbage collection. This mirrors the behavior of pure-Python modules.

Additional module instances may be created in *sub-interpreters* or after Python runtime reinitialization (*`Py_Finalize()`* and *`Py_Initialize()`*). In these cases, sharing Python objects between module instances would

likely cause crashes or undefined behavior.

To avoid such issues, each instance of an extension module should be *isolated* : changes to one instance should not implicitly affect the others, and all state owned by the module, including references to Python objects, should be specific to a particular module instance. See isolating-extensions-howto for more details and a practical guide.

A simpler way to avoid these issues is raising an error on repeated initialization.

All modules are expected to support *sub-interpreters*, or otherwise explicitly signal a lack of support. This is usually achieved by isolation or blocking repeated initialization, as above. A module may also be limited to the main interpreter using the *Py_mod_multiple_interpreters* slot.

6.2 Initialization function

The initialization function defined by an extension module has the following signature :

PyObject ***PyInit_modulename**(void)

Its name should be `PyInit_<name>`, with `<name>` replaced by the name of the module.

For modules with ASCII-only names, the function must instead be named `PyInit_<name>`, with `<name>` replaced by the name of the module. When using *Multi-phase initialization*, non-ASCII module names are allowed. In this case, the initialization function name is `PyInitU_<name>`, with `<name>` encoded using Python's *punycode* encoding with hyphens replaced by underscores. In 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
```

It is recommended to define the initialization function using a helper macro :

PyMODINIT_FUNC

Declare an extension module initialization function. This macro :
— specifies the *PyObject** return type,
— adds any special linkage declarations required by the platform, and
— for C++, declares the function as `extern "C"`.

For example, a module called `spam` would be defined like this :

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

PyMODINIT_FUNC
PyInit_spam(void)
{
    return PyModuleDef_Init(&spam_module);
}
```

It is possible to export multiple modules from a single shared library by defining multiple initialization functions. However, importing them requires using symbolic links or a custom importer, because by default only the function corresponding to the filename is found. See the Multiple modules in one library section in **PEP 489** for details.

The initialization function is typically the only non-`static` item defined in the module's C source.

6.3 Multi-phase initialization

Normally, the *initialization function* (`PyInit_modulename`) returns a *`PyModuleDef`* instance with non-`NULL` *`m_slots`*. Before it is returned, the `PyModuleDef` instance must be initialized using the following function :

PyObject ***PyModuleDef_Init**(*PyModuleDef* *def)

Part of the Stable ABI *Depuis la version 3.5.* Ensure a module definition is a properly initialized Python object that correctly reports its type and a reference count.

Return *def* cast to `PyObject*`, or `NULL` if an error occurred.

Calling this function is required for *Multi-phase initialization*. It should not be used in other contexts.

Note that Python assumes that `PyModuleDef` structures are statically allocated. This function may return either a new reference or a borrowed one ; this reference must not be released.

Ajouté dans la version 3.5.

6.4 Legacy single-phase initialization

Attention

Single-phase initialization is a legacy mechanism to initialize extension modules, with known drawbacks and design flaws. Extension module authors are encouraged to use multi-phase initialization instead.

In single-phase initialization, the *initialization function* (`PyInit_modulename`) should create, populate and return a module object. This is typically done using *`PyModule_Create()`* and functions like *`PyModule_AddObjectRef()`*.

Single-phase initialization differs from the *default* in the following ways :

— Single-phase modules are, or rather *contain*, “singletons”.

When the module is first initialized, Python saves the contents of the module’s `__dict__` (that is, typically, the module’s functions and types).

For subsequent imports, Python does not call the initialization function again. Instead, it creates a new module object with a new `__dict__`, and copies the saved contents to it. For example, given a single-phase module `_testsinglephase` [1] that defines a function `sum` and an exception class `error` :

```
>>> import sys
>>> import _testsinglephase as one
>>> del sys.modules['_testsinglephase']
>>> import _testsinglephase as two
>>> one is two
False
>>> one.__dict__ is two.__dict__
False
>>> one.sum is two.sum
True
>>> one.error is two.error
True
```

The exact behavior should be considered a CPython implementation detail.

— To work around the fact that `PyInit_modulename` does not take a *spec* argument, some state of the import machinery is saved and applied to the first suitable module created during the `PyInit_modulename` call. Specifically, when a sub-module is imported, this mechanism prepends the parent package name to the name of the module.

1. `_testsinglephase` is an internal module used in CPython’s self-test suite ; your installation may or may not include it.

A single-phase `PyInit_modulename` function should create "its" module object as soon as possible, before any other module objects can be created.

— Non-ASCII module names (`PyInitU_modulename`) are not supported.

— Single-phase modules support module lookup functions like *PyState_FindModule()*.

CHAPITRE 7

Utilitaires

Les fonctions de ce chapitre sont utilitaires, certaines aident à rendre le code en C plus portable, d'autres à utiliser des modules Python depuis du C, analyser des arguments de fonctions, ou encore construire des valeurs Python à partir de valeurs C.

7.1 Operating System Utilities

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

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.6.* Return the file system representation for *path*. If the object is a `str` or `bytes` object, then a new *strong reference* is returned. If the object implements the `os.PathLike` interface, then `__fspath__()` is returned as long as it is a `str` or `bytes` object. Otherwise `TypeError` is raised and `NULL` is returned.

Ajouté dans la version 3.6.

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

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

This function must not be called before Python is initialized.

void **PyOS_BeforeFork**()

Part of the Stable ABI *on platforms with fork() Depuis la version 3.7.* Function to prepare some internal state before a process fork. This should be called before calling `fork()` or any similar function that clones the current process. Only available on systems where `fork()` is defined.

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

Ajouté dans la version 3.7.

void **PyOS_AfterFork_Parent**()

Part of the Stable ABI *on platforms with fork() Depuis la version 3.7.* Function to update some internal state

after a process fork. This should be called from the parent process after calling `fork()` or any similar function that clones the current process, regardless of whether process cloning was successful. Only available on systems where `fork()` is defined.

Avertissement

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

Ajouté dans la version 3.7.

void **PyOS_AfterFork_Child**()

Part of the Stable ABI *on platforms with fork() Depuis la version 3.7.* Function to update internal interpreter state after a process fork. This must be called from the child process after calling `fork()`, or any similar function that clones the current process, if there is any chance the process will call back into the Python interpreter. Only available on systems where `fork()` is defined.

Avertissement

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

Ajouté dans la version 3.7.

Voir aussi

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

void **PyOS_AfterFork**()

Part of the Stable ABI *on platforms with fork().* Function to update some internal state after a process fork ; this should be called in the new process if the Python interpreter will continue to be used. If a new executable is loaded into the new process, this function does not need to be called.

Obsolète depuis la version 3.7 : This function is superseded by *`PyOS_AfterFork_Child()`*.

int **PyOS_CheckStack**()

Part of the Stable ABI *on platforms with USE_STACKCHECK Depuis la version 3.7.* Return true when the interpreter runs out of stack space. This is a reliable check, but is only available when `USE_STACKCHECK` is defined (currently on certain versions of Windows using the Microsoft Visual C++ compiler). `USE_STACKCHECK` will be defined automatically ; you should never change the definition in your own code.

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

Part of the Stable ABI.

PyOS_sighandler_t **PyOS_getsig**(int i)

Part of the Stable ABI. Return the current signal handler for signal *i*. This is a thin wrapper around either `sigaction()` or `signal()`. Do not call those functions directly !

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

Part of the Stable ABI. Set the signal handler for signal *i* to be *h* ; return the old signal handler. This is a thin wrapper around either `sigaction()` or `signal()`. Do not call those functions directly !

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

Part of the Stable ABI *Depuis la version 3.7.*

Avertissement

This function should not be called directly : use the `PyConfig` API with the `PyConfig_SetBytesString()` function which ensures that *Python is preinitialized*.

This function must not be called before *Python is preinitialized* and so that the LC_CTYPE locale is properly configured : see the `Py_PreInitialize()` function.

Decode a byte string from the *filesystem encoding and error handler*. If the error handler is surrogateescape error handler, undecodable bytes are decoded as characters in range U+DC80..U+DCFF; and if a byte sequence can be decoded as a surrogate character, the bytes are escaped using the surrogateescape error handler instead of decoding them.

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

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

The *filesystem encoding and error handler* are selected by `PyConfig_Read()` : see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

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

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

Voir aussi

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

Ajouté dans la version 3.5.

Modifié dans la version 3.7 : The function now uses the UTF-8 encoding in the Python UTF-8 Mode.

Modifié dans la version 3.8 : The function now uses the UTF-8 encoding on Windows if `PyPreConfig.legacy_windows_fs_encoding` is zero;

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

Part of the Stable ABI *Depuis la version 3.7.* Encode a wide character string to the *filesystem encoding and error handler*. If the error handler is surrogateescape error handler, surrogate characters in the range U+DC80..U+DCFF are converted to bytes 0x80..0xFF.

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

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

The *filesystem encoding and error handler* are selected by `PyConfig_Read()` : see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

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

Avertissement

This function must not be called before *Python is preinitialized* and so that the LC_CTYPE locale is properly configured : see the `Py_PreInitialize()` function.

> **Voir aussi**
>
> The *PyUnicode_EncodeFSDefault()* and *PyUnicode_EncodeLocale()* functions.

Ajouté dans la version 3.5.

Modifié dans la version 3.7 : The function now uses the UTF-8 encoding in the Python UTF-8 Mode.

Modifié dans la version 3.8 : The function now uses the UTF-8 encoding on Windows if *PyPreConfig.legacy_windows_fs_encoding* is zero.

FILE ***Py_fopen**(*PyObject* *path, const char *mode)

Similar to `fopen()`, but *path* is a Python object and an exception is set on error.

path must be a `str` object, a `bytes` object, or a *path-like object*.

On success, return the new file pointer. On error, set an exception and return `NULL`.

The file must be closed by *Py_fclose()* rather than calling directly `fclose()`.

The file descriptor is created non-inheritable (**PEP 446**).

The caller must have an *attached thread state*.

Ajouté dans la version 3.14.

int **Py_fclose**(FILE *file)

Close a file that was opened by *Py_fopen()*.

On success, return `0`. On error, return `EOF` and `errno` is set to indicate the error. In either case, any further access (including another call to *Py_fclose()*) to the stream results in undefined behavior.

Ajouté dans la version 3.14.

7.2 System Functions

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

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

Valeur de retour : référence empruntée. Part of the Stable ABI. Return the object *name* from the `sys` module or `NULL` if it does not exist, without setting an exception.

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

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

void **PySys_ResetWarnOptions**()

Part of the Stable ABI. Reset `sys.warnoptions` to an empty list. This function may be called prior to *Py_Initialize()*.

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

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

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

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

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

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

Part of the Stable ABI. As *PySys_WriteStdout()*, but write to `sys.stderr` or *stderr* instead.

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

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

Ajouté dans la version 3.2.

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

Part of the Stable ABI. As *PySys_FormatStdout()*, but write to `sys.stderr` or *stderr* instead.

Ajouté dans la version 3.2.

PyObject ***PySys_GetXOptions**()

Valeur de retour : référence empruntée. Part of the Stable ABI *Depuis la version 3.7.* Return the current dictionary of `-X` options, similarly to `sys._xoptions`. On error, `NULL` is returned and an exception is set.

Ajouté dans la version 3.2.

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

Part of the Stable ABI *Depuis la version 3.13.* Raise an auditing event with any active hooks. Return zero for success and non-zero with an exception set on failure.

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

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

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

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

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

See also *PySys_AuditTuple()*.

Ajouté dans la version 3.8.

Modifié dans la version 3.8.2 : Require *Py_ssize_t* for `#` format characters. Previously, an unavoidable deprecation warning was raised.

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

Part of the Stable ABI *Depuis la version 3.13.* Similar to *PySys_Audit()*, but pass arguments as a Python object. *args* must be a `tuple`. To pass no arguments, *args* can be *NULL*.

Ajouté dans la version 3.13.

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

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

The *userData* pointer is passed into the hook function. Since hook functions may be called from different runtimes, this pointer should not refer directly to Python state.

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

The hook function is always called with an *attached thread state* by the Python interpreter that raised the event.

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

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

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

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

Ajouté dans la version 3.8.

7.3 Process Control

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

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

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

Modifié dans la version 3.9 : Log the function name automatically.

void **Py_Exit**(int status)

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

Modifié dans la version 3.6 : Errors from finalization no longer ignored.

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

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

Voir aussi

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

7.4 Importer des modules

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. This is a wrapper around `PyImport_Import()` which takes a `const char*` as an argument instead of a `PyObject*`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. This function is a deprecated alias of `PyImport_ImportModule()`.

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

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

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

Valeur de retour : nouvelle référence. Import a module. This is best described by referring to the built-in Python function `__import__()`.

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

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

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

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.7.* Import a module. This is best described by referring to the built-in Python function `__import__()`, as the standard `__import__()` function calls this function directly.

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

Ajouté dans la version 3.3.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Similar to *PyImport_ImportModuleLevelObject()*, but the name is a UTF-8 encoded string instead of a Unicode object.

Modifié dans la version 3.3 : Negative values for *level* are no longer accepted.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. This is a higher-level interface that calls the current "import hook function" (with an explicit *level* of 0, meaning absolute import). It invokes the `__import__()` function from the `__builtins__` of the current globals. This means that the import is done using whatever import hooks are installed in the current environment.

This function always uses absolute imports.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Reload a module. Return a new reference to the reloaded module, or `NULL` with an exception set on failure (the module still exists in this case).

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

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.13.* Return the module object corresponding to a module name.

The *name* argument may be of the form `package.module`. First check the modules dictionary if there's one there, and if not, create a new one and insert it in the modules dictionary.

Return a *strong reference* to the module on success. Return `NULL` with an exception set on failure.

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

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

Ajouté dans la version 3.13.

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

Valeur de retour : référence empruntée. Part of the Stable ABI *Depuis la version 3.7.* Similar to *PyImport_AddModuleRef()*, but return a *borrowed reference* and *name* is a Python `str` object.

Ajouté dans la version 3.3.

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

Valeur de retour : référence empruntée. Part of the Stable ABI. Similar to `PyImport_AddModuleRef()`, but return a *borrowed reference*.

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

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

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

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

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

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

See also `PyImport_ExecCodeModuleEx()` and `PyImport_ExecCodeModuleWithPathnames()`.

Modifié dans la version 3.12 : The setting of `__cached__` and `__loader__` is deprecated. See `ModuleSpec` for alternatives.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Like `PyImport_ExecCodeModule()`, but the `__file__` attribute of the module object is set to *pathname* if it is non-`NULL`.

See also `PyImport_ExecCodeModuleWithPathnames()`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.7.* Like `PyImport_ExecCodeModuleEx()`, but the `__cached__` attribute of the module object is set to *cpathname* if it is non-`NULL`. Of the three functions, this is the preferred one to use.

Ajouté dans la version 3.3.

Modifié dans la version 3.12 : Setting `__cached__` is deprecated. See `ModuleSpec` for alternatives.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Like `PyImport_ExecCodeModuleObject()`, but *name*, *pathname* and *cpathname* are UTF-8 encoded strings. Attempts are also made to figure out what the value for *pathname* should be from *cpathname* if the former is set to `NULL`.

Ajouté dans la version 3.2.

Modifié dans la version 3.3 : Uses `imp.source_from_cache()` in calculating the source path if only the bytecode path is provided.

Modifié dans la version 3.12 : No longer uses the removed `imp` module.

long **PyImport_GetMagicNumber**()

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

Modifié dans la version 3.3 : Return value of `-1` upon failure.

const char *__PyImport_GetMagicTag__()

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

Ajouté dans la version 3.2.

PyObject *__PyImport_GetModuleDict__()

Valeur de retour : référence empruntée. Part of the Stable ABI. Return the dictionary used for the module administration (a.k.a. `sys.modules`). Note that this is a per-interpreter variable.

PyObject *__PyImport_GetModule__(*PyObject* *name)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.8.* Return the already imported module with the given name. If the module has not been imported yet then returns `NULL` but does not set an error. Returns `NULL` and sets an error if the lookup failed.

Ajouté dans la version 3.7.

PyObject *__PyImport_GetImporter__(*PyObject* *path)

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

int __PyImport_ImportFrozenModuleObject__(*PyObject* *name)

Part of the Stable ABI *Depuis la version 3.7.* Load a frozen module named *name*. Return `1` for success, `0` if the module is not found, and `-1` with an exception set if the initialization failed. To access the imported module on a successful load, use *PyImport_ImportModule()*. (Note the misnomer --- this function would reload the module if it was already imported.)

Ajouté dans la version 3.3.

Modifié dans la version 3.4 : The `__file__` attribute is no longer set on the module.

int __PyImport_ImportFrozenModule__(const char *name)

Part of the Stable ABI. Similar to *PyImport_ImportFrozenModuleObject()*, but the name is a UTF-8 encoded string instead of a Unicode object.

struct ___frozen__

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

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

Modifié dans la version 3.11 : The new `is_package` field indicates whether the module is a package or not. This replaces setting the `size` field to a negative value.

const struct *_frozen* *__PyImport_FrozenModules__

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

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

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

struct **_inittab**

Structure describing a single entry in the list of built-in modules. Programs which embed Python may use an array of these structures in conjunction with *PyImport_ExtendInittab()* to provide additional built-in modules. The structure consists of two members :

const char ***name**

The module name, as an ASCII encoded string.

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

Initialization function for a module built into the interpreter.

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

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

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

PyObject ***PyImport_ImportModuleAttr**(*PyObject* *mod_name, *PyObject* *attr_name)

Valeur de retour : nouvelle référence. Import the module *mod_name* and get its attribute *attr_name*.

Names must be Python `str` objects.

Helper function combining *PyImport_Import()* and *PyObject_GetAttr()*. For example, it can raise `ImportError` if the module is not found, and `AttributeError` if the attribute doesn't exist.

Ajouté dans la version 3.14.

PyObject ***PyImport_ImportModuleAttrString**(const char *mod_name, const char *attr_name)

Valeur de retour : nouvelle référence. Similar to *PyImport_ImportModuleAttr()*, but names are UTF-8 encoded strings instead of Python `str` objects.

Ajouté dans la version 3.14.

7.5 Data marshalling support

These routines allow C code to work with serialized objects using the same data format as the `marshal` module. There are functions to write data into the serialization format, and additional functions that can be used to read the data back. Files used to store marshalled data must be opened in binary mode.

Numeric values are stored with the least significant byte first.

The module supports several versions of the data format ; see the `Python module documentation` for details.

Py_MARSHAL_VERSION

The current format version. See `marshal.version`.

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

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

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

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

Marshal a Python object, *value*, to *file*. *version* indicates the file format.

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

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

Valeur de retour : nouvelle référence. Return a bytes object containing the marshalled representation of *value*. *version* indicates the file format.

The following functions allow marshalled values to be read back in.

long **PyMarshal_ReadLongFromFile**(FILE *file)

Return a C `long` from the data stream in a `FILE*` opened for reading. Only a 32-bit value can be read in using this function, regardless of the native size of `long`.

On error, sets the appropriate exception (`EOFError`) and returns `-1`.

int **PyMarshal_ReadShortFromFile**(FILE *file)

Return a C `short` from the data stream in a `FILE*` opened for reading. Only a 16-bit value can be read in using this function, regardless of the native size of `short`.

On error, sets the appropriate exception (`EOFError`) and returns `-1`.

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

Valeur de retour : nouvelle référence. Return a Python object from the data stream in a `FILE*` opened for reading.

On error, sets the appropriate exception (`EOFError`, `ValueError` or `TypeError`) and returns `NULL`.

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

Valeur de retour : nouvelle référence. Return a Python object from the data stream in a `FILE*` opened for reading. Unlike *PyMarshal_ReadObjectFromFile()*, this function assumes that no further objects will be read from the file, allowing it to aggressively load file data into memory so that the de-serialization can operate from data in memory rather than reading a byte at a time from the file. Only use these variant if you are certain that you won't be reading anything else from the file.

On error, sets the appropriate exception (`EOFError`, `ValueError` or `TypeError`) and returns `NULL`.

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

Valeur de retour : nouvelle référence. Return a Python object from the data stream in a byte buffer containing *len* bytes pointed to by *data*.

On error, sets the appropriate exception (`EOFError`, `ValueError` or `TypeError`) and returns `NULL`.

7.6 Analyse des arguments et construction des valeurs

These functions are useful when creating your own extension functions and methods. Additional information and examples are available in extending-index.

Dans Les trois premières de ces fonctions décrites, *PyArg_ParseTuple()*, *PyArg_ParseTupleAndKeywords()*, et *PyArg_Parse()*, toutes utilisent *des chaînes de format* qui sont utilisées pour indiquer à la fonction les arguments attendus. Les chaînes de format utilise la même syntaxe pour chacune de ces fonctions.

7.6.1 Analyse des arguments

Une chaîne de format se compose de zéro ou plusieurs "unités de format". Une unité de format décrit un objet Python, elle est généralement composée d'un seul caractère ou d'une séquence d'unités de format entre parenthèses. À quelques exceptions près, une unité de format qui n'est pas une séquence entre parenthèses correspond normalement à un argument d'une seule adresse pour ces fonctions. Dans la description qui suit, la forme entre guillemets est l'unité de format, l'entrée entre parenthèses est le type d'objet Python qui correspond à l'unité de format, et l'entrée entre crochets est le type de la variable C (ou des variables) dont l'adresse doit être donnée.

Chaînes et tampons

Note

Dans Python 3.12 et précédents, la macro `PY_SSIZE_T_CLEAN` doit être définie avant d'inclure `Python.h` pour utiliser toutes les variantes `#` de format (`s#`, `y#`, etc.) détaillées ci-dessous. Ce n'est plus nécessaire pour Python 3.13 et les versions ultérieures.

Ces formats permettent d'accéder à un objet sous forme d'un fragment de mémoire contiguë. Il n'est pas nécessaire d'allouer la mémoire pour l'*unicode* ou le *bytes* renvoyé.

Sauf indication contraire, les tampons ne se terminent pas par NUL.

Les chaînes de caractères et tampons peuvent être convertis de trois manières vers C :

— Formats such as `y*` and `s*` fill a *Py_buffer* structure. This locks the underlying buffer so that the caller can subsequently use the buffer even inside a *Py_BEGIN_ALLOW_THREADS* block without the risk of mutable data being resized or destroyed. As a result, **you have to call** *PyBuffer_Release()* after you have finished processing the data (or in any early abort case).

— Les formats `es`, `es#`, `et` et `et#` allouent le tampon de résultats. **Vous devez appeler** *PyMem_Free()* après avoir fini de traiter les données (ou en cas d'abandon prématuré).

— Other formats take a `str` or a read-only *bytes-like object*, such as `bytes`, and provide a `const char *` pointer to its buffer. In this case the buffer is "borrowed" : it is managed by the corresponding Python object, and shares the lifetime of this object. You won't have to release any memory yourself.

  To ensure that the underlying buffer may be safely borrowed, the object's *PyBufferProcs.bf_releasebuffer* field must be `NULL`. This disallows common mutable objects such as `bytearray`, but also some read-only objects such as `memoryview` of `bytes`.

  Besides this `bf_releasebuffer` requirement, there is no check to verify whether the input object is immutable (e.g. whether it would honor a request for a writable buffer, or whether another thread can mutate the data).

s (str) [const char *]
Convertit un objet Unicode en un pointeur vers une chaîne de caractères. S'il s'agit d'un pointeur vers une chaîne de caractères déjà existante, il est stocké dans la variable de type pointeur vers un caractère dont vous avez donné l'adresse. Une chaîne de caractères en C se termine par *NULL*. La chaîne de caractères Python ne doit donc pas contenir de caractère dont le code est *null*. Si elle en contient, une exception `ValueError` est levée. Si la conversion échoue, une `UnicodeError` est levée.

> **Note**
>
> Ce format n'accepte pas les *objets compatibles avec une chaîne d'octets*. Si vous voulez accepter les chemins du système de fichiers et les convertir vers des chaînes de caractères C, il est préférable d'utiliser le format `O&` avec *PyUnicode_FSConverter()* en tant que *converter*.

Modifié dans la version 3.5 : Auparavant, une `TypeError` était levée quand la chaîne de caractères Python contenait des codes NULL.

s* (str ou *bytes-like object*) [Py_buffer]
Ce format accepte les objets Unicode et les *bytes-like object*. Cela remplit une structure *Py_buffer* qui est fournie par l'appelant. Dans ce cas, la chaîne de caractères C qui en résulte peut contenir des octets NULL. Les objets Unicode sont convertis en chaînes de caractères C en utilisant l'encodage `'utf-8'`.

s# (str, read-only *bytes-like object*) [const char *, *Py_ssize_t*]
Like `s*`, except that it provides a *borrowed buffer*. The result is stored into two C variables, the first one a pointer to a C string, the second one its length. The string may contain embedded null bytes. Unicode objects are converted to C strings using `'utf-8'` encoding.

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

z* (str, *bytes-like object* ou None) [Py_buffer]
Comme `s*`, mais l'objet Python peut aussi être `None`, auquel cas le membre `buf`, dont la structure est *Py_buffer* est fixée à *NULL*.

z# (str, read-only *bytes-like object* or None) [const char *, *Py_ssize_t*]
Like `s#`, but the Python object may also be `None`, in which case the C pointer is set to `NULL`.

y (lecture seule *objet compatible avec une chaîne d'octets*) [constante char *]
This format converts a bytes-like object to a C pointer to a *borrowed* character string ; it does not accept Unicode objects. The bytes buffer must not contain embedded null bytes ; if it does, a `ValueError` exception is raised.

Modifié dans la version 3.5 : Auparavant, `TypeError` était levée lorsque des octets *null* étaient rencontrés dans le tampon d'octets.

y* (*bytes-like object*) **[Py_buffer]**
: This variant on `s*` doesn't accept Unicode objects, only bytes-like objects. **This is the recommended way to accept binary data.**

y# (**read-only** *bytes-like object*) **[const char *, Py_ssize_t]**
: Cette variante de `s#` n'accepte pas les objets Unicode, uniquement des objets assimilés à des octets.

S (**bytes**) **[PyBytesObject *]**
: Requires that the Python object is a `bytes` object, without attempting any conversion. Raises `TypeError` if the object is not a bytes object. The C variable may also be declared as *PyObject**.

Y (**bytearray**) **[PyByteArrayObject *]**
: Requires that the Python object is a `bytearray` object, without attempting any conversion. Raises `TypeError` if the object is not a `bytearray` object. The C variable may also be declared as *PyObject**.

U (**str**) **[PyObject *]**
: Requires that the Python object is a Unicode object, without attempting any conversion. Raises `TypeError` if the object is not a Unicode object. The C variable may also be declared as *PyObject**.

w* (**lecture-écriture** *bytes-like object*) **[Py_buffer]**
: This format accepts any object which implements the read-write buffer interface. It fills a *Py_buffer* structure provided by the caller. The buffer may contain embedded null bytes. The caller have to call *PyBuffer_Release()* when it is done with the buffer.

es (**str**) **[const char *encoding, char **buffer]**
: This variant on `s` is used for encoding Unicode into a character buffer. It only works for encoded data without embedded NUL bytes.

  This format requires two arguments. The first is only used as input, and must be a `const char*` which points to the name of an encoding as a NUL-terminated string, or `NULL`, in which case `'utf-8'` encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a `char**`; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument.

  PyArg_ParseTuple() will allocate a buffer of the needed size, copy the encoded data into this buffer and adjust **buffer* to reference the newly allocated storage. The caller is responsible for calling *PyMem_Free()* to free the allocated buffer after use.

et (**str**, **bytes** or **bytearray**) **[const char *encoding, char **buffer]**
: Same as `es` except that byte string objects are passed through without recoding them. Instead, the implementation assumes that the byte string object uses the encoding passed in as parameter.

es# (**str**) **[const char *encoding, char **buffer, Py_ssize_t *buffer_length]**
: This variant on `s#` is used for encoding Unicode into a character buffer. Unlike the `es` format, this variant allows input data which contains NUL characters.

  It requires three arguments. The first is only used as input, and must be a `const char*` which points to the name of an encoding as a NUL-terminated string, or `NULL`, in which case `'utf-8'` encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a `char**`; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument. The third argument must be a pointer to an integer; the referenced integer will be set to the number of bytes in the output buffer.

  Il existe deux modes de fonctionnement :

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

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

  Dans les deux cas, **buffer_length* est la longueur des données encodées, sans l'octet NUL de fin.

et# (**str**, **bytes** or **bytearray**) **[const char *encoding, char **buffer, Py_ssize_t *buffer_length]**

Same as `es#` except that byte string objects are passed through without recoding them. Instead, the implementation assumes that the byte string object uses the encoding passed in as parameter.

Modifié dans la version 3.12 : `u`, `u#`, `Z`, and `Z#` are removed because they used a legacy `Py_UNICODE*` representation.

Les nombres

These formats allow representing Python numbers or single characters as C numbers. Formats that require `int`, `float` or `complex` can also use the corresponding special methods `__index__()`, `__float__()` or `__complex__()` to convert the Python object to the required type.

For signed integer formats, `OverflowError` is raised if the value is out of range for the C type. For unsigned integer formats, no range checking is done --- the most significant bits are silently truncated when the receiving field is too small to receive the value.

`b` (`int`) [`unsigned char`]
Convert a nonnegative Python integer to an unsigned tiny integer, stored in a C `unsigned char`.

`B` (`int`) [`unsigned char`]
Convert a Python integer to a tiny integer without overflow checking, stored in a C `unsigned char`.

`h` (`int`) [`short int`]
Convert a Python integer to a C `short int`.

`H` (`int`) [`unsigned short int`]
Convert a Python integer to a C `unsigned short int`, without overflow checking.

`i` (`int`) [`int`]
Convert a Python integer to a plain C `int`.

`I` (`int`) [`unsigned int`]
Convert a Python integer to a C `unsigned int`, without overflow checking.

`l` (`int`) [`long int`]
Convert a Python integer to a C `long int`.

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

Modifié dans la version 3.14 : Use `__index__()` if available.

`L` (`int`) [`long long`]
Convert a Python integer to a C `long long`.

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

Modifié dans la version 3.14 : Use `__index__()` if available.

`n` (`int`) [*`Py_ssize_t`*]
Convertit un entier Python en un *`Py_ssize_t`*.

`c` (`bytes` ou `bytearray` de longueur 1) [`char`]
Convert a Python byte, represented as a `bytes` or `bytearray` object of length 1, to a C `char`.

Modifié dans la version 3.3 : Allow `bytearray` objects.

`C` (`str` de longueur 1) [`int`]
Convert a Python character, represented as a `str` object of length 1, to a C `int`.

`f` (`float`) [`float`]
Convert a Python floating-point number to a C `float`.

`d` (`float`) [`double`]
Convert a Python floating-point number to a C `double`.

`D` (`complex`) [`Py_complex`]
Convertit un nombre complexe Python vers une structure *`Py_complex`* C.

Autres objets

`O` (objet) [`PyObject *`]
Store a Python object (without any conversion) in a C object pointer. The C program thus receives the actual object that was passed. A new *strong reference* to the object is not created (i.e. its reference count is not increased). The pointer stored is not `NULL`.

O! (**objet**) [*typeobject*, **PyObject ***]
: Store a Python object in a C object pointer. This is similar to `O`, but takes two C arguments : the first is the address of a Python type object, the second is the address of the C variable (of type *PyObject**) into which the object pointer is stored. If the Python object does not have the required type, `TypeError` is raised.

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

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

where *object* is the Python object to be converted and *address* is the `void*` argument that was passed to the `PyArg_Parse*` function. The returned *status* should be `1` for a successful conversion and `0` if the conversion has failed. When the conversion fails, the *converter* function should raise an exception and leave the content of *address* unmodified. If the *converter* returns `Py_CLEANUP_SUPPORTED`, it may get called a second time if the argument parsing eventually fails, giving the converter a chance to release any memory that it had already allocated. In this second call, the *object* parameter will be `NULL`; *address* will have the same value as in the original call.

Examples of converters : *PyUnicode_FSConverter()* and *PyUnicode_FSDecoder()*.

Modifié dans la version 3.1 : `Py_CLEANUP_SUPPORTED` was added.

p (**bool**) [**int**]
: Tests the value passed in for truth (a boolean **p**redicate) and converts the result to its equivalent C true/false integer value. Sets the int to `1` if the expression was true and `0` if it was false. This accepts any valid Python value. See truth for more information about how Python tests values for truth.

Ajouté dans la version 3.3.

(items) (**sequence**) [*matching-items*]
: The object must be a Python sequence (except `str`, `bytes` or `bytearray`) whose length is the number of format units in *items*. The C arguments must correspond to the individual format units in *items*. Format units for sequences may be nested.

If *items* contains format units which store a *borrowed buffer* (`s`, `s#`, `z`, `z#`, `y`, or `y#`) or a *borrowed reference* (`S`, `Y`, `U`, `O`, or `O!`), the object must be a Python tuple. The *converter* for the `O&` format unit in *items* must not store a borrowed buffer or a borrowed reference.

Modifié dans la version 3.14 : `str` and `bytearray` objects no longer accepted as a sequence.

Obsolète depuis la version 3.14 : Non-tuple sequences are deprecated if *items* contains format units which store a borrowed buffer or a borrowed reference.

Quelques autres caractères ont un sens dans une chaîne de format. On ne doit pas les trouvées dans des parenthèses imbriquées. Ce sont :

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

$
: *PyArg_ParseTupleAndKeywords()* only : Indicates that the remaining arguments in the Python argument list are keyword-only. Currently, all keyword-only arguments must also be optional arguments, so `|` must always be specified before `$` in the format string.

Ajouté dans la version 3.3.

:
: The list of format units ends here ; the string after the colon is used as the function name in error messages (the "associated value" of the exception that *PyArg_ParseTuple()* raises).

;
: La liste des unités de format s'arrête ici ; la chaîne après le point-virgule est utilise comme message d'erreur *au lieu* du message d'erreur par défaut. `:` et `;` sont mutuellement exclusifs.

Note that any Python object references which are provided to the caller are *borrowed* references ; do not release them (i.e. do not decrement their reference count) !

Les arguments additionnels qui sont donnés à ces fonctions doivent être des adresses de variables dont le type est déterminé par la chaîne de format. Elles sont utilisées pour stocker les valeurs du *n*-uplet d'entrée. Il y a quelques cas, comme décrit précédemment dans le liste des unités de formats, où ces paramètres sont utilisés comme valeurs d'entrée. Dans ce cas, ils devraient correspondre à ce qui est spécifié pour l'unité de format correspondante.

For the conversion to succeed, the *arg* object must match the format and the format must be exhausted. On success, the `PyArg_Parse*` functions return true, otherwise they return false and raise an appropriate exception. When the `PyArg_Parse*` functions fail due to conversion failure in one of the format units, the variables at the addresses corresponding to that and the following format units are left untouched.

Fonction de l'API

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

Part of the Stable ABI. Parse the parameters of a function that takes only positional parameters into local variables. Returns true on success ; on failure, it returns false and raises the appropriate exception.

int **PyArg_VaParse**(*PyObject* *args, const char *format, va_list vargs)

Part of the Stable ABI. Identical to *PyArg_ParseTuple()*, except that it accepts a va_list rather than a variable number of arguments.

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

Part of the Stable ABI. Parse the parameters of a function that takes both positional and keyword parameters into local variables. The *keywords* argument is a `NULL`-terminated array of keyword parameter names specified as null-terminated ASCII or UTF-8 encoded C strings. Empty names denote *positional-only parameters*. Returns true on success ; on failure, it returns false and raises the appropriate exception.

> **Note**
>
> The *keywords* parameter declaration is `char *const*` in C and `const char *const*` in C++. This can be overridden with the *PY_CXX_CONST* macro.

Modifié dans la version 3.6 : Added support for *positional-only parameters*.

Modifié dans la version 3.13 : The *keywords* parameter has now type `char *const*` in C and `const char *const*` in C++, instead of `char**`. Added support for non-ASCII keyword parameter names.

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

Part of the Stable ABI. Identical to *PyArg_ParseTupleAndKeywords()*, except that it accepts a va_list rather than a variable number of arguments.

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

Part of the Stable ABI. Ensure that the keys in the keywords argument dictionary are strings. This is only needed if *PyArg_ParseTupleAndKeywords()* is not used, since the latter already does this check.

Ajouté dans la version 3.2.

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

Part of the Stable ABI. Parse the parameter of a function that takes a single positional parameter into a local variable. Returns true on success ; on failure, it returns false and raises the appropriate exception.

Exemple :

```
// Function using METH_O calling convention
static PyObject*
my_function(PyObject *module, PyObject *arg)
```

(suite sur la page suivante)

(suite de la page précédente)

```
{
    int value;
    if (!PyArg_Parse(arg, "i:my_function", &value)) {
        return NULL;
    }
    // ... use value ...
}
```

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

Part of the Stable ABI. A simpler form of parameter retrieval which does not use a format string to specify the types of the arguments. Functions which use this method to retrieve their parameters should be declared as `METH_VARARGS` in function or method tables. The tuple containing the actual parameters should be passed as *args*; it must actually be a tuple. The length of the tuple must be at least *min* and no more than *max*; *min* and *max* may be equal. Additional arguments must be passed to the function, each of which should be a pointer to a `PyObject*` variable; these will be filled in with the values from *args*; they will contain *borrowed references*. The variables which correspond to optional parameters not given by *args* will not be filled in; these should be initialized by the caller. This function returns true on success and false if *args* is not a tuple or contains the wrong number of elements; an exception will be set if there was a failure.

This is an example of the use of this function, taken from the sources for the `_weakref` helper module for weak references :

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

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

The call to `PyArg_UnpackTuple()` in this example is entirely equivalent to this call to `PyArg_ParseTuple()` :

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

PY_CXX_CONST

The value to be inserted, if any, before `char *const*` in the *keywords* parameter declaration of `PyArg_ParseTupleAndKeywords()` and `PyArg_VaParseTupleAndKeywords()`. Default empty for C and `const` for C++ (`const char *const*`). To override, define it to the desired value before including `Python.h`.

Ajouté dans la version 3.13.

7.6.2 Construction des valeurs

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a new value based on a format string similar to those accepted by the `PyArg_Parse*` family of functions and a sequence of values. Returns the value or `NULL` in the case of an error; an exception will be raised if `NULL` is returned.

`Py_BuildValue()` does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns

whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

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

In the following description, the quoted form is the format unit ; the entry in (round) parentheses is the Python object type that the format unit will return ; and the entry in [square] brackets is the type of the C value(s) to be passed.

The characters space, tab, colon and comma are ignored in format strings (but not within format units such as `s#`). This can be used to make long format strings a tad more readable.

s (str ou None) [const char *]
Convert a null-terminated C string to a Python `str` object using `'utf-8'` encoding. If the C string pointer is `NULL`, `None` is used.

s# (str or None) [const char *, *Py_ssize_t*]
Convert a C string and its length to a Python `str` object using `'utf-8'` encoding. If the C string pointer is `NULL`, the length is ignored and `None` is returned.

y (bytes) [const char *]
This converts a C string to a Python `bytes` object. If the C string pointer is `NULL`, `None` is returned.

y# (bytes) [const char *, *Py_ssize_t*]
This converts a C string and its lengths to a Python object. If the C string pointer is `NULL`, `None` is returned.

z (str ou None) [const char *]
Same as `s`.

z# (str or None) [const char *, *Py_ssize_t*]
Same as `s#`.

u (str) [const wchar_t *]
Convert a null-terminated `wchar_t` buffer of Unicode (UTF-16 or UCS-4) data to a Python Unicode object. If the Unicode buffer pointer is `NULL`, `None` is returned.

u# (str) [const wchar_t *, *Py_ssize_t*]
Convert a Unicode (UTF-16 or UCS-4) data buffer and its length to a Python Unicode object. If the Unicode buffer pointer is `NULL`, the length is ignored and `None` is returned.

U (str ou None) [const char *]
Same as `s`.

U# (str or None) [const char *, *Py_ssize_t*]
Same as `s#`.

i (int) [int]
Convert a plain C `int` to a Python integer object.

b (int) [char]
Convert a plain C `char` to a Python integer object.

h (int) [short int]
Convert a plain C `short int` to a Python integer object.

l (int) [long int]
Convert a C `long int` to a Python integer object.

B (int) [unsigned char]
Convert a C `unsigned char` to a Python integer object.

H (int) [unsigned short int]
Convert a C `unsigned short int` to a Python integer object.

I (int) [unsigned int]
Convert a C `unsigned int` to a Python integer object.

k (int) [unsigned long]
Convert a C `unsigned long` to a Python integer object.

L (int) [long long]
Convert a C `long long` to a Python integer object.

K (**int**) [**unsigned long long**]
Convert a C `unsigned long long` to a Python integer object.

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

p (**bool**) [**int**]
Convert a C `int` to a Python `bool` object.

Be aware that this format requires an `int` argument. Unlike most other contexts in C, variadic arguments are not coerced to a suitable type automatically. You can convert another type (for example, a pointer or a float) to a suitable `int` value using `(x) ? 1 : 0` or `!!x`.

Ajouté dans la version 3.14.

c (**bytes de taille 1**) [**char**]
Convert a C `int` representing a byte to a Python `bytes` object of length 1.

C (**str de longueur 1**) [**int**]
Convert a C `int` representing a character to Python `str` object of length 1.

d (**float**) [**double**]
Convert a C `double` to a Python floating-point number.

f (**float**) [**float**]
Convert a C `float` to a Python floating-point number.

D (**complex**) [**Py_complex ***]
Convert a C *Py_complex* structure to a Python complex number.

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

S (**objet**) [**PyObject ***]
Same as `O`.

N (**objet**) [**PyObject ***]
Same as `O`, except it doesn't create a new *strong reference*. Useful when the object is created by a call to an object constructor in the argument list.

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

(items) (**tuple**) [***matching-items***]
Convert a sequence of C values to a Python tuple with the same number of items.

[items] (**list**) [***matching-items***]
Convert a sequence of C values to a Python list with the same number of items.

{items} (**dict**) [***matching-items***]
Convert a sequence of C values to a Python dictionary. Each pair of consecutive C values adds one item to the dictionary, serving as key and value, respectively.

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

PyObject ***Py_VaBuildValue**(const char *format, va_list vargs)
Valeur de retour : nouvelle référence. Part of the Stable ABI. Identical to *Py_BuildValue()*, except that it accepts a va_list rather than a variable number of arguments.

7.7 Conversion et formatage de chaînes

Fonctions de conversion pour les nombres et pour la sortie des chaînes formatées.

int **PyOS_snprintf**(char *str, size_t size, const char *format, ...)
Part of the Stable ABI. Output not more than *size* bytes to *str* according to the format string *format* and the extra arguments. See the Unix man page *snprintf(3)*.

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

Part of the Stable ABI. Output not more than *size* bytes to *str* according to the format string *format* and the variable argument list *va*. Unix man page *vsnprintf(3)*.

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

The wrappers ensure that `str[size-1]` is always `'\0'` upon return. They never write more than *size* bytes (including the trailing `'\0'`) into str. Both functions require that `str != NULL`, `size > 0`, `format != NULL` and `size < INT_MAX`. Note that this means there is no equivalent to the C99 `n = snprintf(NULL, 0, ...)` which would determine the necessary buffer size.

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

- When `0 <= rv < size`, the output conversion was successful and *rv* characters were written to *str* (excluding the trailing `'\0'` byte at `str[rv]`).
- When `rv >= size`, the output conversion was truncated and a buffer with `rv + 1` bytes would have been needed to succeed. `str[size-1]` is `'\0'` in this case.
- When `rv < 0`, "something bad happened." `str[size-1]` is `'\0'` in this case too, but the rest of *str* is undefined. The exact cause of the error depends on the underlying platform.

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

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

Part of the Stable ABI. Convert the initial part of the string in `str` to an `unsigned long` value according to the given `base`, which must be between `2` and `36` inclusive, or be the special value `0`.

Leading white space and case of characters are ignored. If `base` is zero it looks for a leading `0b`, `0o` or `0x` to tell which base. If these are absent it defaults to `10`. Base must be 0 or between 2 and 36 (inclusive). If `ptr` is non-`NULL` it will contain a pointer to the end of the scan.

If the converted value falls out of range of corresponding return type, range error occurs (`errno` is set to `ERANGE`) and `ULONG_MAX` is returned. If no conversion can be performed, `0` is returned.

See also the Unix man page *strtoul(3)*.

Ajouté dans la version 3.2.

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

Part of the Stable ABI. Convert the initial part of the string in `str` to an `long` value according to the given `base`, which must be between `2` and `36` inclusive, or be the special value `0`.

Same as *PyOS_strtoul()*, but return a `long` value instead and `LONG_MAX` on overflows.

See also the Unix man page *strtol(3)*.

Ajouté dans la version 3.2.

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

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

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

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

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

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

Ajouté dans la version 3.1.

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

Part of the Stable ABI. Convert a `double` *val* to a string using supplied *format_code*, *precision*, and *flags*.

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

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

— `Py_DTSF_SIGN` means to always precede the returned string with a sign character, even if *val* is non-negative.

— `Py_DTSF_ADD_DOT_0` means to ensure that the returned string will not look like an integer.

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

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

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

Ajouté dans la version 3.1.

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

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

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

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

7.8 PyHash API

See also the *PyTypeObject.tp_hash* member and numeric-hash.

type **Py_hash_t**

Hash value type : signed integer.

Ajouté dans la version 3.2.

type **Py_uhash_t**

Hash value type : unsigned integer.

Ajouté dans la version 3.2.

PyHASH_MODULUS

The Mersenne prime `P = 2**n -1`, used for numeric hash scheme.

Ajouté dans la version 3.13.

PyHASH_BITS

The exponent `n` of `P` in *PyHASH_MODULUS*.

Ajouté dans la version 3.13.

PyHASH_MULTIPLIER

Prime multiplier used in string and various other hashes.

Ajouté dans la version 3.13.

PyHASH_INF

The hash value returned for a positive infinity.

Ajouté dans la version 3.13.

PyHASH_IMAG

The multiplier used for the imaginary part of a complex number.

Ajouté dans la version 3.13.

type **PyHash_FuncDef**

Hash function definition used by *PyHash_GetFuncDef()*.

Py_hash_t (*const **hash**)(const void*, *Py_ssize_t*)

Hash function.

const char ***name**

Hash function name (UTF-8 encoded string).

const int **hash_bits**

Internal size of the hash value in bits.

const int **seed_bits**

Size of seed input in bits.

Ajouté dans la version 3.4.

PyHash_FuncDef ***PyHash_GetFuncDef**(void)

Get the hash function definition.

> **Voir aussi**
>
> **PEP 456** "Secure and interchangeable hash algorithm".

Ajouté dans la version 3.4.

Py_hash_t **Py_HashPointer**(const void *ptr)

Hash a pointer value : process the pointer value as an integer (cast it to `uintptr_t` internally). The pointer is not dereferenced.

The function cannot fail : it cannot return `-1`.

Ajouté dans la version 3.13.

Py_hash_t **Py_HashBuffer**(const void *ptr, *Py_ssize_t* len)

Compute and return the hash value of a buffer of *len* bytes starting at address *ptr*. The hash is guaranteed to match that of `bytes`, `memoryview`, and other built-in objects that implement the *buffer protocol*.

Use this function to implement hashing for immutable objects whose *tp_richcompare* function compares to another object's buffer.

len must be greater than or equal to `0`.

This function always succeeds.

Ajouté dans la version 3.14.

Py_hash_t **PyObject_GenericHash**(*PyObject* *obj)

Generic hashing function that is meant to be put into a type object's `tp_hash` slot. Its result only depends on the object's identity.

Particularité de l'implémentation CPython : In CPython, it is equivalent to *Py_HashPointer()*.

Ajouté dans la version 3.13.

7.9 Réflexion

PyObject ***PyEval_GetBuiltins**(void)

Valeur de retour : référence empruntée. Part of the Stable ABI. Obsolète depuis la version 3.13 : Use `PyEval_GetFrameBuiltins()` instead.

Renvoie un dictionnaire des fonctions natives de la *frame* en cours d'exécution, ou si aucune *frame* n'est exécutée, les fonctions natives du *thread* indiqué par le *thread state*.

PyObject ***PyEval_GetLocals**(void)

Valeur de retour : référence empruntée. Part of the Stable ABI. Obsolète depuis la version 3.13 : Use either `PyEval_GetFrameLocals()` to obtain the same behaviour as calling `locals()` in Python code, or else call `PyFrame_GetLocals()` on the result of `PyEval_GetFrame()` to access the `f_locals` attribute of the currently executing frame.

Return a mapping providing access to the local variables in the current execution frame, or `NULL` if no frame is currently executing.

Refer to `locals()` for details of the mapping returned at different scopes.

As this function returns a *borrowed reference*, the dictionary returned for *optimized scopes* is cached on the frame object and will remain alive as long as the frame object does. Unlike `PyEval_GetFrameLocals()` and `locals()`, subsequent calls to this function in the same frame will update the contents of the cached dictionary to reflect changes in the state of the local variables rather than returning a new snapshot.

Modifié dans la version 3.13 : As part of **PEP 667**, `PyFrame_GetLocals()`, `locals()`, and `FrameType.f_locals` no longer make use of the shared cache dictionary. Refer to the What's New entry for additional details.

PyObject ***PyEval_GetGlobals**(void)

Valeur de retour : référence empruntée. Part of the Stable ABI. Obsolète depuis la version 3.13 : Use `PyEval_GetFrameGlobals()` instead.

Return a dictionary of the global variables in the current execution frame, or `NULL` if no frame is currently executing.

PyFrameObject ***PyEval_GetFrame**(void)

Valeur de retour : référence empruntée. Part of the Stable ABI. Return the *attached thread state*'s frame, which is `NULL` if no frame is currently executing.

See also `PyThreadState_GetFrame()`.

PyObject ***PyEval_GetFrameBuiltins**(void)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.13.* Renvoie un dictionnaire des fonctions natives de la *frame* en cours d'exécution, ou si aucune *frame* n'est exécutée, les fonctions natives du *thread* indiqué par le *thread state*.

Ajouté dans la version 3.13.

PyObject ***PyEval_GetFrameLocals**(void)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.13.* Return a dictionary of the local variables in the current execution frame, or `NULL` if no frame is currently executing. Equivalent to calling `locals()` in Python code.

To access `f_locals` on the current frame without making an independent snapshot in *optimized scopes*, call `PyFrame_GetLocals()` on the result of `PyEval_GetFrame()`.

Ajouté dans la version 3.13.

PyObject ***PyEval_GetFrameGlobals**(void)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.13.* Return a dictionary of the global variables in the current execution frame, or `NULL` if no frame is currently executing. Equivalent to calling `globals()` in Python code.

Ajouté dans la version 3.13.

const char *__PyEval_GetFuncName__(*PyObject* *func)

Part of the Stable ABI. Renvoie le nom de *func* s'il s'agit d'une fonction, d'une classe ou d'un objet d'instance, sinon le nom du type de *func*

const char *__PyEval_GetFuncDesc__(*PyObject* *func)

Part of the Stable ABI. Renvoie une description en chaîne de caractères, en fonction du type de *func*. Les valeurs renvoyées peuvent être `"()"` pour les fonction et les méthodes, `\" constructor\"`, `\" instance\"`, `\" object\"`. Concaténé avec le résultat de *PyEval_GetFuncName()*, le résultat sera une description de *func*

7.10 Codec registry and support functions

int __PyCodec_Register__(*PyObject* *search_function)

Part of the Stable ABI. Register a new codec search function.

As side effect, this tries to load the `encodings` package, if not yet done, to make sure that it is always first in the list of search functions.

int __PyCodec_Unregister__(*PyObject* *search_function)

Part of the Stable ABI *Depuis la version 3.10.* Unregister a codec search function and clear the registry's cache. If the search function is not registered, do nothing. Return 0 on success. Raise an exception and return -1 on error.

Ajouté dans la version 3.10.

int __PyCodec_KnownEncoding__(const char *encoding)

Part of the Stable ABI. Return `1` or `0` depending on whether there is a registered codec for the given *encoding*. This function always succeeds.

PyObject *__PyCodec_Encode__(*PyObject* *object, const char *encoding, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Generic codec based encoding API.

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

PyObject *__PyCodec_Decode__(*PyObject* *object, const char *encoding, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Generic codec based decoding API.

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

7.10.1 Codec lookup API

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

PyObject *__PyCodec_Encoder__(const char *encoding)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Get an encoder function for the given *encoding*.

PyObject *__PyCodec_Decoder__(const char *encoding)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Get a decoder function for the given *encoding*.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Get an `IncrementalEncoder` object for the given *encoding*.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Get an `IncrementalDecoder` object for the given *encoding*.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Get a `StreamReader` factory function for the given *encoding*.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Get a `StreamWriter` factory function for the given *encoding*.

7.10.2 Registry API for Unicode encoding error handlers

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

Part of the Stable ABI. Register the error handling callback function *error* under the given *name*. This callback function will be called by a codec when it encounters unencodable characters/undecodable bytes and *name* is specified as the error parameter in the call to the encode/decode function.

The callback gets a single argument, an instance of `UnicodeEncodeError`, `UnicodeDecodeError` or `UnicodeTranslateError` that holds information about the problematic sequence of characters or bytes and their offset in the original string (see *Objets exception Unicode* for functions to extract this information). The callback must either raise the given exception, or return a two-item tuple containing the replacement for the problematic sequence, and an integer giving the offset in the original string at which encoding/decoding should be resumed.

Return `0` on success, `-1` on error.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Lookup the error handling callback function registered under *name*. As a special case `NULL` can be passed, in which case the error handling callback for ”strict” will be returned.

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

Valeur de retour : toujours NULL. Part of the Stable ABI. Raise *exc* as an exception.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Ignore the unicode error, skipping the faulty input.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Replace the unicode encode error with `?` or `U+FFFD`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Replace the unicode encode error with XML character references.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Replace the unicode encode error with backslash escapes (`\x`, `\u` and `\U`).

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

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.7.* Replace the unicode encode error with `\N{...}` escapes.

Ajouté dans la version 3.5.

7.11 PyTime C API

Ajouté dans la version 3.13.

The clock C API provides access to system clocks. It is similar to the Python `time` module.

For C API related to the `datetime` module, see *Objets DateTime*.

7.11.1 Types

type **PyTime_t**

A timestamp or duration in nanoseconds, represented as a signed 64-bit integer.

The reference point for timestamps depends on the clock used. For example, *PyTime_Time()* returns timestamps relative to the UNIX epoch.

The supported range is around [-292.3 years; +292.3 years]. Using the Unix epoch (January 1st, 1970) as reference, the supported date range is around [1677-09-21; 2262-04-11]. The exact limits are exposed as constants :

PyTime_t **PyTime_MIN**

Minimum value of *PyTime_t*.

PyTime_t **PyTime_MAX**

Maximum value of *PyTime_t*.

7.11.2 Clock Functions

The following functions take a pointer to a *PyTime_t* that they set to the value of a particular clock. Details of each clock are given in the documentation of the corresponding Python function.

The functions return `0` on success, or `-1` (with an exception set) on failure.

On integer overflow, they set the *PyExc_OverflowError* exception and set `*result` to the value clamped to the `[PyTime_MIN; PyTime_MAX]` range. (On current systems, integer overflows are likely caused by misconfigured system time.)

As any other C API (unless otherwise specified), the functions must be called with an *attached thread state*.

int **PyTime_Monotonic**(*PyTime_t* *result)

Read the monotonic clock. See `time.monotonic()` for important details on this clock.

int **PyTime_PerfCounter**(*PyTime_t* *result)

Read the performance counter. See `time.perf_counter()` for important details on this clock.

int **PyTime_Time**(*PyTime_t* *result)

Read the "wall clock" time. See `time.time()` for details important on this clock.

7.11.3 Raw Clock Functions

Similar to clock functions, but don't set an exception on error and don't require the caller to have an *attached thread state*.

On success, the functions return `0`.

On failure, they set `*result` to `0` and return `-1`, *without* setting an exception. To get the cause of the error, *attach* a *thread state*, and call the regular (non-`Raw`) function. Note that the regular function may succeed after the `Raw` one failed.

int **PyTime_MonotonicRaw**(*PyTime_t* *result)

Similar to *PyTime_Monotonic()*, but don't set an exception on error and don't require an *attached thread state*.

int **PyTime_PerfCounterRaw**(*PyTime_t* *result)

Similar to *PyTime_PerfCounter()*, but don't set an exception on error and don't require an *attached thread state*.

int **PyTime_TimeRaw**(*PyTime_t* *result)

Similar to *PyTime_Time()*, but don't set an exception on error and don't require an *attached thread state*.

7.11.4 Conversion functions

double **PyTime_AsSecondsDouble**(*PyTime_t* t)

Convert a timestamp to a number of seconds as a C `double`.

The function cannot fail, but note that `double` has limited accuracy for large values.

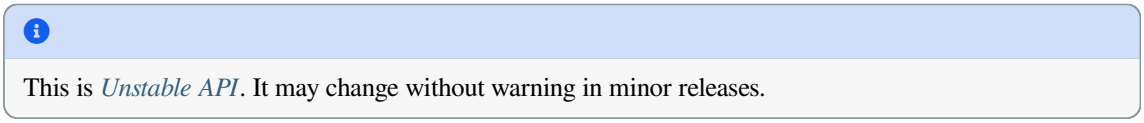
7.12 Support for Perf Maps

On supported platforms (as of this writing, only Linux), the runtime can take advantage of *perf map files* to make Python functions visible to an external profiling tool (such as perf). A running process may create a file in the `/tmp` directory, which contains entries that can map a section of executable code to a name. This interface is described in the documentation of the Linux Perf tool.

In Python, these helper APIs can be used by libraries and features that rely on generating machine code on the fly.

Note that holding an *attached thread state* is not required for these APIs.

int **PyUnstable_PerfMapState_Init**(void)

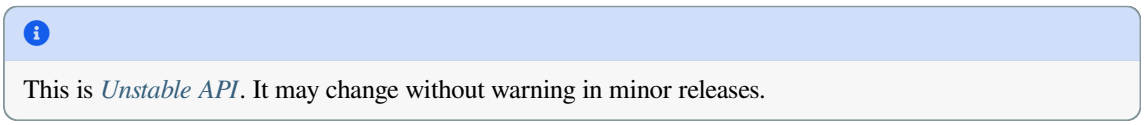


This is *Unstable API*. It may change without warning in minor releases.

Open the `/tmp/perf-$pid.map` file, unless it's already opened, and create a lock to ensure thread-safe writes to the file (provided the writes are done through *PyUnstable_WritePerfMapEntry()*). Normally, there's no need to call this explicitly; just use *PyUnstable_WritePerfMapEntry()* and it will initialize the state on first call.

Returns `0` on success, `-1` on failure to create/open the perf map file, or `-2` on failure to create a lock. Check `errno` for more information about the cause of a failure.

int **PyUnstable_WritePerfMapEntry**(const void *code_addr, unsigned int code_size, const char *entry_name)



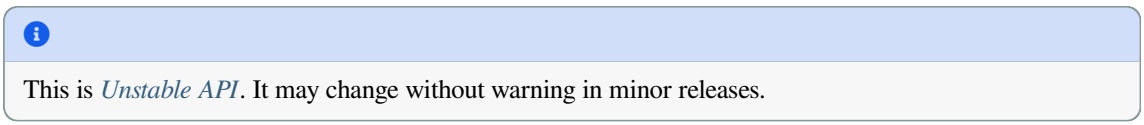
This is *Unstable API*. It may change without warning in minor releases.

Write one single entry to the `/tmp/perf-$pid.map` file. This function is thread safe. Here is what an example entry looks like :

```
# address       size  name
7f3529fcf759 b     py::bar:/run/t.py
```

Will call *PyUnstable_PerfMapState_Init()* before writing the entry, if the perf map file is not already opened. Returns `0` on success, or the same error codes as *PyUnstable_PerfMapState_Init()* on failure.

void **PyUnstable_PerfMapState_Fini**(void)



This is *Unstable API*. It may change without warning in minor releases.

Close the perf map file opened by *PyUnstable_PerfMapState_Init()*. This is called by the runtime itself during interpreter shut-down. In general, there shouldn't be a reason to explicitly call this, except to handle specific scenarios such as forking.

CHAPITRE 8

Couche d'abstraction des objets

Dans ce chapitre, les fonctions s'appliquent à des objets Python sans tenir compte de leur type, ou des classes d'objets au sens large (par exemple, tous les types numériques, ou tous les types de séquence). Quand ils sont utilisés sur des types d'objets qui ne correspondent pas, ils lèveront une exception Python.

Il n'est pas possible d'utiliser ces fonctions sur des objets qui n'ont pas été correctement initialisés, comme un objet liste qui a été créé avec `PyList_New()` mais dont les éléments n'ont pas encore été mis à une valeur non-`NULL`.

8.1 Protocole Objet

PyObject ***Py_GetConstant**(unsigned int constant_id)

Part of the Stable ABI *Depuis la version 3.13.* Get a *strong reference* to a constant.

Set an exception and return `NULL` if *constant_id* is invalid.

constant_id must be one of these constant identifiers :

Constant Identifier	Valeur	Returned object
Py_CONSTANT_NONE	0	None
Py_CONSTANT_FALSE	1	False
Py_CONSTANT_TRUE	2	True
Py_CONSTANT_ELLIPSIS	3	Ellipsis
Py_CONSTANT_NOT_IMPLEMENTI	4	NotImplemented
Py_CONSTANT_ZERO	5	0
Py_CONSTANT_ONE	6	1
Py_CONSTANT_EMPTY_STR	7	''
Py_CONSTANT_EMPTY_BYTES	8	b''
Py_CONSTANT_EMPTY_TUPLE	9	()

Numeric values are only given for projects which cannot use the constant identifiers.

Ajouté dans la version 3.13.

Particularité de l'implémentation CPython : In CPython, all of these constants are *immortal*.

PyObject ***Py_GetConstantBorrowed**(unsigned int constant_id)

Part of the Stable ABI *Depuis la version 3.13.* Similar to `Py_GetConstant()`, but return a *borrowed reference*.

This function is primarily intended for backwards compatibility : using `Py_GetConstant()` is recommended for new code.

The reference is borrowed from the interpreter, and is valid until the interpreter finalization.

Ajouté dans la version 3.13.

PyObject ***Py_NotImplemented**

Le singleton `NotImplemented`, utilisé pour signaler qu'une opération n'est pas implémentée pour la combinaison de types en question.

Py_RETURN_NOTIMPLEMENTED

Properly handle returning `Py_NotImplemented` from within a C function (that is, create a new *strong reference* to `NotImplemented` and return it).

Py_PRINT_RAW

Flag to be used with multiple functions that print the object (like `PyObject_Print()` and `PyFile_WriteObject()`). If passed, these function would use the `str()` of the object instead of the `repr()`.

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

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

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

Part of the Stable ABI *Depuis la version 3.13.* Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This is equivalent to the Python expression `hasattr(o, attr_name)`. On failure, return `-1`.

Ajouté dans la version 3.13.

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

Part of the Stable ABI *Depuis la version 3.13.* This is the same as *PyObject_HasAttrWithError()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

Ajouté dans la version 3.13.

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

Part of the Stable ABI. Returns `1` if *o* has the attribute *attr_name*, and `0` otherwise. This function always succeeds.

> **Note**
>
> Exceptions that occur when this calls `__getattr__()` and `__getattribute__()` methods aren't propagated, but instead given to `sys.unraisablehook()`. For proper error handling, use *PyObject_HasAttrWithError()*, *PyObject_GetOptionalAttr()* or *PyObject_GetAttr()* instead.

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

Part of the Stable ABI. This is the same as *PyObject_HasAttr()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

> **Note**
>
> Exceptions that occur when this calls `__getattr__()` and `__getattribute__()` methods or while creating the temporary `str` object are silently ignored. For proper error handling, use *PyObject_HasAttrStringWithError()*, *PyObject_GetOptionalAttrString()* or *PyObject_GetAttrString()* instead.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Retrieve an attribute named *attr_name* from object *o*. Returns the attribute value on success, or `NULL` on failure. This is the equivalent of the Python expression `o.attr_name`.

If the missing attribute should not be treated as a failure, you can use *PyObject_GetOptionalAttr()* instead.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. This is the same as *PyObject_GetAttr()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

If the missing attribute should not be treated as a failure, you can use *PyObject_GetOptionalAttrString()* instead.

int **PyObject_GetOptionalAttr**(*PyObject* *obj, *PyObject* *attr_name, *PyObject* **result) ;

Part of the Stable ABI *Depuis la version 3.13.* Variant of *PyObject_GetAttr()* which doesn't raise `AttributeError` if the attribute is not found.

If the attribute is found, return `1` and set **result* to a new *strong reference* to the attribute. If the attribute is not found, return `0` and set **result* to `NULL`; the `AttributeError` is silenced. If an error other than

`AttributeError` is raised, return `-1` and set **result* to `NULL`.

Ajouté dans la version 3.13.

int **PyObject_GetOptionalAttrString**(*PyObject* *obj, const char *attr_name, *PyObject* **result) ;

Part of the Stable ABI *Depuis la version 3.13.* This is the same as *PyObject_GetOptionalAttr()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

Ajouté dans la version 3.13.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Accesseur d'attribut générique destiné à être mis dans le *slot* `tp_getattro` d'un objet type. Recherche un descripteur dans le dictionnaire de classes du MRO de l'objet ainsi qu'un attribut dans le `__dict__` de l'objet (si présent). Comme défini dans descriptors, les descripteurs de données sont prioritaires sur les attributs d'instance, contrairement aux autres descripteurs. Sinon, une `AttributeError` est levée.

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

Part of the Stable ABI. Définit la valeur de l'attribut nommé *attr_name*, pour l'objet *o*, à la valeur *v*. Lève une exception et renvoie `-1` en cas d'échec ; renvoie `0` en cas de succès. Ceci est équivalent à l'instruction Python `o.attr_name = v`.

If *v* is `NULL`, the attribute is deleted. This behaviour is deprecated in favour of using *PyObject_DelAttr()*, but there are currently no plans to remove it.

int **PyObject_SetAttrString**(*PyObject* *o, const char *attr_name, *PyObject* *v)

Part of the Stable ABI. This is the same as *PyObject_SetAttr()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

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

The number of different attribute names passed to this function should be kept small, usually by using a statically allocated string as *attr_name*. For attribute names that aren't known at compile time, prefer calling *PyUnicode_FromString()* and *PyObject_SetAttr()* directly. For more details, see *PyUnicode_InternFromString()*, which may be used internally to create a key object.

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

Part of the Stable ABI. Accesseur et suppresseur générique d'attributs qui est fait pour être mis dans le *tp_setattro* d'un objet type. Il cherche un descripteur de données dans le dictionnaire de classes dans le MRO de l'objet et, si ce descripteur est trouvé, c'est lui qui est utilisé de préférence pour la suppression et la définition de l'attribut dans le dictionnaire d'instance. Sinon, l'attribut est défini ou supprimé dans le `__dict__` de l'objet (si présent). En cas de succès, `0` est renvoyé, sinon une `AttributeError` est levée et `-1` est renvoyé.

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

Part of the Stable ABI *Depuis la version 3.13.* Supprime l'attribut nommé *attr_name*, pour l'objet *o*. Renvoie `-1` en cas d'échec. Ceci est l'équivalent de l'expression Python `del o.attr_name`.

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

Part of the Stable ABI *Depuis la version 3.13.* This is the same as *PyObject_DelAttr()*, but *attr_name* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

The number of different attribute names passed to this function should be kept small, usually by using a statically allocated string as *attr_name*. For attribute names that aren't known at compile time, prefer calling *PyUnicode_FromString()* and *PyObject_DelAttr()* directly. For more details, see *PyUnicode_InternFromString()*, which may be used internally to create a key object for lookup.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.10.* Une implémentation générique de l'accesseur d'un descripteur d'un `__dict__`. Crée le dictionnaire si nécessaire.

This function may also be called to get the `__dict__` of the object *o*. Pass `NULL` for *context* when calling it. Since this function may need to allocate memory for the dictionary, it may be more efficient to call *PyObject_GetAttr()* when accessing an attribute on the object.

On failure, returns `NULL` with an exception set.

Ajouté dans la version 3.3.

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

Part of the Stable ABI *Depuis la version 3.7.* Une implémentation générique du mutateur d'un descripteur de `__dict__`. Cette implémentation n'autorise pas la suppression du dictionnaire.

Ajouté dans la version 3.3.

PyObject ***_PyObject_GetDictPtr**(*PyObject* *obj)

Return a pointer to `__dict__` of the object *obj*. If there is no `__dict__`, return `NULL` without setting an exception.

This function may need to allocate memory for the dictionary, so it may be more efficient to call *PyObject_GetAttr()* when accessing an attribute on the object.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Compare the values of *o1* and *o2* using the operation specified by *opid*, which must be one of *Py_LT*, *Py_LE*, *Py_EQ*, *Py_NE*, *Py_GT*, or *Py_GE*, corresponding to `<`, `<=`, `==`, `!=`, `>`, or `>=` respectively. This is the equivalent of the Python expression `o1 op o2`, where `op` is the operator corresponding to *opid*. Returns the value of the comparison on success, or `NULL` on failure.

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

Part of the Stable ABI. Compare the values of *o1* and *o2* using the operation specified by *opid*, like *PyObject_RichCompare()*, but returns `-1` on error, `0` if the result is false, `1` otherwise.

Note

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

PyObject ***PyObject_Format**(*PyObject* *obj, *PyObject* *format_spec)

Part of the Stable ABI. Format *obj* using *format_spec*. This is equivalent to the Python expression `format(obj, format_spec)`.

format_spec may be `NULL`. In this case the call is equivalent to `format(obj)`. Returns the formatted string on success, `NULL` on failure.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Compute a string representation of object *o*. Returns the string representation on success, `NULL` on failure. This is the equivalent of the Python expression `repr(o)`. Called by the `repr()` built-in function.

Modifié dans la version 3.4 : Cette fonction inclut maintenant une assertion de débogage afin d'assurer qu'elle ne passe pas sous silence une exception active.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Comme *PyObject_Repr()*, calcule une représentation en chaîne de caractères de l'objet *o*, mais échappe les caractères non ASCII dans la chaîne de caractères renvoyée par *PyObject_Repr()* avec' `\x`, `\u` ou `\U`. Cela génère une chaîne de caractères similaire à celle renvoyée par *PyObject_Repr()* en Python 2. Appelée par la fonction native `ascii()`.

PyObject ***PyObject_Str**(*PyObject* *o)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Compute a string representation of object *o*. Returns the string representation on success, `NULL` on failure. This is the equivalent of the Python expression `str(o)`. Called by the `str()` built-in function and, therefore, by the `print()` function.

Modifié dans la version 3.4 : Cette fonction inclut maintenant une assertion de débogage afin d'assurer qu'elle ne passe pas sous silence une exception active.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Compute a bytes representation of object *o*. `NULL` is returned on failure and a bytes object on success. This is equivalent to the Python expression `bytes(o)`, when *o* is not an integer. Unlike `bytes(o)`, a TypeError is raised when *o* is an integer instead of a zero-initialized bytes object.

int **PyObject_IsSubclass**(*PyObject* *derived, *PyObject* *cls)

Part of the Stable ABI. Renvoie `1` si la classe *derived* est identique à ou dérivée de la classe *cls*, renvoie `0` sinon. En cas d'erreur, renvoie `-1`.

Si *cls* est un *n*-uplet, la vérification est menée sur chaque entrée de *cls*. Le résultat sera `1` quand au moins une des vérifications renvoie `1`, sinon ce sera `0`.

If *cls* has a `__subclasscheck__()` method, it will be called to determine the subclass status as described in **PEP 3119**. Otherwise, *derived* is a subclass of *cls* if it is a direct or indirect subclass, i.e. contained in `cls.__mro__`.

Normally only class objects, i.e. instances of `type` or a derived class, are considered classes. However, objects can override this by having a `__bases__` attribute (which must be a tuple of base classes).

int **PyObject_IsInstance**(*PyObject* *inst, *PyObject* *cls)

Part of the Stable ABI. Renvoie `1` si *inst* est une instance de la classe *cls* ou une sous-classe de *cls*, ou `0` sinon. En cas d'erreur, renvoie `-1` et initialise une exception.

Si *cls* est un *n*-uplet, la vérification est menée sur chaque entrée de *cls*. Le résultat sera `1` quand au moins une des vérifications renvoie `1`, sinon ce sera `0`.

If *cls* has a `__instancecheck__()` method, it will be called to determine the subclass status as described in **PEP 3119**. Otherwise, *inst* is an instance of *cls* if its class is a subclass of *cls*.

An instance *inst* can override what is considered its class by having a `__class__` attribute.

An object *cls* can override if it is considered a class, and what its base classes are, by having a `__bases__` attribute (which must be a tuple of base classes).

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

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

Modifié dans la version 3.2 : The return type is now Py_hash_t. This is a signed integer the same size as *Py_ssize_t*.

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

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

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

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

int **PyObject_Not**(*PyObject* *o)

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

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. When *o* is non-`NULL`, returns a type object corresponding to the object type of object *o*. On failure, raises `SystemError` and returns `NULL`. This is equivalent to the Python expression `type(o)`. This function creates a new *strong reference* to the return value. There's really no reason to use this function instead of the *Py_TYPE()* function, which returns a pointer of type *PyTypeObject**, except when a new *strong reference* is needed.

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

Return non-zero if the object *o* is of type *type* or a subtype of *type*, and `0` otherwise. Both parameters must be non-`NULL`.

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

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

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

Py_ssize_t **PyObject_LengthHint**(*PyObject* *o, *Py_ssize_t* defaultvalue)

Return an estimated length for the object *o*. First try to return its actual length, then an estimate using `__length_hint__()`, and finally return the default value. On error return `-1`. This is the equivalent to the Python expression `operator.length_hint(o, defaultvalue)`.

Ajouté dans la version 3.4.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return element of *o* corresponding to the object *key* or `NULL` on failure. This is the equivalent of the Python expression `o[key]`.

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

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

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

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

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

Part of the Stable ABI. This is the same as *PyObject_DelItem()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

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

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

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. This is equivalent to the Python expression `iter(o)`. It returns a new iterator for the object argument, or the object itself if the object is already an iterator. Raises `TypeError` and returns `NULL` if the object cannot be iterated.

PyObject ***PyObject_SelfIter**(*PyObject* *obj)

Valeur de retour : nouvelle référence. Part of the Stable ABI. This is equivalent to the Python `__iter__(self): return self` method. It is intended for *iterator* types, to be used in the *PyTypeObject.tp_iter* slot.

PyObject ***PyObject_GetAIter**(*PyObject* *o)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.10.* This is the equivalent to the Python expression `aiter(o)`. Takes an `AsyncIterable` object and returns an `AsyncIterator` for it. This is typically a new iterator but if the argument is an `AsyncIterator`, this returns itself. Raises `TypeError` and returns `NULL` if the object cannot be iterated.

Ajouté dans la version 3.10.

void ***PyObject_GetTypeData**(*PyObject* *o, *PyTypeObject* *cls)

Part of the Stable ABI *Depuis la version 3.12.* Get a pointer to subclass-specific data reserved for *cls*.

The object *o* must be an instance of *cls*, and *cls* must have been created using negative *PyType_Spec.basicsize*. Python does not check this.

On error, set an exception and return `NULL`.

Ajouté dans la version 3.12.

Py_ssize_t **PyType_GetTypeDataSize**(*PyTypeObject* *cls)

Part of the Stable ABI *Depuis la version 3.12.* Return the size of the instance memory space reserved for *cls*, i.e. the size of the memory `PyObject_GetTypeData()` returns.

This may be larger than requested using `-PyType_Spec.basicsize`; it is safe to use this larger size (e.g. with `memset()`).

The type *cls* **must** have been created using negative `PyType_Spec.basicsize`. Python does not check this.

On error, set an exception and return a negative value.

Ajouté dans la version 3.12.

void ***PyObject_GetItemData**(*PyObject* *o)

Get a pointer to per-item data for a class with `Py_TPFLAGS_ITEMS_AT_END`.

On error, set an exception and return `NULL`. `TypeError` is raised if *o* does not have `Py_TPFLAGS_ITEMS_AT_END` set.

Ajouté dans la version 3.12.

int **PyObject_VisitManagedDict**(*PyObject* *obj, *visitproc* visit, void *arg)

Visit the managed dictionary of *obj*.

This function must only be called in a traverse function of the type which has the `Py_TPFLAGS_MANAGED_DICT` flag set.

Ajouté dans la version 3.13.

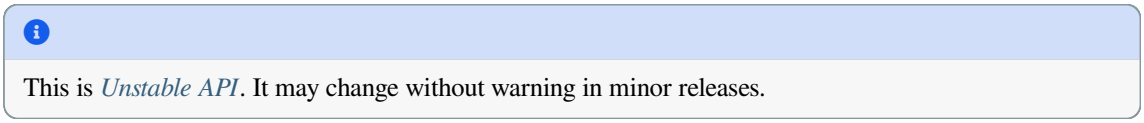
void **PyObject_ClearManagedDict**(*PyObject* *obj)

Clear the managed dictionary of *obj*.

This function must only be called in a traverse function of the type which has the `Py_TPFLAGS_MANAGED_DICT` flag set.

Ajouté dans la version 3.13.

int **PyUnstable_Object_EnableDeferredRefcount**(*PyObject* *obj)



This is *Unstable API*. It may change without warning in minor releases.

Enable deferred reference counting on *obj*, if supported by the runtime. In the *free-threaded* build, this allows the interpreter to avoid reference count adjustments to *obj*, which may improve multi-threaded performance. The tradeoff is that *obj* will only be deallocated by the tracing garbage collector, and not when the interpreter no longer has any references to it.

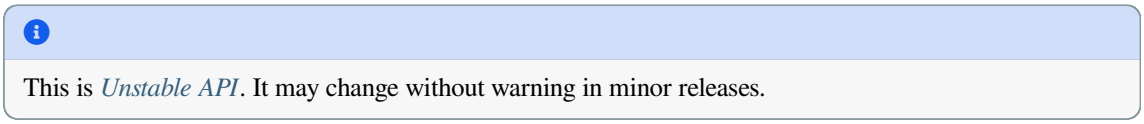
This function returns `1` if deferred reference counting is enabled on *obj*, and `0` if deferred reference counting is not supported or if the hint was ignored by the interpreter, such as when deferred reference counting is already enabled on *obj*. This function is thread-safe, and cannot fail.

This function does nothing on builds with the *GIL* enabled, which do not support deferred reference counting. This also does nothing if *obj* is not an object tracked by the garbage collector (see `gc.is_tracked()` and `PyObject_GC_IsTracked()`).

This function is intended to be used soon after *obj* is created, by the code that creates it, such as in the object's `tp_new` slot.

Ajouté dans la version 3.14.

int **PyUnstable_Object_IsUniqueReferencedTemporary**(*PyObject* *obj)



This is *Unstable API*. It may change without warning in minor releases.

Check if *obj* is a unique temporary object. Returns `1` if *obj* is known to be a unique temporary object, and `0` otherwise. This function cannot fail, but the check is conservative, and may return `0` in some cases even if *obj* is a unique temporary object.

If an object is a unique temporary, it is guaranteed that the current code has the only reference to the object. For arguments to C functions, this should be used instead of checking if the reference count is `1`. Starting with Python 3.14, the interpreter internally avoids some reference count modifications when loading objects onto the operands stack by *borrowing* references when possible, which means that a reference count of `1` by itself does not guarantee that a function argument uniquely referenced.

In the example below, `my_func` is called with a unique temporary object as its argument :

```
my_func([1, 2, 3])
```

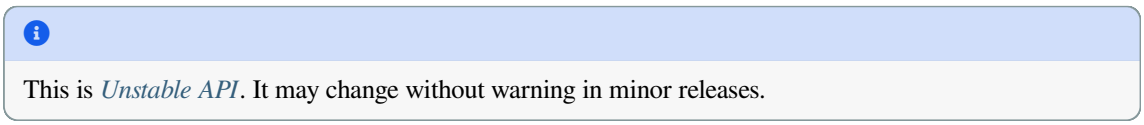
In the example below, `my_func` is **not** called with a unique temporary object as its argument, even if its refcount is `1` :

```
my_list = [1, 2, 3]
my_func(my_list)
```

See also the function *`Py_REFCNT()`*.

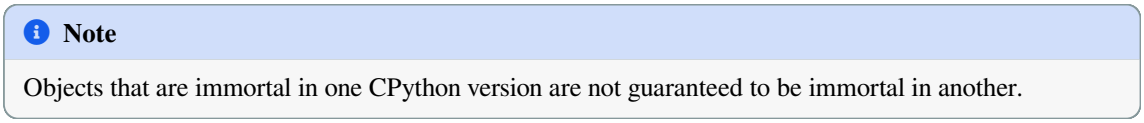
Ajouté dans la version 3.14.

int **`PyUnstable_IsImmortal`**(*PyObject* *obj)



This is *Unstable API*. It may change without warning in minor releases.

This function returns non-zero if *obj* is *immortal*, and zero otherwise. This function cannot fail.

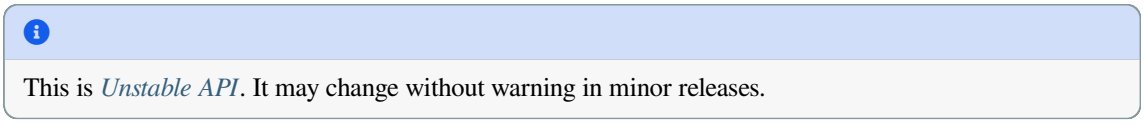


Note

Objects that are immortal in one CPython version are not guaranteed to be immortal in another.

Ajouté dans la version 3.14.

int **`PyUnstable_TryIncRef`**(*PyObject* *obj)



This is *Unstable API*. It may change without warning in minor releases.

Increments the reference count of *obj* if it is not zero. Returns `1` if the object's reference count was successfully incremented. Otherwise, this function returns `0`.

`PyUnstable_EnableTryIncRef()` must have been called earlier on *obj* or this function may spuriously return `0` in the *free threading* build.

This function is logically equivalent to the following C code, except that it behaves atomically in the *free threading* build :

```
if (Py_REFCNT(op) > 0) {
   Py_INCREF(op);
   return 1;
}
return 0;
```

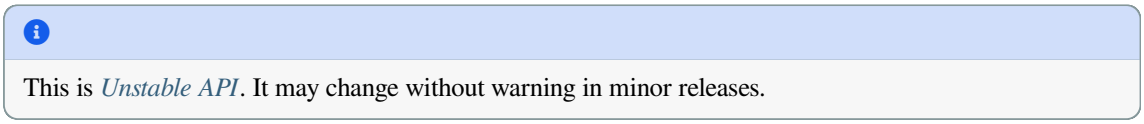
This is intended as a building block for managing weak references without the overhead of a Python *weak reference object*.

Typically, correct use of this function requires support from *obj*'s deallocator (`tp_dealloc`). For example, the following sketch could be adapted to implement a "weakmap" that works like a `WeakValueDictionary` for a specific type :

```
PyMutex mutex;

PyObject *
add_entry(weakmap_key_type *key, PyObject *value)
{
    PyUnstable_EnableTryIncRef(value);
    weakmap_type weakmap = ...;
    PyMutex_Lock(&mutex);
    weakmap_add_entry(weakmap, key, value);
    PyMutex_Unlock(&mutex);
    Py_RETURN_NONE;
}

PyObject *
get_value(weakmap_key_type *key)
{
    weakmap_type weakmap = ...;
    PyMutex_Lock(&mutex);
    PyObject *result = weakmap_find(weakmap, key);
    if (PyUnstable_TryIncRef(result)) {
        // `result` is safe to use
        PyMutex_Unlock(&mutex);
        return result;
    }
    // if we get here, `result` is starting to be garbage-collected,
    // but has not been removed from the weakmap yet
    PyMutex_Unlock(&mutex);
    return NULL;
}

// tp_dealloc function for weakmap values
void
value_dealloc(PyObject *value)
{
    weakmap_type weakmap = ...;
    PyMutex_Lock(&mutex);
    weakmap_remove_value(weakmap, value);

    ...
    PyMutex_Unlock(&mutex);
}
```

Ajouté dans la version 3.14.

void **PyUnstable_EnableTryIncRef**(*PyObject* *obj)

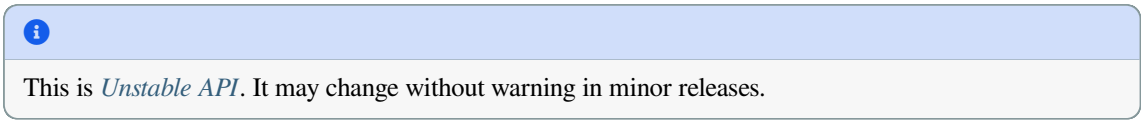


This is *Unstable API*. It may change without warning in minor releases.

Enables subsequent uses of `PyUnstable_TryIncRef()` on *obj*. The caller must hold a *strong reference* to *obj* when calling this.

Ajouté dans la version 3.14.

int **PyUnstable_Object_IsUniquelyReferenced**(*PyObject* *op)



This is *Unstable API*. It may change without warning in minor releases.

Determine if *op* only has one reference.

On GIL-enabled builds, this function is equivalent to `Py_REFCNT(op) == 1`.

On a *free threaded* build, this checks if *op*'s *reference count* is equal to one and additionally checks if *op* is only used by this thread. `Py_REFCNT(op) == 1` is **not** thread-safe on free threaded builds; prefer this function.

The caller must hold an *attached thread state*, despite the fact that this function doesn't call into the Python interpreter. This function cannot fail.

Ajouté dans la version 3.14.

8.2 Call Protocol

CPython supports two different calling protocols : *tp_call* and vectorcall.

8.2.1 The *tp_call* Protocol

Instances of classes that set `tp_call` are callable. The signature of the slot is :

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

A call is made using a tuple for the positional arguments and a dict for the keyword arguments, similarly to `callable(*args, **kwargs)` in Python code. *args* must be non-NULL (use an empty tuple if there are no arguments) but *kwargs* may be *NULL* if there are no keyword arguments.

This convention is not only used by *tp_call* : `tp_new` and `tp_init` also pass arguments this way.

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

8.2.2 The Vectorcall Protocol

Ajouté dans la version 3.9.

The vectorcall protocol was introduced in **PEP 590** as an additional protocol for making calls more efficient.

As rule of thumb, CPython will prefer the vectorcall for internal calls if the callable supports it. However, this is not a hard rule. Additionally, some third-party extensions use *tp_call* directly (rather than using `PyObject_Call()`). Therefore, a class supporting vectorcall must also implement `tp_call`. Moreover, the callable must behave the same regardless of which protocol is used. The recommended way to achieve this is by setting `tp_call` to `PyVectorcall_Call()`. This bears repeating :

Avertissement

A class supporting vectorcall **must** also implement *tp_call* with the same semantics.

Modifié dans la version 3.12 : The *Py_TPFLAGS_HAVE_VECTORCALL* flag is now removed from a class when the class's `__call__()` method is reassigned. (This internally sets *tp_call* only, and thus may make it behave differently than the vectorcall function.) In earlier Python versions, vectorcall should only be used with *immutable* or static types.

A class should not implement vectorcall if that would be slower than *tp_call*. For example, if the callee needs to convert the arguments to an args tuple and kwargs dict anyway, then there is no point in implementing vectorcall.

Classes can implement the vectorcall protocol by enabling the *Py_TPFLAGS_HAVE_VECTORCALL* flag and setting *tp_vectorcall_offset* to the offset inside the object structure where a *vectorcallfunc* appears. This is a pointer to a function with the following signature :

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

Part of the Stable ABI *Depuis la version 3.12.*

- *callable* is the object being called.
- ***args* is a C array consisting of the positional arguments followed by the** values of the keyword arguments. This can be *NULL* if there are no arguments.
- ***nargsf* is the number of positional arguments plus possibly the** *PY_VECTORCALL_ARGUMENTS_OFFSET* flag. To get the actual number of positional arguments from *nargsf*, use *PyVectorcall_NARGS()*.
- ***kwnames* is a tuple containing the names of the keyword arguments ;** in other words, the keys of the kwargs dict. These names must be strings (instances of `str` or a subclass) and they must be unique. If there are no keyword arguments, then *kwnames* can instead be *NULL*.

PY_VECTORCALL_ARGUMENTS_OFFSET

Part of the Stable ABI *Depuis la version 3.12.* If this flag is set in a vectorcall *nargsf* argument, the callee is allowed to temporarily change `args[-1]`. In other words, *args* points to argument 1 (not 0) in the allocated vector. The callee must restore the value of `args[-1]` before returning.

For *PyObject_VectorcallMethod()*, this flag means instead that `args[0]` may be changed.

Whenever they can do so cheaply (without additional allocation), callers are encouraged to use *PY_VECTORCALL_ARGUMENTS_OFFSET*. Doing so will allow callables such as bound methods to make their onward calls (which include a prepended *self* argument) very efficiently.

Ajouté dans la version 3.8.

To call an object that implements vectorcall, use a *call API* function as with any other callable. *PyObject_Vectorcall()* will usually be most efficient.

Contrôle de la récursion

When using *tp_call*, callees do not need to worry about *recursion* : CPython uses *Py_EnterRecursiveCall()* and *Py_LeaveRecursiveCall()* for calls made using *tp_call*.

For efficiency, this is not the case for calls done using vectorcall : the callee should use *Py_EnterRecursiveCall* and *Py_LeaveRecursiveCall* if needed.

Vectorcall Support API

Py_ssize_t **PyVectorcall_NARGS**(size_t nargsf)

Part of the Stable ABI *Depuis la version 3.12.* Given a vectorcall *nargsf* argument, return the actual number of arguments. Currently equivalent to :

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

However, the function `PyVectorcall_NARGS` should be used to allow for future extensions.

Ajouté dans la version 3.8.

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

If *op* does not support the vectorcall protocol (either because the type does not or because the specific instance does not), return *NULL*. Otherwise, return the vectorcall function pointer stored in *op*. This function never raises an exception.

This is mostly useful to check whether or not *op* supports vectorcall, which can be done by checking `PyVectorcall_Function(op) != NULL`.

Ajouté dans la version 3.9.

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

Part of the Stable ABI *Depuis la version 3.12.* Call *callable*'s `vectorcallfunc` with positional and keyword arguments given in a tuple and dict, respectively.

This is a specialized function, intended to be put in the `tp_call` slot or be used in an implementation of `tp_call`. It does not check the `Py_TPFLAGS_HAVE_VECTORCALL` flag and it does not fall back to `tp_call`.

Ajouté dans la version 3.8.

8.2.3 Object Calling API

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

The following table summarizes the available functions; please see individual documentation for details.

Fonction	appelable	args	kwargs
`PyObject_Call()`	`PyObject *`	*n*-uplet	dict/`NULL`
`PyObject_CallNoArgs()`	`PyObject *`	---	---
`PyObject_CallOneArg()`	`PyObject *`	1 object	---
`PyObject_CallObject()`	`PyObject *`	tuple/`NULL`	---
`PyObject_CallFunction()`	`PyObject *`	format	---
`PyObject_CallMethod()`	obj + `char*`	format	---
`PyObject_CallFunctionObjArgs()`	`PyObject *`	variadic	---
`PyObject_CallMethodObjArgs()`	obj + name	variadic	---
`PyObject_CallMethodNoArgs()`	obj + name	---	---
`PyObject_CallMethodOneArg()`	obj + name	1 object	---
`PyObject_Vectorcall()`	`PyObject *`	vectorcall	vectorcall
`PyObject_VectorcallDict()`	`PyObject *`	vectorcall	dict/`NULL`
`PyObject_VectorcallMethod()`	arg + name	vectorcall	vectorcall

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Appelle un objet Python appelable *callable*, avec des arguments donnés par le *n*-uplet *args*, et des arguments nommés donnés par le dictionnaire *kwargs*.

args must not be *NULL*; use an empty tuple if no arguments are needed. If no named arguments are needed, *kwargs* can be *NULL*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Ceci est l'équivalent de l'expression Python : `callable(*args, **kwargs)`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.10.* Call a callable Python object *callable* without any arguments. It is the most efficient way to call a callable Python object without any argument.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Ajouté dans la version 3.9.

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

Valeur de retour : nouvelle référence. Call a callable Python object *callable* with exactly 1 positional argument *arg* and no keyword arguments.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Ajouté dans la version 3.9.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Call a callable Python object *callable*, with arguments given by the tuple *args*. If no arguments are needed, then *args* can be *NULL*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Ceci est l'équivalent de l'expression Python : `callable(*args)`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Call a callable Python object *callable*, with a variable number of C arguments. The C arguments are described using a `Py_BuildValue()` style format string. The format can be *NULL*, indicating that no arguments are provided.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Ceci est l'équivalent de l'expression Python : `callable(*args)`.

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

Modifié dans la version 3.4 : The type of *format* was changed from `char *`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Call the method named *name* of object *obj* with a variable number of C arguments. The C arguments are described by a `Py_BuildValue()` format string that should produce a tuple.

The format can be *NULL*, indicating that no arguments are provided.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

This is the equivalent of the Python expression : `obj.name(arg1, arg2, ...)`.

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

Modifié dans la version 3.4 : The types of *name* and *format* were changed from `char *`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Call a callable Python object *callable*, with a variable number of `PyObject*` arguments. The arguments are provided as a variable number of parameters followed by *NULL*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

This is the equivalent of the Python expression : `callable(arg1, arg2, ...)`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Call a method of the Python object *obj*, where the name of the method is given as a Python string object in *name*. It is called with a variable number of `PyObject*` arguments. The arguments are provided as a variable number of parameters followed by *NULL*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

PyObject ***PyObject_CallMethodNoArgs**(*PyObject* *obj, *PyObject* *name)

Call a method of the Python object *obj* without arguments, where the name of the method is given as a Python string object in *name*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Ajouté dans la version 3.9.

PyObject ***PyObject_CallMethodOneArg**(*PyObject* *obj, *PyObject* *name, *PyObject* *arg)

Call a method of the Python object *obj* with a single positional argument *arg*, where the name of the method is given as a Python string object in *name*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Ajouté dans la version 3.9.

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

Part of the Stable ABI *Depuis la version 3.12.* Call a callable Python object *callable*. The arguments are the same as for `vectorcallfunc`. If *callable* supports *vectorcall*, this directly calls the vectorcall function stored in *callable*.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Ajouté dans la version 3.9.

PyObject ***PyObject_VectorcallDict**(*PyObject* *callable, *PyObject* *const *args, size_t nargsf, *PyObject* *kwdict)

Call *callable* with positional arguments passed exactly as in the *vectorcall* protocol, but with keyword arguments passed as a dictionary *kwdict*. The *args* array contains only the positional arguments.

Regardless of which protocol is used internally, a conversion of arguments needs to be done. Therefore, this function should only be used if the caller already has a dictionary ready to use for the keyword arguments, but not a tuple for the positional arguments.

Ajouté dans la version 3.9.

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

Part of the Stable ABI *Depuis la version 3.12.* Call a method using the vectorcall calling convention. The name of the method is given as a Python string *name*. The object whose method is called is *args[0]*, and the *args* array starting at *args[1]* represents the arguments of the call. There must be at least one positional argument. *nargsf* is the number of positional arguments including *args[0]*, plus `PY_VECTORCALL_ARGUMENTS_OFFSET` if the value of `args[0]` may temporarily be changed. Keyword arguments can be passed just like in `PyObject_Vectorcall()`.

If the object has the `Py_TPFLAGS_METHOD_DESCRIPTOR` feature, this will call the unbound method object with the full *args* vector as arguments.

Return the result of the call on success, or raise an exception and return *NULL* on failure.

Ajouté dans la version 3.9.

8.2.4 Call Support API

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

Part of the Stable ABI. Détermine si l'objet *o* est appelable. Renvoie `1` si c'est le cas, et `0` sinon. Cette fonction réussit toujours.

8.3 Number Protocol

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

Part of the Stable ABI. Returns `1` if the object *o* provides numeric protocols, and false otherwise. This function always succeeds.

Modifié dans la version 3.8 : Returns `1` if *o* is an index integer.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the result of adding *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 + o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the result of subtracting *o2* from *o1*, or `NULL` on failure. This is the equivalent of the Python expression `o1 - o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the result of multiplying *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 * o2`.

PyObject ***PyNumber_MatrixMultiply**(*PyObject* *o1, *PyObject* *o2)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.7.* Returns the result of matrix multiplication on *o1* and *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 @ o2`.

Ajouté dans la version 3.5.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the floor of *o1* divided by *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 // o2`.

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

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

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the remainder of dividing *o1* by *o2*, or `NULL` on failure. This is the equivalent of the Python expression `o1 % o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. See the built-in function `divmod()`. Returns `NULL` on failure. This is the equivalent of the Python expression `divmod(o1, o2)`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. See the built-in function `pow()`. Returns `NULL` on failure. This is the equivalent of the Python expression `pow(o1, o2, o3)`, where *o3* is optional. If *o3* is to be ignored, pass *Py_None* in its place (passing `NULL` for *o3* would cause an illegal memory access).

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the negation of *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `-o`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `+o`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the absolute value of *o*, or `NULL` on failure. This is the equivalent of the Python expression `abs(o)`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the bitwise negation of *o* on success, or `NULL` on failure. This is the equivalent of the Python expression `~o`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the result of left shifting *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 << o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the result of right shifting *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 >> o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the "bitwise and" of *o1* and *o2* on success and `NULL` on failure. This is the equivalent of the Python expression `o1 & o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the "bitwise exclusive or" of *o1* by *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 ^ o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the "bitwise or" of *o1* and *o2* on success, or `NULL` on failure. This is the equivalent of the Python expression `o1 | o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the result of adding *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 += o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the result of subtracting *o2* from *o1*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 -= o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the result of multiplying *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 *= o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.7.* Returns the result of matrix multiplication on *o1* and *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 @= o2`.

Ajouté dans la version 3.5.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the mathematical floor of dividing *o1* by *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 //= o2`.

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

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

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the remainder of dividing *o1* by *o2*, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 %= o2`.

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

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

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the result of left shifting *o1* by *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 <<= o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the result of right shifting *o1* by *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 >>= o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the "bitwise and" of *o1* and *o2* on success and `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 &= o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the "bitwise exclusive or" of *o1* by *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 ^= o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the "bitwise or" of *o1* and *o2* on success, or `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python statement `o1 |= o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the *o* converted to an integer object on success, or `NULL` on failure. This is the equivalent of the Python expression `int(o)`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the *o* converted to a float object on success, or `NULL` on failure. This is the equivalent of the Python expression `float(o)`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the *o* converted to a Python int on success or `NULL` with a `TypeError` exception raised on failure.

Modifié dans la version 3.10 : The result always has exact type `int`. Previously, the result could have been an instance of a subclass of `int`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Returns the integer *n* converted to base *base* as a string. The *base* argument must be one of 2, 8, 10, or 16. For base 2, 8, or 16, the returned string is prefixed with a base marker of `'0b'`, `'0o'`, or `'0x'`, respectively. If *n* is not a Python int, it is converted with *PyNumber_Index()* first.

Py_ssize_t **PyNumber_AsSsize_t**(*PyObject* *o, *PyObject* *exc)

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

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

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

Part of the Stable ABI *Depuis la version 3.8.* Returns `1` if *o* is an index integer (has the `nb_index` slot of the `tp_as_number` structure filled in), and `0` otherwise. This function always succeeds.

8.4 Sequence Protocol

int **PySequence_Check**(*PyObject* *o)

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

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

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

Part of the Stable ABI. Returns the number of objects in sequence *o* on success, and `-1` on failure. This is equivalent to the Python expression `len(o)`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the concatenation of *o1* and *o2* on success, and `NULL` on failure. This is the equivalent of the Python expression `o1 + o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the result of repeating sequence object *o* *count* times, or `NULL` on failure. This is the equivalent of the Python expression `o * count`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the concatenation of *o1* and *o2* on success, and `NULL` on failure. The operation is done *in-place* when *o1* supports it. This is the equivalent of the Python expression `o1 += o2`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the result of repeating sequence object *o* *count* times, or `NULL` on failure. The operation is done *in-place* when *o* supports it. This is the equivalent of the Python expression `o *= count`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the *i*th element of *o*, or `NULL` on failure. This is the equivalent of the Python expression `o[i]`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the slice of sequence object *o* between *i1* and *i2*, or `NULL` on failure. This is the equivalent of the Python expression `o[i1:i2]`.

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

Part of the Stable ABI. Assign object *v* to the *i*th element of *o*. Raise an exception and return `-1` on failure; return `0` on success. This is the equivalent of the Python statement `o[i] = v`. This function *does not* steal a reference to *v*.

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

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

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

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

Part of the Stable ABI. Assign the sequence object *v* to the slice in sequence object *o* from *i1* to *i2*. This is the equivalent of the Python statement `o[i1:i2] = v`.

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

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

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

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

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

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

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

Part of the Stable ABI. Alias for `PySequence_Contains()`.

Obsolète depuis la version 3.14 : The function is *soft deprecated* and should no longer be used to write new code.

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

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

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a list object with the same contents as the sequence or iterable *o*, or `NULL` on failure. The returned list is guaranteed to be new. This is equivalent to the Python expression `list(o)`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a tuple object with the same contents as the sequence or iterable *o*, or `NULL` on failure. If *o* is a tuple, a new reference will be returned, otherwise a tuple will be constructed with the appropriate contents. This is equivalent to the Python expression `tuple(o)`.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the sequence or iterable *o* as an object usable by the other `PySequence_Fast*` family of functions. If the object is not a sequence or iterable, raises `TypeError` with *m* as the message text. Returns `NULL` on failure.

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

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

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

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

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

Valeur de retour : référence empruntée. Return the *i*th element of *o*, assuming that *o* was returned by `PySequence_Fast()`, *o* is not `NULL`, and that *i* is within bounds.

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

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

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

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

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

8.5 Protocole de correspondance

Voir aussi `PyObject_GetItem()`, `PyObject_SetItem()` et `PyObject_DelItem()`.

int **PyMapping_Check**(*PyObject* *o)

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

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

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

Part of the Stable ABI. Renvoie le nombre de clefs dans l'objet *o* et `-1` en cas d'échec. C'est l'équivalent de l'expression Python `len(o)`.

PyObject ***PyMapping_GetItemString**(*PyObject* *o, const char *key)

Valeur de retour : nouvelle référence. Part of the Stable ABI. This is the same as `PyObject_GetItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

int **PyMapping_GetOptionalItem**(*PyObject* *obj, *PyObject* *key, *PyObject* **result)

Part of the Stable ABI *Depuis la version 3.13.* Variant of `PyObject_GetItem()` which doesn't raise `KeyError` if the key is not found.

If the key is found, return `1` and set **result* to a new *strong reference* to the corresponding value. If the key is not found, return `0` and set **result* to `NULL`; the `KeyError` is silenced. If an error other than `KeyError` is raised, return `-1` and set **result* to `NULL`.

Ajouté dans la version 3.13.

int **PyMapping_GetOptionalItemString**(*PyObject* *obj, const char *key, *PyObject* **result)

Part of the Stable ABI *Depuis la version 3.13.* This is the same as `PyMapping_GetOptionalItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Ajouté dans la version 3.13.

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

Part of the Stable ABI. This is the same as `PyObject_SetItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

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

This is an alias of `PyObject_DelItem()`.

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

This is the same as `PyObject_DelItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

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

Part of the Stable ABI *Depuis la version 3.13.* Return `1` if the mapping object has the key *key* and `0` otherwise. This is equivalent to the Python expression `key in o`. On failure, return `-1`.

Ajouté dans la version 3.13.

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

Part of the Stable ABI *Depuis la version 3.13.* This is the same as `PyMapping_HasKeyWithError()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Ajouté dans la version 3.13.

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

Part of the Stable ABI. Renvoie `1` si l'objet de correspondance possède une clef *key* et `0` sinon. C'est l'équivalent de l'expression Python `key in o`. Cette fonction ne provoque jamais d'erreur.

> **Note**
>
> Exceptions which occur when this calls `__getitem__()` method are silently ignored. For proper error handling, use `PyMapping_HasKeyWithError()`, `PyMapping_GetOptionalItem()` or `PyObject_GetItem()` instead.

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

Part of the Stable ABI. This is the same as `PyMapping_HasKey()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

> **Note**
>
> Exceptions that occur when this calls `__getitem__()` method or while creating the temporary `str` object are silently ignored. For proper error handling, use *`PyMapping_HasKeyStringWithError()`*, *`PyMapping_GetOptionalItemString()`* or *`PyMapping_GetItemString()`* instead.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Renvoie la liste des clefs dans l'objet *o*. En cas d'échec, renvoie *NULL*.

Modifié dans la version 3.7 : Auparavant, la fonction renvoyait une liste ou un *n*-uplet.

PyObject ***PyMapping_Values**(*PyObject* *o)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Renvoie la liste des valeurs dans l'objet *o*. En cas d'échec, renvoie *NULL*.

Modifié dans la version 3.7 : Auparavant, la fonction renvoyait une liste ou un *n*-uplet.

PyObject ***PyMapping_Items**(*PyObject* *o)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Renvoie la liste des éléments dans l'objet *o*, où chaque élément est un n-uplet contenant une paire clef-valeur. En cas d'échec, renvoie *NULL*.

Modifié dans la version 3.7 : Auparavant, la fonction renvoyait une liste ou un *n*-uplet.

8.6 Protocole d'itération

Il existe deux fonctions dédiées à l'interaction avec les itérateurs.

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

Part of the Stable ABI *Depuis la version 3.8.* Return non-zero if the object *o* can be safely passed to *`PyIter_NextItem()`* and `0` otherwise. This function always succeeds.

int **PyAIter_Check**(*PyObject* *o)

Part of the Stable ABI *Depuis la version 3.10.* Return non-zero if the object *o* provides the `AsyncIterator` protocol, and `0` otherwise. This function always succeeds.

Ajouté dans la version 3.10.

int **PyIter_NextItem**(*PyObject* *iter, *PyObject* **item)

Part of the Stable ABI *Depuis la version 3.14.* Return `1` and set *item* to a *strong reference* of the next value of the iterator *iter* on success. Return `0` and set *item* to `NULL` if there are no remaining values. Return `-1`, set *item* to `NULL` and set an exception on error.

Ajouté dans la version 3.14.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. This is an older version of `PyIter_NextItem()`, which is retained for backwards compatibility. Prefer *`PyIter_NextItem()`*.

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

type **PySendResult**

The enum value used to represent different results of *`PyIter_Send()`*.

Ajouté dans la version 3.10.

PySendResult **PyIter_Send**(*PyObject* *iter, *PyObject* *arg, *PyObject* **presult)

Part of the Stable ABI *Depuis la version 3.10.* Sends the *arg* value into the iterator *iter*. Returns :

— `PYGEN_RETURN` if iterator returns. Return value is returned via *presult*.

— `PYGEN_NEXT` if iterator yields. Yielded value is returned via *presult*.

— `PYGEN_ERROR` if iterator has raised and exception. *presult* is set to `NULL`.
Ajouté dans la version 3.10.

8.7 Protocole tampon

Certains objets Python enveloppent l'accès à un tableau de mémoire sous-jacente (nommée zone tampon ou simplement tampon, *buffer* en anglais). Les objets natifs `bytes` et `bytearray` en sont des exemples, ainsi que quelques types d'extension comme `array.array`. Les bibliothèques tierces peuvent définir leurs propres types à des fins spéciales, telles que le traitement d'image ou l'analyse numérique.

Alors que chacun de ces types a sa propre sémantique, ils partagent la caractéristique commune d'être soutenus par un tampon de mémoire important. Il est donc souhaitable, dans certains cas, d'accéder à cette mémoire directement sans l'étape intermédiaire de copie.

Python provides such a facility at the C and Python level in the form of the *buffer protocol*. This protocol has two sides :

— on the producer side, a type can export a "buffer interface" which allows objects of that type to expose information about their underlying buffer. This interface is described in the section *Buffer Object Structures* ; for Python see python-buffer-protocol.
— on the consumer side, several means are available to obtain a pointer to the raw underlying data of an object (for example a method parameter). For Python see `memoryview`.

Des objets simples tels que `bytes` et `bytearray` exposent leur tampon sous-jacent dans un format orienté octet. D'autres formes sont possibles ; par exemple, les éléments exposés par un `array.array` peuvent être des valeurs multi-octets.

An example consumer of the buffer interface is the `write()` method of file objects : any object that can export a series of bytes through the buffer interface can be written to a file. While `write()` only needs read-only access to the internal contents of the object passed to it, other methods such as `readinto()` need write access to the contents of their argument. The buffer interface allows objects to selectively allow or reject exporting of read-write and read-only buffers.

Un consommateur de l'interface tampon peut acquérir un tampon sur un objet cible de deux manières :
— appelez *`PyObject_GetBuffer()`* avec les paramètres appropriés ;
— appelez *`PyArg_ParseTuple()`* (ou l'un de ses fonctions sœurs) avec l'un des `y*`, `w*` ou `s*` *format codes*.
Dans les deux cas, *`PyBuffer_Release()`* doit être appelée quand le tampon n'est plus nécessaire. Ne pas le faire peut conduire à divers problèmes tels que des fuites de ressources.

Ajouté dans la version 3.12 : The buffer protocol is now accessible in Python, see python-buffer-protocol and `memoryview`.

8.7.1 La structure *buffer*

Les structures tampons (ou simplement les "tampons", *buffers* en anglais) sont utiles pour exposer les données binaires d'un autre objet au programmeur Python. Elles peuvent également être utilisées comme un mécanisme de découpage sans copie. En utilisant leur capacité à référencer un bloc de mémoire, il est possible d'exposer toutes les données au programmeur Python assez facilement. La mémoire peut être un grand tableau constant dans une extension C, il peut s'agir d'un bloc brut de mémoire à manipuler avant de passer à une bibliothèque de système d'exploitation ou être utilisé pour transmettre des données structurées dans son format natif en mémoire.

Contrairement à la plupart des types de données exposés par l'interpréteur Python, les tampons ne sont pas de simples pointeurs vers *`PyObject`* mais plutôt des structures C simples. Cela leur permet d'être créés et copiés très simplement. lorsque vous avez besoin d'une enveloppe générique (*wrapper* en anglais) pour un tampon, un objet *memoryview* peut être créé.

For short instructions how to write an exporting object, see *Buffer Object Structures*. For obtaining a buffer, see *`PyObject_GetBuffer()`*.

type **`Py_buffer`**

Part of the Stable ABI *(including all members) Depuis la version 3.11.*

void ***buf**

A pointer to the start of the logical structure described by the buffer fields. This can be any location within the underlying physical memory block of the exporter. For example, with negative *strides* the value may point to the end of the memory block.

For *contiguous* arrays, the value points to the beginning of the memory block.

PyObject ***obj**

A new reference to the exporting object. The reference is owned by the consumer and automatically released (i.e. reference count decremented) and set to `NULL` by *PyBuffer_Release()*. The field is the equivalent of the return value of any standard C-API function.

As a special case, for *temporary* buffers that are wrapped by *PyMemoryView_FromBuffer()* or *PyBuffer_FillInfo()* this field is `NULL`. In general, exporting objects MUST NOT use this scheme.

Py_ssize_t **len**

`product(shape) * itemsize`. For contiguous arrays, this is the length of the underlying memory block. For non-contiguous arrays, it is the length that the logical structure would have if it were copied to a contiguous representation.

Accessing `((char *)buf)[0] up to ((char *)buf)[len-1]` is only valid if the buffer has been obtained by a request that guarantees contiguity. In most cases such a request will be *PyBUF_SIMPLE* or *PyBUF_WRITABLE*.

int **readonly**

An indicator of whether the buffer is read-only. This field is controlled by the *PyBUF_WRITABLE* flag.

Py_ssize_t **itemsize**

Item size in bytes of a single element. Same as the value of `struct.calcsize()` called on non-`NULL` *format* values.

Important exception : If a consumer requests a buffer without the *PyBUF_FORMAT* flag, *format* will be set to `NULL`, but *itemsize* still has the value for the original format.

If *shape* is present, the equality `product(shape) * itemsize == len` still holds and the consumer can use *itemsize* to navigate the buffer.

If *shape* is `NULL` as a result of a *PyBUF_SIMPLE* or a *PyBUF_WRITABLE* request, the consumer must disregard *itemsize* and assume `itemsize == 1`.

char ***format**

A *NULL* terminated string in `struct` module style syntax describing the contents of a single item. If this is `NULL`, `"B"` (unsigned bytes) is assumed.

This field is controlled by the *PyBUF_FORMAT* flag.

int **ndim**

The number of dimensions the memory represents as an n-dimensional array. If it is `0`, *buf* points to a single item representing a scalar. In this case, *shape*, *strides* and *suboffsets* MUST be `NULL`. The maximum number of dimensions is given by *PyBUF_MAX_NDIM*.

Py_ssize_t ***shape**

An array of *Py_ssize_t* of length *ndim* indicating the shape of the memory as an n-dimensional array. Note that `shape[0] * ... * shape[ndim-1] * itemsize` MUST be equal to *len*.

Shape values are restricted to `shape[n] >= 0`. The case `shape[n] == 0` requires special attention. See *complex arrays* for further information.

The shape array is read-only for the consumer.

Py_ssize_t ***strides**

An array of *Py_ssize_t* of length *ndim* giving the number of bytes to skip to get to a new element in each dimension.

Stride values can be any integer. For regular arrays, strides are usually positive, but a consumer MUST be able to handle the case `strides[n] <= 0`. See *complex arrays* for further information.

The strides array is read-only for the consumer.

Py_ssize_t ***suboffsets**

An array of *Py_ssize_t* of length *ndim*. If `suboffsets[n] >= 0`, the values stored along the nth dimension are pointers and the suboffset value dictates how many bytes to add to each pointer after de-referencing. A suboffset value that is negative indicates that no de-referencing should occur (striding in a contiguous memory block).

If all suboffsets are negative (i.e. no de-referencing is needed), then this field must be `NULL` (the default value).

This type of array representation is used by the Python Imaging Library (PIL). See *complex arrays* for further information how to access elements of such an array.

The suboffsets array is read-only for the consumer.

void ***internal**

This is for use internally by the exporting object. For example, this might be re-cast as an integer by the exporter and used to store flags about whether or not the shape, strides, and suboffsets arrays must be freed when the buffer is released. The consumer MUST NOT alter this value.

Constants :

PyBUF_MAX_NDIM

The maximum number of dimensions the memory represents. Exporters MUST respect this limit, consumers of multi-dimensional buffers SHOULD be able to handle up to `PyBUF_MAX_NDIM` dimensions. Currently set to 64.

8.7.2 Buffer request types

Buffers are usually obtained by sending a buffer request to an exporting object via *PyObject_GetBuffer()*. Since the complexity of the logical structure of the memory can vary drastically, the consumer uses the *flags* argument to specify the exact buffer type it can handle.

All *Py_buffer* fields are unambiguously defined by the request type.

request-independent fields

The following fields are not influenced by *flags* and must always be filled in with the correct values : *obj*, *buf*, *len*, *itemsize*, *ndim*.

readonly, format

PyBUF_WRITABLE

Controls the *readonly* field. If set, the exporter MUST provide a writable buffer or else report failure. Otherwise, the exporter MAY provide either a read-only or writable buffer, but the choice MUST be consistent for all consumers. For example, *PyBUF_SIMPLE* | *PyBUF_WRITABLE* can be used to request a simple writable buffer.

PyBUF_FORMAT

Controls the *format* field. If set, this field MUST be filled in correctly. Otherwise, this field MUST be `NULL`.

PyBUF_WRITABLE can be |'d to any of the flags in the next section. Since *PyBUF_SIMPLE* is defined as 0, *PyBUF_WRITABLE* can be used as a stand-alone flag to request a simple writable buffer.

PyBUF_FORMAT must be |'d to any of the flags except *PyBUF_SIMPLE*, because the latter already implies format `B` (unsigned bytes). `PyBUF_FORMAT` cannot be used on its own.

shape, strides, suboffsets

The flags that control the logical structure of the memory are listed in decreasing order of complexity. Note that each flag contains all bits of the flags below it.

Request	shape	strides	suboffsets
PyBUF_INDIRECT	oui	oui	si nécessaire
PyBUF_STRIDES	oui	oui	NULL
PyBUF_ND	oui	NULL	NULL
PyBUF_SIMPLE	NULL	NULL	NULL

contiguity requests

C or Fortran *contiguity* can be explicitly requested, with and without stride information. Without stride information, the buffer must be C-contiguous.

Request	shape	strides	suboffsets	contig
PyBUF_C_CONTIGUOUS	oui	oui	NULL	C
PyBUF_F_CONTIGUOUS	oui	oui	NULL	F
PyBUF_ANY_CONTIGUOUS	oui	oui	NULL	C or F
PyBUF_ND	oui	NULL	NULL	C

compound requests

All possible requests are fully defined by some combination of the flags in the previous section. For convenience, the buffer protocol provides frequently used combinations as single flags.

In the following table *U* stands for undefined contiguity. The consumer would have to call *PyBuffer_IsContiguous()* to determine contiguity.

Request	shape	strides	suboffsets	contig	lecture seule	format
PyBUF_FULL	oui	oui	si nécessaire	U	0	oui
PyBUF_FULL_RO	oui	oui	si nécessaire	U	0 ou 1	oui
PyBUF_RECORDS	oui	oui	NULL	U	0	oui
PyBUF_RECORDS_RO	oui	oui	NULL	U	0 ou 1	oui
PyBUF_STRIDED	oui	oui	NULL	U	0	NULL
PyBUF_STRIDED_RO	oui	oui	NULL	U	0 ou 1	NULL
PyBUF_CONTIG	oui	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	oui	NULL	NULL	C	0 ou 1	NULL

8.7.3 Complex arrays

NumPy-style : shape and strides

The logical structure of NumPy-style arrays is defined by *itemsize*, *ndim*, *shape* and *strides*.

If `ndim == 0`, the memory location pointed to by *buf* is interpreted as a scalar of size *itemsize*. In that case, both *shape* and *strides* are `NULL`.

If *strides* is `NULL`, the array is interpreted as a standard n-dimensional C-array. Otherwise, the consumer must access an n-dimensional array as follows :

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

As noted above, *buf* can point to any location within the actual memory block. An exporter can check the validity of a buffer with this function :

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

(suite sur la page suivante)

(suite de la page précédente)

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

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

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

PIL-style : shape, strides and suboffsets

In addition to the regular items, PIL-style arrays can contain pointers that must be followed in order to get to the next element in a dimension. For example, the regular three-dimensional C-array `char v[2][2][3]` can also be viewed as an array of 2 pointers to 2 two-dimensional arrays : `char (*v[2])[2][3]`. In suboffsets representation, those two pointers can be embedded at the start of *buf*, pointing to two `char x[2][3]` arrays that can be located anywhere in memory.

Here is a function that returns a pointer to the element in an N-D array pointed to by an N-dimensional index when there are both non-`NULL` strides and suboffsets :

```
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
                       Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >=0 ) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

8.7.4 Fonctions relatives aux tampons

int **PyObject_CheckBuffer**(*PyObject* *obj)

Part of the Stable ABI *Depuis la version 3.11.* Return `1` if *obj* supports the buffer interface otherwise `0`. When `1` is returned, it doesn't guarantee that *PyObject_GetBuffer()* will succeed. This function always succeeds.

int **PyObject_GetBuffer**(*PyObject* *exporter, *Py_buffer* *view, int flags)

Part of the Stable ABI *Depuis la version 3.11.* Send a request to *exporter* to fill in *view* as specified by *flags*. If the exporter cannot provide a buffer of the exact type, it MUST raise `BufferError`, set `view->obj` to `NULL` and return `-1`.

On success, fill in *view*, set `view->obj` to a new reference to *exporter* and return 0. In the case of chained buffer providers that redirect requests to a single object, `view->obj` MAY refer to this object instead of *exporter* (See *Buffer Object Structures*).

Successful calls to *PyObject_GetBuffer()* must be paired with calls to *PyBuffer_Release()*, similar to `malloc()` and `free()`. Thus, after the consumer is done with the buffer, *PyBuffer_Release()* must be called exactly once.

void **PyBuffer_Release**(*Py_buffer* *view)

Part of the Stable ABI *Depuis la version 3.11.* Release the buffer *view* and release the *strong reference* (i.e.

decrement the reference count) to the view's supporting object, `view->obj`. This function MUST be called when the buffer is no longer being used, otherwise reference leaks may occur.

It is an error to call this function on a buffer that was not obtained via *PyObject_GetBuffer()*.

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

Part of the Stable ABI *Depuis la version 3.11.* Return the implied *itemsize* from *format*. On error, raise an exception and return -1.

Ajouté dans la version 3.9.

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

Part of the Stable ABI *Depuis la version 3.11.* Return `1` if the memory defined by the *view* is C-style (*order* is `'C'`) or Fortran-style (*order* is `'F'`) *contiguous* or either one (*order* is `'A'`). Return `0` otherwise. This function always succeeds.

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

Part of the Stable ABI *Depuis la version 3.11.* Get the memory area pointed to by the *indices* inside the given *view*. *indices* must point to an array of `view->ndim` indices.

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

Part of the Stable ABI *Depuis la version 3.11.* Copy contiguous *len* bytes from *buf* to *view*. *fort* can be `'C'` or `'F'` (for C-style or Fortran-style ordering). `0` is returned on success, `-1` on error.

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

Part of the Stable ABI *Depuis la version 3.11.* Copy *len* bytes from *src* to its contiguous representation in *buf*. *order* can be `'C'` or `'F'` or `'A'` (for C-style or Fortran-style ordering or either one). `0` is returned on success, `-1` on error.

This function fails if *len* != *src->len*.

int **PyObject_CopyData**(*PyObject* *dest, *PyObject* *src)

Part of the Stable ABI *Depuis la version 3.11.* Copy data from *src* to *dest* buffer. Can convert between C-style and or Fortran-style buffers.

`0` is returned on success, `-1` on error.

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

Part of the Stable ABI *Depuis la version 3.11.* Fill the *strides* array with byte-strides of a *contiguous* (C-style if *order* is `'C'` or Fortran-style if *order* is `'F'`) array of the given shape with the given number of bytes per element.

int **PyBuffer_FillInfo**(*Py_buffer* *view, *PyObject* *exporter, void *buf, *Py_ssize_t* len, int readonly, int flags)

Part of the Stable ABI *Depuis la version 3.11.* Handle buffer requests for an exporter that wants to expose *buf* of size *len* with writability set according to *readonly*. *buf* is interpreted as a sequence of unsigned bytes.

The *flags* argument indicates the request type. This function always fills in *view* as specified by flags, unless *buf* has been designated as read-only and *PyBUF_WRITABLE* is set in *flags*.

On success, set `view->obj` to a new reference to *exporter* and return 0. Otherwise, raise `BufferError`, set `view->obj` to `NULL` and return `-1`;

If this function is used as part of a *getbufferproc*, *exporter* MUST be set to the exporting object and *flags* must be passed unmodified. Otherwise, *exporter* MUST be `NULL`.

CHAPITRE 9

Couche des objets concrets

Les fonctions de ce chapitre sont spécifiques à certains types d'objets Python. Leur donner un objet du mauvais type n'est pas une bonne idée, si vous recevez un objet d'un programme Python, et que vous n'êtes pas sûr qu'il soit du bon type, vous devez vérifier son type en premier. Par exemple, pour vérifier qu'un objet est un dictionnaire, utilisez `PyDict_Check()`. Ce chapitre est organisé comme un arbre généalogique de types d'objets Python.

Avertissement

Tandis que les fonctions décrites dans ce chapitre vérifient avec soin le type des objets qui leur sont passés, beaucoup d'entre elles ne vérifient pas que `NULL` est passé au lieu d'un objet valide. Autoriser `NULL` à être passé peut provoquer des violations d'accès à la mémoire et ainsi terminer immédiatement l'interpréteur.

9.1 Objets fondamentaux

Cette section décrit les objets de type Python et l'objet singleton `None`.

9.1.1 Objets type

type **PyTypeObject**

Part of the Limited API *(as an opaque struct).* The C structure of the objects used to describe built-in types.

PyTypeObject **PyType_Type**

Part of the Stable ABI. This is the type object for type objects ; it is the same object as `type` in the Python layer.

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

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

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

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

unsigned int **PyType_ClearCache**()

Part of the Stable ABI. Clear the internal lookup cache. Return the current version tag.

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

Part of the Stable ABI. Return the `tp_flags` member of *type*. This function is primarily meant for use with `Py_LIMITED_API`; the individual flag bits are guaranteed to be stable across Python releases, but access to `tp_flags` itself is not part of the *limited API*.

Ajouté dans la version 3.2.

Modifié dans la version 3.4 : The return type is now `unsigned long` rather than `long`.

PyObject ***PyType_GetDict**(*PyTypeObject* *type)

Return the type object's internal namespace, which is otherwise only exposed via a read-only proxy (`cls.__dict__`). This is a replacement for accessing `tp_dict` directly. The returned dictionary must be treated as read-only.

This function is meant for specific embedding and language-binding cases, where direct access to the dict is necessary and indirect access (e.g. via the proxy or `PyObject_GetAttr()`) isn't adequate.

Extension modules should continue to use `tp_dict`, directly or indirectly, when setting up their own types.

Ajouté dans la version 3.12.

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

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

int **PyType_AddWatcher**(*PyType_WatchCallback* callback)

Register *callback* as a type watcher. Return a non-negative integer ID which must be passed to future calls to `PyType_Watch()`. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

In free-threaded builds, `PyType_AddWatcher()` is not thread-safe, so it must be called at start up (before spawning the first thread).

Ajouté dans la version 3.12.

int **PyType_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* (previously returned from `PyType_AddWatcher()`). Return `0` on success, `-1` on error (e.g. if *watcher_id* was never registered.)

An extension should never call `PyType_ClearWatcher` with a *watcher_id* that was not returned to it by a previous call to `PyType_AddWatcher()`.

Ajouté dans la version 3.12.

int **PyType_Watch**(int watcher_id, *PyObject* *type)

Mark *type* as watched. The callback granted *watcher_id* by `PyType_AddWatcher()` will be called whenever `PyType_Modified()` reports a change to *type*. (The callback may be called only once for a series of consecutive modifications to *type*, if `_PyType_Lookup()` is not called on *type* between the modifications; this is an implementation detail and subject to change.)

An extension should never call `PyType_Watch` with a *watcher_id* that was not returned to it by a previous call to `PyType_AddWatcher()`.

Ajouté dans la version 3.12.

typedef int (***PyType_WatchCallback**)(*PyObject* *type)

Type of a type-watcher callback function.

The callback must not modify *type* or cause `PyType_Modified()` to be called on *type* or any type in its MRO; violating this rule could cause infinite recursion.

Ajouté dans la version 3.12.

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

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

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

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

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

Part of the Stable ABI. Return true if *a* is a subtype of *b*.

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

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Generic handler for the *tp_alloc* slot of a type object. Uses Python's default memory allocation mechanism to allocate memory for a new instance, zeros the memory, then initializes the memory as if by calling *PyObject_Init()* or *PyObject_InitVar()*.

Do not call this directly to allocate memory for an object; call the type's *tp_alloc* slot instead.

For types that support garbage collection (i.e., the *Py_TPFLAGS_HAVE_GC* flag is set), this function behaves like *PyObject_GC_New* or *PyObject_GC_NewVar* (except the memory is guaranteed to be zeroed before initialization), and should be paired with *PyObject_GC_Del()* in *tp_free*. Otherwise, it behaves like *PyObject_New* or *PyObject_NewVar* (except the memory is guaranteed to be zeroed before initialization) and should be paired with *PyObject_Free()* in *tp_free*.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Generic handler for the *tp_new* slot of a type object. Creates a new instance using the type's *tp_alloc* slot and returns the resulting object.

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

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

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

PyObject ***PyType_GetName**(*PyTypeObject* *type)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.11.* Return the type's name. Equivalent to getting the type's `__name__` attribute.

Ajouté dans la version 3.11.

PyObject ***PyType_GetQualName**(*PyTypeObject* *type)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.11.* Return the type's qualified name. Equivalent to getting the type's `__qualname__` attribute.

Ajouté dans la version 3.11.

PyObject ***PyType_GetFullyQualifiedName**(*PyTypeObject* *type)

Part of the Stable ABI *Depuis la version 3.13.* Return the type's fully qualified name. Equivalent to `f"{type.__module__}.{type.__qualname__}"`, or `type.__qualname__` if `type.__module__` is not a string or is equal to `"builtins"`.

Ajouté dans la version 3.13.

PyObject ***PyType_GetModuleName**(*PyTypeObject* *type)

Part of the Stable ABI *Depuis la version 3.13.* Return the type's module name. Equivalent to getting the `type.__module__` attribute.

Ajouté dans la version 3.13.

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

Part of the Stable ABI *Depuis la version 3.4.* Return the function pointer stored in the given slot. If the result is `NULL`, this indicates that either the slot is `NULL`, or that the function was called with invalid parameters. Callers will typically cast the result pointer into the appropriate function type.

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

Ajouté dans la version 3.4.

Modifié dans la version 3.10 : `PyType_GetSlot()` can now accept all types. Previously, it was limited to *heap types*.

PyObject *__PyType_GetModule__(*PyTypeObject* *type)

Part of the Stable ABI *Depuis la version 3.10.* Return the module object associated with the given type when the type was created using `PyType_FromModuleAndSpec()`.

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

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

Ajouté dans la version 3.9.

void *__PyType_GetModuleState__(*PyTypeObject* *type)

Part of the Stable ABI *Depuis la version 3.10.* Return the state of the module object associated with the given type. This is a shortcut for calling `PyModule_GetState()` on the result of `PyType_GetModule()`.

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

If the *type* has an associated module but its state is `NULL`, returns `NULL` without setting an exception.

Ajouté dans la version 3.9.

PyObject *__PyType_GetModuleByDef__(*PyTypeObject* *type, struct *PyModuleDef* *def)

Valeur de retour : référence empruntée. Part of the Stable ABI *Depuis la version 3.13.* Find the first superclass whose module was created from the given `PyModuleDef` *def*, and return that module.

If no module is found, raises a `TypeError` and returns `NULL`.

This function is intended to be used together with `PyModule_GetState()` to get module state from slot methods (such as `tp_init` or `nb_add`) and other places where a method's defining class cannot be passed using the `PyCMethod` calling convention.

The returned reference is *borrowed* from *type*, and will be valid as long as you hold a reference to *type*. Do not release it with `Py_DECREF()` or similar.

Ajouté dans la version 3.11.

int __PyType_GetBaseByToken__(*PyTypeObject* *type, void *token, *PyTypeObject* **result)

Part of the Stable ABI *Depuis la version 3.14.* Find the first superclass in *type*'s *method resolution order* whose `Py_tp_token` token is equal to the given one.

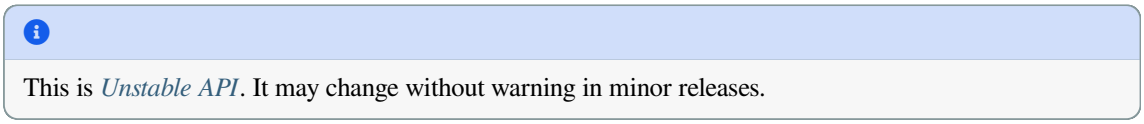
- If found, set **result* to a new *strong reference* to it and return `1`.
- If not found, set **result* to `NULL` and return `0`.
- On error, set **result* to `NULL` and return `-1` with an exception set.

The *result* argument may be `NULL`, in which case **result* is not set. Use this if you need only the return value.

The *token* argument may not be `NULL`.

Ajouté dans la version 3.14.

int __PyUnstable_Type_AssignVersionTag__(*PyTypeObject* *type)



This is *Unstable API*. It may change without warning in minor releases.

Attempt to assign a version tag to the given type.

Returns 1 if the type already had a valid version tag or a new one was assigned, or 0 if a new tag could not be assigned.

Ajouté dans la version 3.12.

Creating Heap-Allocated Types

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

PyObject ***PyType_FromMetaclass**(*PyTypeObject* *metaclass, *PyObject* *module, *PyType_Spec* *spec, *PyObject* *bases)

Part of the Stable ABI *Depuis la version 3.12.* Create and return a *heap type* from the *spec* (see `Py_TPFLAGS_HEAPTYPE`).

The metaclass *metaclass* is used to construct the resulting type object. When *metaclass* is `NULL`, the metaclass is derived from *bases* (or *Py_tp_base[s]* slots if *bases* is `NULL`, see below).

Metaclasses that override `tp_new` are not supported, except if `tp_new` is `NULL`.

The *bases* argument can be used to specify base classes; it can either be only one class or a tuple of classes. If *bases* is `NULL`, the *Py_tp_bases* slot is used instead. If that also is `NULL`, the *Py_tp_base* slot is used instead. If that also is `NULL`, the new type derives from `object`.

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

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

Note that this function does *not* fully match the behavior of calling `type()` or using the `class` statement. With user-provided base types or metaclasses, prefer *calling* `type` (or the metaclass) over `PyType_From*` functions. Specifically :
— `__new__()` is not called on the new class (and it must be set to `type.__new__`).
— `__init__()` is not called on the new class.
— `__init_subclass__()` is not called on any bases.
— `__set_name__()` is not called on new descriptors.
Ajouté dans la version 3.12.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.10.* Equivalent to `PyType_FromMetaclass(NULL, module, spec, bases)`.

Ajouté dans la version 3.9.

Modifié dans la version 3.10 : The function now accepts a single class as the *bases* argument and `NULL` as the `tp_doc` slot.

Modifié dans la version 3.12 : The function now finds and uses a metaclass corresponding to the provided base classes. Previously, only `type` instances were returned.

The `tp_new` of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides `tp_new` is deprecated.

Modifié dans la version 3.14 : Creating classes whose metaclass overrides `tp_new` is no longer allowed.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.3.* Equivalent to `PyType_FromMetaclass(NULL, NULL, spec, bases)`.

Ajouté dans la version 3.3.

Modifié dans la version 3.12 : The function now finds and uses a metaclass corresponding to the provided base classes. Previously, only `type` instances were returned.

The *tp_new* of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides *tp_new* is deprecated.

Modifié dans la version 3.14 : Creating classes whose metaclass overrides *tp_new* is no longer allowed.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Equivalent to `PyType_FromMetaclass(NULL, NULL, spec, NULL)`.

Modifié dans la version 3.12 : The function now finds and uses a metaclass corresponding to the base classes provided in *Py_tp_base[s]* slots. Previously, only `type` instances were returned.

The *tp_new* of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides *tp_new* is deprecated.

Modifié dans la version 3.14 : Creating classes whose metaclass overrides *tp_new* is no longer allowed.

int **PyType_Freeze**(*PyTypeObject* *type)

Part of the Stable ABI *Depuis la version 3.14.* Make a type immutable : set the *Py_TPFLAGS_IMMUTABLETYPE* flag.

All base classes of *type* must be immutable.

On success, return `0`. On error, set an exception and return `-1`.

The type must not be used before it's made immutable. For example, type instances must not be created before the type is made immutable.

Ajouté dans la version 3.14.

type **PyType_Spec**

Part of the Stable ABI *(including all members).* Structure defining a type's behavior.

const char ***name**

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

int **basicsize**

If positive, specifies the size of the instance in bytes. It is used to set *PyTypeObject.tp_basicsize*.

If zero, specifies that *tp_basicsize* should be inherited.

If negative, the absolute value specifies how much space instances of the class need *in addition* to the superclass. Use *PyObject_GetTypeData()* to get a pointer to subclass-specific memory reserved this way. For negative `basicsize`, Python will insert padding when needed to meet *tp_basicsize*'s alignment requirements.

Modifié dans la version 3.12 : Previously, this field could not be negative.

int **itemsize**

Size of one element of a variable-size type, in bytes. Used to set *PyTypeObject.tp_itemsize*. See `tp_itemsize` documentation for caveats.

If zero, *tp_itemsize* is inherited. Extending arbitrary variable-sized classes is dangerous, since some types use a fixed offset for variable-sized memory, which can then overlap fixed-sized memory used by a subclass. To help prevent mistakes, inheriting `itemsize` is only possible in the following situations :

— The base is not variable-sized (its *tp_itemsize*).
— The requested *PyType_Spec.basicsize* is positive, suggesting that the memory layout of the base class is known.
— The requested *PyType_Spec.basicsize* is zero, suggesting that the subclass does not access the instance's memory directly.
— With the *Py_TPFLAGS_ITEMS_AT_END* flag.

unsigned int **flags**

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

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

PyType_Slot ***slots**

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

Each slot ID should be specified at most once.

type **PyType_Slot**

Part of the Stable ABI *(including all members).* Structure defining optional functionality of a type, containing a slot ID and a value pointer.

int **slot**

A slot ID.

Slot IDs are named like the field names of the structures *PyTypeObject*, *PyNumberMethods*, *PySequenceMethods*, *PyMappingMethods* and *PyAsyncMethods* with an added `Py_` prefix. For example, use :
— `Py_tp_dealloc` to set *PyTypeObject.tp_dealloc*
— `Py_nb_add` to set *PyNumberMethods.nb_add*
— `Py_sq_length` to set *PySequenceMethods.sq_length*
An additional slot is supported that does not correspond to a `PyTypeObject` struct field :
— *Py_tp_token*
The following "offset" fields cannot be set using *PyType_Slot* :
— *tp_weaklistoffset* (use *Py_TPFLAGS_MANAGED_WEAKREF* instead if possible)
— *tp_dictoffset* (use *Py_TPFLAGS_MANAGED_DICT* instead if possible)
— *tp_vectorcall_offset* (use `"__vectorcalloffset__"` in *PyMemberDef*)
If it is not possible to switch to a `MANAGED` flag (for example, for vectorcall or to support Python older than 3.12), specify the offset in *Py_tp_members*. See *PyMemberDef documentation* for details.

The following internal fields cannot be set at all when creating a heap type :
— *tp_dict*, *tp_mro*, *tp_cache*, *tp_subclasses*, and *tp_weaklist*.
Setting `Py_tp_bases` or `Py_tp_base` may be problematic on some platforms. To avoid issues, use the *bases* argument of *PyType_FromSpecWithBases()* instead.

Modifié dans la version 3.9 : Slots in *PyBufferProcs* may be set in the unlimited API.

Modifié dans la version 3.11 : *bf_getbuffer* and *bf_releasebuffer* are now available under the *limited API*.

Modifié dans la version 3.14 : The field *tp_vectorcall* can now set using `Py_tp_vectorcall`. See the field's documentation for details.

void ***pfunc**

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

pfunc values may not be `NULL`, except for the following slots :
— `Py_tp_doc`
— *Py_tp_token* (for clarity, prefer *Py_TP_USE_SPEC* rather than `NULL`)

Py_tp_token

A *slot* that records a static memory layout ID for a class.

If the *PyType_Spec* of the class is statically allocated, the token can be set to the spec using the special value *Py_TP_USE_SPEC* :

```
static PyType_Slot foo_slots[] = {
   {Py_tp_token, Py_TP_USE_SPEC},
```

It can also be set to an arbitrary pointer, but you must ensure that :
— The pointer outlives the class, so it's not reused for something else while the class exists.
— It "belongs" to the extension module where the class lives, so it will not clash with other extensions.

Use *PyType_GetBaseByToken()* to check if a class's superclass has a given token -- that is, check whether the memory layout is compatible.

To get the token for a given class (without considering superclasses), use *PyType_GetSlot()* with Py_tp_token.

Ajouté dans la version 3.14.

Py_TP_USE_SPEC

Used as a value with *Py_tp_token* to set the token to the class's *PyType_Spec*. Expands to NULL.

Ajouté dans la version 3.14.

9.1.2 L'objet None

Note that the *PyTypeObject* for None is not directly exposed in the Python/C API. Since None is a singleton, testing for object identity (using == in C) is sufficient. There is no PyNone_Check() function for the same reason.

PyObject ***Py_None**

The Python None object, denoting lack of value. This object has no methods and is *immortal*.

Modifié dans la version 3.12 : *Py_None* is *immortal*.

Py_RETURN_NONE

Return *Py_None* from a function.

9.2 Objets numériques

9.2.1 Objets *Integer*

All integers are implemented as "long" integer objects of arbitrary size.

On error, most PyLong_As* APIs return (return type)-1 which cannot be distinguished from a number. Use *PyErr_Occurred()* to disambiguate.

type **PyLongObject**

Part of the Limited API *(as an opaque struct).* This subtype of *PyObject* represents a Python integer object.

PyTypeObject **PyLong_Type**

Part of the Stable ABI. This instance of *PyTypeObject* represents the Python integer type. This is the same object as int in the Python layer.

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

Return true if its argument is a *PyLongObject* or a subtype of *PyLongObject*. This function always succeeds.

int **PyLong_CheckExact**(*PyObject* *p)

Return true if its argument is a *PyLongObject*, but not a subtype of *PyLongObject*. This function always succeeds.

PyObject ***PyLong_FromLong**(long v)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new *PyLongObject* object from *v*, or NULL on failure.

Particularité de l'implémentation CPython : CPython keeps an array of integer objects for all integers between -5 and 256. When you create an int in that range you actually just get back a reference to the existing object.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new *PyLongObject* object from a C unsigned long, or NULL on failure.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new `PyLongObject` object from a C `Py_ssize_t`, or `NULL` on failure.

PyObject ***PyLong_FromSize_t**(size_t v)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new `PyLongObject` object from a C `size_t`, or `NULL` on failure.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new `PyLongObject` object from a C `long long`, or `NULL` on failure.

PyObject ***PyLong_FromInt32**(int32_t value)

PyObject ***PyLong_FromInt64**(int64_t value)

Part of the Stable ABI *Depuis la version 3.14.* Return a new `PyLongObject` object from a signed C `int32_t` or `int64_t`, or `NULL` with an exception set on failure.

Ajouté dans la version 3.14.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new `PyLongObject` object from a C `unsigned long long`, or `NULL` on failure.

PyObject ***PyLong_FromUInt32**(uint32_t value)

PyObject ***PyLong_FromUInt64**(uint64_t value)

Part of the Stable ABI *Depuis la version 3.14.* Return a new `PyLongObject` object from an unsigned C `uint32_t` or `uint64_t`, or `NULL` with an exception set on failure.

Ajouté dans la version 3.14.

PyObject ***PyLong_FromDouble**(double v)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new `PyLongObject` object from the integer part of *v*, or `NULL` on failure.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new `PyLongObject` based on the string value in *str*, which is interpreted according to the radix in *base*, or `NULL` on failure. If *pend* is non-`NULL`, **pend* will point to the end of *str* on success or to the first character that could not be processed on error. If *base* is `0`, *str* is interpreted using the integers definition; in this case, leading zeros in a non-zero decimal number raises a `ValueError`. If *base* is not `0`, it must be between `2` and `36`, inclusive. Leading and trailing whitespace and single underscores after a base specifier and between digits are ignored. If there are no digits or *str* is not NULL-terminated following the digits and trailing whitespace, `ValueError` will be raised.

Voir aussi

`PyLong_AsNativeBytes()` and `PyLong_FromNativeBytes()` functions can be used to convert a `PyLongObject` to/from an array of bytes in base `256`.

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

Valeur de retour : nouvelle référence. Convert a sequence of Unicode digits in the string *u* to a Python integer value.

Ajouté dans la version 3.3.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a Python integer from the pointer *p*. The pointer value can be retrieved from the resulting value using `PyLong_AsVoidPtr()`.

PyObject ***PyLong_FromNativeBytes**(const void *buffer, size_t n_bytes, int flags)

Part of the Stable ABI *Depuis la version 3.14.* Create a Python integer from the value contained in the first *n_bytes* of *buffer*, interpreted as a two's-complement signed number.

flags are as for *PyLong_AsNativeBytes()*. Passing `-1` will select the native endian that CPython was compiled with and assume that the most-significant bit is a sign bit. Passing `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` will produce the same result as calling *PyLong_FromUnsignedNativeBytes()*. Other flags are ignored.

Ajouté dans la version 3.13.

PyObject ***PyLong_FromUnsignedNativeBytes**(const void *buffer, size_t n_bytes, int flags)

Part of the Stable ABI *Depuis la version 3.14.* Create a Python integer from the value contained in the first *n_bytes* of *buffer*, interpreted as an unsigned number.

flags are as for *PyLong_AsNativeBytes()*. Passing `-1` will select the native endian that CPython was compiled with and assume that the most-significant bit is not a sign bit. Flags other than endian are ignored.

Ajouté dans la version 3.13.

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

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

Raise `OverflowError` if the value of *obj* is out of range for a `long`.

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

Modifié dans la version 3.8 : Use `__index__()` if available.

Modifié dans la version 3.10 : This function will no longer use `__int__()`.

long **PyLong_AS_LONG**(*PyObject* *obj)

A *soft deprecated* alias. Exactly equivalent to the preferred `PyLong_AsLong`. In particular, it can fail with `OverflowError` or another exception.

Obsolète depuis la version 3.14 : The function is soft deprecated.

int **PyLong_AsInt**(*PyObject* *obj)

Part of the Stable ABI *Depuis la version 3.13.* Similar to *PyLong_AsLong()*, but store the result in a C `int` instead of a C `long`.

Ajouté dans la version 3.13.

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

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

If the value of *obj* is greater than `LONG_MAX` or less than `LONG_MIN`, set **overflow* to `1` or `-1`, respectively, and return `-1`; otherwise, set **overflow* to `0`. If any other exception occurs set **overflow* to `0` and return `-1` as usual.

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

Modifié dans la version 3.8 : Use `__index__()` if available.

Modifié dans la version 3.10 : This function will no longer use `__int__()`.

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

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

Raise `OverflowError` if the value of *obj* is out of range for a `long long`.

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

Modifié dans la version 3.8 : Use `__index__()` if available.

Modifié dans la version 3.10 : This function will no longer use `__int__()`.

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

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

If the value of *obj* is greater than `LLONG_MAX` or less than `LLONG_MIN`, set **overflow* to `1` or `-1`, respectively, and return `-1`; otherwise, set **overflow* to `0`. If any other exception occurs set **overflow* to `0` and return `-1` as usual.

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

Ajouté dans la version 3.2.

Modifié dans la version 3.8 : Use `__index__()` if available.

Modifié dans la version 3.10 : This function will no longer use `__int__()`.

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

Part of the Stable ABI. Return a C *Py_ssize_t* representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a *Py_ssize_t*.

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

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

Part of the Stable ABI. Return a C `unsigned long` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a `unsigned long`.

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

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

Part of the Stable ABI. Return a C `size_t` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a `size_t`.

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

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

Part of the Stable ABI. Return a C `unsigned long long` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for an `unsigned long long`.

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

Modifié dans la version 3.1 : A negative *pylong* now raises `OverflowError`, not `TypeError`.

unsigned long **PyLong_AsUnsignedLongMask**(*PyObject* *obj)

Part of the Stable ABI. Return a C `unsigned long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

If the value of *obj* is out of range for an `unsigned long`, return the reduction of that value modulo `ULONG_MAX + 1`.

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

Modifié dans la version 3.8 : Use `__index__()` if available.

Modifié dans la version 3.10 : This function will no longer use `__int__()`.

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

Part of the Stable ABI. Return a C `unsigned long long` representation of *obj*. If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

If the value of *obj* is out of range for an `unsigned long long`, return the reduction of that value modulo `ULLONG_MAX + 1`.

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

Modifié dans la version 3.8 : Use `__index__()` if available.

Modifié dans la version 3.10 : This function will no longer use `__int__()`.

int **PyLong_AsInt32**(*PyObject* *obj, int32_t *value)

int **PyLong_AsInt64**(*PyObject* *obj, int64_t *value)

Part of the Stable ABI *Depuis la version 3.14.* Set **value* to a signed C `int32_t` or `int64_t` representation of *obj*.

If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.

If the *obj* value is out of range, raise an `OverflowError`.

Set **value* and return `0` on success. Set an exception and return `-1` on error.

value must not be `NULL`.

Ajouté dans la version 3.14.

int **PyLong_AsUInt32**(*PyObject* *obj, uint32_t *value)

int **PyLong_AsUInt64**(*PyObject* *obj, uint64_t *value)

Part of the Stable ABI *Depuis la version 3.14.* Set **value* to an unsigned C `uint32_t` or `uint64_t` representation of *obj*.

If *obj* is not an instance of *PyLongObject*, first call its `__index__()` method (if present) to convert it to a *PyLongObject*.
— If *obj* is negative, raise a `ValueError`.
— If the *obj* value is out of range, raise an `OverflowError`.
Set **value* and return `0` on success. Set an exception and return `-1` on error.

value must not be `NULL`.

Ajouté dans la version 3.14.

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

Part of the Stable ABI. Return a C `double` representation of *pylong*. *pylong* must be an instance of *PyLongObject*.

Raise `OverflowError` if the value of *pylong* is out of range for a `double`.

Returns `-1.0` on error. Use *PyErr_Occurred()* to disambiguate.

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

Part of the Stable ABI. Convert a Python integer *pylong* to a C `void` pointer. If *pylong* cannot be converted, an `OverflowError` will be raised. This is only assured to produce a usable `void` pointer for values created with *PyLong_FromVoidPtr()*.

Returns `NULL` on error. Use *PyErr_Occurred()* to disambiguate.

Py_ssize_t **PyLong_AsNativeBytes**(*PyObject* *pylong, void *buffer, *Py_ssize_t* n_bytes, int flags)

Part of the Stable ABI *Depuis la version 3.14.* Copy the Python integer value *pylong* to a native *buffer* of size *n_bytes*. The *flags* can be set to `-1` to behave similarly to a C cast, or to values documented below to control the behavior.

Returns `-1` with an exception raised on error. This may happen if *pylong* cannot be interpreted as an integer, or if *pylong* was negative and the `Py_ASNATIVEBYTES_REJECT_NEGATIVE` flag was set.

Otherwise, returns the number of bytes required to store the value. If this is equal to or less than *n_bytes*, the entire value was copied. All *n_bytes* of the buffer are written : large buffers are padded with zeroes.

If the returned value is greater than *n_bytes*, the value was truncated : as many of the lowest bits of the value as could fit are written, and the higher bits are ignored. This matches the typical behavior of a C-style downcast.

> **Note**
>
> Overflow is not considered an error. If the returned value is larger than *n_bytes*, most significant bits were discarded.

`0` will never be returned.

Values are always copied as two's-complement.

Usage example :

```
int32_t value;
Py_ssize_t bytes = PyLong_AsNativeBytes(pylong, &value, sizeof(value), -1);
if (bytes < 0) {
    // Failed. A Python exception was set with the reason.
    return NULL;
}
else if (bytes <= (Py_ssize_t)sizeof(value)) {
    // Success!
}
else {
    // Overflow occurred, but 'value' contains the truncated
    // lowest bits of pylong.
}
```

Passing zero to *n_bytes* will return the size of a buffer that would be large enough to hold the value. This may be larger than technically necessary, but not unreasonably so. If *n_bytes=0*, *buffer* may be `NULL`.

> **Note**
>
> Passing *n_bytes=0* to this function is not an accurate way to determine the bit length of the value.

To get at the entire Python value of an unknown size, the function can be called twice : first to determine the buffer size, then to fill it :

```
// Ask how much space we need.
Py_ssize_t expected = PyLong_AsNativeBytes(pylong, NULL, 0, -1);
if (expected < 0) {
    // Failed. A Python exception was set with the reason.
    return NULL;
}
assert(expected != 0);  // Impossible per the API definition.
uint8_t *bignum = malloc(expected);
if (!bignum) {
    PyErr_SetString(PyExc_MemoryError, "bignum malloc failed.");
    return NULL;
}
// Safely get the entire value.
Py_ssize_t bytes = PyLong_AsNativeBytes(pylong, bignum, expected, -1);
if (bytes < 0) {  // Exception has been set.
    free(bignum);
    return NULL;
}
else if (bytes > expected) {  // This should not be possible.
    PyErr_SetString(PyExc_RuntimeError,
        "Unexpected bignum truncation after a size check.");
    free(bignum);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return NULL;
}
// The expected success given the above pre-check.
// ... use bignum ...
free(bignum);
```

flags is either `-1` (`Py_ASNATIVEBYTES_DEFAULTS`) to select defaults that behave most like a C cast, or a combination of the other flags in the table below. Note that `-1` cannot be combined with other flags.

Currently, `-1` corresponds to `Py_ASNATIVEBYTES_NATIVE_ENDIAN | Py_ASNATIVEBYTES_UNSIGNED_BUFFER`.

Option	Valeur
Py_ASNATIVEBYTES_DEFAULTS	-1
Py_ASNATIVEBYTES_BIG_ENDIAN	0
Py_ASNATIVEBYTES_LITTLE_ENDIAN	1
Py_ASNATIVEBYTES_NATIVE_ENDIAN	3
Py_ASNATIVEBYTES_UNSIGNED_BUFFER	4
Py_ASNATIVEBYTES_REJECT_NEGATIVE	8
Py_ASNATIVEBYTES_ALLOW_INDEX	16

Specifying `Py_ASNATIVEBYTES_NATIVE_ENDIAN` will override any other endian flags. Passing `2` is reserved.

By default, sufficient buffer will be requested to include a sign bit. For example, when converting 128 with *n_bytes=1*, the function will return 2 (or more) in order to store a zero sign bit.

If `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` is specified, a zero sign bit will be omitted from size calculations. This allows, for example, 128 to fit in a single-byte buffer. If the destination buffer is later treated as signed, a positive input value may become negative. Note that the flag does not affect handling of negative values : for those, space for a sign bit is always requested.

Specifying `Py_ASNATIVEBYTES_REJECT_NEGATIVE` causes an exception to be set if *pylong* is negative. Without this flag, negative values will be copied provided there is enough space for at least one sign bit, regardless of whether `Py_ASNATIVEBYTES_UNSIGNED_BUFFER` was specified.

If `Py_ASNATIVEBYTES_ALLOW_INDEX` is specified and a non-integer value is passed, its `__index__()` method will be called first. This may result in Python code executing and other threads being allowed to run, which could cause changes to other objects or values in use. When *flags* is `-1`, this option is not set, and non-integer values will raise `TypeError`.

Note

With the default *flags* (`-1`, or *UNSIGNED_BUFFER* without *REJECT_NEGATIVE*), multiple Python inte-

gers can map to a single value without overflow. For example, both `255` and `-1` fit a single-byte buffer and set all its bits. This matches typical C cast behavior.

Ajouté dans la version 3.13.

int **PyLong_GetSign**(*PyObject* *obj, int *sign)

Get the sign of the integer object *obj*.

On success, set **sign* to the integer sign (0, -1 or +1 for zero, negative or positive integer, respectively) and return 0.

On failure, return -1 with an exception set. This function always succeeds if *obj* is a `PyLongObject` or its subtype.

Ajouté dans la version 3.14.

int **PyLong_IsPositive**(*PyObject* *obj)

Check if the integer object *obj* is positive (`obj > 0`).

If *obj* is an instance of `PyLongObject` or its subtype, return `1` when it's positive and `0` otherwise. Else set an exception and return `-1`.

Ajouté dans la version 3.14.

int **PyLong_IsNegative**(*PyObject* *obj)

Check if the integer object *obj* is negative (`obj < 0`).

If *obj* is an instance of `PyLongObject` or its subtype, return `1` when it's negative and `0` otherwise. Else set an exception and return `-1`.

Ajouté dans la version 3.14.

int **PyLong_IsZero**(*PyObject* *obj)

Check if the integer object *obj* is zero.

If *obj* is an instance of `PyLongObject` or its subtype, return `1` when it's zero and `0` otherwise. Else set an exception and return `-1`.

Ajouté dans la version 3.14.

PyObject ***PyLong_GetInfo**(void)

Part of the Stable ABI. On success, return a read only *named tuple*, that holds information about Python's internal representation of integers. See `sys.int_info` for description of individual fields.

On failure, return `NULL` with an exception set.

Ajouté dans la version 3.1.

int **PyUnstable_Long_IsCompact**(const *PyLongObject* *op)

This is *Unstable API*. It may change without warning in minor releases.

Return 1 if *op* is compact, 0 otherwise.

This function makes it possible for performance-critical code to implement a "fast path" for small integers. For compact values use `PyUnstable_Long_CompactValue()`; for others fall back to a `PyLong_As*` function or `PyLong_AsNativeBytes()`.

The speedup is expected to be negligible for most users.

Exactly what values are considered compact is an implementation detail and is subject to change.

Ajouté dans la version 3.12.

Py_ssize_t **PyUnstable_Long_CompactValue**(const *PyLongObject* *op)

This is *Unstable API*. It may change without warning in minor releases.

If *op* is compact, as determined by `PyUnstable_Long_IsCompact()`, return its value.

Otherwise, the return value is undefined.

Ajouté dans la version 3.12.

Export API

Ajouté dans la version 3.14.

struct **PyLongLayout**

Layout of an array of "digits" ("limbs" in the GMP terminology), used to represent absolute value for arbitrary precision integers.

Use `PyLong_GetNativeLayout()` to get the native layout of Python `int` objects, used internally for integers with "big enough" absolute value.

See also `sys.int_info` which exposes similar information in Python.

uint8_t **bits_per_digit**

Bits per digit. For example, a 15 bit digit means that bits 0-14 contain meaningful information.

uint8_t **digit_size**

Digit size in bytes. For example, a 15 bit digit will require at least 2 bytes.

int8_t **digits_order**

Digits order :
— `1` for most significant digit first
— `-1` for least significant digit first

int8_t **digit_endianness**

Digit endianness :
— `1` for most significant byte first (big endian)
— `-1` for least significant byte first (little endian)

const *PyLongLayout* ***PyLong_GetNativeLayout**(void)

Get the native layout of Python `int` objects.

See the `PyLongLayout` structure.

The function must not be called before Python initialization nor after Python finalization. The returned layout is valid until Python is finalized. The layout is the same for all Python sub-interpreters in a process, and so it can be cached.

struct **PyLongExport**

Export of a Python `int` object.

There are two cases :
— If `digits` is `NULL`, only use the `value` member.
— If `digits` is not `NULL`, use `negative`, `ndigits` and `digits` members.

int64_t **value**

The native integer value of the exported `int` object. Only valid if `digits` is `NULL`.

uint8_t **negative**

`1` if the number is negative, `0` otherwise. Only valid if `digits` is not `NULL`.

Py_ssize_t **ndigits**

Number of digits in `digits` array. Only valid if `digits` is not `NULL`.

const void ***digits**

Read-only array of unsigned digits. Can be `NULL`.

int **PyLong_Export**(*PyObject* *obj, *PyLongExport* *export_long)

Export a Python `int` object.

export_long must point to a `PyLongExport` structure allocated by the caller. It must not be `NULL`.

On success, fill in **export_long* and return `0`. On error, set an exception and return `-1`.

`PyLong_FreeExport()` must be called when the export is no longer needed.

Particularité de l'implémentation CPython : This function always succeeds if *obj* is a Python `int` object or a subclass.

void **PyLong_FreeExport**(*PyLongExport* *export_long)

Release the export *export_long* created by `PyLong_Export()`.

Particularité de l'implémentation CPython : Calling `PyLong_FreeExport()` is optional if *export_long->digits* is `NULL`.

PyLongWriter API

The `PyLongWriter` API can be used to import an integer.

Ajouté dans la version 3.14.

struct **PyLongWriter**

A Python `int` writer instance.

The instance must be destroyed by `PyLongWriter_Finish()` or `PyLongWriter_Discard()`.

PyLongWriter ***PyLongWriter_Create**(int negative, *Py_ssize_t* ndigits, void **digits)

Create a `PyLongWriter`.

On success, allocate **digits* and return a writer. On error, set an exception and return `NULL`.

negative is `1` if the number is negative, or `0` otherwise.

ndigits is the number of digits in the *digits* array. It must be greater than 0.

digits must not be NULL.

After a successful call to this function, the caller should fill in the array of digits *digits* and then call `PyLongWriter_Finish()` to get a Python `int`. The layout of *digits* is described by `PyLong_GetNativeLayout()`.

Digits must be in the range [`0`; `(1 << bits_per_digit) - 1`] (where the `bits_per_digit` is the number of bits per digit). Any unused most significant digits must be set to `0`.

Alternately, call `PyLongWriter_Discard()` to destroy the writer instance without creating an `int` object.

PyObject ***PyLongWriter_Finish**(*PyLongWriter* *writer)

Valeur de retour : nouvelle référence. Finish a `PyLongWriter` created by `PyLongWriter_Create()`.

On success, return a Python `int` object. On error, set an exception and return `NULL`.

The function takes care of normalizing the digits and converts the object to a compact integer if needed.

The writer instance and the *digits* array are invalid after the call.

void **PyLongWriter_Discard**(*PyLongWriter* *writer)

Discard a `PyLongWriter` created by `PyLongWriter_Create()`.

If *writer* is `NULL`, no operation is performed.

The writer instance and the *digits* array are invalid after the call.

9.2.2 Les objets booléens

Booleans in Python are implemented as a subclass of integers. There are only two booleans, `Py_False` and `Py_True`. As such, the normal creation and deletion functions don't apply to booleans. The following macros are available, however.

PyTypeObject **PyBool_Type**

Part of the Stable ABI. This instance of `PyTypeObject` represents the Python boolean type ; it is the same object as `bool` in the Python layer.

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

Renvoie vrai si *o* est de type `PyBook_Type`. Cette fonction réussit systématiquement.

PyObject ***Py_False**

The Python `False` object. This object has no methods and is *immortal*.

Modifié dans la version 3.12 : `Py_False` is *immortal*.

PyObject ***Py_True**

The Python `True` object. This object has no methods and is *immortal*.

Modifié dans la version 3.12 : `Py_True` is *immortal*.

Py_RETURN_FALSE

Return `Py_False` from a function.

Py_RETURN_TRUE

Return `Py_True` from a function.

PyObject ***PyBool_FromLong**(long v)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return `Py_True` or `Py_False`, depending on the truth value of *v*.

9.2.3 Floating-Point Objects

type **PyFloatObject**

This subtype of `PyObject` represents a Python floating-point object.

PyTypeObject **PyFloat_Type**

Part of the Stable ABI. This instance of `PyTypeObject` represents the Python floating-point type. This is the same object as `float` in the Python layer.

int **PyFloat_Check**(*PyObject* *p)

Return true if its argument is a `PyFloatObject` or a subtype of `PyFloatObject`. This function always succeeds.

int **PyFloat_CheckExact**(*PyObject* *p)

Return true if its argument is a `PyFloatObject`, but not a subtype of `PyFloatObject`. This function always succeeds.

PyObject ***PyFloat_FromString**(*PyObject* *str)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a `PyFloatObject` object based on the string value in *str*, or `NULL` on failure.

PyObject ***PyFloat_FromDouble**(double v)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a `PyFloatObject` object from *v*, or `NULL` on failure.

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

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

Modifié dans la version 3.8 : Use `__index__()` if available.

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

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

PyObject ***PyFloat_GetInfo**(void)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Renvoie une instance *structseq* qui contient les informations sur la précision et les valeurs minimales et maximales pour un nombre à virgule flottante. C'est une enveloppe autour du fichier d'entête `float.h`.

double **PyFloat_GetMax**()

Part of the Stable ABI. Return the maximum representable finite float *DBL_MAX* as C `double`.

double **PyFloat_GetMin**()

Part of the Stable ABI. Return the minimum normalized positive float *DBL_MIN* as C `double`.

Pack and Unpack functions

The pack and unpack functions provide an efficient platform-independent way to store floating-point values as byte strings. The Pack routines produce a bytes string from a C `double`, and the Unpack routines produce a C `double` from such a bytes string. The suffix (2, 4 or 8) specifies the number of bytes in the bytes string.

On platforms that appear to use IEEE 754 formats these functions work by copying bits. On other platforms, the 2-byte format is identical to the IEEE 754 binary16 half-precision format, the 4-byte format (32-bit) is identical to the IEEE 754 binary32 single precision format, and the 8-byte format to the IEEE 754 binary64 double precision format, although the packing of INFs and NaNs (if such things exist on the platform) isn't handled correctly, and attempting to unpack a bytes string containing an IEEE INF or NaN will raise an exception.

Note that NaNs type may not be preserved on IEEE platforms (silent NaN become quiet), for example on x86 systems in 32-bit mode.

On non-IEEE platforms with more precision, or larger dynamic range, than IEEE 754 supports, not all values can be packed ; on non-IEEE platforms with less precision, or smaller dynamic range, not all values can be unpacked. What happens in such cases is partly accidental (alas).

Ajouté dans la version 3.11.

Pack functions

The pack routines write 2, 4 or 8 bytes, starting at *p*. *le* is an `int` argument, non-zero if you want the bytes string in little-endian format (exponent last, at `p+1`, `p+3`, or `p+6` `p+7`), zero if you want big-endian format (exponent first, at *p*). The `PY_BIG_ENDIAN` constant can be used to use the native endian : it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value : `0` if all is OK, `-1` if error (and an exception is set, most likely `OverflowError`).

There are two problems on non-IEEE platforms :

— What this does is undefined if *x* is a NaN or infinity.

— `-0.0` and `+0.0` produce the same bytes string.

int **PyFloat_Pack2**(double x, char *p, int le)

Pack a C double as the IEEE 754 binary16 half-precision format.

int **PyFloat_Pack4**(double x, char *p, int le)

Pack a C double as the IEEE 754 binary32 single precision format.

int **PyFloat_Pack8**(double x, char *p, int le)

Pack a C double as the IEEE 754 binary64 double precision format.

Unpack functions

The unpack routines read 2, 4 or 8 bytes, starting at *p*. *le* is an `int` argument, non-zero if the bytes string is in little-endian format (exponent last, at `p+1`, `p+3` or `p+6` and `p+7`), zero if big-endian (exponent first, at *p*). The `PY_BIG_ENDIAN` constant can be used to use the native endian : it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value : The unpacked double. On error, this is `-1.0` and *`PyErr_Occurred()`* is true (and an exception is set, most likely `OverflowError`).

Note that on a non-IEEE platform this will refuse to unpack a bytes string that represents a NaN or infinity.

double **`PyFloat_Unpack2`**(const char *p, int le)

Unpack the IEEE 754 binary16 half-precision format as a C double.

double **`PyFloat_Unpack4`**(const char *p, int le)

Unpack the IEEE 754 binary32 single precision format as a C double.

double **`PyFloat_Unpack8`**(const char *p, int le)

Unpack the IEEE 754 binary64 double precision format as a C double.

9.2.4 Objets représentant des nombres complexes

Les nombres complexes Python sont implémentés comme deux types distincts, lorsqu'ils sont vus de l'API C : l'un est l'objet Python tel qu'il est vu par les programmes Python, et l'autre est une structure C qui représente la valeur exacte du nombre complexe. L'API fournit des fonctions pour travailler avec ces deux représentations.

Nombres complexes en tant que structures C

Les fonctions qui acceptent ces structures comme paramètres et les renvoient comme résultats le font en fonction de leur *valeur* au lieu de les dé-référencer en utilisant des pointeurs. C'est constant dans toute l'API.

type **`Py_complex`**

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

double **`real`**

double **`imag`**

The structure is defined as :

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

Py_complex **`_Py_c_sum`**(*Py_complex* left, *Py_complex* right)

Renvoie la somme de deux nombres complexes, sous la forme d'un *`Py_complex`* en C.

Py_complex **`_Py_c_diff`**(*Py_complex* left, *Py_complex* right)

Renvoie la différence de deux nombres complexes, sous la forme d'un *`Py_complex`* en C.

Py_complex **`_Py_c_neg`**(*Py_complex* num)

Return the negation of the complex number *num*, using the C *`Py_complex`* representation.

Py_complex **`_Py_c_prod`**(*Py_complex* left, *Py_complex* right)

Renvoie le produit de deux nombres complexes, sous la forme d'un *`Py_complex`* en C.

Py_complex **`_Py_c_quot`**(*Py_complex* dividend, *Py_complex* divisor)

Renvoie le quotient de deux nombres complexes, sous la forme d'un *`Py_complex`* en C.

If *divisor* is null, this method returns zero and sets `errno` to `EDOM`.

Py_complex **_Py_c_pow**(*Py_complex* num, *Py_complex* exp)

Renvoie *num* à la puissance *exp*, sous la forme d'un *Py_complex* en C.

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

Set `errno` to `ERANGE` on overflows.

Nombres complexes en tant qu'objets Python

type **PyComplexObject**

Ce sous-type de l'objet *PyObject* représente un nombre complexe en Python.

PyTypeObject **PyComplex_Type**

Part of the Stable ABI. Cette instance de *PyTypeObject* représente le type nombre complexe Python. C'est le même objet que la classe `complex` de la couche Python.

int **PyComplex_Check**(*PyObject* *p)

Return true if its argument is a *PyComplexObject* or a subtype of *PyComplexObject*. This function always succeeds.

int **PyComplex_CheckExact**(*PyObject* *p)

Return true if its argument is a *PyComplexObject*, but not a subtype of *PyComplexObject*. This function always succeeds.

PyObject ***PyComplex_FromCComplex**(*Py_complex* v)

Valeur de retour : nouvelle référence. Create a new Python complex number object from a C *Py_complex* value. Return `NULL` with an exception set on error.

PyObject ***PyComplex_FromDoubles**(double real, double imag)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new *PyComplexObject* object from *real* and *imag*. Return `NULL` with an exception set on error.

double **PyComplex_RealAsDouble**(*PyObject* *op)

Part of the Stable ABI. Return the real part of *op* as a C `double`.

If *op* is not a Python complex number object but has a `__complex__()` method, this method will first be called to convert *op* to a Python complex number object. If `__complex__()` is not defined then it falls back to call *PyFloat_AsDouble()* and returns its result.

Upon failure, this method returns `-1.0` with an exception set, so one should call *PyErr_Occurred()* to check for errors.

Modifié dans la version 3.13 : Use `__complex__()` if available.

double **PyComplex_ImagAsDouble**(*PyObject* *op)

Part of the Stable ABI. Return the imaginary part of *op* as a C `double`.

If *op* is not a Python complex number object but has a `__complex__()` method, this method will first be called to convert *op* to a Python complex number object. If `__complex__()` is not defined then it falls back to call *PyFloat_AsDouble()* and returns `0.0` on success.

Upon failure, this method returns `-1.0` with an exception set, so one should call *PyErr_Occurred()* to check for errors.

Modifié dans la version 3.13 : Use `__complex__()` if available.

Py_complex **PyComplex_AsCComplex**(*PyObject* *op)

Renvoie la valeur du nombre complexe *op* sous la forme d'un *Py_complex* en C.

If *op* is not a Python complex number object but has a `__complex__()` method, this method will first be called to convert *op* to a Python complex number object. If `__complex__()` is not defined then it falls back to `__float__()`. If `__float__()` is not defined then it falls back to `__index__()`.

Upon failure, this method returns *Py_complex* with *real* set to `-1.0` and with an exception set, so one should call *PyErr_Occurred()* to check for errors.

Modifié dans la version 3.8 : Use `__index__()` if available.

9.3 Objets séquences

Les opérations génériques sur les objets séquences ont été discutées dans le chapitre précédent. Cette section traite des genres spécifiques d'objets séquences qui sont intrinsèques au langage Python.

9.3.1 Objets *bytes*

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

type **PyBytesObject**

This subtype of *PyObject* represents a Python bytes object.

PyTypeObject **PyBytes_Type**

Part of the Stable ABI. This instance of *PyTypeObject* represents the Python bytes type ; it is the same object as `bytes` in the Python layer.

int **PyBytes_Check**(*PyObject* *o)

Return true if the object *o* is a bytes object or an instance of a subtype of the bytes type. This function always succeeds.

int **PyBytes_CheckExact**(*PyObject* *o)

Return true if the object *o* is a bytes object, but not an instance of a subtype of the bytes type. This function always succeeds.

PyObject ***PyBytes_FromString**(const char *v)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new bytes object with a copy of the string *v* as value on success, and `NULL` on failure. The parameter *v* must not be `NULL` ; it will not be checked.

PyObject ***PyBytes_FromStringAndSize**(const char *v, *Py_ssize_t* len)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new bytes object with a copy of the string *v* as value and length *len* on success, and `NULL` on failure. If *v* is `NULL`, the contents of the bytes object are uninitialized.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python bytes object and return a bytes object with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* string. The following format characters are allowed :

Caractères de format	Type	Comment
`%%`	*n/a*	The literal % character.
`%c`	*int*	A single byte, represented as a C int.
`%d`	*int*	Equivalent to `printf("%d")`. [1]
`%u`	`unsigned int`	Equivalent to `printf("%u")`.[page 149, 1]
`%ld`	long	Equivalent to `printf("%ld")`.[page 149, 1]
`%lu`	`unsigned long`	Equivalent to `printf("%lu")`.[page 149, 1]
`%zd`	*Py_ssize_t*	Equivalent to `printf("%zd")`.[page 149, 1]
`%zu`	size_t	Equivalent to `printf("%zu")`.[page 149, 1]
`%i`	*int*	Equivalent to `printf("%i")`.[page 149, 1]
`%x`	*int*	Equivalent to `printf("%x")`.[page 149, 1]
`%s`	const char*	A null-terminated C character array.
`%p`	const void*	The hex representation of a C pointer. Mostly equivalent to `printf("%p")` except that it is guaranteed to start with the literal `0x` regardless of what the platform's `printf` yields.

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

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Identical to `PyBytes_FromFormat()` except that it takes exactly two arguments.

PyObject ***PyBytes_FromObject**(*PyObject* *o)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the bytes representation of object *o* that implements the buffer protocol.

Py_ssize_t **PyBytes_Size**(*PyObject* *o)

Part of the Stable ABI. Return the length of the bytes in bytes object *o*.

Py_ssize_t **PyBytes_GET_SIZE**(*PyObject* *o)

Similar to `PyBytes_Size()`, but without error checking.

char ***PyBytes_AsString**(*PyObject* *o)

Part of the Stable ABI. Return a pointer to the contents of *o*. The pointer refers to the internal buffer of *o*, which consists of `len(o) + 1` bytes. The last byte in the buffer is always null, regardless of whether there are any other null bytes. The data must not be modified in any way, unless the object was just created using `PyBytes_FromStringAndSize(NULL, size)`. It must not be deallocated. If *o* is not a bytes object at all, `PyBytes_AsString()` returns `NULL` and raises `TypeError`.

char ***PyBytes_AS_STRING**(*PyObject* *string)

Similar to `PyBytes_AsString()`, but without error checking.

int **PyBytes_AsStringAndSize**(*PyObject* *obj, char **buffer, *Py_ssize_t* *length)

Part of the Stable ABI. Return the null-terminated contents of the object *obj* through the output variables *buffer* and *length*. Returns `0` on success.

If *length* is `NULL`, the bytes object may not contain embedded null bytes; if it does, the function returns `-1` and a `ValueError` is raised.

The buffer refers to an internal buffer of *obj*, which includes an additional null byte at the end (not counted in *length*). The data must not be modified in any way, unless the object was just created using `PyBytes_FromStringAndSize(NULL, size)`. It must not be deallocated. If *obj* is not a bytes object at all, `PyBytes_AsStringAndSize()` returns `-1` and raises `TypeError`.

Modifié dans la version 3.5 : Previously, `TypeError` was raised when embedded null bytes were encountered in the bytes object.

void **PyBytes_Concat**(*PyObject* **bytes, *PyObject* *newpart)

Part of the Stable ABI. Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*; the caller will own the new reference. The reference to the old value of *bytes* will be stolen. If the new object cannot be created, the old reference to *bytes* will still be discarded and the value of **bytes* will be set to `NULL`; the appropriate exception will be set.

void **PyBytes_ConcatAndDel**(*PyObject* **bytes, *PyObject* *newpart)

Part of the Stable ABI. Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*. This version releases the *strong reference* to *newpart* (i.e. decrements its reference count).

PyObject ***PyBytes_Join**(*PyObject* *sep, *PyObject* *iterable)

Similar to `sep.join(iterable)` in Python.

sep must be Python `bytes` object. (Note that `PyUnicode_Join()` accepts `NULL` separator and treats it as a space, whereas `PyBytes_Join()` doesn't accept `NULL` separator.)

iterable must be an iterable object yielding objects that implement the *buffer protocol*.

On success, return a new `bytes` object. On error, set an exception and return `NULL`.

Ajouté dans la version 3.14.

1. For integer specifiers (d, u, ld, lu, zd, zu, i, x) : the 0-conversion flag has effect even when a precision is given.

int **_PyBytes_Resize**(*PyObject* **bytes, *Py_ssize_t* newsize)

Resize a bytes object. *newsize* will be the new length of the bytes object. You can think of it as creating a new bytes object and destroying the old one, only more efficiently. Pass the address of an existing bytes object as an lvalue (it may be written into), and the new size desired. On success, **bytes* holds the resized bytes object and `0` is returned; the address in **bytes* may differ from its input value. If the reallocation fails, the original bytes object at **bytes* is deallocated, **bytes* is set to `NULL`, `MemoryError` is set, and `-1` is returned.

9.3.2 Objets tableau d'octets

type **PyByteArrayObject**

Ce sous-type de `PyObject` représente un objet `bytearray` Python.

PyTypeObject **PyByteArray_Type**

Part of the Stable ABI. Cette instance de `PyTypeObject` représente le type Python *bytearray*, c'est le même que `bytearray` côté Python.

Macros de vérification de type

int **PyByteArray_Check**(*PyObject* *o)

Renvoie vrai si l'objet *o* est un `bytearray` ou une instance d'un sous-type du type `bytearray`. Cette méthode réussit toujours.

int **PyByteArray_CheckExact**(*PyObject* *o)

Renvoie vrai si l'objet *o* est un `bytearray`, mais pas une instance d'un sous-type du type `bytearray`. Cette méthode réussit toujours.

Fonctions directes sur l'API

PyObject ***PyByteArray_FromObject**(*PyObject* *o)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Renvoie un nouvel objet *bytearray* depuis n'importe quel objet, *o*, qui implémente le *protocole buffer*.

On failure, return `NULL` with an exception set.

PyObject ***PyByteArray_FromStringAndSize**(const char *string, *Py_ssize_t* len)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a new bytearray object from *string* and its length, *len*.

On failure, return `NULL` with an exception set.

PyObject ***PyByteArray_Concat**(*PyObject* *a, *PyObject* *b)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Concatène les `bytearrays` *a* et *b* et renvoie un nouveau `bytearray` avec le résultat.

On failure, return `NULL` with an exception set.

Py_ssize_t **PyByteArray_Size**(*PyObject* *bytearray)

Part of the Stable ABI. Renvoie la taille de *bytearray* après vérification de la présence d'un pointeur `NULL`.

char ***PyByteArray_AsString**(*PyObject* *bytearray)

Part of the Stable ABI. Renvoie le contenu de *bytearray* sous forme d'un tableau de caractères, en vérifiant que ce n'est pas un pointeur `NULL`. Le tableau renvoyé a toujours un caractère *null* rajouté.

int **PyByteArray_Resize**(*PyObject* *bytearray, *Py_ssize_t* len)

Part of the Stable ABI. Resize the internal buffer of *bytearray* to *len*. Failure is a `-1` return with an exception set.

Modifié dans la version 3.14 : A negative *len* will now result in an exception being set and -1 returned.

Macros

Ces macros sont taillées pour la vitesse d'exécution et ne vérifient pas les pointeurs.

char ***PyByteArray_AS_STRING**(*PyObject* *bytearray)

Similar to `PyByteArray_AsString()`, but without error checking.

Py_ssize_t **PyByteArray_GET_SIZE**(*PyObject* *bytearray)

Similar to `PyByteArray_Size()`, but without error checking.

9.3.3 Objets Unicode et Codecs

Objets Unicode

Depuis l'implémentation de **PEP 393** dans Python 3.3, les objets Unicode utilisent une variété de représentations internes, pour permettre de gérer l'intervalle complet des caractères Unicode en restant efficace en termes de mémoire. Il y a des cas spéciaux pour les chaînes où tous les points de code sont inférieurs à 128, 256, ou 65536 ; sinon, les points de code doivent être inférieurs à 1114112 (qui est l'intervalle Unicode complet).

UTF-8 representation is created on demand and cached in the Unicode object.

Note

The `Py_UNICODE` representation has been removed since Python 3.12 with deprecated APIs. See **PEP 623** for more information.

Type Unicode

Voici les types d'objets Unicode utilisés pour l'implémentation Unicode en Python :

PyTypeObject **PyUnicode_Type**

Part of the Stable ABI. This instance of `PyTypeObject` represents the Python Unicode type. It is exposed to Python code as `str`.

PyTypeObject **PyUnicodeIter_Type**

Part of the Stable ABI. This instance of `PyTypeObject` represents the Python Unicode iterator type. It is used to iterate over Unicode string objects.

type **Py_UCS4**

type **Py_UCS2**

type **Py_UCS1**

Part of the Stable ABI. Voici les *typedefs* pour les types entiers non signés suffisamment large pour contenir des caractères de 32 bits, 16 bits et 8 bits, respectivement. Pour traiter des caractères uniques, utilisez `Py_UCS4`.

Ajouté dans la version 3.3.

type **PyASCIIObject**

type **PyCompactUnicodeObject**

type **PyUnicodeObject**

These subtypes of `PyObject` represent a Python Unicode object. In almost all cases, they shouldn't be used directly, since all API functions that deal with Unicode objects take and return `PyObject` pointers.

Ajouté dans la version 3.3.

The following APIs are C macros and static inlined functions for fast checks and access to internal read-only data of Unicode objects :

int **PyUnicode_Check**(*PyObject* *obj)

Return true if the object *obj* is a Unicode object or an instance of a Unicode subtype. This function always succeeds.

int **PyUnicode_CheckExact**(*PyObject* *obj)

Return true if the object *obj* is a Unicode object, but not an instance of a subtype. This function always succeeds.

Py_ssize_t **PyUnicode_GET_LENGTH**(*PyObject* *unicode)

Return the length of the Unicode string, in code points. *unicode* has to be a Unicode object in the "canonical" representation (not checked).

Ajouté dans la version 3.3.

Py_UCS1 ***PyUnicode_1BYTE_DATA**(*PyObject* *unicode)

Py_UCS2 ***PyUnicode_2BYTE_DATA**(*PyObject* *unicode)

Py_UCS4 ***PyUnicode_4BYTE_DATA**(*PyObject* *unicode)

Return a pointer to the canonical representation cast to UCS1, UCS2 or UCS4 integer types for direct character access. No checks are performed if the canonical representation has the correct character size; use *PyUnicode_KIND()* to select the right function.

Ajouté dans la version 3.3.

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND

PyUnicode_4BYTE_KIND

Return values of the *PyUnicode_KIND()* macro.

Ajouté dans la version 3.3.

Modifié dans la version 3.12 : `PyUnicode_WCHAR_KIND` has been removed.

int **PyUnicode_KIND**(*PyObject* *unicode)

Return one of the PyUnicode kind constants (see above) that indicate how many bytes per character this Unicode object uses to store its data. *unicode* has to be a Unicode object in the "canonical" representation (not checked).

Ajouté dans la version 3.3.

void ***PyUnicode_DATA**(*PyObject* *unicode)

Return a void pointer to the raw Unicode buffer. *unicode* has to be a Unicode object in the "canonical" representation (not checked).

Ajouté dans la version 3.3.

void **PyUnicode_WRITE**(int kind, void *data, *Py_ssize_t* index, *Py_UCS4* value)

Write the code point *value* to the given zero-based *index* in a string.

The *kind* value and *data* pointer must have been obtained from a string using *PyUnicode_KIND()* and *PyUnicode_DATA()* respectively. You must hold a reference to that string while calling `PyUnicode_WRITE()`. All requirements of *PyUnicode_WriteChar()* also apply.

The function performs no checks for any of its requirements, and is intended for usage in loops.

Ajouté dans la version 3.3.

Py_UCS4 **PyUnicode_READ**(int kind, void *data, *Py_ssize_t* index)

Read a code point from a canonical representation *data* (as obtained with *PyUnicode_DATA()*). No checks or ready calls are performed.

Ajouté dans la version 3.3.

Py_UCS4 **PyUnicode_READ_CHAR**(*PyObject* *unicode, *Py_ssize_t* index)

Read a character from a Unicode object *unicode*, which must be in the "canonical" representation. This is less efficient than *PyUnicode_READ()* if you do multiple consecutive reads.

Ajouté dans la version 3.3.

Py_UCS4 **PyUnicode_MAX_CHAR_VALUE**(*PyObject* *unicode)

Return the maximum code point that is suitable for creating another string based on *unicode*, which must be in the ”canonical” representation. This is always an approximation but more efficient than iterating over the string.

Ajouté dans la version 3.3.

int **PyUnicode_IsIdentifier**(*PyObject* *unicode)

Part of the Stable ABI. Return `1` if the string is a valid identifier according to the language definition, section identifiers. Return `0` otherwise.

Modifié dans la version 3.9 : The function does not call `Py_FatalError()` anymore if the string is not ready.

unsigned int **PyUnicode_IS_ASCII**(*PyObject* *unicode)

Return true if the string only contains ASCII characters. Equivalent to `str.isascii()`.

Ajouté dans la version 3.2.

Unicode Character Properties

Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration.

int **Py_UNICODE_ISSPACE**(*Py_UCS4* ch)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

int **Py_UNICODE_ISALPHA**(*Py_UCS4* ch)

Return `1` or `0` depending on whether *ch* is an alphabetic character.

int **Py_UNICODE_ISALNUM**(*Py_UCS4* ch)

Return `1` or `0` depending on whether *ch* is an alphanumeric character.

int **Py_UNICODE_ISPRINTABLE**(*Py_UCS4* ch)

Return `1` or `0` depending on whether *ch* is a printable character, in the sense of `str.isprintable()`.

These APIs can be used for fast direct character conversions :

Py_UCS4 **Py_UNICODE_TOLOWER**(*Py_UCS4* ch)

Return the character *ch* converted to lower case.

Py_UCS4 **Py_UNICODE_TOUPPER**(*Py_UCS4* ch)

Return the character *ch* converted to upper case.

Py_UCS4 **Py_UNICODE_TOTITLE**(*Py_UCS4* ch)

Return the character *ch* converted to title case.

int **Py_UNICODE_TODECIMAL**(*Py_UCS4* ch)

Return the character *ch* converted to a decimal positive integer. Return `-1` if this is not possible. This function does not raise exceptions.

int **Py_UNICODE_TODIGIT**(*Py_UCS4* ch)

Return the character *ch* converted to a single digit integer. Return `-1` if this is not possible. This function does not raise exceptions.

double **Py_UNICODE_TONUMERIC**(*Py_UCS4* ch)

Return the character *ch* converted to a double. Return `-1.0` if this is not possible. This function does not raise exceptions.

These APIs can be used to work with surrogates :

int **Py_UNICODE_IS_SURROGATE**(*Py_UCS4* ch)

Check if *ch* is a surrogate (`0xD800 <= ch <= 0xDFFF`).

int **Py_UNICODE_IS_HIGH_SURROGATE**(*Py_UCS4* ch)

Check if *ch* is a high surrogate (`0xD800 <= ch <= 0xDBFF`).

int **Py_UNICODE_IS_LOW_SURROGATE**(*Py_UCS4* ch)

Check if *ch* is a low surrogate (`0xDC00 <= ch <= 0xDFFF`).

Py_UCS4 **Py_UNICODE_JOIN_SURROGATES**(*Py_UCS4* high, *Py_UCS4* low)

Join two surrogate code points and return a single *Py_UCS4* value. *high* and *low* are respectively the leading and trailing surrogates in a surrogate pair. *high* must be in the range [0xD800 ; 0xDBFF] and *low* must be in the range [0xDC00 ; 0xDFFF].

Creating and accessing Unicode strings

To create Unicode objects and access their basic sequence properties, use these APIs :

PyObject ***PyUnicode_New**(*Py_ssize_t* size, *Py_UCS4* maxchar)

Valeur de retour : nouvelle référence. Create a new Unicode object. *maxchar* should be the true maximum code point to be placed in the string. As an approximation, it can be rounded up to the nearest value in the sequence 127, 255, 65535, 1114111.

On error, set an exception and return `NULL`.

After creation, the string can be filled by *PyUnicode_WriteChar()*, *PyUnicode_CopyCharacters()*, *PyUnicode_Fill()*, *PyUnicode_WRITE()* or similar. Since strings are supposed to be immutable, take care to not “use” the result while it is being modified. In particular, before it’s filled with its final contents, a string :
— must not be hashed,
— must not be *converted to UTF-8*, or another non-”canonical” representation,
— must not have its reference count changed,
— must not be shared with code that might do one of the above.
This list is not exhaustive. Avoiding these uses is your responsibility ; Python does not always check these requirements.

To avoid accidentally exposing a partially-written string object, prefer using the *PyUnicodeWriter* API, or one of the `PyUnicode_From*` functions below.

Ajouté dans la version 3.3.

PyObject ***PyUnicode_FromKindAndData**(int kind, const void *buffer, *Py_ssize_t* size)

Valeur de retour : nouvelle référence. Create a new Unicode object with the given *kind* (possible values are *PyUnicode_1BYTE_KIND* etc., as returned by *PyUnicode_KIND()*). The *buffer* must point to an array of *size* units of 1, 2 or 4 bytes per character, as given by the kind.

If necessary, the input *buffer* is copied and transformed into the canonical representation. For example, if the *buffer* is a UCS4 string (*PyUnicode_4BYTE_KIND*) and it consists only of codepoints in the UCS1 range, it will be transformed into UCS1 (*PyUnicode_1BYTE_KIND*).

Ajouté dans la version 3.3.

PyObject ***PyUnicode_FromStringAndSize**(const char *str, *Py_ssize_t* size)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a Unicode object from the char buffer *str*. The bytes will be interpreted as being UTF-8 encoded. The buffer is copied into the new object. The return value might be a shared object, i.e. modification of the data is not allowed.

This function raises `SystemError` when :
— *size* < 0,
— *str* is `NULL` and *size* > 0
Modifié dans la version 3.12 : *str* == `NULL` with *size* > 0 is not allowed anymore.

PyObject ***PyUnicode_FromString**(const char *str)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a Unicode object from a UTF-8 encoded null-terminated char buffer *str*.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python Unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* ASCII-encoded string.

Un indicateur de conversion contient deux ou plusieurs caractères et comporte les éléments suivants, qui doivent apparaître dans cet ordre :

1. le caractère `'%'`, qui marque le début du marqueur ;
2. des indications de conversion, facultatives, qui affectent le résultat de certains types de conversion ;
3. Minimum field width (optional). If specified as an `'*'` (asterisk), the actual width is given in the next argument, which must be of type `int`, and the object to convert comes after the minimum field width and optional precision.
4. Precision (optional), given as a `'.'` (dot) followed by the precision. If specified as `'*'` (an asterisk), the actual precision is given in the next argument, which must be of type `int`, and the value to convert comes after the precision.
5. modificateur de longueur (facultatif) ;
6. type de conversion.

Les caractères indicateurs de conversion sont :

Option	Signification
`0`	Les valeurs numériques converties sont complétées de zéros.
`-`	The converted value is left adjusted (overrides the `0` flag if both are given).

The length modifiers for following integer conversions (`d`, `i`, `o`, `u`, `x`, or `X`) specify the type of the argument (`int` by default) :

Modifier	Types
`l`	`long` or `unsigned long`
`ll`	`long long` or `unsigned long long`
`j`	`intmax_t` or `uintmax_t`
`z`	`size_t` or `ssize_t`
`t`	`ptrdiff_t`

The length modifier `l` for following conversions `s` or `V` specify that the type of the argument is `const wchar_t*`.

The conversion specifiers are :

Conversion Specifier	Type	Comment
`%`	*n/a*	The literal `%` character.
`d`, `i`	Specified by the length modifier	The decimal representation of a signed C integer.
`u`	Specified by the length modifier	The decimal representation of an unsigned C integer.
`o`	Specified by the length modifier	The octal representation of an unsigned C integer.
`x`	Specified by the length modifier	The hexadecimal representation of an unsigned C integer (lowercase).
`X`	Specified by the length modifier	The hexadecimal representation of an unsigned C integer (uppercase).
`c`	`int`	A single character.
`s`	`const char*` or `const wchar_t*`	A null-terminated C character array.
`p`	`const void*`	The hex representation of a C pointer. Mostly equivalent to `printf("%p")` except that it is guaranteed to start with the literal `0x` regardless of what the platform's `printf` yields.
`A`	*`PyObject`*`*`	The result of calling `ascii()`.
`U`	*`PyObject`*`*`	A Unicode object.
`V`	*`PyObject`*`*`, `const char*` or `const wchar_t*`	A Unicode object (which may be `NULL`) and a null-terminated C character array as a second parameter (which will be used, if the first parameter is `NULL`).
`S`	*`PyObject`*`*`	The result of calling *`PyObject_Str()`*.
`R`	*`PyObject`*`*`	The result of calling *`PyObject_Repr()`*.
`T`	*`PyObject`*`*`	Get the fully qualified name of an object type; call *`PyType_GetFullyQualifiedName()`*.
`#T`	*`PyObject`*`*`	Similar to `T` format, but use a colon (`:`) as separator between the module name and the qualified name.
`N`	*`PyTypeObject`*`*`	Get the fully qualified name of a type; call *`PyType_GetFullyQualifiedName()`*.
`#N`	*`PyTypeObject`*`*`	Similar to `N` format, but use a colon (`:`) as separator between the module name and the qualified name.

Note

The width formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes or `wchar_t` items (if the length modifier `l` is used) for `"%s"` and `"%V"` (if the `PyObject*` argument is `NULL`), and a number of characters for `"%A"`, `"%U"`, `"%S"`, `"%R"` and `"%V"` (if the `PyObject*` argument is not `NULL`).

Note

Unlike to C `printf()` the `0` flag has effect even when a precision is given for integer conversions (`d`, `i`, `u`, `o`, `x`, or `X`).

Modifié dans la version 3.2 : Support for `"%lld"` and `"%llu"` added.

Modifié dans la version 3.3 : Support for `"%li"`, `"%lli"` and `"%zi"` added.

Modifié dans la version 3.4 : Support width and precision formatter for `"%s"`, `"%A"`, `"%U"`, `"%V"`, `"%S"`, `"%R"` added.

Modifié dans la version 3.12 : Support for conversion specifiers `o` and `X`. Support for length modifiers `j` and `t`. Length modifiers are now applied to all integer conversions. Length modifier `l` is now applied to conversion specifiers `s` and `V`. Support for variable width and precision `*`. Support for flag `-`.

An unrecognized format character now sets a `SystemError`. In previous versions it caused all the rest of the format string to be copied as-is to the result string, and any extra arguments discarded.

Modifié dans la version 3.13 : Support for `%T`, `%#T`, `%N` and `%#N` formats added.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Identical to *`PyUnicode_FromFormat()`* except that it takes exactly two arguments.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Copy an instance of a Unicode subtype to a new true Unicode object if necessary. If *obj* is already a true Unicode object (not a subtype), return a new *strong reference* to the object.

Objects other than Unicode or its subtypes will cause a `TypeError`.

PyObject ***PyUnicode_FromOrdinal**(int ordinal)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a Unicode Object from the given Unicode code point *ordinal*.

The ordinal must be in `range(0x110000)`. A `ValueError` is raised in the case it is not.

PyObject ***PyUnicode_FromEncodedObject**(*PyObject* *obj, const char *encoding, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Decode an encoded object *obj* to a Unicode object.

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

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

The API returns `NULL` if there was an error. The caller is responsible for decref'ing the returned objects.

void **PyUnicode_Append**(*PyObject* **p_left, *PyObject* *right)

Part of the Stable ABI. Append the string *right* to the end of *p_left*. *p_left* must point to a *strong reference* to a Unicode object; `PyUnicode_Append()` releases ("steals") this reference.

On error, set **p_left* to `NULL` and set an exception.

On success, set **p_left* to a new strong reference to the result.

void **PyUnicode_AppendAndDel**(*PyObject* **p_left, *PyObject* *right)

Part of the Stable ABI. The function is similar to *`PyUnicode_Append()`*, with the only difference being that it decrements the reference count of *right* by one.

PyObject ***PyUnicode_BuildEncodingMap**(*PyObject* *string)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a mapping suitable for decoding a custom single-byte encoding. Given a Unicode string *string* of up to 256 characters representing an encoding table, returns either a compact internal mapping object or a dictionary mapping character ordinals to byte values. Raises a `TypeError` and return `NULL` on invalid input.

Ajouté dans la version 3.2.

const char ***PyUnicode_GetDefaultEncoding**(void)

Part of the Stable ABI. Return the name of the default string encoding, `"utf-8"`. See `sys.getdefaultencoding()`.

The returned string does not need to be freed, and is valid until interpreter shutdown.

Py_ssize_t **PyUnicode_GetLength**(*PyObject* *unicode)

Part of the Stable ABI *Depuis la version 3.7.* Return the length of the Unicode object, in code points.

On error, set an exception and return `-1`.

Ajouté dans la version 3.3.

Py_ssize_t **PyUnicode_CopyCharacters**(*PyObject* *to, *Py_ssize_t* to_start, *PyObject* *from, *Py_ssize_t* from_start, *Py_ssize_t* how_many)

Copy characters from one Unicode object into another. This function performs character conversion when necessary and falls back to `memcpy()` if possible. Returns `-1` and sets an exception on error, otherwise returns the number of copied characters.

The string must not have been "used" yet. See *PyUnicode_New()* for details.

Ajouté dans la version 3.3.

int **PyUnicode_Resize**(*PyObject* **unicode, *Py_ssize_t* length);

Part of the Stable ABI. Resize a Unicode object **unicode* to the new *length* in code points.

Try to resize the string in place (which is usually faster than allocating a new string and copying characters), or create a new string.

**unicode* is modified to point to the new (resized) object and `0` is returned on success. Otherwise, `-1` is returned and an exception is set, and **unicode* is left untouched.

The function doesn't check string content, the result may not be a string in canonical representation.

Py_ssize_t **PyUnicode_Fill**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* length, *Py_UCS4* fill_char)

Fill a string with a character : write *fill_char* into `unicode[start:start+length]`.

Fail if *fill_char* is bigger than the string maximum character, or if the string has more than 1 reference.

The string must not have been "used" yet. See *PyUnicode_New()* for details.

Return the number of written character, or return `-1` and raise an exception on error.

Ajouté dans la version 3.3.

int **PyUnicode_WriteChar**(*PyObject* *unicode, *Py_ssize_t* index, *Py_UCS4* character)

Part of the Stable ABI *Depuis la version 3.7.* Write a *character* to the string *unicode* at the zero-based *index*. Return `0` on success, `-1` on error with an exception set.

This function checks that *unicode* is a Unicode object, that the index is not out of bounds, and that the object's reference count is one). See *PyUnicode_WRITE()* for a version that skips these checks, making them your responsibility.

The string must not have been "used" yet. See *PyUnicode_New()* for details.

Ajouté dans la version 3.3.

Py_UCS4 **PyUnicode_ReadChar**(*PyObject* *unicode, *Py_ssize_t* index)

Part of the Stable ABI *Depuis la version 3.7.* Read a character from a string. This function checks that *unicode* is a Unicode object and the index is not out of bounds, in contrast to *PyUnicode_READ_CHAR()*, which performs no error checking.

Return character on success, `-1` on error with an exception set.

Ajouté dans la version 3.3.

PyObject ***PyUnicode_Substring**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* end)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.7.* Return a substring of *unicode*, from character index *start* (included) to character index *end* (excluded). Negative indices are not supported. On error, set an exception and return `NULL`.

Ajouté dans la version 3.3.

Py_UCS4 ***PyUnicode_AsUCS4**(*PyObject* *unicode, *Py_UCS4* *buffer, *Py_ssize_t* buflen, int copy_null)

Part of the Stable ABI *Depuis la version 3.7.* Copy the string *unicode* into a UCS4 buffer, including a null character, if *copy_null* is set. Returns `NULL` and sets an exception on error (in particular, a `SystemError` if *buflen* is smaller than the length of *unicode*). *buffer* is returned on success.

Ajouté dans la version 3.3.

Py_UCS4 ***PyUnicode_AsUCS4Copy**(*PyObject* *unicode)

Part of the Stable ABI *Depuis la version 3.7.* Copy the string *unicode* into a new UCS4 buffer that is allocated using *PyMem_Malloc()*. If this fails, `NULL` is returned with a `MemoryError` set. The returned buffer always has an extra null code point appended.

Ajouté dans la version 3.3.

Locale Encoding

The current locale encoding can be used to decode text from the operating system.

PyObject ***PyUnicode_DecodeLocaleAndSize**(const char *str, *Py_ssize_t* length, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.7.* Decode a string from UTF-8 on Android and VxWorks, or from the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The decoder uses `"strict"` error handler if *errors* is `NULL`. *str* must end with a null character but cannot contain embedded null characters.

Use *PyUnicode_DecodeFSDefaultAndSize()* to decode a string from the *filesystem encoding and error handler*.

This function ignores the Python UTF-8 Mode.

Voir aussi

The *Py_DecodeLocale()* function.

Ajouté dans la version 3.3.

Modifié dans la version 3.7 : The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, *Py_DecodeLocale()* was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

PyObject ***PyUnicode_DecodeLocale**(const char *str, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.7.* Similar to *PyUnicode_DecodeLocaleAndSize()*, but compute the string length using `strlen()`.

Ajouté dans la version 3.3.

PyObject ***PyUnicode_EncodeLocale**(*PyObject* *unicode, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.7.* Encode a Unicode object to UTF-8 on Android and VxWorks, or to the current locale encoding on other platforms. The supported error handlers are `"strict"` and `"surrogateescape"` (**PEP 383**). The encoder uses `"strict"` error handler if *errors* is `NULL`. Return a `bytes` object. *unicode* cannot contain embedded null characters.

Use *PyUnicode_EncodeFSDefault()* to encode a string to the *filesystem encoding and error handler*.

This function ignores the Python UTF-8 Mode.

Voir aussi

The *Py_EncodeLocale()* function.

Ajouté dans la version 3.3.

Modifié dans la version 3.7 : The function now also uses the current locale encoding for the `surrogateescape` error handler, except on Android. Previously, *`Py_EncodeLocale()`* was used for the `surrogateescape`, and the current locale encoding was used for `strict`.

File System Encoding

Functions encoding to and decoding from the *filesystem encoding and error handler* (**PEP 383** and **PEP 529**).

To encode file names to `bytes` during argument parsing, the `"O&"` converter should be used, passing `PyUnicode_FSConverter()` as the conversion function :

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

Part of the Stable ABI. *PyArg_Parse* converter* : encode `str` objects -- obtained directly or through the `os.PathLike` interface -- to `bytes` using *`PyUnicode_EncodeFSDefault()`* ; `bytes` objects are output as-is. *result* must be an address of a C variable of type *`PyObject*`* (or *`PyBytesObject*`*). On success, set the variable to a new *strong reference* to a *bytes object* which must be released when it is no longer used and return a non-zero value (*`Py_CLEANUP_SUPPORTED`*). Embedded null bytes are not allowed in the result. On failure, return `0` with an exception set.

If *obj* is `NULL`, the function releases a strong reference stored in the variable referred by *result* and returns `1`.

Ajouté dans la version 3.1.

Modifié dans la version 3.6 : Accepte un *path-like object*.

To decode file names to `str` during argument parsing, the `"O&"` converter should be used, passing `PyUnicode_FSDecoder()` as the conversion function :

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

Part of the Stable ABI. *PyArg_Parse* converter* : decode `bytes` objects -- obtained either directly or indirectly through the `os.PathLike` interface -- to `str` using *`PyUnicode_DecodeFSDefaultAndSize()`* ; `str` objects are output as-is. *result* must be an address of a C variable of type *`PyObject*`* (or *`PyUnicodeObject*`*). On success, set the variable to a new *strong reference* to a *Unicode object* which must be released when it is no longer used and return a non-zero value (*`Py_CLEANUP_SUPPORTED`*). Embedded null characters are not allowed in the result. On failure, return `0` with an exception set.

If *obj* is `NULL`, release the strong reference to the object referred to by *result* and return `1`.

Ajouté dans la version 3.2.

Modifié dans la version 3.6 : Accepte un *path-like object*.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Decode a string from the *filesystem encoding and error handler*.

If you need to decode a string from the current locale encoding, use *`PyUnicode_DecodeLocaleAndSize()`*.

> **Voir aussi**
>
> The *`Py_DecodeLocale()`* function.

Modifié dans la version 3.6 : The *filesystem error handler* is now used.

PyObject ***PyUnicode_DecodeFSDefault**(const char *str)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Decode a null-terminated string from the *filesystem encoding and error handler*.

If the string length is known, use *`PyUnicode_DecodeFSDefaultAndSize()`*.

Modifié dans la version 3.6 : The *filesystem error handler* is now used.

PyObject ***PyUnicode_EncodeFSDefault**(*PyObject* *unicode)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Encode a Unicode object to the *filesystem encoding and error handler*, and return `bytes`. Note that the resulting `bytes` object can contain null bytes.

If you need to encode a string to the current locale encoding, use `PyUnicode_EncodeLocale()`.

Voir aussi

The `Py_EncodeLocale()` function.

Ajouté dans la version 3.2.

Modifié dans la version 3.6 : The *filesystem error handler* is now used.

wchar_t Support

`wchar_t` support for platforms which support it :

PyObject ***PyUnicode_FromWideChar**(const wchar_t *wstr, *Py_ssize_t* size)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a Unicode object from the `wchar_t` buffer *wstr* of the given *size*. Passing `-1` as the *size* indicates that the function must itself compute the length, using `wcslen()`. Return `NULL` on failure.

Py_ssize_t ***PyUnicode_AsWideChar**(*PyObject* *unicode, wchar_t *wstr, *Py_ssize_t* size)

Part of the Stable ABI. Copy the Unicode object contents into the `wchar_t` buffer *wstr*. At most *size* `wchar_t` characters are copied (excluding a possibly trailing null termination character). Return the number of `wchar_t` characters copied or `-1` in case of an error.

When *wstr* is `NULL`, instead return the *size* that would be required to store all of *unicode* including a terminating null.

Note that the resulting `wchar_t*` string may or may not be null-terminated. It is the responsibility of the caller to make sure that the `wchar_t*` string is null-terminated in case this is required by the application. Also, note that the `wchar_t*` string might contain null characters, which would cause the string to be truncated when used with most C functions.

wchar_t ***PyUnicode_AsWideCharString**(*PyObject* *unicode, *Py_ssize_t* *size)

Part of the Stable ABI *Depuis la version 3.7.* Convert the Unicode object to a wide character string. The output string always ends with a null character. If *size* is not `NULL`, write the number of wide characters (excluding the trailing null termination character) into **size*. Note that the resulting `wchar_t` string might contain null characters, which would cause the string to be truncated when used with most C functions. If *size* is `NULL` and the `wchar_t*` string contains null characters a `ValueError` is raised.

Returns a buffer allocated by `PyMem_New` (use `PyMem_Free()` to free it) on success. On error, returns `NULL` and **size* is undefined. Raises a `MemoryError` if memory allocation is failed.

Ajouté dans la version 3.2.

Modifié dans la version 3.7 : Raises a `ValueError` if *size* is `NULL` and the `wchar_t*` string contains null characters.

Built-in Codecs

Python provides a set of built-in codecs which are written in C for speed. All of these codecs are directly usable via the following functions.

Many of the following APIs take two arguments encoding and errors, and they have the same semantics as the ones of the built-in `str()` string object constructor.

Setting encoding to `NULL` causes the default encoding to be used which is UTF-8. The file system calls should use `PyUnicode_FSConverter()` for encoding file names. This uses the *filesystem encoding and error handler* internally.

Error handling is set by errors which may also be set to `NULL` meaning to use the default handling defined for the codec. Default error handling for all built-in codecs is "strict" (`ValueError` is raised).

The codecs all use a similar interface. Only deviations from the following generic ones are documented for simplicity.

Generic Codecs

The following macro is provided :

Py_UNICODE_REPLACEMENT_CHARACTER

The Unicode code point `U+FFFD` (replacement character).

This Unicode character is used as the replacement character during decoding if the *errors* argument is set to "replace".

These are the generic codec APIs :

PyObject ***PyUnicode_Decode**(const char *str, *Py_ssize_t* size, const char *encoding, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the encoded string *str*. *encoding* and *errors* have the same meaning as the parameters of the same name in the `str()` built-in function. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsEncodedString**(*PyObject* *unicode, const char *encoding, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Encode a Unicode object and return the result as Python bytes object. *encoding* and *errors* have the same meaning as the parameters of the same name in the Unicode `encode()` method. The codec to be used is looked up using the Python codec registry. Return `NULL` if an exception was raised by the codec.

UTF-8 Codecs

These are the UTF-8 codec APIs :

PyObject ***PyUnicode_DecodeUTF8**(const char *str, *Py_ssize_t* size, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the UTF-8 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeUTF8Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Valeur de retour : nouvelle référence. Part of the Stable ABI. If *consumed* is `NULL`, behave like *PyUnicode_DecodeUTF8()*. If *consumed* is not `NULL`, trailing incomplete UTF-8 byte sequences will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF8String**(*PyObject* *unicode)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Encode a Unicode object using UTF-8 and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

The function fails if the string contains surrogate code points (`U+D800` - `U+DFFF`).

const char ***PyUnicode_AsUTF8AndSize**(*PyObject* *unicode, *Py_ssize_t* *size)

Part of the Stable ABI *Depuis la version 3.10.* Return a pointer to the UTF-8 encoding of the Unicode object, and store the size of the encoded representation (in bytes) in *size*. The *size* argument can be `NULL`; in this case no size will be stored. The returned buffer always has an extra null byte appended (not included in *size*), regardless of whether there are any other null code points.

On error, set an exception, set *size* to `-1` (if it's not NULL) and return `NULL`.

The function fails if the string contains surrogate code points (`U+D800` - `U+DFFF`).

This caches the UTF-8 representation of the string in the Unicode object, and subsequent calls will return a pointer to the same buffer. The caller is not responsible for deallocating the buffer. The buffer is deallocated and pointers to it become invalid when the Unicode object is garbage collected.

Ajouté dans la version 3.3.

Modifié dans la version 3.7 : The return type is now `const char *` rather of `char *`.

Modifié dans la version 3.10 : This function is a part of the *limited API*.

const char ***PyUnicode_AsUTF8**(*PyObject* *unicode)

As *PyUnicode_AsUTF8AndSize()*, but does not store the size.

> **Avertissement**
>
> This function does not have any special behavior for null characters embedded within *unicode*. As a result, strings containing null characters will remain in the returned string, which some C functions might interpret as the end of the string, leading to truncation. If truncation is an issue, it is recommended to use *PyUnicode_AsUTF8AndSize()* instead.

Ajouté dans la version 3.3.

Modifié dans la version 3.7 : The return type is now `const char *` rather of `char *`.

UTF-32 Codecs

These are the UTF-32 codec APIs :

PyObject ***PyUnicode_DecodeUTF32**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Decode *size* bytes from a UTF-32 encoded buffer string and return the corresponding Unicode object. *errors* (if non-`NULL`) defines the error handling. It defaults to "strict".

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

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

If `*byteorder` is zero, and the first four bytes of the input data are a byte order mark (BOM), the decoder switches to this byte order and the BOM is not copied into the resulting Unicode string. If `*byteorder` is `-1` or `1`, any byte order mark is copied to the output.

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

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

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

PyObject ***PyUnicode_DecodeUTF32Stateful**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

Valeur de retour : nouvelle référence. Part of the Stable ABI. If *consumed* is `NULL`, behave like *PyUnicode_DecodeUTF32()*. If *consumed* is not `NULL`, *PyUnicode_DecodeUTF32Stateful()* will not treat trailing incomplete UTF-32 byte sequences (such as a number of bytes not divisible by four) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF32String**(*PyObject* *unicode)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a Python byte string using the UTF-32 encoding in native byte order. The string always starts with a BOM mark. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

UTF-16 Codecs

These are the UTF-16 codec APIs :

PyObject ***PyUnicode_DecodeUTF16**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Decode *size* bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. *errors* (if non-`NULL`) defines the error handling. It defaults to "strict".

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

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

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

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

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

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

PyObject ***PyUnicode_DecodeUTF16Stateful**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

Valeur de retour : nouvelle référence. Part of the Stable ABI. If *consumed* is `NULL`, behave like *PyUnicode_DecodeUTF16()*. If *consumed* is not `NULL`, *PyUnicode_DecodeUTF16Stateful()* will not treat trailing incomplete UTF-16 byte sequences (such as an odd number of bytes or a split surrogate pair) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_AsUTF16String**(*PyObject* *unicode)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a Python byte string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

UTF-7 Codecs

These are the UTF-7 codec APIs :

PyObject ***PyUnicode_DecodeUTF7**(const char *str, *Py_ssize_t* size, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the UTF-7 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeUTF7Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Valeur de retour : nouvelle référence. Part of the Stable ABI. If *consumed* is `NULL`, behave like *PyUnicode_DecodeUTF7()*. If *consumed* is not `NULL`, trailing incomplete UTF-7 base-64 sections will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in *consumed*.

Unicode-Escape Codecs

These are the "Unicode Escape" codec APIs :

PyObject ***PyUnicode_DecodeUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the Unicode-Escape encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsUnicodeEscapeString**(*PyObject* *unicode)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Encode a Unicode object using Unicode-Escape and return the result as a bytes object. Error handling is ”strict”. Return `NULL` if an exception was raised by the codec.

Raw-Unicode-Escape Codecs

These are the ”Raw Unicode Escape” codec APIs :

PyObject ***PyUnicode_DecodeRawUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the Raw-Unicode-Escape encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsRawUnicodeEscapeString**(*PyObject* *unicode)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Encode a Unicode object using Raw-Unicode-Escape and return the result as a bytes object. Error handling is ”strict”. Return `NULL` if an exception was raised by the codec.

Latin-1 Codecs

These are the Latin-1 codec APIs : Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

PyObject ***PyUnicode_DecodeLatin1**(const char *str, *Py_ssize_t* size, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the Latin-1 encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsLatin1String**(*PyObject* *unicode)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Encode a Unicode object using Latin-1 and return the result as Python bytes object. Error handling is ”strict”. Return `NULL` if an exception was raised by the codec.

ASCII Codecs

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

PyObject ***PyUnicode_DecodeASCII**(const char *str, *Py_ssize_t* size, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the ASCII encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_AsASCIIString**(*PyObject* *unicode)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Encode a Unicode object using ASCII and return the result as Python bytes object. Error handling is ”strict”. Return `NULL` if an exception was raised by the codec.

Character Map Codecs

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the `encodings` package). The codec uses mappings to encode and decode characters. The mapping objects provided must support the `__getitem__()` mapping interface ; dictionaries and sequences work well.

These are the mapping codec APIs :

PyObject ***PyUnicode_DecodeCharmap**(const char *str, *Py_ssize_t* length, *PyObject* *mapping, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a Unicode object by decoding *size* bytes of the encoded string *str* using the given *mapping* object. Return `NULL` if an exception was raised by the codec.

If *mapping* is `NULL`, Latin-1 decoding will be applied. Else *mapping* must map bytes ordinals (integers in the range from 0 to 255) to Unicode strings, integers (which are then interpreted as Unicode ordinals) or `None`.

Unmapped data bytes -- ones which cause a `LookupError`, as well as ones which get mapped to `None`, `0xFFFE` or `'\ufffe'`, are treated as undefined mappings and cause an error.

PyObject ***PyUnicode_AsCharmapString**(*PyObject* *unicode, *PyObject* *mapping)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Encode a Unicode object using the given *mapping* object and return the result as a bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

The *mapping* object must map Unicode ordinal integers to bytes objects, integers in the range from 0 to 255 or `None`. Unmapped character ordinals (ones which cause a `LookupError`) as well as mapped to `None` are treated as "undefined mapping" and cause an error.

The following codec API is special in that maps Unicode to Unicode.

PyObject ***PyUnicode_Translate**(*PyObject* *unicode, *PyObject* *table, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Translate a string by applying a character mapping table to it and return the resulting Unicode object. Return `NULL` if an exception was raised by the codec.

The mapping table must map Unicode ordinal integers to Unicode ordinal integers or `None` (causing deletion of the character).

Mapping tables need only provide the `__getitem__()` interface; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a `LookupError`) are left untouched and are copied as-is.

errors has the usual meaning for codecs. It may be `NULL` which indicates to use the default error handling.

MBCS codecs for Windows

These are the MBCS codec APIs. They are currently only available on Windows and use the Win32 MBCS converters to implement the conversions. Note that MBCS (or DBCS) is a class of encodings, not just one. The target encoding is defined by the user settings on the machine running the codec.

PyObject ***PyUnicode_DecodeMBCS**(const char *str, *Py_ssize_t* size, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI *on Windows Depuis la version 3.7.* Create a Unicode object by decoding *size* bytes of the MBCS encoded string *str*. Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_DecodeMBCSStateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Valeur de retour : nouvelle référence. Part of the Stable ABI *on Windows Depuis la version 3.7.* If *consumed* is `NULL`, behave like *`PyUnicode_DecodeMBCS()`*. If *consumed* is not `NULL`, *`PyUnicode_DecodeMBCSStateful()`* will not decode trailing lead byte and the number of bytes that have been decoded will be stored in *consumed*.

PyObject ***PyUnicode_DecodeCodePageStateful**(int code_page, const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

Valeur de retour : nouvelle référence. Part of the Stable ABI *on Windows Depuis la version 3.7.* Similar to *`PyUnicode_DecodeMBCSStateful()`*, except uses the code page specified by *code_page*.

PyObject ***PyUnicode_AsMBCSString**(*PyObject* *unicode)

Valeur de retour : nouvelle référence. Part of the Stable ABI *on Windows Depuis la version 3.7.* Encode a Unicode object using MBCS and return the result as Python bytes object. Error handling is "strict". Return `NULL` if an exception was raised by the codec.

PyObject ***PyUnicode_EncodeCodePage**(int code_page, *PyObject* *unicode, const char *errors)

Valeur de retour : nouvelle référence. Part of the Stable ABI *on Windows Depuis la version 3.7.* Encode the Unicode object using the specified code page and return a Python bytes object. Return `NULL` if an exception was raised by the codec. Use `CP_ACP` code page to get the MBCS encoder.

Ajouté dans la version 3.3.

Methods and Slot Functions

The following APIs are capable of handling Unicode objects and strings on input (we refer to them as strings in the descriptions) and return Unicode objects or integers as appropriate.

They all return `NULL` or `-1` if an exception occurs.

PyObject ***PyUnicode_Concat**(*PyObject* *left, *PyObject* *right)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Concat two strings giving a new Unicode string.

PyObject ***PyUnicode_Split**(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Split a string giving a list of Unicode strings. If *sep* is `NULL`, splitting will be done at all whitespace substrings. Otherwise, splits occur at the given separator. At most *maxsplit* splits will be done. If negative, no limit is set. Separators are not included in the resulting list.

On error, return `NULL` with an exception set.

Equivalent to `str.split()`.

PyObject ***PyUnicode_RSplit**(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Similar to *PyUnicode_Split()*, but splitting will be done beginning at the end of the string.

On error, return `NULL` with an exception set.

Equivalent to `str.rsplit()`.

PyObject ***PyUnicode_Splitlines**(*PyObject* *unicode, int keepends)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If *keepends* is `0`, the Line break characters are not included in the resulting strings.

PyObject ***PyUnicode_Partition**(*PyObject* *unicode, *PyObject* *sep)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Split a Unicode string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings.

sep must not be empty.

On error, return `NULL` with an exception set.

Equivalent to `str.partition()`.

PyObject ***PyUnicode_RPartition**(*PyObject* *unicode, *PyObject* *sep)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Similar to *PyUnicode_Partition()*, but split a Unicode string at the last occurrence of *sep*. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself.

sep must not be empty.

On error, return `NULL` with an exception set.

Equivalent to `str.rpartition()`.

PyObject ***PyUnicode_Join**(*PyObject* *separator, *PyObject* *seq)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Join a sequence of strings using the given *separator* and return the resulting Unicode string.

Py_ssize_t **PyUnicode_Tailmatch**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Part of the Stable ABI. Return `1` if *substr* matches `unicode[start:end]` at the given tail end (*direction* == `-1` means to do a prefix match, *direction* == `1` a suffix match), `0` otherwise. Return `-1` if an error occurred.

Py_ssize_t **PyUnicode_Find**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Part of the Stable ABI. Return the first position of *substr* in `unicode[start:end]` using the given *direction*

(*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Py_ssize_t **PyUnicode_FindChar**(*PyObject* *unicode, *Py_UCS4* ch, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

Part of the Stable ABI *Depuis la version 3.7.* Return the first position of the character *ch* in `unicode[start:end]` using the given *direction* (*direction* == `1` means to do a forward search, *direction* == `-1` a backward search). The return value is the index of the first match; a value of `-1` indicates that no match was found, and `-2` indicates that an error occurred and an exception has been set.

Ajouté dans la version 3.3.

Modifié dans la version 3.7 : *start* and *end* are now adjusted to behave like `unicode[start:end]`.

Py_ssize_t **PyUnicode_Count**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end)

Part of the Stable ABI. Return the number of non-overlapping occurrences of *substr* in `unicode[start:end]`. Return `-1` if an error occurred.

PyObject ***PyUnicode_Replace**(*PyObject* *unicode, *PyObject* *substr, *PyObject* *replstr, *Py_ssize_t* maxcount)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Replace at most *maxcount* occurrences of *substr* in *unicode* with *replstr* and return the resulting Unicode object. *maxcount* == `-1` means replace all occurrences.

int **PyUnicode_Compare**(*PyObject* *left, *PyObject* *right)

Part of the Stable ABI. Compare two strings and return `-1`, `0`, `1` for less than, equal, and greater than, respectively.

This function returns `-1` upon failure, so one should call *PyErr_Occurred()* to check for errors.

Voir aussi

The *PyUnicode_Equal()* function.

int **PyUnicode_Equal**(*PyObject* *a, *PyObject* *b)

Part of the Stable ABI *Depuis la version 3.14.* Test if two strings are equal :
— Return `1` if *a* is equal to *b*.
— Return `0` if *a* is not equal to *b*.
— Set a `TypeError` exception and return `-1` if *a* or *b* is not a `str` object.
The function always succeeds if *a* and *b* are `str` objects.

The function works for `str` subclasses, but does not honor custom `__eq__()` method.

Voir aussi

The *PyUnicode_Compare()* function.

Ajouté dans la version 3.14.

int **PyUnicode_EqualToUTF8AndSize**(*PyObject* *unicode, const char *string, *Py_ssize_t* size)

Part of the Stable ABI *Depuis la version 3.13.* Compare a Unicode object with a char buffer which is interpreted as being UTF-8 or ASCII encoded and return true (`1`) if they are equal, or false (`0`) otherwise. If the Unicode object contains surrogate code points (`U+D800 - U+DFFF`) or the C string is not valid UTF-8, false (`0`) is returned.

This function does not raise exceptions.

Ajouté dans la version 3.13.

int **PyUnicode_EqualToUTF8**(*PyObject* *unicode, const char *string)

Part of the Stable ABI *Depuis la version 3.13.* Similar to *PyUnicode_EqualToUTF8AndSize()*, but compute *string* length using `strlen()`. If the Unicode object contains null characters, false (`0`) is returned.

Ajouté dans la version 3.13.

int **PyUnicode_CompareWithASCIIString**(*PyObject* *unicode, const char *string)

Part of the Stable ABI. Compare a Unicode object, *unicode*, with *string* and return `-1`, `0`, `1` for less than, equal, and greater than, respectively. It is best to pass only ASCII-encoded strings, but the function interprets the input string as ISO-8859-1 if it contains non-ASCII characters.

This function does not raise exceptions.

PyObject ***PyUnicode_RichCompare**(*PyObject* *left, *PyObject* *right, int op)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Rich compare two Unicode strings and return one of the following :

— `NULL` in case an exception was raised

— *`Py_True`* or *`Py_False`* for successful comparisons

— *`Py_NotImplemented`* in case the type combination is unknown

Possible values for *op* are *`Py_GT`*, *`Py_GE`*, *`Py_EQ`*, *`Py_NE`*, *`Py_LT`*, and *`Py_LE`*.

PyObject ***PyUnicode_Format**(*PyObject* *format, *PyObject* *args)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new string object from *format* and *args* ; this is analogous to `format % args`.

int **PyUnicode_Contains**(*PyObject* *unicode, *PyObject* *substr)

Part of the Stable ABI. Check whether *substr* is contained in *unicode* and return true or false accordingly.

substr has to coerce to a one element Unicode string. `-1` is returned if there was an error.

void **PyUnicode_InternInPlace**(*PyObject* **p_unicode)

Part of the Stable ABI. Intern the argument *`*p_unicode`* in place. The argument must be the address of a pointer variable pointing to a Python Unicode string object. If there is an existing interned string that is the same as *`*p_unicode`*, it sets *`*p_unicode`* to it (releasing the reference to the old string object and creating a new *strong reference* to the interned string object), otherwise it leaves *`*p_unicode`* alone and interns it.

(Clarification : even though there is a lot of talk about references, think of this function as reference-neutral. You must own the object you pass in ; after the call you no longer own the passed-in reference, but you newly own the result.)

This function never raises an exception. On error, it leaves its argument unchanged without interning it.

Instances of subclasses of `str` may not be interned, that is, *`PyUnicode_CheckExact`*`(`*`*p_unicode`*`)` must be true. If it is not, then -- as with any other error -- the argument is left unchanged.

Note that interned strings are not “immortal”. You must keep a reference to the result to benefit from interning.

PyObject ***PyUnicode_InternFromString**(const char *str)

Valeur de retour : nouvelle référence. Part of the Stable ABI. A combination of *`PyUnicode_FromString()`* and *`PyUnicode_InternInPlace()`*, meant for statically allocated strings.

Return a new (”owned”) reference to either a new Unicode string object that has been interned, or an earlier interned string object with the same value.

Python may keep a reference to the result, or make it *immortal*, preventing it from being garbage-collected promptly. For interning an unbounded number of different strings, such as ones coming from user input, prefer calling *`PyUnicode_FromString()`* and *`PyUnicode_InternInPlace()`* directly.

unsigned int **PyUnicode_CHECK_INTERNED**(*PyObject* *str)

Return a non-zero value if *str* is interned, zero if not. The *str* argument must be a string ; this is not checked. This function always succeeds.

Particularité de l’implémentation CPython : A non-zero return value may carry additional information about *how* the string is interned. The meaning of such non-zero values, as well as each specific string’s intern-related details, may change between CPython versions.

PyUnicodeWriter

The *PyUnicodeWriter* API can be used to create a Python `str` object.

Ajouté dans la version 3.14.

type **PyUnicodeWriter**

A Unicode writer instance.

The instance must be destroyed by *PyUnicodeWriter_Finish()* on success, or *PyUnicodeWriter_Discard()* on error.

PyUnicodeWriter ***PyUnicodeWriter_Create**(*Py_ssize_t* length)

Create a Unicode writer instance.

length must be greater than or equal to `0`.

If *length* is greater than `0`, preallocate an internal buffer of *length* characters.

Set an exception and return `NULL` on error.

PyObject ***PyUnicodeWriter_Finish**(*PyUnicodeWriter* *writer)

Return the final Python `str` object and destroy the writer instance.

Set an exception and return `NULL` on error.

The writer instance is invalid after this call.

void **PyUnicodeWriter_Discard**(*PyUnicodeWriter* *writer)

Discard the internal Unicode buffer and destroy the writer instance.

If *writer* is `NULL`, no operation is performed.

The writer instance is invalid after this call.

int **PyUnicodeWriter_WriteChar**(*PyUnicodeWriter* *writer, *Py_UCS4* ch)

Write the single Unicode character *ch* into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteUTF8**(*PyUnicodeWriter* *writer, const char *str, *Py_ssize_t* size)

Decode the string *str* from UTF-8 in strict mode and write the output into *writer*.

size is the string length in bytes. If *size* is equal to `-1`, call `strlen(str)` to get the string length.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

See also *PyUnicodeWriter_DecodeUTF8Stateful()*.

int **PyUnicodeWriter_WriteASCII**(*PyUnicodeWriter* *writer, const char *str, *Py_ssize_t* size)

Write the ASCII string *str* into *writer*.

size is the string length in bytes. If *size* is equal to `-1`, call `strlen(str)` to get the string length.

str must only contain ASCII characters. The behavior is undefined if *str* contains non-ASCII characters.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

Ajouté dans la version 3.14.

int **PyUnicodeWriter_WriteWideChar**(*PyUnicodeWriter* *writer, const wchar_t *str, *Py_ssize_t* size)

Write the wide string *str* into *writer*.

size is a number of wide characters. If *size* is equal to `-1`, call `wcslen(str)` to get the string length.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteUCS4**(*PyUnicodeWriter* *writer, *Py_UCS4* *str, *Py_ssize_t* size)

Writer the UCS4 string *str* into *writer*.

size is a number of UCS4 characters.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteStr**(*PyUnicodeWriter* *writer, *PyObject* *obj)

Call *PyObject_Str()* on *obj* and write the output into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteRepr**(*PyUnicodeWriter* *writer, *PyObject* *obj)

Call *PyObject_Repr()* on *obj* and write the output into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_WriteSubstring**(*PyUnicodeWriter* *writer, *PyObject* *str, *Py_ssize_t* start, *Py_ssize_t* end)

Write the substring `str[start:end]` into *writer*.

str must be Python `str` object. *start* must be greater than or equal to 0, and less than or equal to *end*. *end* must be less than or equal to *str* length.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_Format**(*PyUnicodeWriter* *writer, const char *format, ...)

Similar to *PyUnicode_FromFormat()*, but write the output directly into *writer*.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

int **PyUnicodeWriter_DecodeUTF8Stateful**(*PyUnicodeWriter* *writer, const char *string, *Py_ssize_t* length, const char *errors, *Py_ssize_t* *consumed)

Decode the string *str* from UTF-8 with *errors* error handler and write the output into *writer*.

size is the string length in bytes. If *size* is equal to `-1`, call `strlen(str)` to get the string length.

errors is an error handler name, such as `"replace"`. If *errors* is `NULL`, use the strict error handler.

If *consumed* is not `NULL`, set **consumed* to the number of decoded bytes on success. If *consumed* is `NULL`, treat trailing incomplete UTF-8 byte sequences as an error.

On success, return `0`. On error, set an exception, leave the writer unchanged, and return `-1`.

See also *PyUnicodeWriter_WriteUTF8()*.

Deprecated API

The following API is deprecated.

type **Py_UNICODE**

This is a typedef of `wchar_t`, which is a 16-bit type or 32-bit type depending on the platform. Please use `wchar_t` directly instead.

Modifié dans la version 3.3 : Dans les versions précédentes, ceci était un type 16-bit ou un type 32-bit selon que vous choisissiez une version Unicode *narrow* ou *wide* de Python à la compilation.

Deprecated since version 3.13, will be removed in version 3.15.

int **PyUnicode_READY**(*PyObject* *unicode)

Do nothing and return `0`. This API is kept only for backward compatibility, but there are no plans to remove it.

Ajouté dans la version 3.3.

Obsolète depuis la version 3.10 : This API does nothing since Python 3.12. Previously, this needed to be called for each string created using the old API (`PyUnicode_FromUnicode()` or similar).

unsigned int **PyUnicode_IS_READY**(*PyObject* *unicode)

Do nothing and return `1`. This API is kept only for backward compatibility, but there are no plans to remove it.

Ajouté dans la version 3.3.

Obsolète depuis la version 3.14 : This API does nothing since Python 3.12. Previously, this could be called to check if *PyUnicode_READY()* is necessary.

9.3.4 Tuple Objects

type **PyTupleObject**

This subtype of *PyObject* represents a Python tuple object.

PyTypeObject **PyTuple_Type**

Part of the Stable ABI. This instance of *PyTypeObject* represents the Python tuple type ; it is the same object as `tuple` in the Python layer.

int **PyTuple_Check**(*PyObject* *p)

Return true if *p* is a tuple object or an instance of a subtype of the tuple type. This function always succeeds.

int **PyTuple_CheckExact**(*PyObject* *p)

Return true if *p* is a tuple object, but not an instance of a subtype of the tuple type. This function always succeeds.

PyObject ***PyTuple_New**(*Py_ssize_t* len)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new tuple object of size *len*, or `NULL` with an exception set on failure.

PyObject ***PyTuple_Pack**(*Py_ssize_t* n, ...)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new tuple object of size *n*, or `NULL` with an exception set on failure. The tuple values are initialized to the subsequent *n* C arguments pointing to Python objects. `PyTuple_Pack(2, a, b)` is equivalent to `Py_BuildValue("(OO)", a, b)`.

Py_ssize_t **PyTuple_Size**(*PyObject* *p)

Part of the Stable ABI. Take a pointer to a tuple object, and return the size of that tuple. On error, return `-1` and with an exception set.

Py_ssize_t **PyTuple_GET_SIZE**(*PyObject* *p)

Like *PyTuple_Size()*, but without error checking.

PyObject ***PyTuple_GetItem**(*PyObject* *p, *Py_ssize_t* pos)

Valeur de retour : référence empruntée. Part of the Stable ABI. Return the object at position *pos* in the tuple pointed to by *p*. If *pos* is negative or out of bounds, return `NULL` and set an `IndexError` exception.

The returned reference is borrowed from the tuple *p* (that is : it is only valid as long as you hold a reference to *p*). To get a *strong reference*, use *Py_NewRef(PyTuple_GetItem(...))* or *PySequence_GetItem()*.

PyObject ***PyTuple_GET_ITEM**(*PyObject* *p, *Py_ssize_t* pos)

Valeur de retour : référence empruntée. Like *PyTuple_GetItem()*, but does no checking of its arguments.

PyObject ***PyTuple_GetSlice**(*PyObject* *p, *Py_ssize_t* low, *Py_ssize_t* high)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the slice of the tuple pointed to by *p* between *low* and *high*, or `NULL` with an exception set on failure.

This is the equivalent of the Python expression `p[low:high]`. Indexing from the end of the tuple is not supported.

int **PyTuple_SetItem**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Part of the Stable ABI. Insert a reference to object *o* at position *pos* of the tuple pointed to by *p*. Return `0` on success. If *pos* is out of bounds, return `-1` and set an `IndexError` exception.

> **Note**
>
> This function ”steals” a reference to *o* and discards a reference to an item already in the tuple at the affected position.

void **PyTuple_SET_ITEM**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Like `PyTuple_SetItem()`, but does no error checking, and should *only* be used to fill in brand new tuples.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

> **Note**
>
> This function "steals" a reference to *o*, and, unlike `PyTuple_SetItem()`, does *not* discard a reference to any item that is being replaced; any reference in the tuple at position *pos* will be leaked.

> **Avertissement**
>
> This macro should *only* be used on tuples that are newly created. Using this macro on a tuple that is already in use (or in other words, has a refcount > 1) could lead to undefined behavior.

int **_PyTuple_Resize**(*PyObject* **p, *Py_ssize_t* newsize)

Can be used to resize a tuple. *newsize* will be the new length of the tuple. Because tuples are *supposed* to be immutable, this should only be used if there is only one reference to the object. Do *not* use this if the tuple may already be known to some other part of the code. The tuple will always grow or shrink at the end. Think of this as destroying the old tuple and creating a new one, only more efficiently. Returns `0` on success. Client code should never assume that the resulting value of `*p` will be the same as before calling this function. If the object referenced by `*p` is replaced, the original `*p` is destroyed. On failure, returns `-1` and sets `*p` to `NULL`, and raises `MemoryError` or `SystemError`.

9.3.5 Struct Sequence Objects

Struct sequence objects are the C equivalent of `namedtuple()` objects, i.e. a sequence whose items can also be accessed through attributes. To create a struct sequence, you first have to create a specific struct sequence type.

PyTypeObject ***PyStructSequence_NewType**(*PyStructSequence_Desc* *desc)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a new struct sequence type from the data in *desc*, described below. Instances of the resulting type can be created with `PyStructSequence_New()`.

Return `NULL` with an exception set on failure.

void **PyStructSequence_InitType**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

Initializes a struct sequence type *type* from *desc* in place.

int **PyStructSequence_InitType2**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

Like `PyStructSequence_InitType()`, but returns `0` on success and `-1` with an exception set on failure.

Ajouté dans la version 3.4.

type **PyStructSequence_Desc**

Part of the Stable ABI *(including all members).* Contains the meta information of a struct sequence type to create.

const char ***name**

Fully qualified name of the type; null-terminated UTF-8 encoded. The name must contain the module name.

const char ***doc**

Pointer to docstring for the type or `NULL` to omit.

PyStructSequence_Field ***fields**

Pointer to `NULL`-terminated array with field names of the new type.

int **n_in_sequence**

Number of fields visible to the Python side (if used as tuple).

type **PyStructSequence_Field**

Part of the Stable ABI *(including all members).* Describes a field of a struct sequence. As a struct sequence is modeled as a tuple, all fields are typed as *PyObject**. The index in the *fields* array of the *PyStructSequence_Desc* determines which field of the struct sequence is described.

const char ***name**

Name for the field or NULL to end the list of named fields, set to *PyStructSequence_UnnamedField* to leave unnamed.

const char ***doc**

Field docstring or NULL to omit.

const char *const **PyStructSequence_UnnamedField**

Part of the Stable ABI *Depuis la version 3.11.* Special value for a field name to leave it unnamed.

Modifié dans la version 3.9 : The type was changed from `char *`.

PyObject ***PyStructSequence_New**(*PyTypeObject* *type)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Creates an instance of *type*, which must have been created with *PyStructSequence_NewType()*.

Return NULL with an exception set on failure.

PyObject ***PyStructSequence_GetItem**(*PyObject* *p, *Py_ssize_t* pos)

Valeur de retour : référence empruntée. Part of the Stable ABI. Return the object at position *pos* in the struct sequence pointed to by *p*.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

PyObject ***PyStructSequence_GET_ITEM**(*PyObject* *p, *Py_ssize_t* pos)

Valeur de retour : référence empruntée. Alias to *PyStructSequence_GetItem()*.

Modifié dans la version 3.13 : Now implemented as an alias to *PyStructSequence_GetItem()*.

void **PyStructSequence_SetItem**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

Part of the Stable ABI. Sets the field at index *pos* of the struct sequence *p* to value *o*. Like *PyTuple_SET_ITEM()*, this should only be used to fill in brand new instances.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

> **Note**
>
> This function "steals" a reference to *o*.

void **PyStructSequence_SET_ITEM**(*PyObject* *p, *Py_ssize_t* *pos, *PyObject* *o)

Alias to *PyStructSequence_SetItem()*.

Modifié dans la version 3.13 : Now implemented as an alias to *PyStructSequence_SetItem()*.

9.3.6 List Objects

type **PyListObject**

This subtype of *PyObject* represents a Python list object.

PyTypeObject **PyList_Type**

Part of the Stable ABI. This instance of *PyTypeObject* represents the Python list type. This is the same object as `list` in the Python layer.

int **PyList_Check**(*PyObject* *p)

Return true if *p* is a list object or an instance of a subtype of the list type. This function always succeeds.

int **PyList_CheckExact**(*PyObject* *p)

Return true if *p* is a list object, but not an instance of a subtype of the list type. This function always succeeds.

PyObject ***PyList_New**(*Py_ssize_t* len)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new list of length *len* on success, or `NULL` on failure.

> **Note**
>
> If *len* is greater than zero, the returned list object's items are set to `NULL`. Thus you cannot use abstract API functions such as `PySequence_SetItem()` or expose the object to Python code before setting all items to a real object with `PyList_SetItem()` or `PyList_SET_ITEM()`. The following APIs are safe APIs before the list is fully initialized : `PyList_SetItem()` and `PyList_SET_ITEM()`.

Py_ssize_t **PyList_Size**(*PyObject* *list)

Part of the Stable ABI. Return the length of the list object in *list*; this is equivalent to `len(list)` on a list object.

Py_ssize_t **PyList_GET_SIZE**(*PyObject* *list)

Similar to `PyList_Size()`, but without error checking.

PyObject ***PyList_GetItemRef**(*PyObject* *list, *Py_ssize_t* index)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.13.* Return the object at position *index* in the list pointed to by *list*. The position must be non-negative; indexing from the end of the list is not supported. If *index* is out of bounds (`<0 or >=len(list)`), return `NULL` and set an `IndexError` exception.

Ajouté dans la version 3.13.

PyObject ***PyList_GetItem**(*PyObject* *list, *Py_ssize_t* index)

Valeur de retour : référence empruntée. Part of the Stable ABI. Like `PyList_GetItemRef()`, but returns a *borrowed reference* instead of a *strong reference*.

PyObject ***PyList_GET_ITEM**(*PyObject* *list, *Py_ssize_t* i)

Valeur de retour : référence empruntée. Similar to `PyList_GetItem()`, but without error checking.

int **PyList_SetItem**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

Part of the Stable ABI. Set the item at index *index* in list to *item*. Return `0` on success. If *index* is out of bounds, return `-1` and set an `IndexError` exception.

> **Note**
>
> This function "steals" a reference to *item* and discards a reference to an item already in the list at the affected position.

void **PyList_SET_ITEM**(*PyObject* *list, *Py_ssize_t* i, *PyObject* *o)

Macro form of `PyList_SetItem()` without error checking. This is normally only used to fill in new lists where there is no previous content.

Bounds checking is performed as an assertion if Python is built in debug mode or `with assertions`.

> **Note**
>
> This macro "steals" a reference to *item*, and, unlike `PyList_SetItem()`, does *not* discard a reference to any item that is being replaced; any reference in *list* at position *i* will be leaked.

int **PyList_Insert**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

Part of the Stable ABI. Insert the item *item* into list *list* in front of index *index*. Return `0` if successful; return `-1` and set an exception if unsuccessful. Analogous to `list.insert(index, item)`.

int **PyList_Append**(*PyObject* *list, *PyObject* *item)

Part of the Stable ABI. Append the object *item* at the end of list *list*. Return `0` if successful; return `-1` and set an exception if unsuccessful. Analogous to `list.append(item)`.

PyObject ***PyList_GetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a list of the objects in *list* containing the objects *between low* and *high*. Return `NULL` and set an exception if unsuccessful. Analogous to `list[low:high]`. Indexing from the end of the list is not supported.

int **PyList_SetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high, *PyObject* *itemlist)

Part of the Stable ABI. Set the slice of *list* between *low* and *high* to the contents of *itemlist*. Analogous to `list[low:high] = itemlist`. The *itemlist* may be `NULL`, indicating the assignment of an empty list (slice deletion). Return `0` on success, `-1` on failure. Indexing from the end of the list is not supported.

int **PyList_Extend**(*PyObject* *list, *PyObject* *iterable)

Extend *list* with the contents of *iterable*. This is the same as `PyList_SetSlice(list, PY_SSIZE_T_MAX, PY_SSIZE_T_MAX, iterable)` and analogous to `list.extend(iterable)` or `list += iterable`.

Raise an exception and return `-1` if *list* is not a `list` object. Return 0 on success.

Ajouté dans la version 3.13.

int **PyList_Clear**(*PyObject* *list)

Remove all items from *list*. This is the same as `PyList_SetSlice(list, 0, PY_SSIZE_T_MAX, NULL)` and analogous to `list.clear()` or `del list[:]`.

Raise an exception and return `-1` if *list* is not a `list` object. Return 0 on success.

Ajouté dans la version 3.13.

int **PyList_Sort**(*PyObject* *list)

Part of the Stable ABI. Sort the items of *list* in place. Return `0` on success, `-1` on failure. This is equivalent to `list.sort()`.

int **PyList_Reverse**(*PyObject* *list)

Part of the Stable ABI. Reverse the items of *list* in place. Return `0` on success, `-1` on failure. This is the equivalent of `list.reverse()`.

PyObject ***PyList_AsTuple**(*PyObject* *list)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new tuple object containing the contents of *list*; equivalent to `tuple(list)`.

9.4 Objets conteneurs

9.4.1 Objets dictionnaires

type **PyDictObject**

This subtype of *PyObject* represents a Python dictionary object.

PyTypeObject **PyDict_Type**

Part of the Stable ABI. This instance of *PyTypeObject* represents the Python dictionary type. This is the same object as `dict` in the Python layer.

int **PyDict_Check**(*PyObject* *p)

Return true if *p* is a dict object or an instance of a subtype of the dict type. This function always succeeds.

int **PyDict_CheckExact**(*PyObject* *p)

Return true if *p* is a dict object, but not an instance of a subtype of the dict type. This function always succeeds.

PyObject ***PyDict_New**()

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new empty dictionary, or `NULL` on failure.

PyObject ***PyDictProxy_New**(*PyObject* *mapping)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a `types.MappingProxyType` object for a mapping which enforces read-only behavior. This is normally used to create a view to prevent modification of the dictionary for non-dynamic class types.

void **PyDict_Clear**(*PyObject* *p)

Part of the Stable ABI. Empty an existing dictionary of all key-value pairs.

int **PyDict_Contains**(*PyObject* *p, *PyObject* *key)

Part of the Stable ABI. Determine if dictionary *p* contains *key*. If an item in *p* is matches *key*, return `1`, otherwise return `0`. On error, return `-1`. This is equivalent to the Python expression `key in p`.

int **PyDict_ContainsString**(*PyObject* *p, const char *key)

This is the same as *PyDict_Contains()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

Ajouté dans la version 3.13.

PyObject ***PyDict_Copy**(*PyObject* *p)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new dictionary that contains the same key-value pairs as *p*.

int **PyDict_SetItem**(*PyObject* *p, *PyObject* *key, *PyObject* *val)

Part of the Stable ABI. Insert *val* into the dictionary *p* with a key of *key*. *key* must be *hashable*; if it isn't, `TypeError` will be raised. Return `0` on success or `-1` on failure. This function *does not* steal a reference to *val*.

int **PyDict_SetItemString**(*PyObject* *p, const char *key, *PyObject* *val)

Part of the Stable ABI. This is the same as *PyDict_SetItem()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

int **PyDict_DelItem**(*PyObject* *p, *PyObject* *key)

Part of the Stable ABI. Remove the entry in dictionary *p* with key *key*. *key* must be *hashable*; if it isn't, `TypeError` is raised. If *key* is not in the dictionary, `KeyError` is raised. Return `0` on success or `-1` on failure.

int **PyDict_DelItemString**(*PyObject* *p, const char *key)

Part of the Stable ABI. This is the same as *PyDict_DelItem()*, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a *PyObject**.

int **PyDict_GetItemRef**(*PyObject* *p, *PyObject* *key, *PyObject* **result)

Part of the Stable ABI *Depuis la version 3.13.* Return a new *strong reference* to the object from dictionary *p* which has a key *key* :
— If the key is present, set **result* to a new *strong reference* to the value and return `1`.
— If the key is missing, set **result* to `NULL` and return `0`.
— On error, raise an exception and return `-1`.
Ajouté dans la version 3.13.

See also the *PyObject_GetItem()* function.

PyObject ***PyDict_GetItem**(*PyObject* *p, *PyObject* *key)

Valeur de retour : référence empruntée. Part of the Stable ABI. Return a *borrowed reference* to the object from dictionary *p* which has a key *key*. Return `NULL` if the key *key* is missing *without* setting an exception.

Note

Exceptions that occur while this calls `__hash__()` and `__eq__()` methods are silently ignored. Prefer the *PyDict_GetItemWithError()* function instead.

Modifié dans la version 3.10 : Calling this API without an *attached thread state* had been allowed for historical reason. It is no longer allowed.

PyObject ***PyDict_GetItemWithError**(*PyObject* *p, *PyObject* *key)

Valeur de retour : référence empruntée. Part of the Stable ABI. Variant of `PyDict_GetItem()` that does not suppress exceptions. Return `NULL` **with** an exception set if an exception occurred. Return `NULL` **without** an exception set if the key wasn't present.

PyObject ***PyDict_GetItemString**(*PyObject* *p, const char *key)

Valeur de retour : référence empruntée. Part of the Stable ABI. This is the same as `PyDict_GetItem()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

> **Note**
>
> Exceptions that occur while this calls `__hash__()` and `__eq__()` methods or while creating the temporary `str` object are silently ignored. Prefer using the `PyDict_GetItemWithError()` function with your own `PyUnicode_FromString()` *key* instead.

int **PyDict_GetItemStringRef**(*PyObject* *p, const char *key, *PyObject* **result)

Part of the Stable ABI *Depuis la version 3.13.* Similar to `PyDict_GetItemRef()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Ajouté dans la version 3.13.

PyObject ***PyDict_SetDefault**(*PyObject* *p, *PyObject* *key, *PyObject* *defaultobj)

Valeur de retour : référence empruntée. This is the same as the Python-level `dict.setdefault()`. If present, it returns the value corresponding to *key* from the dictionary *p*. If the key is not in the dict, it is inserted with value *defaultobj* and *defaultobj* is returned. This function evaluates the hash function of *key* only once, instead of evaluating it independently for the lookup and the insertion.

Ajouté dans la version 3.4.

int **PyDict_SetDefaultRef**(*PyObject* *p, *PyObject* *key, *PyObject* *default_value, *PyObject* **result)

Inserts *default_value* into the dictionary *p* with a key of *key* if the key is not already present in the dictionary. If *result* is not `NULL`, then **result* is set to a *strong reference* to either *default_value*, if the key was not present, or the existing value, if *key* was already present in the dictionary. Returns `1` if the key was present and *default_value* was not inserted, or `0` if the key was not present and *default_value* was inserted. On failure, returns `-1`, sets an exception, and sets `*result` to `NULL`.

For clarity : if you have a strong reference to *default_value* before calling this function, then after it returns, you hold a strong reference to both *default_value* and **result* (if it's not `NULL`). These may refer to the same object : in that case you hold two separate references to it.

Ajouté dans la version 3.13.

int **PyDict_Pop**(*PyObject* *p, *PyObject* *key, *PyObject* **result)

Remove *key* from dictionary *p* and optionally return the removed value. Do not raise `KeyError` if the key missing.

— If the key is present, set **result* to a new reference to the removed value if *result* is not `NULL`, and return `1`.
— If the key is missing, set **result* to `NULL` if *result* is not `NULL`, and return `0`.
— On error, raise an exception and return `-1`.

Similar to `dict.pop()`, but without the default value and not raising `KeyError` if the key missing.

Ajouté dans la version 3.13.

int **PyDict_PopString**(*PyObject* *p, const char *key, *PyObject* **result)

Similar to `PyDict_Pop()`, but *key* is specified as a `const char*` UTF-8 encoded bytes string, rather than a `PyObject*`.

Ajouté dans la version 3.13.

PyObject ***PyDict_Items**(*PyObject* *p)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a `PyListObject` containing all the items from the dictionary.

PyObject ***PyDict_Keys**(*PyObject* *p)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a `PyListObject` containing all the keys from the dictionary.

PyObject ***PyDict_Values**(*PyObject* *p)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a `PyListObject` containing all the values from the dictionary *p*.

Py_ssize_t **PyDict_Size**(*PyObject* *p)

Part of the Stable ABI. Return the number of items in the dictionary. This is equivalent to `len(p)` on a dictionary.

int **PyDict_Next**(*PyObject* *p, *Py_ssize_t* *ppos, *PyObject* **pkey, *PyObject* **pvalue)

Part of the Stable ABI. Iterate over all key-value pairs in the dictionary *p*. The `Py_ssize_t` referred to by *ppos* must be initialized to `0` prior to the first call to this function to start the iteration ; the function returns true for each pair in the dictionary, and false once all pairs have been reported. The parameters *pkey* and *pvalue* should either point to `PyObject*` variables that will be filled in with each key and value, respectively, or may be `NULL`. Any references returned through them are borrowed. *ppos* should not be altered during iteration. Its value represents offsets within the internal dictionary structure, and since the structure is sparse, the offsets are not consecutive.

Par exemple :

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}
```

The dictionary *p* should not be mutated during iteration. It is safe to modify the values of the keys as you iterate over the dictionary, but only so long as the set of keys does not change. For example :

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    long i = PyLong_AsLong(value);
    if (i == -1 && PyErr_Occurred()) {
        return -1;
    }
    PyObject *o = PyLong_FromLong(i + 1);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
    Py_DECREF(o);
}
```

The function is not thread-safe in the *free-threaded* build without external synchronization. You can use `Py_BEGIN_CRITICAL_SECTION` to lock the dictionary while iterating over it :

```
Py_BEGIN_CRITICAL_SECTION(self->dict);
while (PyDict_Next(self->dict, &pos, &key, &value)) {
    ...
}
Py_END_CRITICAL_SECTION();
```

Note

On the free-threaded build, this function can be used safely inside a critical section. However, the references returned for *pkey* and *pvalue* are *borrowed* and are only valid while the critical section is held. If you need to use these objects outside the critical section or when the critical section can be suspended, create a *strong reference* (for example, using `Py_NewRef()`).

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

Part of the Stable ABI. Iterate over mapping object *b* adding key-value pairs to dictionary *a*. *b* may be a dictionary, or any object supporting `PyMapping_Keys()` and `PyObject_GetItem()`. If *override* is true, existing pairs in *a* will be replaced if a matching key is found in *b*, otherwise pairs will only be added if there is not a matching key in *a*. Return `0` on success or `-1` if an exception was raised.

int **PyDict_Update**(*PyObject* *a, *PyObject* *b)

Part of the Stable ABI. This is the same as `PyDict_Merge(a, b, 1)` in C, and is similar to `a.update(b)` in Python except that `PyDict_Update()` doesn't fall back to the iterating over a sequence of key value pairs if the second argument has no "keys" attribute. Return `0` on success or `-1` if an exception was raised.

int **PyDict_MergeFromSeq2**(*PyObject* *a, *PyObject* *seq2, int override)

Part of the Stable ABI. Update or merge into dictionary *a*, from the key-value pairs in *seq2*. *seq2* must be an iterable object producing iterable objects of length 2, viewed as key-value pairs. In case of duplicate keys, the last wins if *override* is true, else the first wins. Return `0` on success or `-1` if an exception was raised. Equivalent Python (except for the return value) :

```
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value
```

int **PyDict_AddWatcher**(*PyDict_WatchCallback* callback)

Register *callback* as a dictionary watcher. Return a non-negative integer id which must be passed to future calls to `PyDict_Watch()`. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Ajouté dans la version 3.12.

int **PyDict_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from `PyDict_AddWatcher()`. Return `0` on success, `-1` on error (e.g. if the given *watcher_id* was never registered.)

Ajouté dans la version 3.12.

int **PyDict_Watch**(int watcher_id, *PyObject* *dict)

Mark dictionary *dict* as watched. The callback granted *watcher_id* by `PyDict_AddWatcher()` will be called when *dict* is modified or deallocated. Return `0` on success or `-1` on error.

Ajouté dans la version 3.12.

int **PyDict_Unwatch**(int watcher_id, *PyObject* *dict)

Mark dictionary *dict* as no longer watched. The callback granted *watcher_id* by `PyDict_AddWatcher()` will no longer be called when *dict* is modified or deallocated. The dict must previously have been watched by this watcher. Return `0` on success or `-1` on error.

Ajouté dans la version 3.12.

type **PyDict_WatchEvent**

Enumeration of possible dictionary watcher events : `PyDict_EVENT_ADDED`, `PyDict_EVENT_MODIFIED`, `PyDict_EVENT_DELETED`, `PyDict_EVENT_CLONED`, `PyDict_EVENT_CLEARED`, or `PyDict_EVENT_DEALLOCATED`.

Ajouté dans la version 3.12.

typedef int (***PyDict_WatchCallback**)(*PyDict_WatchEvent* event, *PyObject* *dict, *PyObject* *key, *PyObject* *new_value)

Type of a dict watcher callback function.

If *event* is `PyDict_EVENT_CLEARED` or `PyDict_EVENT_DEALLOCATED`, both *key* and *new_value* will be `NULL`. If *event* is `PyDict_EVENT_ADDED` or `PyDict_EVENT_MODIFIED`, *new_value* will be the new value for *key*. If *event* is `PyDict_EVENT_DELETED`, *key* is being deleted from the dictionary and *new_value* will be `NULL`.

`PyDict_EVENT_CLONED` occurs when *dict* was previously empty and another dict is merged into it. To maintain efficiency of this operation, per-key `PyDict_EVENT_ADDED` events are not issued in this case ; instead a single `PyDict_EVENT_CLONED` is issued, and *key* will be the source dictionary.

The callback may inspect but must not modify *dict* ; doing so could have unpredictable effects, including infinite recursion. Do not trigger Python code execution in the callback, as it could modify the dict as a side effect.

If *event* is `PyDict_EVENT_DEALLOCATED`, taking a new reference in the callback to the about-to-be-destroyed dictionary will resurrect it and prevent it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

Callbacks occur before the notified modification to *dict* takes place, so the prior state of *dict* can be inspected.

If the callback sets an exception, it must return `-1` ; this exception will be printed as an unraisable exception using *PyErr_WriteUnraisable()*. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Ajouté dans la version 3.12.

9.4.2 Set Objects

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

type **PySetObject**

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

PyTypeObject **PySet_Type**

Part of the Stable ABI. This is an instance of *PyTypeObject* representing the Python `set` type.

PyTypeObject **PyFrozenSet_Type**

Part of the Stable ABI. This is an instance of *PyTypeObject* representing the Python `frozenset` type.

The following type check macros work on pointers to any Python object. Likewise, the constructor functions work with any iterable Python object.

int **PySet_Check**(*PyObject* *p)

Return true if *p* is a `set` object or an instance of a subtype. This function always succeeds.

int **PyFrozenSet_Check**(*PyObject* *p)

Return true if *p* is a `frozenset` object or an instance of a subtype. This function always succeeds.

int **PyAnySet_Check**(*PyObject* *p)

Return true if *p* is a `set` object, a `frozenset` object, or an instance of a subtype. This function always succeeds.

int **PySet_CheckExact**(*PyObject* *p)

Return true if *p* is a `set` object but not an instance of a subtype. This function always succeeds.

Ajouté dans la version 3.10.

int **PyAnySet_CheckExact**(*PyObject* *p)

Return true if *p* is a `set` object or a `frozenset` object but not an instance of a subtype. This function always succeeds.

int **PyFrozenSet_CheckExact**(*PyObject* *p)

Return true if *p* is a `frozenset` object but not an instance of a subtype. This function always succeeds.

PyObject ***PySet_New**(*PyObject* *iterable)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new `set` containing objects returned by the *iterable*. The *iterable* may be `NULL` to create a new empty set. Return the new set on success or `NULL` on failure. Raise `TypeError` if *iterable* is not actually iterable. The constructor is also useful for copying a set (`c=set(s)`).

PyObject ***PyFrozenSet_New**(*PyObject* *iterable)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new `frozenset` containing objects returned by the *iterable*. The *iterable* may be `NULL` to create a new empty frozenset. Return the new set on success or `NULL` on failure. Raise `TypeError` if *iterable* is not actually iterable.

The following functions and macros are available for instances of `set` or `frozenset` or instances of their subtypes.

Py_ssize_t **PySet_Size**(*PyObject* *anyset)

Part of the Stable ABI. Return the length of a `set` or `frozenset` object. Equivalent to `len(anyset)`. Raises a `SystemError` if *anyset* is not a `set`, `frozenset`, or an instance of a subtype.

Py_ssize_t **PySet_GET_SIZE**(*PyObject* *anyset)

Macro form of *`PySet_Size()`* without error checking.

int **PySet_Contains**(*PyObject* *anyset, *PyObject* *key)

Part of the Stable ABI. Return `1` if found, `0` if not found, and `-1` if an error is encountered. Unlike the Python `__contains__()` method, this function does not automatically convert unhashable sets into temporary frozensets. Raise a `TypeError` if the *key* is unhashable. Raise `SystemError` if *anyset* is not a `set`, `frozenset`, or an instance of a subtype.

int **PySet_Add**(*PyObject* *set, *PyObject* *key)

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

The following functions are available for instances of `set` or its subtypes but not for instances of `frozenset` or its subtypes.

int **PySet_Discard**(*PyObject* *set, *PyObject* *key)

Part of the Stable ABI. Return `1` if found and removed, `0` if not found (no action taken), and `-1` if an error is encountered. Does not raise `KeyError` for missing keys. Raise a `TypeError` if the *key* is unhashable. Unlike the Python `discard()` method, this function does not automatically convert unhashable sets into temporary frozensets. Raise `SystemError` if *set* is not an instance of `set` or its subtype.

PyObject ***PySet_Pop**(*PyObject* *set)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new reference to an arbitrary object in the *set*, and removes the object from the *set*. Return `NULL` on failure. Raise `KeyError` if the set is empty. Raise a `SystemError` if *set* is not an instance of `set` or its subtype.

int **PySet_Clear**(*PyObject* *set)

Part of the Stable ABI. Empty an existing set of all elements. Return `0` on success. Return `-1` and raise `SystemError` if *set* is not an instance of `set` or its subtype.

9.5 Objets fonctions

9.5.1 Objets fonction

Certaines fonctions sont spécifiques aux fonctions Python.

type **PyFunctionObject**

La structure C utilisée pour les fonctions.

PyTypeObject **PyFunction_Type**

C'est une instance de *PyTypeObject* et représente le type fonction en Python. Il est exposé aux développeurs comme `types.FunctionType`.

int **PyFunction_Check**(*PyObject* *o)

Renvoie vrai si *o* est un objet de type fonction (a comme type *PyFunction_Type*). Le paramètre ne doit pas être `NULL`. Cette fonction réussit toujours.

PyObject ***PyFunction_New**(*PyObject* *code, *PyObject* *globals)

Valeur de retour : nouvelle référence. Renvoie une nouvelle fonction associée avec l'objet *code*. *globals* doit être un dictionnaire avec les variables globales accessibles à la fonction.

The function's docstring and name are retrieved from the code object. `__module__` is retrieved from *globals*. The argument defaults, annotations and closure are set to `NULL`. `__qualname__` is set to the same value as the code object's `co_qualname` field.

PyObject ***PyFunction_NewWithQualName**(*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)

Valeur de retour : nouvelle référence. As *PyFunction_New()*, but also allows setting the function object's `__qualname__` attribute. *qualname* should be a unicode object or `NULL`; if `NULL`, the `__qualname__` attribute is set to the same value as the code object's `co_qualname` field.

Ajouté dans la version 3.3.

PyObject ***PyFunction_GetCode**(*PyObject* *op)

Valeur de retour : référence empruntée. Renvoie l'objet code associé avec l'objet de la fonction *op*.

PyObject ***PyFunction_GetGlobals**(*PyObject* *op)

Valeur de retour : référence empruntée. Renvoie le dictionnaire global associé avec l'objet de la fonction *op*.

PyObject ***PyFunction_GetModule**(*PyObject* *op)

Valeur de retour : référence empruntée. Return a *borrowed reference* to the `__module__` attribute of the function object *op*. It can be *NULL*.

This is normally a `string` containing the module name, but can be set to any other object by Python code.

PyObject ***PyFunction_GetDefaults**(*PyObject* *op)

Valeur de retour : référence empruntée. Renvoie les valeurs par défaut de l'argument de l'objet de la fonction *op*. Cela peut être un tuple d'arguments ou `NULL`.

int **PyFunction_SetDefaults**(*PyObject* *op, *PyObject* *defaults)

Définir les valeurs par défaut de l'argument pour l'objet de la fonction *op*. *defaults* doit être `Py_None` ou un tuple.

Lève `SystemError` et renvoie `-1` en cas de d'échec.

void **PyFunction_SetVectorcall**(*PyFunctionObject* *func, *vectorcallfunc* vectorcall)

Set the vectorcall field of a given function object *func*.

Warning : extensions using this API must preserve the behavior of the unaltered (default) vectorcall function !

Ajouté dans la version 3.12.

PyObject ***PyFunction_GetKwDefaults**(*PyObject* *op)

Valeur de retour : référence empruntée. Return the keyword-only argument default values of the function object *op*. This can be a dictionary of arguments or `NULL`.

PyObject ***PyFunction_GetClosure**(*PyObject* *op)

Valeur de retour : référence empruntée. Renvoie la fermeture associée avec l'objet de la fonction *op*. Cela peut être `NULL` ou un tuple d'objets cellule.

int **PyFunction_SetClosure**(*PyObject* *op, *PyObject* *closure)

Définir la fermeture associée avec l'objet de la fonction *op*. *closure* doit être `Py_None` ou un tuple d'objets cellule.

Lève `SystemError` et renvoie `-1` en cas de d'échec.

PyObject ***PyFunction_GetAnnotations**(*PyObject* *op)

Valeur de retour : référence empruntée. Renvoie les annotations de l'objet de la fonction *op*. Cela peut être un dictionnaire mutable ou `NULL`.

int **PyFunction_SetAnnotations**(*PyObject* *op, *PyObject* *annotations)

Définir les annotations pour l'objet de la fonction *op*. *annotations* doit être un dictionnaire ou `Py_None`.

Lève `SystemError` et renvoie `-1` en cas de d'échec.

PyObject ***PyFunction_GET_CODE**(*PyObject* *op)

PyObject ***PyFunction_GET_GLOBALS**(*PyObject* *op)

PyObject ***PyFunction_GET_MODULE**(*PyObject* *op)

PyObject ***PyFunction_GET_DEFAULTS**(*PyObject* *op)

PyObject ***PyFunction_GET_KW_DEFAULTS**(*PyObject* *op)

PyObject ***PyFunction_GET_CLOSURE**(*PyObject* *op)

PyObject ***PyFunction_GET_ANNOTATIONS**(*PyObject* *op)

Valeur de retour : référence empruntée. These functions are similar to their `PyFunction_Get*` counterparts, but do not do type checking. Passing anything other than an instance of *`PyFunction_Type`* is undefined behavior.

int **PyFunction_AddWatcher**(*PyFunction_WatchCallback* callback)

Register *callback* as a function watcher for the current interpreter. Return an ID which may be passed to *`PyFunction_ClearWatcher()`*. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Ajouté dans la version 3.12.

int **PyFunction_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from *`PyFunction_AddWatcher()`* for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

Ajouté dans la version 3.12.

type **PyFunction_WatchEvent**

Enumeration of possible function watcher events :
— `PyFunction_EVENT_CREATE`
— `PyFunction_EVENT_DESTROY`
— `PyFunction_EVENT_MODIFY_CODE`
— `PyFunction_EVENT_MODIFY_DEFAULTS`
— `PyFunction_EVENT_MODIFY_KWDEFAULTS`

Ajouté dans la version 3.12.

typedef int (***PyFunction_WatchCallback**)(*PyFunction_WatchEvent* event, *PyFunctionObject* *func, *PyObject* *new_value)

Type of a function watcher callback function.

If *event* is `PyFunction_EVENT_CREATE` or `PyFunction_EVENT_DESTROY` then *new_value* will be `NULL`. Otherwise, *new_value* will hold a *borrowed reference* to the new value that is about to be stored in *func* for the attribute that is being modified.

The callback may inspect but must not modify *func*; doing so could have unpredictable effects, including infinite recursion.

If *event* is `PyFunction_EVENT_CREATE`, then the callback is invoked after *func* has been fully initialized. Otherwise, the callback is invoked before the modification to *func* takes place, so the prior state of *func* can be inspected. The runtime is permitted to optimize away the creation of function objects when possible. In such cases no event will be emitted. Although this creates the possibility of an observable difference of runtime behavior depending on optimization decisions, it does not change the semantics of the Python code being executed.

If *event* is `PyFunction_EVENT_DESTROY`, Taking a reference in the callback to the about-to-be-destroyed function will resurrect it, preventing it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using *PyErr_WriteUnraisable()*. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Ajouté dans la version 3.12.

9.5.2 Instance Method Objects

An instance method is a wrapper for a *PyCFunction* and the new way to bind a *PyCFunction* to a class object. It replaces the former call `PyMethod_New(func, NULL, class)`.

PyTypeObject **PyInstanceMethod_Type**

This instance of *PyTypeObject* represents the Python instance method type. It is not exposed to Python programs.

int **PyInstanceMethod_Check**(*PyObject* *o)

Return true if *o* is an instance method object (has type *PyInstanceMethod_Type*). The parameter must not be `NULL`. This function always succeeds.

PyObject ***PyInstanceMethod_New**(*PyObject* *func)

Valeur de retour : nouvelle référence. Return a new instance method object, with *func* being any callable object. *func* is the function that will be called when the instance method is called.

PyObject ***PyInstanceMethod_Function**(*PyObject* *im)

Valeur de retour : référence empruntée. Return the function object associated with the instance method *im*.

PyObject ***PyInstanceMethod_GET_FUNCTION**(*PyObject* *im)

Valeur de retour : référence empruntée. Macro version of *PyInstanceMethod_Function()* which avoids error checking.

9.5.3 Objets méthode

Methods are bound function objects. Methods are always bound to an instance of a user-defined class. Unbound methods (methods bound to a class object) are no longer available.

PyTypeObject **PyMethod_Type**

This instance of *PyTypeObject* represents the Python method type. This is exposed to Python programs as `types.MethodType`.

int **PyMethod_Check**(*PyObject* *o)

Return true if *o* is a method object (has type *PyMethod_Type*). The parameter must not be `NULL`. This function always succeeds.

PyObject ***PyMethod_New**(*PyObject* *func, *PyObject* *self)

Valeur de retour : nouvelle référence. Return a new method object, with *func* being any callable object and *self* the instance the method should be bound. *func* is the function that will be called when the method is called. *self* must not be `NULL`.

PyObject ***PyMethod_Function**(*PyObject* *meth)

Valeur de retour : référence empruntée. Return the function object associated with the method *meth*.

PyObject ***PyMethod_GET_FUNCTION**(*PyObject* *meth)

Valeur de retour : référence empruntée. Macro version of *PyMethod_Function()* which avoids error checking.

PyObject ***PyMethod_Self**(*PyObject* *meth)

Valeur de retour : référence empruntée. Return the instance associated with the method *meth*.

PyObject ***PyMethod_GET_SELF**(*PyObject* *meth)

Valeur de retour : référence empruntée. Macro version of *PyMethod_Self()* which avoids error checking.

9.5.4 Objets Cellules

Les objets "Cellules" (*cell* en anglais) sont utilisés pour implémenter des variables référencées dans de multiples environnements. Pour chacune de ces variables, un objet cellule est créé pour stocker sa valeur ; les variables locales de chaque pile d'exécution qui référence cette valeur contiennent une référence sur les cellules des autres environnements qui utilisent aussi cette variable. Quand la valeur est accédée, la valeur de la cellule est utilisée, au lei de celle de l'objet cellule proprement dit. Ce dé-référencement de l'objet cellule requiert l'intervention du *bytecode* généré ; il n'est pas automatiquement dé-référencé quand il est accédé. Il est plausible que les objets cellules ne soit utilisés ailleurs.

type **PyCellObject**

Structure C utilisée pour les objets cellules.

PyTypeObject **PyCell_Type**

Type objet correspondant aux objets cellules.

int **PyCell_Check**(*PyObject* *ob)

Return true if *ob* is a cell object ; *ob* must not be `NULL`. This function always succeeds.

PyObject ***PyCell_New**(*PyObject* *ob)

Valeur de retour : nouvelle référence. Create and return a new cell object containing the value *ob*. The parameter may be `NULL`.

PyObject ***PyCell_Get**(*PyObject* *cell)

Valeur de retour : nouvelle référence. Return the contents of the cell *cell*, which can be `NULL`. If *cell* is not a cell object, returns `NULL` with an exception set.

PyObject ***PyCell_GET**(*PyObject* *cell)

Valeur de retour : référence empruntée. Return the contents of the cell *cell*, but without checking that *cell* is non-`NULL` and a cell object.

int **PyCell_Set**(*PyObject* *cell, *PyObject* *value)

Set the contents of the cell object *cell* to *value*. This releases the reference to any current content of the cell. *value* may be `NULL`. *cell* must be non-`NULL`.

On success, return `0`. If *cell* is not a cell object, set an exception and return `-1`.

void **PyCell_SET**(*PyObject* *cell, *PyObject* *value)

Sets the value of the cell object *cell* to *value*. No reference counts are adjusted, and no checks are made for safety; *cell* must be non-`NULL` and must be a cell object.

9.5.5 Objets code

Les objets *Code* sont spécifiques à l'implémentation bas niveau de CPython. Chacun d'eux représente une partie de code exécutable, qui n'a pas encore été lié dans une fonction.

type **PyCodeObject**

La structure C utilisée pour décrire les objets *Code*. Les attributs de cette structure sont sujets à changer à tout moment.

PyTypeObject **PyCode_Type**

This is an instance of `PyTypeObject` representing the Python code object.

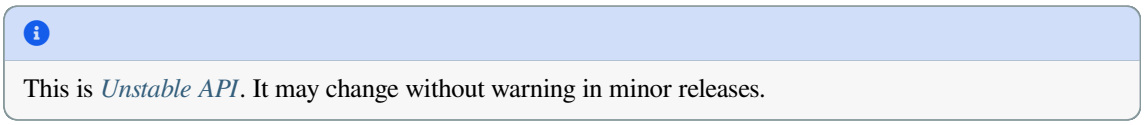
int **PyCode_Check**(*PyObject* *co)

Return true if *co* is a code object. This function always succeeds.

Py_ssize_t **PyCode_GetNumFree**(*PyCodeObject* *co)

Return the number of *free (closure) variables* in a code object.

int **PyUnstable_Code_GetFirstFree**(*PyCodeObject* *co)

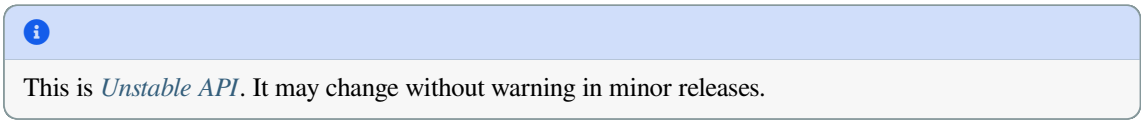


This is *Unstable API*. It may change without warning in minor releases.

Return the position of the first *free (closure) variable* in a code object.

Modifié dans la version 3.13 : Renamed from `PyCode_GetFirstFree` as part of *Unstable C API*. The old name is deprecated, but will remain available until the signature changes again.

PyCodeObject ***PyUnstable_Code_New**(int argcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)



This is *Unstable API*. It may change without warning in minor releases.

Return a new code object. If you need a dummy code object to create a frame, use `PyCode_NewEmpty()` instead.

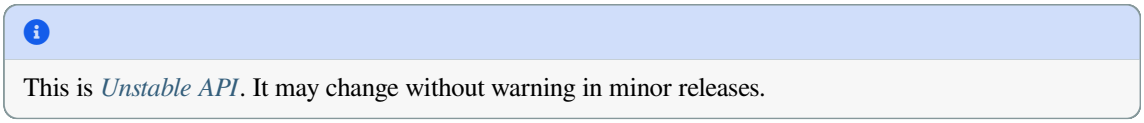
Since the definition of the bytecode changes often, calling `PyUnstable_Code_New()` directly can bind you to a precise Python version.

The many arguments of this function are inter-dependent in complex ways, meaning that subtle changes to values are likely to result in incorrect execution or VM crashes. Use this function only with extreme care.

Modifié dans la version 3.11 : Added `qualname` and `exceptiontable` parameters.

Modifié dans la version 3.12 : Renamed from `PyCode_New` as part of *Unstable C API*. The old name is deprecated, but will remain available until the signature changes again.

PyCodeObject ***PyUnstable_Code_NewWithPosOnlyArgs**(int argcount, int posonlyargcount, int kwonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, *PyObject* *qualname, int firstlineno, *PyObject* *linetable, *PyObject* *exceptiontable)



This is *Unstable API*. It may change without warning in minor releases.

Similar to `PyUnstable_Code_New()`, but with an extra "posonlyargcount" for positional-only arguments. The same caveats that apply to `PyUnstable_Code_New` also apply to this function.

Ajouté dans la version 3.8 : as `PyCode_NewWithPosOnlyArgs`

Modifié dans la version 3.11 : Added `qualname` and `exceptiontable` parameters.

Modifié dans la version 3.12 : Renamed to `PyUnstable_Code_NewWithPosOnlyArgs`. The old name is deprecated, but will remain available until the signature changes again.

PyCodeObject ***PyCode_NewEmpty**(const char *filename, const char *funcname, int firstlineno)

Valeur de retour : nouvelle référence. Return a new empty code object with the specified filename, function name, and first line number. The resulting code object will raise an `Exception` if executed.

int **PyCode_Addr2Line**(*PyCodeObject* *co, int byte_offset)

Return the line number of the instruction that occurs on or before `byte_offset` and ends after it. If you just need the line number of a frame, use `PyFrame_GetLineNumber()` instead.

For efficiently iterating over the line numbers in a code object, use **the API described in PEP 626**.

int **PyCode_Addr2Location**(*PyObject* *co, int byte_offset, int *start_line, int *start_column, int *end_line, int *end_column)

Sets the passed `int` pointers to the source code line and column numbers for the instruction at `byte_offset`. Sets the value to `0` when information is not available for any particular element.

Returns `1` if the function succeeds and 0 otherwise.

Ajouté dans la version 3.11.

PyObject ***PyCode_GetCode**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_code')`. Returns a strong reference to a `PyBytesObject` representing the bytecode in a code object. On error, `NULL` is returned and an exception is raised.

This `PyBytesObject` may be created on-demand by the interpreter and does not necessarily represent the bytecode actually executed by CPython. The primary use case for this function is debuggers and profilers.

Ajouté dans la version 3.11.

PyObject ***PyCode_GetVarnames**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_varnames')`. Returns a new reference to a `PyTupleObject` containing the names of the local variables. On error, `NULL` is returned and an exception is raised.

Ajouté dans la version 3.11.

PyObject ***PyCode_GetCellvars**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_cellvars')`. Returns a new reference to a `PyTupleObject` containing the names of the local variables that are referenced by nested functions. On error, `NULL` is returned and an exception is raised.

Ajouté dans la version 3.11.

PyObject ***PyCode_GetFreevars**(*PyCodeObject* *co)

Equivalent to the Python code `getattr(co, 'co_freevars')`. Returns a new reference to a *PyTupleObject* containing the names of the *free (closure) variables*. On error, `NULL` is returned and an exception is raised.

Ajouté dans la version 3.11.

int **PyCode_AddWatcher**(*PyCode_WatchCallback* callback)

Register *callback* as a code object watcher for the current interpreter. Return an ID which may be passed to *PyCode_ClearWatcher()*. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Ajouté dans la version 3.12.

int **PyCode_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from *PyCode_AddWatcher()* for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

Ajouté dans la version 3.12.

type **PyCodeEvent**

Enumeration of possible code object watcher events : - `PY_CODE_EVENT_CREATE` - `PY_CODE_EVENT_DESTROY`

Ajouté dans la version 3.12.

typedef int (***PyCode_WatchCallback**)(*PyCodeEvent* event, *PyCodeObject* *co)

Type of a code object watcher callback function.

If *event* is `PY_CODE_EVENT_CREATE`, then the callback is invoked after *co* has been fully initialized. Otherwise, the callback is invoked before the destruction of *co* takes place, so the prior state of *co* can be inspected.

If *event* is `PY_CODE_EVENT_DESTROY`, taking a reference in the callback to the about-to-be-destroyed code object will resurrect it and prevent it from being freed at this time. When the resurrected object is destroyed later, any watcher callbacks active at that time will be called again.

Users of this API should not rely on internal runtime implementation details. Such details may include, but are not limited to, the exact order and timing of creation and destruction of code objects. While changes in these details may result in differences observable by watchers (including whether a callback is invoked or not), it does not change the semantics of the Python code being executed.

If the callback sets an exception, it must return `-1`; this exception will be printed as an unraisable exception using *PyErr_WriteUnraisable()*. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Ajouté dans la version 3.12.

9.5.6 Code Object Flags

Code objects contain a bit-field of flags, which can be retrieved as the `co_flags` Python attribute (for example using *PyObject_GetAttrString()*), and set using a *flags* argument to *PyUnstable_Code_New()* and similar functions.

Flags whose names start with `CO_FUTURE_` correspond to features normally selectable by future statements. These flags can be used in *PyCompilerFlags.cf_flags*. Note that many `CO_FUTURE_` flags are mandatory in current versions of Python, and setting them has no effect.

The following flags are available. For their meaning, see the linked documentation of their Python equivalents.

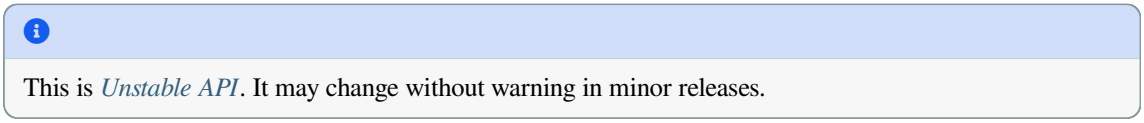
Flag	Meaning
CO_OPTIMIZED	inspect.CO_OPTIMIZED
CO_NEWLOCALS	inspect.CO_NEWLOCALS
CO_VARARGS	inspect.CO_VARARGS
CO_VARKEYWORDS	inspect.CO_VARKEYWORDS
CO_NESTED	inspect.CO_NESTED
CO_GENERATOR	inspect.CO_GENERATOR
CO_COROUTINE	inspect.CO_COROUTINE
CO_ITERABLE_COROUTINE	inspect.CO_ITERABLE_COROUTINE
CO_ASYNC_GENERATOR	inspect.CO_ASYNC_GENERATOR
CO_HAS_DOCSTRING	inspect.CO_HAS_DOCSTRING
CO_METHOD	inspect.CO_METHOD
CO_FUTURE_DIVISION	no effect (__future__.division)
CO_FUTURE_ABSOLUTE_IMPORT	no effect (__future__.absolute_import)
CO_FUTURE_WITH_STATEMENT	no effect (__future__.with_statement)
CO_FUTURE_PRINT_FUNCTION	no effect (__future__.print_function)
CO_FUTURE_UNICODE_LITERALS	no effect (__future__.unicode_literals)
CO_FUTURE_GENERATOR_STOP	no effect (__future__.generator_stop)
CO_FUTURE_ANNOTATIONS	__future__.annotations

9.5.7 Extra information

To support low-level extensions to frame evaluation, such as external just-in-time compilers, it is possible to attach arbitrary extra data to code objects.

These functions are part of the unstable C API tier : this functionality is a CPython implementation detail, and the API may change without deprecation warnings.

Py_ssize_t **PyUnstable_Eval_RequestCodeExtraIndex**(*freefunc* free)



This is *Unstable API*. It may change without warning in minor releases.

Return a new an opaque index value used to adding data to code objects.

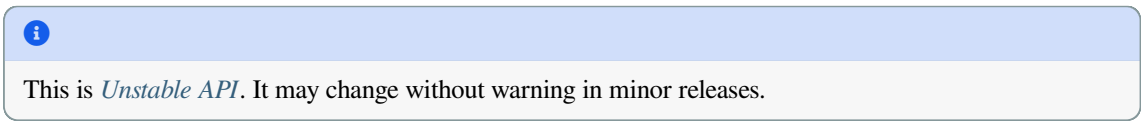
You generally call this function once (per interpreter) and use the result with `PyCode_GetExtra` and `PyCode_SetExtra` to manipulate data on individual code objects.

If *free* is not `NULL` : when a code object is deallocated, *free* will be called on non-`NULL` data stored under the new index. Use *`Py_DecRef()`* when storing *`PyObject`*.

Ajouté dans la version 3.6 : as `_PyEval_RequestCodeExtraIndex`

Modifié dans la version 3.12 : Renamed to `PyUnstable_Eval_RequestCodeExtraIndex`. The old private name is deprecated, but will be available until the API changes.

int **PyUnstable_Code_GetExtra**(*PyObject* *code, *Py_ssize_t* index, void **extra)



This is *Unstable API*. It may change without warning in minor releases.

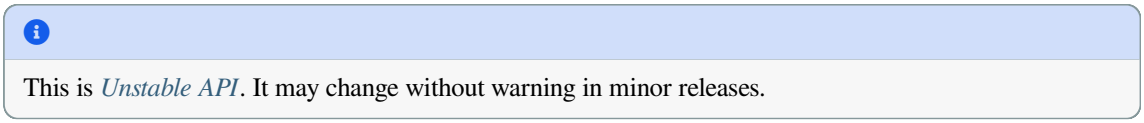
Set *extra* to the extra data stored under the given index. Return 0 on success. Set an exception and return -1 on failure.

If no data was set under the index, set *extra* to `NULL` and return 0 without setting an exception.

Ajouté dans la version 3.6 : as `_PyCode_GetExtra`

Modifié dans la version 3.12 : Renamed to `PyUnstable_Code_GetExtra`. The old private name is deprecated, but will be available until the API changes.

int **PyUnstable_Code_SetExtra**(*PyObject* *code, *Py_ssize_t* index, void *extra)



This is *Unstable API*. It may change without warning in minor releases.

Set the extra data stored under the given index to *extra*. Return 0 on success. Set an exception and return -1 on failure.

Ajouté dans la version 3.6 : as `_PyCode_SetExtra`

Modifié dans la version 3.12 : Renamed to `PyUnstable_Code_SetExtra`. The old private name is deprecated, but will be available until the API changes.

9.6 Autres objets

9.6.1 Objets fichiers

These APIs are a minimal emulation of the Python 2 C API for built-in file objects, which used to rely on the buffered I/O (`FILE*`) support from the C standard library. In Python 3, files and streams use the new `io` module, which defines several layers over the low-level unbuffered I/O of the operating system. The functions described below are convenience C wrappers over these new APIs, and meant mostly for internal error reporting in the interpreter ; third-party code is advised to access the `io` APIs instead.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a Python file object from the file descriptor of an already opened file *fd*. The arguments *name*, *encoding*, *errors* and *newline* can be `NULL` to use the defaults ; *buffering* can be *-1* to use the default. *name* is ignored and kept for backward compatibility. Return `NULL` on failure. For a more comprehensive description of the arguments, please refer to the `io.open()` function documentation.

> **Avertissement**
>
> Étant donné que les flux de données Python possèdent leur propre couche de tampon, les mélanger avec des descripteurs de fichiers du niveau du système d'exploitation peut produire des erreurs (comme par exemple un ordre des données inattendu).

Modifié dans la version 3.2 : ignore l'attribut *name*

int **PyObject_AsFileDescriptor**(*PyObject* *p)

Part of the Stable ABI. Return the file descriptor associated with *p* as an `int`. If the object is an integer, its value is returned. If not, the object's `fileno()` method is called if it exists ; the method must return an integer, which is returned as the file descriptor value. Sets an exception and returns `-1` on failure.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI. Cette fonction, équivalente à `p.readline([n])`, lit une ligne de l'objet *p*. *p* peut être un objet fichier ou n'importe quel objet qui possède une méthode `readline()`. Si *n* vaut `0`, une seule ligne est lue, indépendamment de la taille de la ligne. Si *n* est plus grand que `0`, un maximum de *n* octets seront lus en provenance du fichier ; une ligne partielle peut être renvoyée. Dans les deux cas, une chaîne de caractères vide est renvoyée si la fin du fichier est atteinte immédiatement. Cependant, si *n* est plus petit que `0`, une ligne est lue indépendamment de sa taille, mais `EOFError` est levée si la fin du fichier est atteinte immédiatement.

int **PyFile_SetOpenCodeHook**(*Py_OpenCodeHookFunction* handler)

Overrides the normal behavior of `io.open_code()` to pass its parameter through the provided handler.

The *handler* is a function of type :

typedef *PyObject* *(***Py_OpenCodeHookFunction**)(*PyObject**, void*)

Equivalent of `PyObject *(*)(PyObject *path, void *userData)`, where *path* is guaranteed to be `PyUnicodeObject`.

The *userData* pointer is passed into the hook function. Since hook functions may be called from different runtimes, this pointer should not refer directly to Python state.

As this hook is intentionally used during import, avoid importing new modules during its execution unless they are known to be frozen or available in `sys.modules`.

Once a hook has been set, it cannot be removed or replaced, and later calls to `PyFile_SetOpenCodeHook()` will fail. On failure, the function returns -1 and sets an exception if the interpreter has been initialized.

This function is safe to call before `Py_Initialize()`.

Raises an auditing event `setopencodehook` with no arguments.

Ajouté dans la version 3.8.

int **PyFile_WriteObject**(*PyObject* *obj, *PyObject* *p, int flags)

Part of the Stable ABI. Write object *obj* to file object *p*. The only supported flag for *flags* is `Py_PRINT_RAW`; if given, the `str()` of the object is written instead of the `repr()`. Return `0` on success or `-1` on failure; the appropriate exception will be set.

int **PyFile_WriteString**(const char *s, *PyObject* *p)

Part of the Stable ABI. Écrit la chaîne de caractères *s* dans l'objet fichier *p*. Retourne `0` en cas de succès ou `-1` en cas d'échec ; l'exception appropriée sera mise en place.

9.6.2 Module Objects

PyTypeObject **PyModule_Type**

Part of the Stable ABI. This instance of `PyTypeObject` represents the Python module type. This is exposed to Python programs as `types.ModuleType`.

int **PyModule_Check**(*PyObject* *p)

Return true if *p* is a module object, or a subtype of a module object. This function always succeeds.

int **PyModule_CheckExact**(*PyObject* *p)

Return true if *p* is a module object, but not a subtype of `PyModule_Type`. This function always succeeds.

PyObject ***PyModule_NewObject**(*PyObject* *name)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.7.* Return a new module object with `module.__name__` set to *name*. The module's `__name__`, `__doc__`, `__package__` and `__loader__` attributes are filled in (all but `__name__` are set to `None`). The caller is responsible for setting a `__file__` attribute.

Return `NULL` with an exception set on error.

Ajouté dans la version 3.3.

Modifié dans la version 3.4 : `__package__` and `__loader__` are now set to `None`.

PyObject ***PyModule_New**(const char *name)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Similar to `PyModule_NewObject()`, but the name is a UTF-8 encoded string instead of a Unicode object.

PyObject ***PyModule_GetDict**(*PyObject* *module)

Valeur de retour : référence empruntée. Part of the Stable ABI. Return the dictionary object that implements *module*'s namespace; this object is the same as the `__dict__` attribute of the module object. If *module* is not a module object (or a subtype of a module object), `SystemError` is raised and `NULL` is returned.

It is recommended extensions use other `PyModule_*` and `PyObject_*` functions rather than directly manipulate a module's `__dict__`.

PyObject ***PyModule_GetNameObject**(*PyObject* *module)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.7.* Return *module*'s `__name__` value. If the module does not provide one, or if it is not a string, `SystemError` is raised and `NULL` is returned.

Ajouté dans la version 3.3.

const char ***PyModule_GetName**(*PyObject* *module)

Part of the Stable ABI. Similar to `PyModule_GetNameObject()` but return the name encoded to `'utf-8'`.

void ***PyModule_GetState**(*PyObject* *module)

Part of the Stable ABI. Return the "state" of the module, that is, a pointer to the block of memory allocated at module creation time, or `NULL`. See `PyModuleDef.m_size`.

PyModuleDef ***PyModule_GetDef**(*PyObject* *module)

Part of the Stable ABI. Return a pointer to the `PyModuleDef` struct from which the module was created, or `NULL` if the module wasn't created from a definition.

PyObject ***PyModule_GetFilenameObject**(*PyObject* *module)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return the name of the file from which *module* was loaded using *module*'s `__file__` attribute. If this is not defined, or if it is not a string, raise `SystemError` and return `NULL`; otherwise return a reference to a Unicode object.

Ajouté dans la version 3.2.

const char ***PyModule_GetFilename**(*PyObject* *module)

Part of the Stable ABI. Similar to *PyModule_GetFilenameObject()* but return the filename encoded to 'utf-8'.

Obsolète depuis la version 3.2 : *PyModule_GetFilename()* raises `UnicodeEncodeError` on unencodable filenames, use *PyModule_GetFilenameObject()* instead.

9.6.3 Module definitions

The functions in the previous section work on any module object, including modules imported from Python code.

Modules defined using the C API typically use a *module definition*, *PyModuleDef* -- a statically allocated, constant "description" of how a module should be created.

The definition is usually used to define an extension's "main" module object (see *Defining extension modules* for details). It is also used to *create extension modules dynamically*.

Unlike *PyModule_New()*, the definition allows management of *module state* -- a piece of memory that is allocated and cleared together with the module object. Unlike the module's Python attributes, Python code cannot replace or delete data stored in module state.

type **PyModuleDef**

Part of the Stable ABI *(including all members).* The module definition struct, which holds all information needed to create a module object. This structure must be statically allocated (or be otherwise guaranteed to be valid while any modules created from it exist). Usually, there is only one variable of this type for each extension module.

PyModuleDef_Base **m_base**

Always initialize this member to `PyModuleDef_HEAD_INIT`.

const char ***m_name**

Name for the new module.

const char ***m_doc**

Docstring for the module; usually a docstring variable created with *PyDoc_STRVAR* is used.

Py_ssize_t **m_size**

Module state may be kept in a per-module memory area that can be retrieved with *PyModule_GetState()*, rather than in static globals. This makes modules safe for use in multiple sub-interpreters.

This memory area is allocated based on *m_size* on module creation, and freed when the module object is deallocated, after the *m_free* function has been called, if present.

Setting it to a non-negative value means that the module can be re-initialized and specifies the additional amount of memory it requires for its state.

Setting `m_size` to `-1` means that the module does not support sub-interpreters, because it has global state. Negative `m_size` is only allowed when using *legacy single-phase initialization* or when *creating modules dynamically*.

See **PEP 3121** for more details.

PyMethodDef ***m_methods**

A pointer to a table of module-level functions, described by *PyMethodDef* values. Can be `NULL` if no functions are present.

PyModuleDef_Slot ***m_slots***

An array of slot definitions for multi-phase initialization, terminated by a `{0, NULL}` entry. When using legacy single-phase initialization, *m_slots* must be `NULL`.

Modifié dans la version 3.5 : Prior to version 3.5, this member was always set to `NULL`, and was defined as :

inquiry **m_reload**

traverseproc **m_traverse**

A traversal function to call during GC traversal of the module object, or `NULL` if not needed.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (*Py_mod_exec* function). More precisely, this function is not called if *m_size* is greater than 0 and the module state (as returned by *PyModule_GetState()*) is `NULL`.

Modifié dans la version 3.9 : No longer called before the module state is allocated.

inquiry **m_clear**

A clear function to call during GC clearing of the module object, or `NULL` if not needed.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (*Py_mod_exec* function). More precisely, this function is not called if *m_size* is greater than 0 and the module state (as returned by *PyModule_GetState()*) is `NULL`.

Like *PyTypeObject.tp_clear*, this function is not *always* called before a module is deallocated. For example, when reference counting is enough to determine that an object is no longer used, the cyclic garbage collector is not involved and *m_free* is called directly.

Modifié dans la version 3.9 : No longer called before the module state is allocated.

freefunc **m_free**

A function to call during deallocation of the module object, or `NULL` if not needed.

This function is not called if the module state was requested but is not allocated yet. This is the case immediately after the module is created and before the module is executed (*Py_mod_exec* function). More precisely, this function is not called if *m_size* is greater than 0 and the module state (as returned by *PyModule_GetState()*) is `NULL`.

Modifié dans la version 3.9 : No longer called before the module state is allocated.

Module slots

type **PyModuleDef_Slot**

int **slot**

A slot ID, chosen from the available values explained below.

void ***value**

Value of the slot, whose meaning depends on the slot ID.

Ajouté dans la version 3.5.

The available slot types are :

Py_mod_create

Specifies a function that is called to create the module object itself. The *value* pointer of this slot must point to a function of the signature :

PyObject ***create_module**(*PyObject* *spec, *PyModuleDef* *def)

The function receives a `ModuleSpec` instance, as defined in **PEP 451**, and the module definition. It should return a new module object, or set an error and return `NULL`.

This function should be kept minimal. In particular, it should not call arbitrary Python code, as trying to import the same module again may result in an infinite loop.

Multiple `Py_mod_create` slots may not be specified in one module definition.

If `Py_mod_create` is not specified, the import machinery will create a normal module object using *PyModule_New()*. The name is taken from *spec*, not the definition, to allow extension modules to dynamically adjust to their place in the module hierarchy and be imported under different names through symlinks, all while sharing a single module definition.

There is no requirement for the returned object to be an instance of *PyModule_Type*. Any type can be used, as long as it supports setting and getting import-related attributes. However, only `PyModule_Type` instances may be returned if the `PyModuleDef` has non-`NULL` `m_traverse`, `m_clear`, `m_free`; non-zero `m_size`; or slots other than `Py_mod_create`.

Py_mod_exec

Specifies a function that is called to *execute* the module. This is equivalent to executing the code of a Python module : typically, this function adds classes and constants to the module. The signature of the function is :

int **exec_module**(*PyObject* *module)

If multiple `Py_mod_exec` slots are specified, they are processed in the order they appear in the *m_slots* array.

Py_mod_multiple_interpreters

Specifies one of the following values :

Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED

The module does not support being imported in subinterpreters.

Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED

The module supports being imported in subinterpreters, but only when they share the main interpreter's GIL. (See isolating-extensions-howto.)

Py_MOD_PER_INTERPRETER_GIL_SUPPORTED

The module supports being imported in subinterpreters, even when they have their own GIL. (See isolating-extensions-howto.)

This slot determines whether or not importing this module in a subinterpreter will fail.

Multiple `Py_mod_multiple_interpreters` slots may not be specified in one module definition.

If `Py_mod_multiple_interpreters` is not specified, the import machinery defaults to `Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED`.

Ajouté dans la version 3.12.

Py_mod_gil

Specifies one of the following values :

Py_MOD_GIL_USED

The module depends on the presence of the global interpreter lock (GIL), and may access global state without synchronization.

Py_MOD_GIL_NOT_USED

The module is safe to run without an active GIL.

This slot is ignored by Python builds not configured with `--disable-gil`. Otherwise, it determines whether or not importing this module will cause the GIL to be automatically enabled. See whatsnew313-free-threaded-cpython for more detail.

Multiple `Py_mod_gil` slots may not be specified in one module definition.

If `Py_mod_gil` is not specified, the import machinery defaults to `Py_MOD_GIL_USED`.

Ajouté dans la version 3.13.

9.6.4 Creating extension modules dynamically

The following functions may be used to create a module outside of an extension's *initialization function*. They are also used in *single-phase initialization*.

PyObject ***PyModule_Create**(*PyModuleDef* *def)

Valeur de retour : nouvelle référence. Create a new module object, given the definition in *def*. This is a macro that calls `PyModule_Create2()` with *module_api_version* set to `PYTHON_API_VERSION`, or to `PYTHON_ABI_VERSION` if using the *limited API*.

PyObject ***PyModule_Create2**(*PyModuleDef* *def, int module_api_version)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a new module object, given the definition in *def*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

Return `NULL` with an exception set on error.

This function does not support slots. The `m_slots` member of *def* must be `NULL`.

> **Note**
>
> Most uses of this function should be using `PyModule_Create()` instead; only use this if you are sure you need it.

PyObject ***PyModule_FromDefAndSpec**(*PyModuleDef* *def, *PyObject* *spec)

Valeur de retour : nouvelle référence. This macro calls `PyModule_FromDefAndSpec2()` with *module_api_version* set to `PYTHON_API_VERSION`, or to `PYTHON_ABI_VERSION` if using the *limited API*.

Ajouté dans la version 3.5.

PyObject ***PyModule_FromDefAndSpec2**(*PyModuleDef* *def, *PyObject* *spec, int module_api_version)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.7.* Create a new module object, given the definition in *def* and the ModuleSpec *spec*, assuming the API version *module_api_version*. If that version does not match the version of the running interpreter, a `RuntimeWarning` is emitted.

Return `NULL` with an exception set on error.

Note that this does not process execution slots (`Py_mod_exec`). Both `PyModule_FromDefAndSpec` and `PyModule_ExecDef` must be called to fully initialize a module.

> **Note**
>
> Most uses of this function should be using `PyModule_FromDefAndSpec()` instead; only use this if you are sure you need it.

Ajouté dans la version 3.5.

int **PyModule_ExecDef**(*PyObject* *module, *PyModuleDef* *def)

Part of the Stable ABI *Depuis la version 3.7.* Process any execution slots (`Py_mod_exec`) given in *def*.

Ajouté dans la version 3.5.

PYTHON_API_VERSION

The C API version. Defined for backwards compatibility.

Currently, this constant is not updated in new Python versions, and is not useful for versioning. This may change in the future.

PYTHON_ABI_VERSION

Defined as `3` for backwards compatibility.

Currently, this constant is not updated in new Python versions, and is not useful for versioning. This may change in the future.

9.6.5 Support functions

The following functions are provided to help initialize a module state. They are intended for a module's execution slots (*Py_mod_exec*), the initialization function for legacy *single-phase initialization*, or code that creates modules dynamically.

int **PyModule_AddObjectRef**(*PyObject* *module, const char *name, *PyObject* *value)

Part of the Stable ABI *Depuis la version 3.10.* Add an object to *module* as *name*. This is a convenience function which can be used from the module's initialization function.

On success, return `0`. On error, raise an exception and return `-1`.

Exemple d'utilisation :

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    if (obj == NULL) {
        return -1;
    }
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_DECREF(obj);
    return res;
}
```

To be convenient, the function accepts `NULL` *value* with an exception set. In this case, return `-1` and just leave the raised exception unchanged.

The example can also be written without checking explicitly if *obj* is `NULL` :

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    int res = PyModule_AddObjectRef(module, "spam", obj);
    Py_XDECREF(obj);
    return res;
}
```

Note that `Py_XDECREF()` should be used instead of `Py_DECREF()` in this case, since *obj* can be `NULL`.

The number of different *name* strings passed to this function should be kept small, usually by only using statically allocated strings as *name*. For names that aren't known at compile time, prefer calling *PyUnicode_FromString()* and *PyObject_SetAttr()* directly. For more details, see *PyUnicode_InternFromString()*, which may be used internally to create a key object.

Ajouté dans la version 3.10.

int **PyModule_Add**(*PyObject* *module, const char *name, *PyObject* *value)

Part of the Stable ABI *Depuis la version 3.13.* Similar to *PyModule_AddObjectRef()*, but "steals" a reference to *value*. It can be called with a result of function that returns a new reference without bothering to check its result or even saving it to a variable.

Exemple d'utilisation :

```
if (PyModule_Add(module, "spam", PyBytes_FromString(value)) < 0) {
    goto error;
}
```

Ajouté dans la version 3.13.

int **PyModule_AddObject**(*PyObject* *module, const char *name, *PyObject* *value)

Part of the Stable ABI. Similar to `PyModule_AddObjectRef()`, but steals a reference to *value* on success (if it returns `0`).

The new `PyModule_Add()` or `PyModule_AddObjectRef()` functions are recommended, since it is easy to introduce reference leaks by misusing the `PyModule_AddObject()` function.

Note

Unlike other functions that steal references, `PyModule_AddObject()` only releases the reference to *value* **on success**.

This means that its return value must be checked, and calling code must `Py_XDECREF()` *value* manually on error.

Exemple d'utilisation :

```
PyObject *obj = PyBytes_FromString(value);
if (PyModule_AddObject(module, "spam", obj) < 0) {
    // If 'obj' is not NULL and PyModule_AddObject() failed,
    // 'obj' strong reference must be deleted with Py_XDECREF().
    // If 'obj' is NULL, Py_XDECREF() does nothing.
    Py_XDECREF(obj);
    goto error;
}
// PyModule_AddObject() stole a reference to obj:
// Py_XDECREF(obj) is not needed here.
```

Obsolète depuis la version 3.13 : `PyModule_AddObject()` is *soft deprecated*.

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

Part of the Stable ABI. Add an integer constant to *module* as *name*. This convenience function can be used from the module's initialization function. Return `-1` with an exception set on error, `0` on success.

This is a convenience function that calls `PyLong_FromLong()` and `PyModule_AddObjectRef()`; see their documentation for details.

int **PyModule_AddStringConstant**(*PyObject* *module, const char *name, const char *value)

Part of the Stable ABI. Add a string constant to *module* as *name*. This convenience function can be used from the module's initialization function. The string *value* must be `NULL`-terminated. Return `-1` with an exception set on error, `0` on success.

This is a convenience function that calls `PyUnicode_InternFromString()` and `PyModule_AddObjectRef()`; see their documentation for details.

PyModule_AddIntMacro(module, macro)

Add an int constant to *module*. The name and the value are taken from *macro*. For example `PyModule_AddIntMacro(module, AF_INET)` adds the int constant *AF_INET* with the value of *AF_INET* to *module*. Return `-1` with an exception set on error, `0` on success.

PyModule_AddStringMacro(module, macro)

Add a string constant to *module*.

int **PyModule_AddType**(*PyObject* *module, *PyTypeObject* *type)

Part of the Stable ABI *Depuis la version 3.10.* Add a type object to *module*. The type object is finalized by calling internally `PyType_Ready()`. The name of the type object is taken from the last component of `tp_name` after dot. Return `-1` with an exception set on error, `0` on success.

Ajouté dans la version 3.9.

int **PyModule_AddFunctions**(*PyObject* *module, *PyMethodDef* *functions)

Part of the Stable ABI *Depuis la version 3.7.* Add the functions from the `NULL` terminated *functions* array to *module*. Refer to the `PyMethodDef` documentation for details on individual entries (due to the lack of a shared module namespace, module level "functions" implemented in C typically receive the module as their first parameter, making them similar to instance methods on Python classes).

This function is called automatically when creating a module from `PyModuleDef` (such as when using *Multi-phase initialization*, `PyModule_Create`, or `PyModule_FromDefAndSpec`). Some module authors may prefer defining functions in multiple `PyMethodDef` arrays; in that case they should call this function directly.

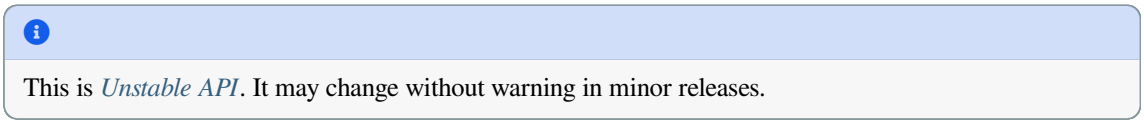
Ajouté dans la version 3.5.

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

Part of the Stable ABI *Depuis la version 3.7.* Set the docstring for *module* to *docstring*. This function is called automatically when creating a module from `PyModuleDef` (such as when using *Multi-phase initialization*, `PyModule_Create`, or `PyModule_FromDefAndSpec`).

Ajouté dans la version 3.5.

int **PyUnstable_Module_SetGIL**(*PyObject* *module, void *gil)



This is *Unstable API*. It may change without warning in minor releases.

Indicate that *module* does or does not support running without the global interpreter lock (GIL), using one of the values from `Py_mod_gil`. It must be called during *module*'s initialization function when using *Legacy single-phase initialization*. If this function is not called during module initialization, the import machinery assumes the module does not support running without the GIL. This function is only available in Python builds configured with `--disable-gil`. Return `-1` with an exception set on error, `0` on success.

Ajouté dans la version 3.13.

Module lookup (single-phase initialization)

The legacy *single-phase initialization* initialization scheme creates singleton modules that can be looked up in the context of the current interpreter. This allows the module object to be retrieved later with only a reference to the module definition.

These functions will not work on modules created using multi-phase initialization, since multiple such modules can be created from a single definition.

PyObject ***PyState_FindModule**(*PyModuleDef* *def)

Valeur de retour : référence empruntée. Part of the Stable ABI. Returns the module object that was created from *def* for the current interpreter. This method requires that the module object has been attached to the interpreter state with `PyState_AddModule()` beforehand. In case the corresponding module object is not found or has not been attached to the interpreter state yet, it returns `NULL`.

int **PyState_AddModule**(*PyObject* *module, *PyModuleDef* *def)

Part of the Stable ABI *Depuis la version 3.3.* Attaches the module object passed to the function to the interpreter state. This allows the module object to be accessible via `PyState_FindModule()`.

Only effective on modules created using single-phase initialization.

Python calls `PyState_AddModule` automatically after importing a module that uses *single-phase initialization*, so it is unnecessary (but harmless) to call it from module initialization code. An explicit call is needed only if the module's own init code subsequently calls `PyState_FindModule`. The function is mainly intended for implementing alternative import mechanisms (either by calling it directly, or by referring to its implementation for details of the required state updates).

If a module was attached previously using the same *def*, it is replaced by the new *module*.

The caller must have an *attached thread state*.

Return `-1` with an exception set on error, `0` on success.

Ajouté dans la version 3.3.

int **PyState_RemoveModule**(*PyModuleDef* *def)

Part of the Stable ABI *Depuis la version 3.3.* Removes the module object created from *def* from the interpreter state. Return `-1` with an exception set on error, `0` on success.

The caller must have an *attached thread state*.

Ajouté dans la version 3.3.

9.6.6 Itérateurs

Python provides two general-purpose iterator objects. The first, a sequence iterator, works with an arbitrary sequence supporting the `__getitem__()` method. The second works with a callable object and a sentinel value, calling the callable for each item in the sequence, and ending the iteration when the sentinel value is returned.

PyTypeObject **PySeqIter_Type**

Part of the Stable ABI. Type des itérateurs renvoyés par les fonctions `PySeqIter_New()` et la forme à un argument de la fonction native `iter()` pour les séquences natives.

int **PySeqIter_Check**(*PyObject* *op)

Return true if the type of *op* is `PySeqIter_Type`. This function always succeeds.

PyObject ***PySeqIter_New**(*PyObject* *seq)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Renvoie un itérateur sur la séquence *seq*. L'itération prend fin lorsque la séquence lève `IndexError` lors d'une tentative d'accès.

PyTypeObject **PyCallIter_Type**

Part of the Stable ABI. Type de l'itérateur renvoyé par les fonctions `PyCallIter_New()` et `iter()` à deux arguments.

int **PyCallIter_Check**(*PyObject* *op)

Return true if the type of *op* is `PyCallIter_Type`. This function always succeeds.

PyObject ***PyCallIter_New**(*PyObject* *callable, *PyObject* *sentinel)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Renvoie un nouvel itérateur. Le premier paramètre, *callable*, peut être n'importe quel objet Python appelable sans aucun paramètre ; chaque appel doit renvoyer l'élément suivant de l'itération. Lorsque *callable* renvoie une valeur égale à *sentinel*, l'itération prend fin.

9.6.7 Les descripteurs

Les "Descripteurs" sont des objets décrivant des attributs pour un objet. Ils se trouvent dans le dictionnaire du type de l'objet.

PyTypeObject **PyProperty_Type**

Part of the Stable ABI. L'objet *type* des descripteurs natifs.

PyObject ***PyDescr_NewGetSet**(*PyTypeObject* *type, struct *PyGetSetDef* *getset)

Valeur de retour : nouvelle référence. Part of the Stable ABI.

PyObject ***PyDescr_NewMember**(*PyTypeObject* *type, struct *PyMemberDef* *meth)

Valeur de retour : nouvelle référence. Part of the Stable ABI.

PyObject ***PyDescr_NewMethod**(*PyTypeObject* *type, struct *PyMethodDef* *meth)

Valeur de retour : nouvelle référence. Part of the Stable ABI.

PyObject ***PyDescr_NewWrapper**(*PyTypeObject* *type, struct wrapperbase *wrapper, void *wrapped)

Valeur de retour : nouvelle référence.

PyObject ***PyDescr_NewClassMethod**(*PyTypeObject* *type, *PyMethodDef* *method)

Valeur de retour : nouvelle référence. Part of the Stable ABI.

int **PyDescr_IsData**(*PyObject* *descr)

Return non-zero if the descriptor objects *descr* describes a data attribute, or `0` if it describes a method. *descr* must be a descriptor object; there is no error checking.

PyObject ***PyWrapper_New**(*PyObject**, *PyObject**)

Valeur de retour : nouvelle référence. Part of the Stable ABI.

9.6.8 Slice Objects

PyTypeObject **PySlice_Type**

Part of the Stable ABI. The type object for slice objects. This is the same as `slice` in the Python layer.

int **PySlice_Check**(*PyObject* *ob)

Return true if *ob* is a slice object; *ob* must not be `NULL`. This function always succeeds.

PyObject ***PySlice_New**(*PyObject* *start, *PyObject* *stop, *PyObject* *step)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a new slice object with the given values. The *start*, *stop*, and *step* parameters are used as the values of the slice object attributes of the same names. Any of the values may be `NULL`, in which case the `None` will be used for the corresponding attribute.

Return `NULL` with an exception set if the new object could not be allocated.

int **PySlice_GetIndices**(*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step)

Part of the Stable ABI. Retrieve the start, stop and step indices from the slice object *slice*, assuming a sequence of length *length*. Treats indices greater than *length* as errors.

Returns `0` on success and `-1` on error with no exception set (unless one of the indices was not `None` and failed to be converted to an integer, in which case `-1` is returned with an exception set).

You probably do not want to use this function.

Modifié dans la version 3.2 : The parameter type for the *slice* parameter was `PySliceObject*` before.

int **PySlice_GetIndicesEx**(*PyObject* *slice, *Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* *step, *Py_ssize_t* *slicelength)

Part of the Stable ABI. Usable replacement for *`PySlice_GetIndices()`*. Retrieve the start, stop, and step indices from the slice object *slice* assuming a sequence of length *length*, and store the length of the slice in *slicelength*. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Return `0` on success and `-1` on error with an exception set.

Note

This function is considered not safe for resizable sequences. Its invocation should be replaced by a combination of *`PySlice_Unpack()`* and *`PySlice_AdjustIndices()`* where

```
if (PySlice_GetIndicesEx(slice, length, &start, &stop, &step, &slicelength)
↪< 0) {
    // return error
}
```

is replaced by

```
if (PySlice_Unpack(slice, &start, &stop, &step) < 0) {
    // return error
}
slicelength = PySlice_AdjustIndices(length, &start, &stop, step);
```

Modifié dans la version 3.2 : The parameter type for the *slice* parameter was `PySliceObject*` before.

Modifié dans la version 3.6.1 : If `Py_LIMITED_API` is not set or set to the value between `0x03050400` and `0x03060000` (not including) or `0x03060100` or higher `PySlice_GetIndicesEx()` is implemented as a macro using `PySlice_Unpack()` and `PySlice_AdjustIndices()`. Arguments *start*, *stop* and *step* are evaluated more than once.

Obsolète depuis la version 3.6.1 : If `Py_LIMITED_API` is set to the value less than `0x03050400` or between `0x03060000` and `0x03060100` (not including) `PySlice_GetIndicesEx()` is a deprecated function.

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

Part of the Stable ABI *Depuis la version 3.7.* Extract the start, stop and step data members from a slice object as C integers. Silently reduce values larger than `PY_SSIZE_T_MAX` to `PY_SSIZE_T_MAX`, silently boost the start and stop values less than `PY_SSIZE_T_MIN` to `PY_SSIZE_T_MIN`, and silently boost the step values less than `-PY_SSIZE_T_MAX` to `-PY_SSIZE_T_MAX`.

Return `-1` with an exception set on error, `0` on success.

Ajouté dans la version 3.6.1.

Py_ssize_t **PySlice_AdjustIndices**(*Py_ssize_t* length, *Py_ssize_t* *start, *Py_ssize_t* *stop, *Py_ssize_t* step)

Part of the Stable ABI *Depuis la version 3.7.* Adjust start/end slice indices assuming a sequence of the specified length. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Return the length of the slice. Always successful. Doesn't call Python code.

Ajouté dans la version 3.6.1.

Ellipsis Object

PyTypeObject **PyEllipsis_Type**

Part of the Stable ABI. The type of Python `Ellipsis` object. Same as `types.EllipsisType` in the Python layer.

PyObject ***Py_Ellipsis**

The Python `Ellipsis` object. This object has no methods. Like *Py_None*, it is an *immortal* singleton object.

Modifié dans la version 3.12 : *Py_Ellipsis* is immortal.

9.6.9 Objets de type MemoryView

Un objet Python `memoryview` expose le *protocole tampon* du C. Cet objet peut ensuite être passé comme n'importe quel objet.

PyObject ***PyMemoryView_FromObject**(*PyObject* *obj)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Crée un objet *memoryview* à partir d'un objet implémentant le protocole tampon. Si *obj* permet d'exporter des tampons modifiables, l'objet *memoryview* crée acceptera la lecture et écriture, sinon l'objet crée est soit en lecture seule ou lecture/écriture, à la discrétion de l'*exporteur*.

PyBUF_READ

Flag to request a readonly buffer.

PyBUF_WRITE

Flag to request a writable buffer.

PyObject ***PyMemoryView_FromMemory**(char *mem, *Py_ssize_t* size, int flags)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.7.* Crée un objet *memoryview* utilisant *mem* comme un tampon sous-jacent. *flags* peut être *PyBUF_READ* ou *PyBUF_WRITE*.

Ajouté dans la version 3.3.

PyObject ***PyMemoryView_FromBuffer**(const *Py_buffer* *view)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.11.* Crée un objet *memoryview* à partir de la structure tampon *view*. Pour de simples tampons d'octets, *PyMemoryView_FromMemory()* est préférée.

PyObject ***PyMemoryView_GetContiguous**(*PyObject* *obj, int buffertype, char order)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Crée un objet *memoryview* vers un segment de mémoire *contiguous* (organisé comme en `'C'` ou comme en `'F'` pour Fortran) à partir d'un objet qui expose le protocole tampon. Si la mémoire est contiguë, l'objet *memoryview* pointe vers la mémoire d'origine. Sinon une copie est faite et la *memoryview* pointe vers un nouvel objet *bytes*.

buffertype can be one of `PyBUF_READ` or `PyBUF_WRITE`.

int **PyMemoryView_Check**(*PyObject* *obj)

Return true if the object *obj* is a memoryview object. It is not currently allowed to create subclasses of `memoryview`. This function always succeeds.

Py_buffer ***PyMemoryView_GET_BUFFER**(*PyObject* *mview)

Retourne un pointeur vers la copie privée du tampon de l'*exporteur* de *memoryview*. *mview* **doit** être une instance de *memoryview* ; cette macro ne vérifie pas le type, vous devez le faire vous-même sinon vous pourriez subir un crash.

PyObject ***PyMemoryView_GET_BASE**(*PyObject* *mview)

Renvoie soit un pointeur vers l'objet exporté sur lequel est basé la *memoryview* ou `NULL` si la *memoryview* a été crée par `PyMemoryView_FromMemory()` ou `PyMemoryView_FromBuffer()`. *mview* **doit** être une instance de *memoryview*.

9.6.10 Objets à références faibles

Python gère les *références faibles* comme des objets de première classe. Il existe deux types d'objets spécifiques qui implémentent directement les références faibles. Le premier est un objet de référence simple, et le second agit autant que possible comme un mandataire vers l'objet original.

int **PyWeakref_Check**(*PyObject* *ob)

Return non-zero if *ob* is either a reference or proxy object. This function always succeeds.

int **PyWeakref_CheckRef**(*PyObject* *ob)

Return non-zero if *ob* is a reference object. This function always succeeds.

int **PyWeakref_CheckProxy**(*PyObject* *ob)

Return non-zero if *ob* is a proxy object. This function always succeeds.

PyObject ***PyWeakref_NewRef**(*PyObject* *ob, *PyObject* *callback)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a weak reference object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object ; an existing reference object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected ; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be `None` or `NULL`. If *ob* is not a weakly referenceable object, or if *callback* is not callable, `None`, or `NULL`, this will return `NULL` and raise `TypeError`.

PyObject ***PyWeakref_NewProxy**(*PyObject* *ob, *PyObject* *callback)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Return a weak reference proxy object for the object *ob*. This will always return a new reference, but is not guaranteed to create a new object ; an existing proxy object may be returned. The second parameter, *callback*, can be a callable object that receives notification when *ob* is garbage collected ; it should accept a single parameter, which will be the weak reference object itself. *callback* may also be `None` or `NULL`. If *ob* is not a weakly referenceable object, or if *callback* is not callable, `None`, or `NULL`, this will return `NULL` and raise `TypeError`.

int **PyWeakref_GetRef**(*PyObject* *ref, *PyObject* **pobj)

Part of the Stable ABI *Depuis la version 3.13.* Get a *strong reference* to the referenced object from a weak reference, *ref*, into **pobj*.

— On success, set **pobj* to a new *strong reference* to the referenced object and return 1.
— If the reference is dead, set **pobj* to `NULL` and return 0.
— On error, raise an exception and return -1.

Ajouté dans la version 3.13.

PyObject ***PyWeakref_GetObject**(*PyObject* *ref)

Valeur de retour : référence empruntée. Part of the Stable ABI. Return a *borrowed reference* to the referenced object from a weak reference, *ref*. If the referent is no longer live, returns `Py_None`.

> **Note**
>
> This function returns a *borrowed reference* to the referenced object. This means that you should always call `Py_INCREF()` on the object except when it cannot be destroyed before the last usage of the borrowed reference.

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

PyObject ***PyWeakref_GET_OBJECT**(*PyObject* *ref)

Valeur de retour : référence empruntée. Similar to `PyWeakref_GetObject()`, but does no error checking.

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

int **PyWeakref_IsDead**(*PyObject* *ref)

Test if the weak reference *ref* is dead. Returns 1 if the reference is dead, 0 if it is alive, and -1 with an error set if *ref* is not a weak reference object.

Ajouté dans la version 3.14.

void **PyObject_ClearWeakRefs**(*PyObject* *object)

Part of the Stable ABI. This function is called by the `tp_dealloc` handler to clear weak references.

This iterates through the weak references for *object* and calls callbacks for those references which have one. It returns when all callbacks have been attempted.

void **PyUnstable_Object_ClearWeakRefsNoCallbacks**(*PyObject* *object)

> This is *Unstable API*. It may change without warning in minor releases.

Clears the weakrefs for *object* without calling the callbacks.

This function is called by the `tp_dealloc` handler for types with finalizers (i.e., `__del__()`). The handler for those objects first calls `PyObject_ClearWeakRefs()` to clear weakrefs and call their callbacks, then the finalizer, and finally this function to clear any weakrefs that may have been created by the finalizer.

In most circumstances, it's more appropriate to use `PyObject_ClearWeakRefs()` to clear weakrefs instead of this function.

Ajouté dans la version 3.13.

9.6.11 Capsules

Reportez-vous à using-capsules pour plus d'informations sur l'utilisation de ces objets.

Ajouté dans la version 3.1.

type **PyCapsule**

This subtype of `PyObject` represents an opaque value, useful for C extension modules who need to pass an opaque value (as a `void*` pointer) through Python code to other C code. It is often used to make a C function pointer defined in one module available to other modules, so the regular import mechanism can be used to access C APIs defined in dynamically loaded modules.

type **PyCapsule_Destructor**

Part of the Stable ABI. The type of a destructor callback for a capsule. Defined as :

```
typedef void (*PyCapsule_Destructor)(PyObject *);
```

See *PyCapsule_New()* for the semantics of PyCapsule_Destructor callbacks.

int **PyCapsule_CheckExact**(*PyObject* *p)

Return true if its argument is a *PyCapsule*. This function always succeeds.

PyObject ***PyCapsule_New**(void *pointer, const char *name, *PyCapsule_Destructor* destructor)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Create a *PyCapsule* encapsulating the *pointer*. The *pointer* argument may not be `NULL`.

On failure, set an exception and return `NULL`.

The *name* string may either be `NULL` or a pointer to a valid C string. If non-`NULL`, this string must outlive the capsule. (Though it is permitted to free it inside the *destructor*.)

If the *destructor* argument is not `NULL`, it will be called with the capsule as its argument when it is destroyed.

If this capsule will be stored as an attribute of a module, the *name* should be specified as `modulename.attributename`. This will enable other modules to import the capsule using *PyCapsule_Import()*.

void ***PyCapsule_GetPointer**(*PyObject* *capsule, const char *name)

Part of the Stable ABI. Retrieve the *pointer* stored in the capsule. On failure, set an exception and return `NULL`.

The *name* parameter must compare exactly to the name stored in the capsule. If the name stored in the capsule is `NULL`, the *name* passed in must also be `NULL`. Python uses the C function `strcmp()` to compare capsule names.

PyCapsule_Destructor **PyCapsule_GetDestructor**(*PyObject* *capsule)

Part of the Stable ABI. Return the current destructor stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` destructor. This makes a `NULL` return code somewhat ambiguous; use *PyCapsule_IsValid()* or *PyErr_Occurred()* to disambiguate.

void ***PyCapsule_GetContext**(*PyObject* *capsule)

Part of the Stable ABI. Return the current context stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` context. This makes a `NULL` return code somewhat ambiguous; use *PyCapsule_IsValid()* or *PyErr_Occurred()* to disambiguate.

const char ***PyCapsule_GetName**(*PyObject* *capsule)

Part of the Stable ABI. Return the current name stored in the capsule. On failure, set an exception and return `NULL`.

It is legal for a capsule to have a `NULL` name. This makes a `NULL` return code somewhat ambiguous; use *PyCapsule_IsValid()* or *PyErr_Occurred()* to disambiguate.

void ***PyCapsule_Import**(const char *name, int no_block)

Part of the Stable ABI. Import a pointer to a C object from a capsule attribute in a module. The *name* parameter should specify the full name to the attribute, as in `module.attribute`. The *name* stored in the capsule must match this string exactly.

This function splits *name* on the `.` character, and imports the first element. It then processes further elements using attribute lookups.

Return the capsule's internal *pointer* on success. On failure, set an exception and return `NULL`.

Note

If *name* points to an attribute of some submodule or subpackage, this submodule or subpackage must be previously imported using other means (for example, by using *PyImport_ImportModule()*) for the attribute lookups to succeed.

Modifié dans la version 3.3 : *no_block* has no effect anymore.

int **PyCapsule_IsValid**(*PyObject* *capsule, const char *name)

Part of the Stable ABI. Determines whether or not *capsule* is a valid capsule. A valid capsule is non-NULL, passes *PyCapsule_CheckExact()*, has a non-NULL pointer stored in it, and its internal name matches the *name* parameter. (See *PyCapsule_GetPointer()* for information on how capsule names are compared.)

In other words, if *PyCapsule_IsValid()* returns a true value, calls to any of the accessors (any function starting with PyCapsule_Get) are guaranteed to succeed.

Return a nonzero value if the object is valid and matches the name passed in. Return 0 otherwise. This function will not fail.

int **PyCapsule_SetContext**(*PyObject* *capsule, void *context)

Part of the Stable ABI. Set the context pointer inside *capsule* to *context*.

Return 0 on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetDestructor**(*PyObject* *capsule, *PyCapsule_Destructor* destructor)

Part of the Stable ABI. Set the destructor inside *capsule* to *destructor*.

Return 0 on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetName**(*PyObject* *capsule, const char *name)

Part of the Stable ABI. Set the name inside *capsule* to *name*. If non-NULL, the name must outlive the capsule. If the previous *name* stored in the capsule was not NULL, no attempt is made to free it.

Return 0 on success. Return nonzero and set an exception on failure.

int **PyCapsule_SetPointer**(*PyObject* *capsule, void *pointer)

Part of the Stable ABI. Set the void pointer inside *capsule* to *pointer*. The pointer may not be NULL.

Return 0 on success. Return nonzero and set an exception on failure.

9.6.12 Objets décrivant les *frames*

type **PyFrameObject**

Part of the Limited API *(as an opaque struct).* Structure des objets C utilisée pour décrire les objets *frame*.

Il n'y a pas de membres publics dans cette structure.

Modifié dans la version 3.11 : Les membres de la structure API publique ont été retirés de l'API publique de C. Visiter l'entrée dans la page des nouveautés de Python pour plus détails.

Les fonctions PyEval_GetFrame() et *PyThreadState_GetFrame()* peuvent être utilisées pour avoir un objet de *frame*.

Voir aussi *Réflexion*

PyTypeObject **PyFrame_Type**

Le type des objets de *frame*. C'est le même objet que types.FrameType dans la couche de Python.

Modifié dans la version 3.11 : Précédemment, ce type était disponible uniquement après l'inclusion de <frameobject.h>.

int **PyFrame_Check**(*PyObject* *obj)

Renvoie un entier non nul si *obj* est un objet *frame*.

Modifié dans la version 3.11 : Précédemment, ce type était disponible uniquement après l'inclusion de <frameobject.h>.

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

Valeur de retour : nouvelle référence. Renvoie la *frame* encadrant immédiatement à *frame*.

Renvoie un *strong reference*, ou NULL si *frame* n'a pas de *frame* encadrante.

Ajouté dans la version 3.9.

PyObject ***PyFrame_GetBuiltins**(*PyFrameObject* *frame)

Valeur de retour : nouvelle référence. Get the *frame*'s `f_builtins` attribute.

Renvoie un *strong reference*. Le résultat ne peut être `NULL`.

Ajouté dans la version 3.11.

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

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.10.* Obtenir le code *frame*.

Renvoie un *strong reference*.

Le résultat (code *frame*) ne peut être `NULL`.

Ajouté dans la version 3.9.

PyObject ***PyFrame_GetGenerator**(*PyFrameObject* *frame)

Valeur de retour : nouvelle référence. Obtenir le générateur, coroutine, ou un générateur asynchrone qui maintient cette *frame*, ou `NULL` si cette *frame* n'est pas maintenu par un générateur. Il ne lève pas d'exception, même si la valeur renvoyée est `NULL`.

Renvoie un *strong reference*, ou `NULL`.

Ajouté dans la version 3.11.

PyObject ***PyFrame_GetGlobals**(*PyFrameObject* *frame)

Valeur de retour : nouvelle référence. Get the *frame*'s `f_globals` attribute.

Renvoie un *strong reference*. Le résultat ne peut être `NULL`.

Ajouté dans la version 3.11.

int **PyFrame_GetLasti**(*PyFrameObject* *frame)

Get the *frame*'s `f_lasti` attribute.

Renvoie -1 si `frame.f_lasti` est `None`.

Ajouté dans la version 3.11.

PyObject ***PyFrame_GetVar**(*PyFrameObject* *frame, *PyObject* *name)

Valeur de retour : nouvelle référence. Get the variable *name* of *frame*.
— Return a *strong reference* to the variable value on success.
— Raise `NameError` and return `NULL` if the variable does not exist.
— Raise an exception and return `NULL` on error.
name type must be a `str`.

Ajouté dans la version 3.12.

PyObject ***PyFrame_GetVarString**(*PyFrameObject* *frame, const char *name)

Valeur de retour : nouvelle référence. Similar to *`PyFrame_GetVar()`*, but the variable name is a C string encoded in UTF-8.

Ajouté dans la version 3.12.

PyObject ***PyFrame_GetLocals**(*PyFrameObject* *frame)

Valeur de retour : nouvelle référence. Get the *frame*'s `f_locals` attribute. If the frame refers to an *optimized scope*, this returns a write-through proxy object that allows modifying the locals. In all other cases (classes, modules, `exec()`, `eval()`) it returns the mapping representing the frame locals directly (as described for `locals()`).

Renvoie un *strong reference*.

Ajouté dans la version 3.11.

Modifié dans la version 3.13 : As part of **PEP 667**, return an instance of *`PyFrameLocalsProxy_Type`*.

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

Part of the Stable ABI *Depuis la version 3.10.* Renvoie le numéro de ligne que cette *frame* est en train d'exécuter.

Frame Locals Proxies

Ajouté dans la version 3.13.

The `f_locals` attribute on a frame object is an instance of a "frame-locals proxy". The proxy object exposes a write-through view of the underlying locals dictionary for the frame. This ensures that the variables exposed by `f_locals` are always up to date with the live local variables in the frame itself.

See **PEP 667** for more information.

PyTypeObject **PyFrameLocalsProxy_Type**

The type of frame `locals()` proxy objects.

int **PyFrameLocalsProxy_Check**(*PyObject* *obj)

Return non-zero if *obj* is a frame `locals()` proxy.

Internal Frames

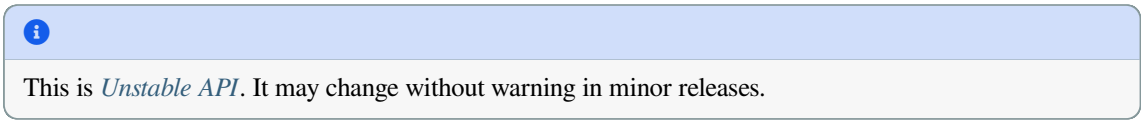
Unless using **PEP 523**, you will not need this.

struct **_PyInterpreterFrame**

The interpreter's internal frame representation.

Ajouté dans la version 3.11.

PyObject ***PyUnstable_InterpreterFrame_GetCode**(struct *_PyInterpreterFrame* *frame);

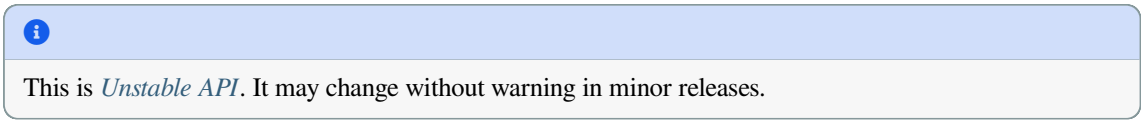


This is *Unstable API*. It may change without warning in minor releases.

Return a *strong reference* to the code object for the frame.

Ajouté dans la version 3.12.

int **PyUnstable_InterpreterFrame_GetLasti**(struct *_PyInterpreterFrame* *frame);

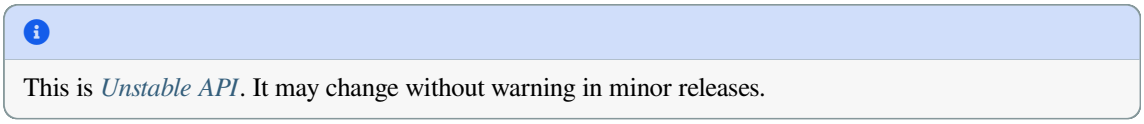


This is *Unstable API*. It may change without warning in minor releases.

Return the byte offset into the last executed instruction.

Ajouté dans la version 3.12.

int **PyUnstable_InterpreterFrame_GetLine**(struct *_PyInterpreterFrame* *frame);



This is *Unstable API*. It may change without warning in minor releases.

Return the currently executing line number, or -1 if there is no line number.

Ajouté dans la version 3.12.

9.6.13 Objets générateur

Python utilise des objets générateurs pour implémenter les itérations de générateurs. Ils sont normalement crées en itérant sur une fonction donnant des valeurs via `yield`, au lieu d'appeler explicitement *PyGen_New()* ou *PyGen_NewWithQualName()*.

type **PyGenObject**

La structure C utilisée pour les objets générateurs.

PyTypeObject **PyGen_Type**

Le type objet correspondant aux objets générateurs.

int **PyGen_Check**(*PyObject* *ob)

Return true if *ob* is a generator object; *ob* must not be `NULL`. This function always succeeds.

int **PyGen_CheckExact**(*PyObject* *ob)

Return true if *ob*'s type is *PyGen_Type*; *ob* must not be `NULL`. This function always succeeds.

PyObject ***PyGen_New**(*PyFrameObject* *frame)

Valeur de retour : nouvelle référence. Create and return a new generator object based on the *frame* object. A reference to *frame* is stolen by this function. The argument must not be `NULL`.

PyObject ***PyGen_NewWithQualName**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

Valeur de retour : nouvelle référence. Create and return a new generator object based on the *frame* object, with `__name__` and `__qualname__` set to *name* and *qualname*. A reference to *frame* is stolen by this function. The *frame* argument must not be `NULL`.

9.6.14 Objets coroutines

Ajouté dans la version 3.5.

Les objets coroutines sont les objets renvoyés par les fonctions déclarées avec le mot clef `async`.

type **PyCoroObject**

La structure C utilisée pour les objets coroutine.

PyTypeObject **PyCoro_Type**

L'objet type correspondant aux objets coroutines.

int **PyCoro_CheckExact**(*PyObject* *ob)

Return true if *ob*'s type is *PyCoro_Type*; *ob* must not be `NULL`. This function always succeeds.

PyObject ***PyCoro_New**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

Valeur de retour : nouvelle référence. Create and return a new coroutine object based on the *frame* object, with `__name__` and `__qualname__` set to *name* and *qualname*. A reference to *frame* is stolen by this function. The *frame* argument must not be `NULL`.

9.6.15 Context Variables Objects

Ajouté dans la version 3.7.

Modifié dans la version 3.7.1 :

Note

In Python 3.7.1 the signatures of all context variables C APIs were **changed** to use *PyObject* pointers instead of *PyContext*, *PyContextVar*, and *PyContextToken*, e.g. :

```
// in 3.7.0:
PyContext *PyContext_New(void);

// in 3.7.1+:
PyObject *PyContext_New(void);
```

See bpo-34762 for more details.

This section details the public C API for the `contextvars` module.

type **PyContext**

The C structure used to represent a `contextvars.Context` object.

type **PyContextVar**

The C structure used to represent a `contextvars.ContextVar` object.

type **PyContextToken**

The C structure used to represent a `contextvars.Token` object.

PyTypeObject **PyContext_Type**

The type object representing the *context* type.

PyTypeObject **PyContextVar_Type**

The type object representing the *context variable* type.

PyTypeObject **PyContextToken_Type**

The type object representing the *context variable token* type.

Macros pour vérifier les types :

int **PyContext_CheckExact**(*PyObject* *o)

Return true if *o* is of type *PyContext_Type*. *o* must not be `NULL`. This function always succeeds.

int **PyContextVar_CheckExact**(*PyObject* *o)

Return true if *o* is of type *PyContextVar_Type*. *o* must not be `NULL`. This function always succeeds.

int **PyContextToken_CheckExact**(*PyObject* *o)

Return true if *o* is of type *PyContextToken_Type*. *o* must not be `NULL`. This function always succeeds.

Context object management functions :

PyObject ***PyContext_New**(void)

Valeur de retour : nouvelle référence. Create a new empty context object. Returns `NULL` if an error has occurred.

PyObject ***PyContext_Copy**(*PyObject* *ctx)

Valeur de retour : nouvelle référence. Create a shallow copy of the passed *ctx* context object. Returns `NULL` if an error has occurred.

PyObject ***PyContext_CopyCurrent**(void)

Valeur de retour : nouvelle référence. Create a shallow copy of the current thread context. Returns `NULL` if an error has occurred.

int **PyContext_Enter**(*PyObject* *ctx)

Set *ctx* as the current context for the current thread. Returns `0` on success, and `-1` on error.

int **PyContext_Exit**(*PyObject* *ctx)

Deactivate the *ctx* context and restore the previous context as the current context for the current thread. Returns `0` on success, and `-1` on error.

int **PyContext_AddWatcher**(*PyContext_WatchCallback* callback)

Register *callback* as a context object watcher for the current interpreter. Return an ID which may be passed to *PyContext_ClearWatcher()*. In case of error (e.g. no more watcher IDs available), return `-1` and set an exception.

Ajouté dans la version 3.14.

int **PyContext_ClearWatcher**(int watcher_id)

Clear watcher identified by *watcher_id* previously returned from *PyContext_AddWatcher()* for the current interpreter. Return `0` on success, or `-1` and set an exception on error (e.g. if the given *watcher_id* was never registered.)

Ajouté dans la version 3.14.

type **PyContextEvent**

Enumeration of possible context object watcher events :
— `Py_CONTEXT_SWITCHED` : The *current context* has switched to a different context. The object passed to the watch callback is the now-current `contextvars.Context` object, or None if no context is current.
Ajouté dans la version 3.14.

typedef int (***PyContext_WatchCallback**)(*PyContextEvent* event, *PyObject* *obj)

Context object watcher callback function. The object passed to the callback is event-specific ; see *PyContextEvent* for details.

If the callback returns with an exception set, it must return `-1` ; this exception will be printed as an unraisable exception using *PyErr_FormatUnraisable()*. Otherwise it should return `0`.

There may already be a pending exception set on entry to the callback. In this case, the callback should return `0` with the same exception still set. This means the callback may not call any other API that can set an exception unless it saves and clears the exception state first, and restores it before returning.

Ajouté dans la version 3.14.

Context variable functions :

PyObject ***PyContextVar_New**(const char *name, *PyObject* *def)

Valeur de retour : nouvelle référence. Create a new `ContextVar` object. The *name* parameter is used for introspection and debug purposes. The *def* parameter specifies a default value for the context variable, or `NULL` for no default. If an error has occurred, this function returns `NULL`.

int **PyContextVar_Get**(*PyObject* *var, *PyObject* *default_value, *PyObject* **value)

Get the value of a context variable. Returns `-1` if an error has occurred during lookup, and `0` if no error occurred, whether or not a value was found.

If the context variable was found, *value* will be a pointer to it. If the context variable was *not* found, *value* will point to :
— *default_value*, if not `NULL` ;
— the default value of *var*, if not `NULL` ;
— `NULL`
Except for `NULL`, the function returns a new reference.

PyObject ***PyContextVar_Set**(*PyObject* *var, *PyObject* *value)

Valeur de retour : nouvelle référence. Set the value of *var* to *value* in the current context. Returns a new token object for this change, or `NULL` if an error has occurred.

int **PyContextVar_Reset**(*PyObject* *var, *PyObject* *token)

Reset the state of the *var* context variable to that it was in before *PyContextVar_Set()* that returned the *token* was called. This function returns `0` on success and `-1` on error.

9.6.16 Objets DateTime

Various date and time objects are supplied by the `datetime` module. Before using any of these functions, the header file `datetime.h` must be included in your source (note that this is not included by `Python.h`), and the macro `PyDateTime_IMPORT` must be invoked, usually as part of the module initialisation function. The macro puts a pointer to a C structure into a static variable, `PyDateTimeAPI`, that is used by the following macros.

type **PyDateTime_Date**

This subtype of *PyObject* represents a Python date object.

type **PyDateTime_DateTime**

This subtype of *PyObject* represents a Python datetime object.

type **PyDateTime_Time**

This subtype of *PyObject* represents a Python time object.

type **PyDateTime_Delta**

This subtype of *PyObject* represents the difference between two datetime values.

PyTypeObject **PyDateTime_DateType**

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

PyTypeObject **PyDateTime_DateTimeType**

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

PyTypeObject **PyDateTime_TimeType**

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

PyTypeObject **PyDateTime_DeltaType**

This instance of *PyTypeObject* represents Python type for the difference between two datetime values ; it is the same object as `datetime.timedelta` in the Python layer.

PyTypeObject **PyDateTime_TZInfoType**

This instance of *PyTypeObject* represents the Python time zone info type ; it is the same object as `datetime.tzinfo` in the Python layer.

Macro pour accéder au singleton UTC :

PyObject ***PyDateTime_TimeZone_UTC**

Renvoie le singleton du fuseau horaire UTC, qui est le même objet que `datetime.timezone.utc`.

Ajouté dans la version 3.7.

Macros pour vérifier les types :

int **PyDate_Check**(*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_DateType* or a subtype of `PyDateTime_DateType`. *ob* must not be `NULL`. This function always succeeds.

int **PyDate_CheckExact**(*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_DateType*. *ob* must not be `NULL`. This function always succeeds.

int **PyDateTime_Check**(*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_DateTimeType* or a subtype of `PyDateTime_DateTimeType`. *ob* must not be `NULL`. This function always succeeds.

int **PyDateTime_CheckExact**(*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_DateTimeType*. *ob* must not be `NULL`. This function always succeeds.

int **PyTime_Check**(*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_TimeType* or a subtype of `PyDateTime_TimeType`. *ob* must not be `NULL`. This function always succeeds.

int **PyTime_CheckExact**(*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_TimeType*. *ob* must not be `NULL`. This function always succeeds.

int **PyDelta_Check**(*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_DeltaType* or a subtype of `PyDateTime_DeltaType`. *ob* must not be `NULL`. This function always succeeds.

int **PyDelta_CheckExact**(*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_DeltaType*. *ob* must not be `NULL`. This function always succeeds.

int **PyTZInfo_Check**(*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_TZInfoType* or a subtype of `PyDateTime_TZInfoType`. *ob* must not be `NULL`. This function always succeeds.

int **PyTZInfo_CheckExact**(*PyObject* *ob)

Return true if *ob* is of type *PyDateTime_TZInfoType*. *ob* must not be `NULL`. This function always succeeds.

Macros pour créer des objets :

PyObject ***PyDate_FromDate**(int year, int month, int day)

Valeur de retour : nouvelle référence. Renvoie un objet `datetime.date` avec l'année, le mois et le jour spécifiés.

PyObject ***PyDateTime_FromDateAndTime**(int year, int month, int day, int hour, int minute, int second, int usecond)

Valeur de retour : nouvelle référence. Renvoie un objet `datetime.datetime` avec l'année, le mois, le jour, l'heure, la minute, la seconde et la microseconde spécifiés.

PyObject ***PyDateTime_FromDateAndTimeAndFold**(int year, int month, int day, int hour, int minute, int second, int usecond, int fold)

Valeur de retour : nouvelle référence. Renvoie un objet `datetime.datetime` avec l'année, le mois, le jour, l'heure, la minute, la seconde, la microseconde et le pli (*fold* en anglais) spécifiés.

Ajouté dans la version 3.6.

PyObject ***PyTime_FromTime**(int hour, int minute, int second, int usecond)

Valeur de retour : nouvelle référence. Renvoie un objet `datetime.time` avec l'heure, la minute, la seconde et la microseconde spécifiées.

PyObject ***PyTime_FromTimeAndFold**(int hour, int minute, int second, int usecond, int fold)

Valeur de retour : nouvelle référence. Renvoie un objet `datetime.time` avec l'heure, la minute, la seconde, la microseconde et le pli (*fold* en anglais) spécifiés.

Ajouté dans la version 3.6.

PyObject ***PyDelta_FromDSU**(int days, int seconds, int useconds)

Valeur de retour : nouvelle référence. Renvoie un objet `datetime.timedelta` représentant le nombre passé en paramètre de jours, de secondes et de microsecondes. Le résultat est normalisé pour que le nombre de microsecondes et de secondes tombe dans la plage documentée pour les objets `datetime.timedelta`.

PyObject ***PyTimeZone_FromOffset**(*PyObject* *offset)

Valeur de retour : nouvelle référence. Renvoie un objet `datetime.timezone` avec un décalage anonyme fixe représenté par l'argument *offset*.

Ajouté dans la version 3.7.

PyObject ***PyTimeZone_FromOffsetAndName**(*PyObject* *offset, *PyObject* *name)

Valeur de retour : nouvelle référence. Renvoie un objet `datetime.timezone` avec un décalage fixe représenté par l'argument *offset* et avec le nom de fuseau horaire *name*.

Ajouté dans la version 3.7.

Macros to extract fields from date objects. The argument must be an instance of *PyDateTime_Date*, including subclasses (such as *PyDateTime_DateTime*). The argument must not be `NULL`, and the type is not checked :

int **PyDateTime_GET_YEAR**(*PyDateTime_Date* *o)

Renvoie l'année, sous forme d'entier positif.

int **PyDateTime_GET_MONTH**(*PyDateTime_Date* *o)

Renvoie le mois, sous forme d'entier allant de 1 à 12.

int **PyDateTime_GET_DAY**(*PyDateTime_Date* *o)

Renvoie le jour, sous forme d'entier allant de 1 à 31.

Macros to extract fields from datetime objects. The argument must be an instance of *PyDateTime_DateTime*, including subclasses. The argument must not be `NULL`, and the type is not checked :

int **PyDateTime_DATE_GET_HOUR**(*PyDateTime_DateTime* *o)

Renvoie l'heure, sous forme d'entier allant de 0 à 23.

int **PyDateTime_DATE_GET_MINUTE**(*PyDateTime_DateTime* *o)

Renvoie la minute, sous forme d'entier allant de 0 à 59.

int **PyDateTime_DATE_GET_SECOND**(*PyDateTime_DateTime* *o)

Renvoie la seconde, sous forme d'entier allant de 0 à 59.

int **PyDateTime_DATE_GET_MICROSECOND**(*PyDateTime_DateTime* *o)

Renvoie la microseconde, sous forme d'entier allant de 0 à 999999.

int **PyDateTime_DATE_GET_FOLD**(*PyDateTime_DateTime* *o)

Return the fold, as an int from 0 through 1.

Ajouté dans la version 3.6.

PyObject ***PyDateTime_DATE_GET_TZINFO**(*PyDateTime_DateTime* *o)

Return the tzinfo (which may be `None`).

Ajouté dans la version 3.10.

Macros to extract fields from time objects. The argument must be an instance of *`PyDateTime_Time`*, including subclasses. The argument must not be `NULL`, and the type is not checked :

int **PyDateTime_TIME_GET_HOUR**(*PyDateTime_Time* *o)

Renvoie l'heure, sous forme d'entier allant de 0 à 23.

int **PyDateTime_TIME_GET_MINUTE**(*PyDateTime_Time* *o)

Renvoie la minute, sous forme d'entier allant de 0 à 59.

int **PyDateTime_TIME_GET_SECOND**(*PyDateTime_Time* *o)

Renvoie la seconde, sous forme d'entier allant de 0 à 59.

int **PyDateTime_TIME_GET_MICROSECOND**(*PyDateTime_Time* *o)

Renvoie la microseconde, sous forme d'entier allant de 0 à 999999.

int **PyDateTime_TIME_GET_FOLD**(*PyDateTime_Time* *o)

Return the fold, as an int from 0 through 1.

Ajouté dans la version 3.6.

PyObject ***PyDateTime_TIME_GET_TZINFO**(*PyDateTime_Time* *o)

Return the tzinfo (which may be `None`).

Ajouté dans la version 3.10.

Macros to extract fields from time delta objects. The argument must be an instance of *`PyDateTime_Delta`*, including subclasses. The argument must not be `NULL`, and the type is not checked :

int **PyDateTime_DELTA_GET_DAYS**(*PyDateTime_Delta* *o)

Renvoie le nombre de jours, sous forme d'entier allant de `-999999999` à `999999999`.

Ajouté dans la version 3.3.

int **PyDateTime_DELTA_GET_SECONDS**(*PyDateTime_Delta* *o)

Renvoie le nombre de secondes sous forme d'entier allant de 0 à 86399.

Ajouté dans la version 3.3.

int **PyDateTime_DELTA_GET_MICROSECONDS**(*PyDateTime_Delta* *o)

Renvoie le nombre de microsecondes, sous forme d'entier allant de 0 à 999999.

Ajouté dans la version 3.3.

Macros de confort pour les modules implémentant l'API DB :

PyObject ***PyDateTime_FromTimestamp**(*PyObject* *args)

Valeur de retour : nouvelle référence. Create and return a new `datetime.datetime` object given an argument tuple suitable for passing to `datetime.datetime.fromtimestamp()`.

PyObject ***PyDate_FromTimestamp**(*PyObject* *args)

Valeur de retour : nouvelle référence. Create and return a new `datetime.date` object given an argument tuple suitable for passing to `datetime.date.fromtimestamp()`.

9.6.17 Objects for Type Hinting

Various built-in types for type hinting are provided. Currently, two types exist -- GenericAlias and Union. Only `GenericAlias` is exposed to C.

PyObject ***Py_GenericAlias**(*PyObject* *origin, *PyObject* *args)

Part of the Stable ABI *Depuis la version 3.9.* Create a GenericAlias object. Equivalent to calling the Python class `types.GenericAlias`. The *origin* and *args* arguments set the `GenericAlias`'s `__origin__` and `__args__` attributes respectively. *origin* should be a *`PyTypeObject`*`*`, and *args* can be a *`PyTupleObject`*`*` or any `PyObject*`. If *args* passed is not a tuple, a 1-tuple is automatically constructed and `__args__` is set to `(args,)`. Minimal checking is done for the arguments, so the function will succeed even if *origin* is not a type. The `GenericAlias`'s `__parameters__` attribute is constructed lazily from `__args__`. On failure, an exception is raised and `NULL` is returned.

Here's an example of how to make an extension type generic :

```
...
static PyMethodDef my_obj_methods[] = {
    // Other methods.
    ...
    {"__class_getitem__", Py_GenericAlias, METH_O|METH_CLASS, "See PEP 585"}
    ...
}
```

Voir aussi

The data model method `__class_getitem__()`.

Ajouté dans la version 3.9.

PyTypeObject **Py_GenericAliasType**

Part of the Stable ABI *Depuis la version 3.9.* The C type of the object returned by *`Py_GenericAlias()`*. Equivalent to `types.GenericAlias` in Python.

Ajouté dans la version 3.9.

CHAPITRE 10

Initialization, Finalization, and Threads

See *Python Initialization Configuration* for details on how to configure the interpreter prior to initialization.

10.1 Before Python Initialization

In an application embedding Python, the `Py_Initialize()` function must be called before using any other Python/C API functions ; with the exception of a few functions and the *global configuration variables*.

The following functions can be safely called before Python is initialized :

- Functions that initialize the interpreter :
  - `Py_Initialize()`
  - `Py_InitializeEx()`
  - `Py_InitializeFromConfig()`
  - `Py_BytesMain()`
  - `Py_Main()`
  - the runtime pre-initialization functions covered in *Python Initialization Configuration*
- Configuration functions :
  - `PyImport_AppendInittab()`
  - `PyImport_ExtendInittab()`
  - `PyInitFrozenExtensions()`
  - `PyMem_SetAllocator()`
  - `PyMem_SetupDebugHooks()`
  - `PyObject_SetArenaAllocator()`
  - `Py_SetProgramName()`
  - `Py_SetPythonHome()`
  - `PySys_ResetWarnOptions()`
  - the configuration functions covered in *Python Initialization Configuration*
- Informative functions :
  - `Py_IsInitialized()`
  - `PyMem_GetAllocator()`
  - `PyObject_GetArenaAllocator()`
  - `Py_GetBuildInfo()`
  - `Py_GetCompiler()`
  - `Py_GetCopyright()`
  - `Py_GetPlatform()`
  - `Py_GetVersion()`
  - `Py_IsInitialized()`

- Utilities :
  - *Py_DecodeLocale()*
  - the status reporting and utility functions covered in *Python Initialization Configuration*
- Memory allocators :
  - *PyMem_RawMalloc()*
  - *PyMem_RawRealloc()*
  - *PyMem_RawCalloc()*
  - *PyMem_RawFree()*
- Synchronization :
  - *PyMutex_Lock()*
  - *PyMutex_Unlock()*

Note

Despite their apparent similarity to some of the functions listed above, the following functions **should not be called** before the interpreter has been initialized : *Py_EncodeLocale()*, *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()*, *Py_GetProgramFullPath()*, *Py_GetPythonHome()*, *Py_GetProgramName()*, *PyEval_InitThreads()*, and *Py_RunMain()*.

10.2 Global configuration variables

Python has variables for the global configuration to control different features and options. By default, these flags are controlled by command line options.

When a flag is set by an option, the value of the flag is the number of times that the option was set. For example, `-b` sets *Py_BytesWarningFlag* to 1 and `-bb` sets *Py_BytesWarningFlag* to 2.

int **Py_BytesWarningFlag**

This API is kept for backward compatibility : setting *PyConfig.bytes_warning* should be used instead, see *Python Initialization Configuration*.

Issue a warning when comparing `bytes` or `bytearray` with `str` or `bytes` with `int`. Issue an error if greater or equal to `2`.

Set by the `-b` option.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_DebugFlag**

This API is kept for backward compatibility : setting *PyConfig.parser_debug* should be used instead, see *Python Initialization Configuration*.

Turn on parser debugging output (for expert only, depending on compilation options).

Set by the `-d` option and the `PYTHONDEBUG` environment variable.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_DontWriteBytecodeFlag**

This API is kept for backward compatibility : setting *PyConfig.write_bytecode* should be used instead, see *Python Initialization Configuration*.

If set to non-zero, Python won't try to write `.pyc` files on the import of source modules.

Set by the `-B` option and the `PYTHONDONTWRITEBYTECODE` environment variable.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_FrozenFlag**

This API is kept for backward compatibility : setting *PyConfig.pathconfig_warnings* should be used instead, see *Python Initialization Configuration*.

Suppress error messages when calculating the module search path in *Py_GetPath()*.

Private flag used by `_freeze_module` and `frozenmain` programs.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_HashRandomizationFlag**

This API is kept for backward compatibility : setting *PyConfig.hash_seed* and *PyConfig.use_hash_seed* should be used instead, see *Python Initialization Configuration*.

Set to `1` if the `PYTHONHASHSEED` environment variable is set to a non-empty string.

If the flag is non-zero, read the `PYTHONHASHSEED` environment variable to initialize the secret hash seed.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_IgnoreEnvironmentFlag**

This API is kept for backward compatibility : setting *PyConfig.use_environment* should be used instead, see *Python Initialization Configuration*.

Ignore all `PYTHON*` environment variables, e.g. `PYTHONPATH` and `PYTHONHOME`, that might be set.

Set by the `-E` and `-I` options.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_InspectFlag**

This API is kept for backward compatibility : setting *PyConfig.inspect* should be used instead, see *Python Initialization Configuration*.

When a script is passed as first argument or the `-c` option is used, enter interactive mode after executing the script or the command, even when `sys.stdin` does not appear to be a terminal.

Set by the `-i` option and the `PYTHONINSPECT` environment variable.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_InteractiveFlag**

This API is kept for backward compatibility : setting *PyConfig.interactive* should be used instead, see *Python Initialization Configuration*.

Set by the `-i` option.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_IsolatedFlag**

This API is kept for backward compatibility : setting *PyConfig.isolated* should be used instead, see *Python Initialization Configuration*.

Run Python in isolated mode. In isolated mode `sys.path` contains neither the script's directory nor the user's site-packages directory.

Set by the `-I` option.

Ajouté dans la version 3.4.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_LegacyWindowsFSEncodingFlag**

This API is kept for backward compatibility : setting *PyPreConfig.legacy_windows_fs_encoding* should be used instead, see *Python Initialization Configuration*.

If the flag is non-zero, use the `mbcs` encoding with `replace` error handler, instead of the UTF-8 encoding with `surrogatepass` error handler, for the *filesystem encoding and error handler*.

Set to `1` if the `PYTHONLEGACYWINDOWSFSENCODING` environment variable is set to a non-empty string.

Voir la **PEP 529** pour plus d'informations.

Availability : Windows.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_LegacyWindowsStdioFlag**

This API is kept for backward compatibility : setting *PyConfig.legacy_windows_stdio* should be used instead, see *Python Initialization Configuration*.

If the flag is non-zero, use `io.FileIO` instead of `io._WindowsConsoleIO` for `sys` standard streams.

Set to `1` if the `PYTHONLEGACYWINDOWSSTDIO` environment variable is set to a non-empty string.

See **PEP 528** for more details.

Availability : Windows.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_NoSiteFlag**

This API is kept for backward compatibility : setting *PyConfig.site_import* should be used instead, see *Python Initialization Configuration*.

Désactive l'importation du module `site` et les modifications locales de `sys.path` qu'il implique. Désactive aussi ces manipulations si `site` est importé explicitement plus tard (appelez `site.main()` si vous voulez les déclencher).

Set by the `-S` option.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_NoUserSiteDirectory**

This API is kept for backward compatibility : setting *PyConfig.user_site_directory* should be used instead, see *Python Initialization Configuration*.

N'ajoute pas le répertoire utilisateur `site-packages` à `sys.path`.

Set by the `-s` and `-I` options, and the `PYTHONNOUSERSITE` environment variable.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_OptimizeFlag**

This API is kept for backward compatibility : setting *PyConfig.optimization_level* should be used instead, see *Python Initialization Configuration*.

Set by the `-O` option and the `PYTHONOPTIMIZE` environment variable.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_QuietFlag**

This API is kept for backward compatibility : setting *PyConfig.quiet* should be used instead, see *Python Initialization Configuration*.

N'affiche pas le copyright et la version, même en mode interactif.

Set by the `-q` option.

Ajouté dans la version 3.2.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_UnbufferedStdioFlag**

This API is kept for backward compatibility : setting *PyConfig.buffered_stdio* should be used instead, see *Python Initialization Configuration*.

Force the stdout and stderr streams to be unbuffered.

Set by the `-u` option and the `PYTHONUNBUFFERED` environment variable.

Deprecated since version 3.12, will be removed in version 3.15.

int **Py_VerboseFlag**

This API is kept for backward compatibility : setting *PyConfig.verbose* should be used instead, see *Python Initialization Configuration*.

Print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded. If greater or equal to `2`, print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Set by the `-v` option and the `PYTHONVERBOSE` environment variable.

Deprecated since version 3.12, will be removed in version 3.15.

10.3 Initializing and finalizing the interpreter

void **Py_Initialize**()

Part of the Stable ABI. Initialize the Python interpreter. In an application embedding Python, this should be called before using any other Python/C API functions; see *Before Python Initialization* for the few exceptions.

This initializes the table of loaded modules (`sys.modules`), and creates the fundamental modules `builtins`, `__main__` and `sys`. It also initializes the module search path (`sys.path`). It does not set `sys.argv`; use the *Python Initialization Configuration* API for that. This is a no-op when called for a second time (without calling `Py_FinalizeEx()` first). There is no return value; it is a fatal error if the initialization fails.

Use `Py_InitializeFromConfig()` to customize the *Python Initialization Configuration*.

> **Note**
>
> On Windows, changes the console mode from `O_TEXT` to `O_BINARY`, which will also affect non-Python uses of the console using the C Runtime.

void **Py_InitializeEx**(int initsigs)

Part of the Stable ABI. This function works like `Py_Initialize()` if *initsigs* is `1`. If *initsigs* is `0`, it skips initialization registration of signal handlers, which may be useful when CPython is embedded as part of a larger application.

Use `Py_InitializeFromConfig()` to customize the *Python Initialization Configuration*.

PyStatus **Py_InitializeFromConfig**(const *PyConfig* *config)

Initialize Python from *config* configuration, as described in *Initialization with PyConfig*.

See the *Python Initialization Configuration* section for details on pre-initializing the interpreter, populating the runtime configuration structure, and querying the returned status structure.

int **Py_IsInitialized**()

Part of the Stable ABI. Return true (nonzero) when the Python interpreter has been initialized, false (zero) if not. After `Py_FinalizeEx()` is called, this returns false until `Py_Initialize()` is called again.

int **Py_IsFinalizing**()

Part of the Stable ABI *Depuis la version 3.13.* Return true (non-zero) if the main Python interpreter is *shutting down*. Return false (zero) otherwise.

Ajouté dans la version 3.13.

int **Py_FinalizeEx**()

Part of the Stable ABI *Depuis la version 3.6.* Undo all initializations made by `Py_Initialize()` and subsequent use of Python/C API functions, and destroy all sub-interpreters (see `Py_NewInterpreter()` below) that were created and not yet destroyed since the last call to `Py_Initialize()`. This is a no-op when called for a second time (without calling `Py_Initialize()` again first).

Since this is the reverse of `Py_Initialize()`, it should be called in the same thread with the same interpreter active. That means the main thread and the main interpreter. This should never be called while `Py_RunMain()` is running.

Normally the return value is `0`. If there were errors during finalization (flushing buffered data), `-1` is returned.

Note that Python will do a best effort at freeing all memory allocated by the Python interpreter. Therefore, any C-Extension should make sure to correctly clean up all of the preveiously allocated PyObjects before using them in subsequent calls to *Py_Initialize()*. Otherwise it could introduce vulnerabilities and incorrect behavior.

This function is provided for a number of reasons. An embedding application might want to restart Python without having to restart the application itself. An application that has loaded the Python interpreter from a dynamically loadable library (or DLL) might want to free all memory allocated by Python before unloading the DLL. During a hunt for memory leaks in an application a developer might want to free all memory allocated by Python before exiting from the application.

Bugs and caveats : The destruction of modules and objects in modules is done in random order ; this may cause destructors (`__del__()` methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Interned strings will all be deallocated regardless of their reference count. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once ; this can happen if an application calls *Py_Initialize()* and *Py_FinalizeEx()* more than once. *Py_FinalizeEx()* must not be called recursively from within itself. Therefore, it must not be called by any code that may be run as part of the interpreter shutdown process, such as `atexit` handlers, object finalizers, or any code that may be run while flushing the stdout and stderr files.

Raises an auditing event `cpython._PySys_ClearAuditHooks` with no arguments.

Ajouté dans la version 3.6.

void **Py_Finalize**()

Part of the Stable ABI. This is a backwards-compatible version of *Py_FinalizeEx()* that disregards the return value.

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

Part of the Stable ABI *Depuis la version 3.8.* Similar to *Py_Main()* but *argv* is an array of bytes strings, allowing the calling application to delegate the text decoding step to the CPython runtime.

Ajouté dans la version 3.8.

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

Part of the Stable ABI. The main program for the standard interpreter, encapsulating a full initialization/finalization cycle, as well as additional behaviour to implement reading configurations settings from the environment and command line, and then executing `__main__` in accordance with using-on-cmdline.

This is made available for programs which wish to support the full CPython command line interface, rather than just embedding a Python runtime in a larger application.

The *argc* and *argv* parameters are similar to those which are passed to a C program's `main()` function, except that the *argv* entries are first converted to `wchar_t` using *Py_DecodeLocale()*. It is also important to note that the argument list entries may be modified to point to strings other than those passed in (however, the contents of the strings pointed to by the argument list are not modified).

The return value is `2` if the argument list does not represent a valid Python command line, and otherwise the same as *Py_RunMain()*.

In terms of the CPython runtime configuration APIs documented in the *runtime configuration* section (and without accounting for error handling), `Py_Main` is approximately equivalent to :

```
PyConfig config;
PyConfig_InitPythonConfig(&config);
PyConfig_SetArgv(&config, argc, argv);
Py_InitializeFromConfig(&config);
PyConfig_Clear(&config);

Py_RunMain();
```

In normal usage, an embedding application will call this function *instead* of calling `Py_Initialize()`, `Py_InitializeEx()` or `Py_InitializeFromConfig()` directly, and all settings will be applied as described elsewhere in this documentation. If this function is instead called *after* a preceding runtime initialization API call, then exactly which environmental and command line configuration settings will be updated is version dependent (as it depends on which settings correctly support being modified after they have already been set once when the runtime was first initialized).

int **Py_RunMain**(void)

Executes the main module in a fully configured CPython runtime.

Executes the command (`PyConfig.run_command`), the script (`PyConfig.run_filename`) or the module (`PyConfig.run_module`) specified on the command line or in the configuration. If none of these values are set, runs the interactive Python prompt (REPL) using the `__main__` module's global namespace.

If `PyConfig.inspect` is not set (the default), the return value will be `0` if the interpreter exits normally (that is, without raising an exception), the exit status of an unhandled `SystemExit`, or `1` for any other unhandled exception.

If `PyConfig.inspect` is set (such as when the `-i` option is used), rather than returning when the interpreter exits, execution will instead resume in an interactive Python prompt (REPL) using the `__main__` module's global namespace. If the interpreter exited with an exception, it is immediately raised in the REPL session. The function return value is then determined by the way the *REPL session* terminates : `0`, `1`, or the status of a `SystemExit`, as specified above.

This function always finalizes the Python interpreter before it returns.

See *Python Configuration* for an example of a customized Python that always runs in isolated mode using `Py_RunMain()`.

int **PyUnstable_AtExit**(*PyInterpreterState* *interp, void (*func)(void*), void *data)

This is *Unstable API*. It may change without warning in minor releases.

Register an `atexit` callback for the target interpreter *interp*. This is similar to `Py_AtExit()`, but takes an explicit interpreter and data pointer for the callback.

There must be an *attached thread state* for *interp*.

Ajouté dans la version 3.13.

10.4 Process-wide parameters

void **Py_SetProgramName**(const wchar_t *name)

Part of the Stable ABI. This API is kept for backward compatibility : setting `PyConfig.program_name` should be used instead, see *Python Initialization Configuration*.

This function should be called before `Py_Initialize()` is called for the first time, if it is called at all. It tells the interpreter the value of the `argv[0]` argument to the `main()` function of the program (converted to wide characters). This is used by `Py_GetPath()` and some other functions below to find the Python run-time libraries relative to the interpreter executable. The default value is `'python'`. The argument should point to a zero-terminated wide character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_t*` string.

Deprecated since version 3.11, will be removed in version 3.15.

wchar_t ***Py_GetProgramName**()

Part of the Stable ABI. Return the program name set with `PyConfig.program_name`, or the default. The returned string points into static storage ; the caller should not modify its value.

This function should not be called before *Py_Initialize()*, otherwise it returns NULL.

Modifié dans la version 3.10 : It now returns NULL if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15 : Use *PyConfig_Get("executable")* (sys.executable) instead.

wchar_t ***Py_GetPrefix**()

Part of the Stable ABI. Return the *prefix* for installed platform-independent files. This is derived through a number of complicated rules from the program name set with *PyConfig.program_name* and some environment variables ; for example, if the program name is '/usr/local/bin/python', the prefix is '/usr/local'. The returned string points into static storage ; the caller should not modify its value. This corresponds to the **prefix** variable in the top-level Makefile and the --prefix argument to the **configure** script at build time. The value is available to Python code as sys.base_prefix. It is only useful on Unix. See also the next function.

This function should not be called before *Py_Initialize()*, otherwise it returns NULL.

Modifié dans la version 3.10 : It now returns NULL if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15 : Use *PyConfig_Get("base_prefix")* (sys.base_prefix) instead. Use *PyConfig_Get("prefix")* (sys.prefix) if virtual environments need to be handled.

wchar_t ***Py_GetExecPrefix**()

Part of the Stable ABI. Return the *exec-prefix* for installed platform-*dependent* files. This is derived through a number of complicated rules from the program name set with *PyConfig.program_name* and some environment variables ; for example, if the program name is '/usr/local/bin/python', the exec-prefix is '/usr/local'. The returned string points into static storage ; the caller should not modify its value. This corresponds to the **exec_prefix** variable in the top-level Makefile and the --exec-prefix argument to the **configure** script at build time. The value is available to Python code as sys.base_exec_prefix. It is only useful on Unix.

Background : The exec-prefix differs from the prefix when platform dependent files (such as executables and shared libraries) are installed in a different directory tree. In a typical installation, platform dependent files may be installed in the /usr/local/plat subtree while platform independent may be installed in /usr/local.

Generally speaking, a platform is a combination of hardware and software families, e.g. Sparc machines running the Solaris 2.x operating system are considered the same platform, but Intel machines running Solaris 2.x are another platform, and Intel machines running Linux are yet another platform. Different major revisions of the same operating system generally also form different platforms. Non-Unix operating systems are a different story ; the installation strategies on those systems are so different that the prefix and exec-prefix are meaningless, and set to the empty string. Note that compiled Python bytecode files are platform independent (but not independent from the Python version by which they were compiled !).

System administrators will know how to configure the **mount** or **automount** programs to share /usr/local between platforms while having /usr/local/plat be a different filesystem for each platform.

This function should not be called before *Py_Initialize()*, otherwise it returns NULL.

Modifié dans la version 3.10 : It now returns NULL if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15 : Use *PyConfig_Get("base_exec_prefix")* (sys.base_exec_prefix) instead. Use *PyConfig_Get("exec_prefix")* (sys.exec_prefix) if virtual environments need to be handled.

wchar_t ***Py_GetProgramFullPath**()

Part of the Stable ABI. Return the full program name of the Python executable ; this is computed as a side-effect of deriving the default module search path from the program name (set by *PyConfig.program_name*). The returned string points into static storage ; the caller should not modify its value. The value is available to Python code as sys.executable.

This function should not be called before *Py_Initialize()*, otherwise it returns NULL.

Modifié dans la version 3.10 : It now returns NULL if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15 : Use *PyConfig_Get("executable")* (`sys.executable`) instead.

wchar_t ***Py_GetPath**()

Part of the Stable ABI. Return the default module search path ; this is computed from the program name (set by *PyConfig.program_name*) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is `':'` on Unix and macOS, `';'` on Windows. The returned string points into static storage ; the caller should not modify its value. The list `sys.path` is initialized with this value on interpreter startup ; it can be (and usually is) modified later to change the search path for loading modules.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

Modifié dans la version 3.10 : It now returns `NULL` if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15 : Use *PyConfig_Get("module_search_paths")* (`sys.path`) instead.

const char ***Py_GetVersion**()

Part of the Stable ABI. Return the version of this Python interpreter. This is a string that looks something like

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

The first word (up to the first space character) is the current Python version ; the first characters are the major and minor version separated by a period. The returned string points into static storage ; the caller should not modify its value. The value is available to Python code as `sys.version`.

See also the *Py_Version* constant.

const char ***Py_GetPlatform**()

Part of the Stable ABI. Return the platform identifier for the current platform. On Unix, this is formed from the ”official” name of the operating system, converted to lower case, followed by the major revision number ; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is `'sunos5'`. On macOS, it is `'darwin'`. On Windows, it is `'win'`. The returned string points into static storage ; the caller should not modify its value. The value is available to Python code as `sys.platform`.

const char ***Py_GetCopyright**()

Part of the Stable ABI. Return the official copyright string for the current Python version, for example

```
'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'
```

The returned string points into static storage ; the caller should not modify its value. The value is available to Python code as `sys.copyright`.

const char ***Py_GetCompiler**()

Part of the Stable ABI. Return an indication of the compiler used to build the current Python version, in square brackets, for example :

```
"[GCC 2.7.2.2]"
```

The returned string points into static storage ; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

const char ***Py_GetBuildInfo**()

Part of the Stable ABI. Return information about the sequence number and build date and time of the current Python interpreter instance, for example

```
"#67, Aug  1 1997, 22:34:28"
```

The returned string points into static storage ; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

void **PySys_SetArgvEx**(int argc, wchar_t **argv, int updatepath)

Part of the Stable ABI. This API is kept for backward compatibility : setting `PyConfig.argv`, `PyConfig.parse_argv` and `PyConfig.safe_path` should be used instead, see *Python Initialization Configuration*.

Set `sys.argv` based on *argc* and *argv*. These parameters are similar to those passed to the program's `main()` function with the difference that the first entry should refer to the script file to be executed rather than the executable hosting the Python interpreter. If there isn't a script that will be run, the first entry in *argv* can be an empty string. If this function fails to initialize `sys.argv`, a fatal condition is signalled using `Py_FatalError()`.

If *updatepath* is zero, this is all the function does. If *updatepath* is non-zero, the function also modifies `sys.path` according to the following algorithm :

— If the name of an existing script is passed in `argv[0]`, the absolute path of the directory where the script is located is prepended to `sys.path`.

— Otherwise (that is, if *argc* is `0` or `argv[0]` doesn't point to an existing file name), an empty string is prepended to `sys.path`, which is the same as prepending the current working directory (`"."`).

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_t*` string.

See also `PyConfig.orig_argv` and `PyConfig.argv` members of the *Python Initialization Configuration*.

Note

It is recommended that applications embedding the Python interpreter for purposes other than executing a single script pass `0` as *updatepath*, and update `sys.path` themselves if desired. See **CVE 2008-5983**.

On versions before 3.1.3, you can achieve the same effect by manually popping the first `sys.path` element after having called `PySys_SetArgv()`, for example using :

```
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```

Ajouté dans la version 3.1.3.

Deprecated since version 3.11, will be removed in version 3.15.

void **PySys_SetArgv**(int argc, wchar_t **argv)

Part of the Stable ABI. This API is kept for backward compatibility : setting `PyConfig.argv` and `PyConfig.parse_argv` should be used instead, see *Python Initialization Configuration*.

This function works like `PySys_SetArgvEx()` with *updatepath* set to `1` unless the **python** interpreter was started with the `-I`.

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_t*` string.

See also `PyConfig.orig_argv` and `PyConfig.argv` members of the *Python Initialization Configuration*.

Modifié dans la version 3.4 : The *updatepath* value depends on `-I`.

Deprecated since version 3.11, will be removed in version 3.15.

void **Py_SetPythonHome**(const wchar_t *home)

Part of the Stable ABI. This API is kept for backward compatibility : setting `PyConfig.home` should be used instead, see *Python Initialization Configuration*.

Set the default "home" directory, that is, the location of the standard Python libraries. See `PYTHONHOME` for the meaning of the argument string.

The argument should point to a zero-terminated character string in static storage whose contents will not change for the duration of the program's execution. No code in the Python interpreter will change the contents of this storage.

Use `Py_DecodeLocale()` to decode a bytes string to get a `wchar_t*` string.

Deprecated since version 3.11, will be removed in version 3.15.

wchar_t ***Py_GetPythonHome**()

Part of the Stable ABI. Return the default "home", that is, the value set by `PyConfig.home`, or the value of

the PYTHONHOME environment variable if it is set.

This function should not be called before *Py_Initialize()*, otherwise it returns NULL.

Modifié dans la version 3.10 : It now returns NULL if called before *Py_Initialize()*.

Deprecated since version 3.13, will be removed in version 3.15 : Use *PyConfig_Get("home")* or the PYTHONHOME environment variable instead.

10.5 Thread State and the Global Interpreter Lock

Unless on a *free-threaded* build of *CPython*, the Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the *global interpreter lock* or *GIL*, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program : for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

Therefore, the rule exists that only the thread that has acquired the *GIL* may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads (see sys.setswitchinterval()). The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.

The Python interpreter keeps some thread-specific bookkeeping information inside a data structure called *PyThreadState*, known as a *thread state*. Each OS thread has a thread-local pointer to a *PyThreadState*; a thread state referenced by this pointer is considered to be *attached*.

A thread can only have one *attached thread state* at a time. An attached thread state is typically analogous with holding the *GIL*, except on *free-threaded* builds. On builds with the *GIL* enabled, *attaching* a thread state will block until the *GIL* can be acquired. However, even on builds with the *GIL* disabled, it is still required to have an attached thread state to call most of the C API.

In general, there will always be an *attached thread state* when using Python's C API. Only in some specific cases (such as in a *Py_BEGIN_ALLOW_THREADS* block) will the thread not have an attached thread state. If uncertain, check if *PyThreadState_GetUnchecked()* returns NULL.

10.5.1 Detaching the thread state from extension code

Most extension code manipulating the *thread state* has the following simple structure :

```
Save the thread state in a local variable.
... Do some blocking I/O operation ...
Restore the thread state from the local variable.
```

This is so common that a pair of macros exists to simplify it :

```
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

The *Py_BEGIN_ALLOW_THREADS* macro opens a new block and declares a hidden local variable; the *Py_END_ALLOW_THREADS* macro closes the block.

The block above expands to the following code :

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

Here is how these functions work :

The *attached thread state* holds the *GIL* for the entire interpreter. When detaching the *attached thread state*, the *GIL* is released, allowing other threads to attach a thread state to their own thread, thus getting the *GIL* and can start executing. The pointer to the prior *attached thread state* is stored as a local variable. Upon reaching *Py_END_ALLOW_THREADS*, the thread state that was previously *attached* is passed to *PyEval_RestoreThread()*. This function will block until another releases its *thread state*, thus allowing the old *thread state* to get re-attached and the C API can be called again.

For *free-threaded* builds, the *GIL* is normally out of the question, but detaching the *thread state* is still required for blocking I/O and long operations. The difference is that threads don't have to wait for the *GIL* to be released to attach their thread state, allowing true multi-core parallelism.

Note

Calling system I/O functions is the most common use case for detaching the *thread state*, but it can also be useful before calling long-running computations which don't need access to Python objects, such as compression or cryptographic functions operating over memory buffers. For example, the standard `zlib` and `hashlib` modules detach the *thread state* when compressing or hashing data.

10.5.2 Non-Python created threads

When threads are created using the dedicated Python APIs (such as the `threading` module), a thread state is automatically associated to them and the code showed above is therefore correct. However, when threads are created from C (for example by a third-party library with its own thread management), they don't hold the *GIL*, because they don't have an *attached thread state*.

If you need to call Python code from these threads (often this will be part of a callback API provided by the aforementioned third-party library), you must first register these threads with the interpreter by creating an *attached thread state* before you can start using the Python/C API. When you are done, you should detach the *thread state*, and finally free it.

The *PyGILState_Ensure()* and *PyGILState_Release()* functions do all of the above automatically. The typical idiom for calling into Python from a C thread is :

```
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Release the thread. No Python API allowed beyond this point. */
PyGILState_Release(gstate);
```

Note that the `PyGILState_*` functions assume there is only one global interpreter (created automatically by *Py_Initialize()*). Python supports the creation of additional interpreters (using *Py_NewInterpreter()*), but mixing multiple interpreters and the `PyGILState_*` API is unsupported. This is because *PyGILState_Ensure()* and similar functions default to *attaching* a *thread state* for the main interpreter, meaning that the thread can't safely interact with the calling subinterpreter.

10.5.3 Supporting subinterpreters in non-Python threads

If you would like to support subinterpreters with non-Python created threads, you must use the `PyThreadState_*` API instead of the traditional `PyGILState_*` API.

In particular, you must store the interpreter state from the calling function and pass it to *PyThreadState_New()*, which will ensure that the *thread state* is targeting the correct interpreter :

```
/* The return value of PyInterpreterState_Get() from the
   function that created this thread. */
```

(suite sur la page suivante)

(suite de la page précédente)

```
PyInterpreterState *interp = ThreadData->interp;
PyThreadState *tstate = PyThreadState_New(interp);
PyThreadState_Swap(tstate);

/* GIL of the subinterpreter is now held.
   Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Destroy the thread state. No Python API allowed beyond this point. */
PyThreadState_Clear(tstate);
PyThreadState_DeleteCurrent();
```

10.5.4 Cautions about fork()

Another important thing to note about threads is their behaviour in the face of the C `fork()` call. On most systems with `fork()`, after a process forks only the thread that issued the fork will exist. This has a concrete impact both on how locks must be handled and on all stored state in CPython's runtime.

The fact that only the "current" thread remains means any locks held by other threads will never be released. Python solves this for `os.fork()` by acquiring the locks it uses internally before the fork, and releasing them afterwards. In addition, it resets any lock-objects in the child. When extending or embedding Python, there is no way to inform Python of additional (non-Python) locks that need to be acquired before or reset after a fork. OS facilities such as `pthread_atfork()` would need to be used to accomplish the same thing. Additionally, when extending or embedding Python, calling `fork()` directly rather than through `os.fork()` (and returning to or calling into Python) may result in a deadlock by one of Python's internal locks being held by a thread that is defunct after the fork. *`PyOS_AfterFork_Child()`* tries to reset the necessary locks, but is not always able to.

The fact that all other threads go away also means that CPython's runtime state there must be cleaned up properly, which `os.fork()` does. This means finalizing all other *`PyThreadState`* objects belonging to the current interpreter and all other *`PyInterpreterState`* objects. Due to this and the special nature of the *"main" interpreter*, `fork()` should only be called in that interpreter's "main" thread, where the CPython global runtime was originally initialized. The only exception is if `exec()` will be called immediately after.

10.5.5 Cautions regarding runtime finalization

In the late stage of *interpreter shutdown*, after attempting to wait for non-daemon threads to exit (though this can be interrupted by `KeyboardInterrupt`) and running the `atexit` functions, the runtime is marked as *finalizing* : *`Py_IsFinalizing()`* and `sys.is_finalizing()` return true. At this point, only the *finalization thread* that initiated finalization (typically the main thread) is allowed to acquire the *GIL*.

If any thread, other than the finalization thread, attempts to attach a *thread state* during finalization, either explicitly or implicitly, the thread enters **a permanently blocked state** where it remains until the program exits. In most cases this is harmless, but this can result in deadlock if a later stage of finalization attempts to acquire a lock owned by the blocked thread, or otherwise waits on the blocked thread.

Gross ? Yes. This prevents random crashes and/or unexpectedly skipped C++ finalizations further up the call stack when such threads were forcibly exited here in CPython 3.13 and earlier. The CPython runtime *thread state* C APIs have never had any error reporting or handling expectations at *thread state* attachment time that would've allowed for graceful exit from this situation. Changing that would require new stable C APIs and rewriting the majority of C code in the CPython ecosystem to use those with error handling.

10.5.6 High-level API

These are the most commonly used types and functions when writing C extension code, or when embedding the Python interpreter :

type **`PyInterpreterState`**

Part of the Limited API *(as an opaque struct).* This data structure represents the state shared by a number of

cooperating threads. Threads belonging to the same interpreter share their module administration and a few other internal items. There are no public members in this structure.

Threads belonging to different interpreters initially share nothing, except process state like available memory, open file descriptors and such. The global interpreter lock is also shared by all threads, regardless of to which interpreter they belong.

Modifié dans la version 3.12 : **PEP 684** introduced the possibility of a *per-interpreter GIL*. See `Py_NewInterpreterFromConfig()`.

type **PyThreadState**

Part of the Limited API *(as an opaque struct)*. This data structure represents the state of a single thread. The only public data member is :

PyInterpreterState ***interp**

This thread's interpreter state.

void **PyEval_InitThreads**()

Part of the Stable ABI. Deprecated function which does nothing.

In Python 3.6 and older, this function created the GIL if it didn't exist.

Modifié dans la version 3.9 : The function now does nothing.

Modifié dans la version 3.7 : This function is now called by `Py_Initialize()`, so you don't have to call it yourself anymore.

Modifié dans la version 3.2 : This function cannot be called before `Py_Initialize()` anymore.

Obsolète depuis la version 3.9.

PyThreadState ***PyEval_SaveThread**()

Part of the Stable ABI. Detach the *attached thread state* and return it. The thread will have no *thread state* upon returning.

void **PyEval_RestoreThread**(*PyThreadState* *tstate)

Part of the Stable ABI. Set the *attached thread state* to *tstate*. The passed *thread state* **should not** be *attached*, otherwise deadlock ensues. *tstate* will be attached upon returning.

> **Note**
>
> Calling this function from a thread when the runtime is finalizing will hang the thread until the program exits, even if the thread was not created by Python. Refer to *Cautions regarding runtime finalization* for more details.

Modifié dans la version 3.14 : Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

PyThreadState ***PyThreadState_Get**()

Part of the Stable ABI. Return the *attached thread state*. If the thread has no attached thread state, (such as when inside of `Py_BEGIN_ALLOW_THREADS` block), then this issues a fatal error (so that the caller needn't check for `NULL`).

See also `PyThreadState_GetUnchecked()`.

PyThreadState ***PyThreadState_GetUnchecked**()

Similar to `PyThreadState_Get()`, but don't kill the process with a fatal error if it is NULL. The caller is responsible to check if the result is NULL.

Ajouté dans la version 3.13 : In Python 3.5 to 3.12, the function was private and known as `_PyThreadState_UncheckedGet()`.

PyThreadState ***PyThreadState_Swap**(*PyThreadState* *tstate)

Part of the Stable ABI. Set the *attached thread state* to *tstate*, and return the *thread state* that was attached prior to calling.

This function is safe to call without an *attached thread state*; it will simply return `NULL` indicating that there was no prior thread state.

Note

Similar to `PyGILState_Ensure()`, this function will hang the thread if the runtime is finalizing.

The following functions use thread-local storage, and are not compatible with sub-interpreters :

PyGILState_STATE **PyGILState_Ensure**()

Part of the Stable ABI. Ensure that the current thread is ready to call the Python C API regardless of the current state of Python, or of the *attached thread state*. This may be called as many times as desired by a thread as long as each call is matched with a call to `PyGILState_Release()`. In general, other thread-related APIs may be used between `PyGILState_Ensure()` and `PyGILState_Release()` calls as long as the thread state is restored to its previous state before the Release(). For example, normal usage of the `Py_BEGIN_ALLOW_THREADS` and `Py_END_ALLOW_THREADS` macros is acceptable.

The return value is an opaque ”handle” to the *attached thread state* when `PyGILState_Ensure()` was called, and must be passed to `PyGILState_Release()` to ensure Python is left in the same state. Even though recursive calls are allowed, these handles *cannot* be shared - each unique call to `PyGILState_Ensure()` must save the handle for its call to `PyGILState_Release()`.

When the function returns, there will be an *attached thread state* and the thread will be able to call arbitrary Python code. Failure is a fatal error.

Avertissement

Calling this function when the runtime is finalizing is unsafe. Doing so will either hang the thread until the program ends, or fully crash the interpreter in rare cases. Refer to *Cautions regarding runtime finalization* for more details.

Modifié dans la version 3.14 : Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

void **PyGILState_Release**(PyGILState_STATE)

Part of the Stable ABI. Release any resources previously acquired. After this call, Python's state will be the same as it was prior to the corresponding `PyGILState_Ensure()` call (but generally this state will be unknown to the caller, hence the use of the GILState API).

Every call to `PyGILState_Ensure()` must be matched by a call to `PyGILState_Release()` on the same thread.

PyThreadState ***PyGILState_GetThisThreadState**()

Part of the Stable ABI. Get the *attached thread state* for this thread. May return `NULL` if no GILState API has been used on the current thread. Note that the main thread always has such a thread-state, even if no auto-thread-state call has been made on the main thread. This is mainly a helper/diagnostic function.

Note

This function does not account for *thread states* created by something other than `PyGILState_Ensure()` (such as `PyThreadState_New()`). Prefer `PyThreadState_Get()` or `PyThreadState_GetUnchecked()` for most cases.

int **PyGILState_Check**()

Return `1` if the current thread is holding the *GIL* and `0` otherwise. This function can be called from any thread at any time. Only if it has had its *thread state* initialized via *PyGILState_Ensure()* will it return `1`. This is mainly a helper/diagnostic function. It can be useful for example in callback contexts or memory allocation functions when knowing that the *GIL* is locked can allow the caller to perform sensitive actions or otherwise behave differently.

Note

If the current Python process has ever created a subinterpreter, this function will *always* return `1`. Prefer *PyThreadState_GetUnchecked()* for most cases.

Ajouté dans la version 3.4.

The following macros are normally used without a trailing semicolon ; look for example usage in the Python source distribution.

Py_BEGIN_ALLOW_THREADS

Part of the Stable ABI. This macro expands to `{ PyThreadState *_save; _save = PyEval_SaveThread();`. Note that it contains an opening brace ; it must be matched with a following *Py_END_ALLOW_THREADS* macro. See above for further discussion of this macro.

Py_END_ALLOW_THREADS

Part of the Stable ABI. This macro expands to `PyEval_RestoreThread(_save); }`. Note that it contains a closing brace ; it must be matched with an earlier *Py_BEGIN_ALLOW_THREADS* macro. See above for further discussion of this macro.

Py_BLOCK_THREADS

Part of the Stable ABI. This macro expands to `PyEval_RestoreThread(_save);` : it is equivalent to *Py_END_ALLOW_THREADS* without the closing brace.

Py_UNBLOCK_THREADS

Part of the Stable ABI. This macro expands to `_save = PyEval_SaveThread();` : it is equivalent to *Py_BEGIN_ALLOW_THREADS* without the opening brace and variable declaration.

10.5.7 Low-level API

All of the following functions must be called after *Py_Initialize()*.

Modifié dans la version 3.7 : *Py_Initialize()* now initializes the *GIL* and sets an *attached thread state*.

PyInterpreterState ***PyInterpreterState_New**()

Part of the Stable ABI. Create a new interpreter state object. An *attached thread state* is not needed, but may optionally exist if it is necessary to serialize calls to this function.

Raises an auditing event `cpython.PyInterpreterState_New` with no arguments.

void **PyInterpreterState_Clear**(*PyInterpreterState* *interp)

Part of the Stable ABI. Reset all information in an interpreter state object. There must be an *attached thread state* for the interpreter.

Raises an auditing event `cpython.PyInterpreterState_Clear` with no arguments.

void **PyInterpreterState_Delete**(*PyInterpreterState* *interp)

Part of the Stable ABI. Destroy an interpreter state object. There **should not** be an *attached thread state* for the target interpreter. The interpreter state must have been reset with a previous call to *PyInterpreterState_Clear()*.

PyThreadState ***PyThreadState_New**(*PyInterpreterState* *interp)

Part of the Stable ABI. Create a new thread state object belonging to the given interpreter object. An *attached thread state* is not needed.

void **PyThreadState_Clear**(*PyThreadState* *tstate)

Part of the Stable ABI. Reset all information in a *thread state* object. *tstate* must be *attached*

Modifié dans la version 3.9 : This function now calls the `PyThreadState.on_delete` callback. Previously, that happened in *PyThreadState_Delete()*.

Modifié dans la version 3.13 : The `PyThreadState.on_delete` callback was removed.

void **PyThreadState_Delete**(*PyThreadState* *tstate)

Part of the Stable ABI. Destroy a *thread state* object. *tstate* should not be *attached* to any thread. *tstate* must have been reset with a previous call to *PyThreadState_Clear()*.

void **PyThreadState_DeleteCurrent**(void)

Detach the *attached thread state* (which must have been reset with a previous call to *PyThreadState_Clear()*) and then destroy it.

No *thread state* will be *attached* upon returning.

PyFrameObject ***PyThreadState_GetFrame**(*PyThreadState* *tstate)

Part of the Stable ABI *Depuis la version 3.10.* Get the current frame of the Python thread state *tstate*.

Return a *strong reference*. Return `NULL` if no frame is currently executing.

See also *PyEval_GetFrame()*.

tstate must not be `NULL`, and must be *attached*.

Ajouté dans la version 3.9.

uint64_t **PyThreadState_GetID**(*PyThreadState* *tstate)

Part of the Stable ABI *Depuis la version 3.10.* Get the unique *thread state* identifier of the Python thread state *tstate*.

tstate must not be `NULL`, and must be *attached*.

Ajouté dans la version 3.9.

PyInterpreterState ***PyThreadState_GetInterpreter**(*PyThreadState* *tstate)

Part of the Stable ABI *Depuis la version 3.10.* Get the interpreter of the Python thread state *tstate*.

tstate must not be `NULL`, and must be *attached*.

Ajouté dans la version 3.9.

void **PyThreadState_EnterTracing**(*PyThreadState* *tstate)

Suspend tracing and profiling in the Python thread state *tstate*.

Resume them using the *PyThreadState_LeaveTracing()* function.

Ajouté dans la version 3.11.

void **PyThreadState_LeaveTracing**(*PyThreadState* *tstate)

Resume tracing and profiling in the Python thread state *tstate* suspended by the *PyThreadState_EnterTracing()* function.

See also *PyEval_SetTrace()* and *PyEval_SetProfile()* functions.

Ajouté dans la version 3.11.

PyInterpreterState ***PyInterpreterState_Get**(void)

Part of the Stable ABI *Depuis la version 3.9.* Get the current interpreter.

Issue a fatal error if there no *attached thread state*. It cannot return NULL.

Ajouté dans la version 3.9.

int64_t **PyInterpreterState_GetID**(*PyInterpreterState* *interp)

Part of the Stable ABI *Depuis la version 3.7.* Return the interpreter's unique ID. If there was any error in doing so then `-1` is returned and an error is set.

The caller must have an *attached thread state*.

Ajouté dans la version 3.7.

PyObject ***PyInterpreterState_GetDict**(*PyInterpreterState* *interp)

Valeur de retour : référence empruntée. Part of the Stable ABI *Depuis la version 3.8.* Return a dictionary in which interpreter-specific data may be stored. If this function returns `NULL` then no exception has been raised and the caller should assume no interpreter-specific dict is available.

This is not a replacement for `PyModule_GetState()`, which extensions should use to store interpreter-specific state information.

The returned dictionary is borrowed from the interpreter and is valid until interpreter shutdown.

Ajouté dans la version 3.8.

typedef *PyObject* *(***_PyFrameEvalFunction**)(*PyThreadState* *tstate, *_PyInterpreterFrame* *frame, int throwflag)

Type of a frame evaluation function.

The *throwflag* parameter is used by the `throw()` method of generators : if non-zero, handle the current exception.

Modifié dans la version 3.9 : The function now takes a *tstate* parameter.

Modifié dans la version 3.11 : The *frame* parameter changed from `PyFrameObject*` to `_PyInterpreterFrame*`.

_PyFrameEvalFunction **_PyInterpreterState_GetEvalFrameFunc**(*PyInterpreterState* *interp)

Get the frame evaluation function.

See the **PEP 523** "Adding a frame evaluation API to CPython".

Ajouté dans la version 3.9.

void **_PyInterpreterState_SetEvalFrameFunc**(*PyInterpreterState* *interp, *_PyFrameEvalFunction* eval_frame)

Set the frame evaluation function.

See the **PEP 523** "Adding a frame evaluation API to CPython".

Ajouté dans la version 3.9.

PyObject ***PyThreadState_GetDict**()

Valeur de retour : référence empruntée. Part of the Stable ABI. Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no *thread state* is *attached*. If this function returns `NULL`, no exception has been raised and the caller should assume no thread state is attached.

int **PyThreadState_SetAsyncExc**(unsigned long id, *PyObject* *exc)

Part of the Stable ABI. Asynchronously raise an exception in a thread. The *id* argument is the thread id of the target thread; *exc* is the exception object to be raised. This function does not steal any references to *exc*. To prevent naive misuse, you must write your own C extension to call this. Must be called with an *attached thread state*. Returns the number of thread states modified; this is normally one, but will be zero if the thread id isn't found. If *exc* is `NULL`, the pending exception (if any) for the thread is cleared. This raises no exceptions.

Modifié dans la version 3.7 : The type of the *id* parameter changed from `long` to `unsigned long`.

void **PyEval_AcquireThread**(*PyThreadState* *tstate)

Part of the Stable ABI. *Attach tstate* to the current thread, which must not be `NULL` or already *attached*.

The calling thread must not already have an *attached thread state*.

Note

Calling this function from a thread when the runtime is finalizing will hang the thread until the program exits, even if the thread was not created by Python. Refer to *Cautions regarding runtime finalization* for more details.

Modifié dans la version 3.8 : Updated to be consistent with `PyEval_RestoreThread()`, `Py_END_ALLOW_THREADS()`, and `PyGILState_Ensure()`, and terminate the current thread if called while the interpreter is finalizing.

Modifié dans la version 3.14 : Hangs the current thread, rather than terminating it, if called while the interpreter is finalizing.

`PyEval_RestoreThread()` is a higher-level function which is always available (even when threads have not been initialized).

void **PyEval_ReleaseThread**(*PyThreadState* *tstate)

Part of the Stable ABI. Detach the *attached thread state*. The *tstate* argument, which must not be `NULL`, is only used to check that it represents the *attached thread state* --- if it isn't, a fatal error is reported.

`PyEval_SaveThread()` is a higher-level function which is always available (even when threads have not been initialized).

10.6 Sub-interpreter support

While in most uses, you will only embed a single Python interpreter, there are cases where you need to create several independent interpreters in the same process and perhaps even in the same thread. Sub-interpreters allow you to do that.

The "main" interpreter is the first one created when the runtime initializes. It is usually the only Python interpreter in a process. Unlike sub-interpreters, the main interpreter has unique process-global responsibilities like signal handling. It is also responsible for execution during runtime initialization and is usually the active interpreter during runtime finalization. The `PyInterpreterState_Main()` function returns a pointer to its state.

You can switch between sub-interpreters using the `PyThreadState_Swap()` function. You can create and destroy them using the following functions :

type **PyInterpreterConfig**

Structure containing most parameters to configure a sub-interpreter. Its values are used only in `Py_NewInterpreterFromConfig()` and never modified by the runtime.

Ajouté dans la version 3.12.

Structure fields :

int **use_main_obmalloc**

If this is `0` then the sub-interpreter will use its own "object" allocator state. Otherwise it will use (share) the main interpreter's.

If this is `0` then `check_multi_interp_extensions` must be `1` (non-zero). If this is `1` then `gil` must not be `PyInterpreterConfig_OWN_GIL`.

int **allow_fork**

If this is `0` then the runtime will not support forking the process in any thread where the sub-interpreter is currently active. Otherwise fork is unrestricted.

Note that the `subprocess` module still works when fork is disallowed.

int **allow_exec**

If this is `0` then the runtime will not support replacing the current process via exec (e.g. `os.execv()`) in any thread where the sub-interpreter is currently active. Otherwise exec is unrestricted.

Note that the `subprocess` module still works when exec is disallowed.

int **allow_threads**

If this is `0` then the sub-interpreter's `threading` module won't create threads. Otherwise threads are allowed.

int **allow_daemon_threads**

If this is `0` then the sub-interpreter's `threading` module won't create daemon threads. Otherwise daemon threads are allowed (as long as *allow_threads* is non-zero).

int **check_multi_interp_extensions**

If this is `0` then all extension modules may be imported, including legacy (single-phase init) modules, in any thread where the sub-interpreter is currently active. Otherwise only multi-phase init extension modules (see **PEP 489**) may be imported. (Also see *Py_mod_multiple_interpreters*.)

This must be `1` (non-zero) if *use_main_obmalloc* is `0`.

int **gil**

This determines the operation of the GIL for the sub-interpreter. It may be one of the following :

PyInterpreterConfig_DEFAULT_GIL

Use the default selection (*PyInterpreterConfig_SHARED_GIL*).

PyInterpreterConfig_SHARED_GIL

Use (share) the main interpreter's GIL.

PyInterpreterConfig_OWN_GIL

Use the sub-interpreter's own GIL.

If this is *PyInterpreterConfig_OWN_GIL* then *PyInterpreterConfig.use_main_obmalloc* must be `0`.

PyStatus **Py_NewInterpreterFromConfig**(*PyThreadState* **tstate_p, const *PyInterpreterConfig* *config)

Create a new sub-interpreter. This is an (almost) totally separate environment for the execution of Python code. In particular, the new interpreter has separate, independent versions of all imported modules, including the fundamental modules `builtins`, `__main__` and `sys`. The table of loaded modules (`sys.modules`) and the module search path (`sys.path`) are also separate. The new environment has no `sys.argv` variable. It has new standard I/O stream file objects `sys.stdin`, `sys.stdout` and `sys.stderr` (however these refer to the same underlying file descriptors).

The given *config* controls the options with which the interpreter is initialized.

Upon success, *tstate_p* will be set to the first *thread state* created in the new sub-interpreter. This thread state is *attached*. Note that no actual thread is created ; see the discussion of thread states below. If creation of the new interpreter is unsuccessful, *tstate_p* is set to `NULL` ; no exception is set since the exception state is stored in the *attached thread state*, which might not exist.

Like all other Python/C API functions, an *attached thread state* must be present before calling this function, but it might be detached upon returning. On success, the returned thread state will be *attached*. If the sub-interpreter is created with its own *GIL* then the *attached thread state* of the calling interpreter will be detached. When the function returns, the new interpreter's *thread state* will be *attached* to the current thread and the previous interpreter's *attached thread state* will remain detached.

Ajouté dans la version 3.12.

Sub-interpreters are most effective when isolated from each other, with certain functionality restricted :

```
PyInterpreterConfig config = {
    .use_main_obmalloc = 0,
    .allow_fork = 0,
    .allow_exec = 0,
    .allow_threads = 1,
    .allow_daemon_threads = 0,
    .check_multi_interp_extensions = 1,
    .gil = PyInterpreterConfig_OWN_GIL,
```

(suite sur la page suivante)

(suite de la page précédente)

```
};
PyThreadState *tstate = NULL;
PyStatus status = Py_NewInterpreterFromConfig(&tstate, &config);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}
```

Note that the config is used only briefly and does not get modified. During initialization the config's values are converted into various *PyInterpreterState* values. A read-only copy of the config may be stored internally on the *PyInterpreterState*.

Extension modules are shared between (sub-)interpreters as follows :

— For modules using multi-phase initialization, e.g. *PyModule_FromDefAndSpec()*, a separate module object is created and initialized for each interpreter. Only C-level static and global variables are shared between these module objects.

— For modules using single-phase initialization, e.g. *PyModule_Create()*, the first time a particular extension is imported, it is initialized normally, and a (shallow) copy of its module's dictionary is squirreled away. When the same extension is imported by another (sub-)interpreter, a new module is initialized and filled with the contents of this copy ; the extension's `init` function is not called. Objects in the module's dictionary thus end up shared across (sub-)interpreters, which might cause unwanted behavior (see *Bugs and caveats* below).

Note that this is different from what happens when an extension is imported after the interpreter has been completely re-initialized by calling *Py_FinalizeEx()* and *Py_Initialize()* ; in that case, the extension's `initmodule` function *is* called again. As with multi-phase initialization, this means that only C-level static and global variables are shared between these modules.

PyThreadState ***Py_NewInterpreter**(void)

Part of the Stable ABI. Create a new sub-interpreter. This is essentially just a wrapper around *Py_NewInterpreterFromConfig()* with a config that preserves the existing behavior. The result is an unisolated sub-interpreter that shares the main interpreter's GIL, allows fork/exec, allows daemon threads, and allows single-phase init modules.

void **Py_EndInterpreter**(*PyThreadState* *tstate)

Part of the Stable ABI. Destroy the (sub-)interpreter represented by the given *thread state*. The given thread state must be *attached*. When the call returns, there will be no *attached thread state*. All thread states associated with this interpreter are destroyed.

Py_FinalizeEx() will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

10.6.1 A Per-Interpreter GIL

Using *Py_NewInterpreterFromConfig()* you can create a sub-interpreter that is completely isolated from other interpreters, including having its own GIL. The most important benefit of this isolation is that such an interpreter can execute Python code without being blocked by other interpreters or blocking any others. Thus a single Python process can truly take advantage of multiple CPU cores when running Python code. The isolation also encourages a different approach to concurrency than that of just using threads. (See **PEP 554** and **PEP 684**.)

Using an isolated interpreter requires vigilance in preserving that isolation. That especially means not sharing any objects or mutable state without guarantees about thread-safety. Even objects that are otherwise immutable (e.g. `None`, `(1, 5)`) can't normally be shared because of the refcount. One simple but less-efficient approach around this is to use a global lock around all use of some state (or object). Alternately, effectively immutable objects (like integers or strings) can be made safe in spite of their refcounts by making them *immortal*. In fact, this has been done for the builtin singletons, small integers, and a number of other builtin objects.

If you preserve isolation then you will have access to proper multi-core computing without the complications that come with free-threading. Failure to preserve isolation will expose you to the full consequences of free-threading, including races and hard-to-debug crashes.

Aside from that, one of the main challenges of using multiple isolated interpreters is how to communicate between them safely (not break isolation) and efficiently. The runtime and stdlib do not provide any standard approach to

this yet. A future stdlib module would help mitigate the effort of preserving isolation and expose effective tools for communicating (and sharing) data between interpreters.

Ajouté dans la version 3.12.

10.6.2 Bugs and caveats

Because sub-interpreters (and the main interpreter) are part of the same process, the insulation between them isn't perfect --- for example, using low-level file operations like `os.close()` they can (accidentally or maliciously) affect each other's open files. Because of the way extensions are shared between (sub-)interpreters, some extensions may not work properly ; this is especially likely when using single-phase initialization or (static) global variables. It is possible to insert objects created in one sub-interpreter into a namespace of another (sub-)interpreter ; this should be avoided if possible.

Special care should be taken to avoid sharing user-defined functions, methods, instances or classes between sub-interpreters, since import operations executed by such objects may affect the wrong (sub-)interpreter's dictionary of loaded modules. It is equally important to avoid sharing objects from which the above are reachable.

Also note that combining this functionality with `PyGILState_*` APIs is delicate, because these APIs assume a bijection between Python thread states and OS-level threads, an assumption broken by the presence of sub-interpreters. It is highly recommended that you don't switch sub-interpreters between a pair of matching *`PyGILState_Ensure()`* and *`PyGILState_Release()`* calls. Furthermore, extensions (such as `ctypes`) using these APIs to allow calling of Python code from non-Python created threads will probably be broken when using sub-interpreters.

10.7 Asynchronous Notifications

A mechanism is provided to make asynchronous notifications to the main interpreter thread. These notifications take the form of a function pointer and a void pointer argument.

int **`Py_AddPendingCall`**(int (*func)(void*), void *arg)

Part of the Stable ABI. Schedule a function to be called from the main interpreter thread. On success, `0` is returned and *func* is queued for being called in the main thread. On failure, `-1` is returned without setting any exception.

When successfully queued, *func* will be *eventually* called from the main interpreter thread with the argument *arg*. It will be called asynchronously with respect to normally running Python code, but with both these conditions met :
— on a *bytecode* boundary ;
— with the main thread holding an *attached thread state* (*func* can therefore use the full C API).
func must return `0` on success, or `-1` on failure with an exception set. *func* won't be interrupted to perform another asynchronous notification recursively, but it can still be interrupted to switch threads if the *thread state* is detached.

This function doesn't need an *attached thread state*. However, to call this function in a subinterpreter, the caller must have an *attached thread state*. Otherwise, the function *func* can be scheduled to be called from the wrong interpreter.

> **Avertissement**
>
> This is a low-level function, only useful for very special cases. There is no guarantee that *func* will be called as quick as possible. If the main thread is busy executing a system call, *func* won't be called before the system call returns. This function is generally **not** suitable for calling Python code from arbitrary C threads. Instead, use the *PyGILState API*.

Ajouté dans la version 3.1.

Modifié dans la version 3.9 : If this function is called in a subinterpreter, the function *func* is now scheduled to be called from the subinterpreter, rather than being called from the main interpreter. Each subinterpreter now has its own list of scheduled calls.

10.8 Profiling and Tracing

The Python interpreter provides some low-level support for attaching profiling and execution tracing facilities. These are used for profiling, debugging, and coverage analysis tools.

This C interface allows the profiling or tracing code to avoid the overhead of calling through Python-level callable objects, making a direct C function call instead. The essential attributes of the facility have not changed ; the interface allows trace functions to be installed per-thread, and the basic events reported to the trace function are the same as had been reported to the Python-level trace functions in previous versions.

typedef int (***Py_tracefunc**)(*PyObject* *obj, *PyFrameObject* *frame, int what, *PyObject* *arg)

The type of the trace function registered using `PyEval_SetProfile()` and `PyEval_SetTrace()`. The first parameter is the object passed to the registration function as *obj*, *frame* is the frame object to which the event pertains, *what* is one of the constants `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION`, `PyTrace_C_RETURN`, or `PyTrace_OPCODE`, and *arg* depends on the value of *what* :

Value of *what*	Meaning of *arg*
`PyTrace_CALL`	Always `Py_None`.
`PyTrace_EXCEPTION`	Exception information as returned by `sys.exc_info()`.
`PyTrace_LINE`	Always `Py_None`.
`PyTrace_RETURN`	Value being returned to the caller, or `NULL` if caused by an exception.
`PyTrace_C_CALL`	Function object being called.
`PyTrace_C_EXCEPTION`	Function object being called.
`PyTrace_C_RETURN`	Function object being called.
`PyTrace_OPCODE`	Always `Py_None`.

int **PyTrace_CALL**

The value of the *what* parameter to a `Py_tracefunc` function when a new call to a function or method is being reported, or a new entry into a generator. Note that the creation of the iterator for a generator function is not reported as there is no control transfer to the Python bytecode in the corresponding frame.

int **PyTrace_EXCEPTION**

The value of the *what* parameter to a `Py_tracefunc` function when an exception has been raised. The callback function is called with this value for *what* when after any bytecode is processed after which the exception becomes set within the frame being executed. The effect of this is that as exception propagation causes the Python stack to unwind, the callback is called upon return to each frame as the exception propagates. Only trace functions receives these events ; they are not needed by the profiler.

int **PyTrace_LINE**

The value passed as the *what* parameter to a `Py_tracefunc` function (but not a profiling function) when a line-number event is being reported. It may be disabled for a frame by setting `f_trace_lines` to *0* on that frame.

int **PyTrace_RETURN**

The value for the *what* parameter to `Py_tracefunc` functions when a call is about to return.

int **PyTrace_C_CALL**

The value for the *what* parameter to `Py_tracefunc` functions when a C function is about to be called.

int **PyTrace_C_EXCEPTION**

The value for the *what* parameter to `Py_tracefunc` functions when a C function has raised an exception.

int **PyTrace_C_RETURN**

The value for the *what* parameter to `Py_tracefunc` functions when a C function has returned.

int **PyTrace_OPCODE**

The value for the *what* parameter to `Py_tracefunc` functions (but not profiling functions) when a new opcode is about to be executed. This event is not emitted by default : it must be explicitly requested by setting `f_trace_opcodes` to *1* on the frame.

void **PyEval_SetProfile**(*Py_tracefunc* func, *PyObject* *obj)

Set the profiler function to *func*. The *obj* parameter is passed to the function as its first parameter, and may be any Python object, or `NULL`. If the profile function needs to maintain state, using a different value for *obj* for each thread provides a convenient and thread-safe place to store it. The profile function is called for all monitored events except `PyTrace_LINE` `PyTrace_OPCODE` and `PyTrace_EXCEPTION`.

See also the `sys.setprofile()` function.

The caller must have an *attached thread state*.

void **PyEval_SetProfileAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

Like `PyEval_SetProfile()` but sets the profile function in all running threads belonging to the current interpreter instead of the setting it only on the current thread.

The caller must have an *attached thread state*.

As `PyEval_SetProfile()`, this function ignores any exceptions raised while setting the profile functions in all threads.

Ajouté dans la version 3.12.

void **PyEval_SetTrace**(*Py_tracefunc* func, *PyObject* *obj)

Set the tracing function to *func*. This is similar to `PyEval_SetProfile()`, except the tracing function does receive line-number events and per-opcode events, but does not receive any event related to C function objects being called. Any trace function registered using `PyEval_SetTrace()` will not receive `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION` or `PyTrace_C_RETURN` as a value for the *what* parameter.

See also the `sys.settrace()` function.

The caller must have an *attached thread state*.

void **PyEval_SetTraceAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

Like `PyEval_SetTrace()` but sets the tracing function in all running threads belonging to the current interpreter instead of the setting it only on the current thread.

The caller must have an *attached thread state*.

As `PyEval_SetTrace()`, this function ignores any exceptions raised while setting the trace functions in all threads.

Ajouté dans la version 3.12.

10.9 Reference tracing

Ajouté dans la version 3.13.

typedef int (***PyRefTracer**)(*PyObject**, int event, void *data)

The type of the trace function registered using `PyRefTracer_SetTracer()`. The first parameter is a Python object that has been just created (when **event** is set to `PyRefTracer_CREATE`) or about to be destroyed (when **event** is set to `PyRefTracer_DESTROY`). The **data** argument is the opaque pointer that was provided when `PyRefTracer_SetTracer()` was called.

Ajouté dans la version 3.13.

int **PyRefTracer_CREATE**

The value for the *event* parameter to `PyRefTracer` functions when a Python object has been created.

int **PyRefTracer_DESTROY**

The value for the *event* parameter to `PyRefTracer` functions when a Python object has been destroyed.

int **PyRefTracer_SetTracer**(*PyRefTracer* tracer, void *data)

Register a reference tracer function. The function will be called when a new Python has been created or when an object is going to be destroyed. If **data** is provided it must be an opaque pointer that will be provided when the tracer function is called. Return `0` on success. Set an exception and return `-1` on error.

Not that tracer functions **must not** create Python objects inside or otherwise the call will be re-entrant. The tracer also **must not** clear any existing exception or set an exception. A *thread state* will be active every time the tracer function is called.

There must be an *attached thread state* when calling this function.

Ajouté dans la version 3.13.

PyRefTracer **PyRefTracer_GetTracer**(void **data)

Get the registered reference tracer function and the value of the opaque data pointer that was registered when `PyRefTracer_SetTracer()` was called. If no tracer was registered this function will return NULL and will set the **data** pointer to NULL.

There must be an *attached thread state* when calling this function.

Ajouté dans la version 3.13.

10.10 Support avancé du débogueur

These functions are only intended to be used by advanced debugging tools.

PyInterpreterState ***PyInterpreterState_Head**()

Return the interpreter state object at the head of the list of all such objects.

PyInterpreterState ***PyInterpreterState_Main**()

Return the main interpreter state object.

PyInterpreterState ***PyInterpreterState_Next**(*PyInterpreterState* *interp)

Return the next interpreter state object after *interp* from the list of all such objects.

PyThreadState ***PyInterpreterState_ThreadHead**(*PyInterpreterState* *interp)

Return the pointer to the first `PyThreadState` object in the list of threads associated with the interpreter *interp*.

PyThreadState ***PyThreadState_Next**(*PyThreadState* *tstate)

Return the next thread state object after *tstate* from the list of all such objects belonging to the same `PyInterpreterState` object.

10.11 Thread Local Storage Support

The Python interpreter provides low-level support for thread-local storage (TLS) which wraps the underlying native TLS implementation to support the Python-level thread local storage API (`threading.local`). The CPython C level APIs are similar to those offered by pthreads and Windows : use a thread key and functions to associate a `void*` value per thread.

A *thread state* does *not* need to be *attached* when calling these functions ; they suppl their own locking.

Note that `Python.h` does not include the declaration of the TLS APIs, you need to include `pythread.h` to use thread-local storage.

Note

None of these API functions handle memory management on behalf of the `void*` values. You need to allocate and deallocate them yourself. If the `void*` values happen to be `PyObject*`, these functions don't do refcount operations on them either.

10.11.1 Thread Specific Storage (TSS) API

TSS API is introduced to supersede the use of the existing TLS API within the CPython interpreter. This API uses a new type *Py_tss_t* instead of `int` to represent thread keys.

Ajouté dans la version 3.7.

Voir aussi

"A New C-API for Thread-Local Storage in CPython" (**PEP 539**)

type **Py_tss_t**

This data structure represents the state of a thread key, the definition of which may depend on the underlying TLS implementation, and it has an internal field representing the key's initialization state. There are no public members in this structure.

When *Py_LIMITED_API* is not defined, static allocation of this type by *Py_tss_NEEDS_INIT* is allowed.

Py_tss_NEEDS_INIT

This macro expands to the initializer for *Py_tss_t* variables. Note that this macro won't be defined with *Py_LIMITED_API*.

Dynamic Allocation

Dynamic allocation of the *Py_tss_t*, required in extension modules built with *Py_LIMITED_API*, where static allocation of this type is not possible due to its implementation being opaque at build time.

Py_tss_t ***PyThread_tss_alloc**()

Part of the Stable ABI *Depuis la version 3.7.* Return a value which is the same state as a value initialized with *Py_tss_NEEDS_INIT*, or `NULL` in the case of dynamic allocation failure.

void **PyThread_tss_free**(*Py_tss_t* *key)

Part of the Stable ABI *Depuis la version 3.7.* Free the given *key* allocated by *PyThread_tss_alloc()*, after first calling *PyThread_tss_delete()* to ensure any associated thread locals have been unassigned. This is a no-op if the *key* argument is `NULL`.

Note

A freed key becomes a dangling pointer. You should reset the key to `NULL`.

Méthodes

The parameter *key* of these functions must not be `NULL`. Moreover, the behaviors of *PyThread_tss_set()* and *PyThread_tss_get()* are undefined if the given *Py_tss_t* has not been initialized by *PyThread_tss_create()*.

int **PyThread_tss_is_created**(*Py_tss_t* *key)

Part of the Stable ABI *Depuis la version 3.7.* Return a non-zero value if the given *Py_tss_t* has been initialized by *PyThread_tss_create()*.

int **PyThread_tss_create**(*Py_tss_t* *key)

Part of the Stable ABI *Depuis la version 3.7.* Return a zero value on successful initialization of a TSS key. The behavior is undefined if the value pointed to by the *key* argument is not initialized by *Py_tss_NEEDS_INIT*. This function can be called repeatedly on the same key -- calling it on an already initialized key is a no-op and immediately returns success.

void **PyThread_tss_delete**(*Py_tss_t* *key)

Part of the Stable ABI *Depuis la version 3.7.* Destroy a TSS key to forget the values associated with the key

across all threads, and change the key's initialization state to uninitialized. A destroyed key is able to be initialized again by `PyThread_tss_create()`. This function can be called repeatedly on the same key -- calling it on an already destroyed key is a no-op.

int **PyThread_tss_set**(*Py_tss_t* *key, void *value)

Part of the Stable ABI *Depuis la version 3.7.* Return a zero value to indicate successfully associating a `void*` value with a TSS key in the current thread. Each thread has a distinct mapping of the key to a `void*` value.

void ***PyThread_tss_get**(*Py_tss_t* *key)

Part of the Stable ABI *Depuis la version 3.7.* Return the `void*` value associated with a TSS key in the current thread. This returns `NULL` if no value is associated with the key in the current thread.

10.11.2 Thread Local Storage (TLS) API

Obsolète depuis la version 3.7 : This API is superseded by *Thread Specific Storage (TSS) API*.

Note

This version of the API does not support platforms where the native TLS key is defined in a way that cannot be safely cast to `int`. On such platforms, `PyThread_create_key()` will return immediately with a failure status, and the other TLS functions will all be no-ops on such platforms.

Due to the compatibility problem noted above, this version of the API should not be used in new code.

int **PyThread_create_key**()

Part of the Stable ABI.

void **PyThread_delete_key**(int key)

Part of the Stable ABI.

int **PyThread_set_key_value**(int key, void *value)

Part of the Stable ABI.

void ***PyThread_get_key_value**(int key)

Part of the Stable ABI.

void **PyThread_delete_key_value**(int key)

Part of the Stable ABI.

void **PyThread_ReInitTLS**()

Part of the Stable ABI.

10.12 Primitives de synchronisation

The C-API provides a basic mutual exclusion lock.

type **PyMutex**

A mutual exclusion lock. The `PyMutex` should be initialized to zero to represent the unlocked state. For example :

```
PyMutex mutex = {0};
```

Instances of `PyMutex` should not be copied or moved. Both the contents and address of a `PyMutex` are meaningful, and it must remain at a fixed, writable location in memory.

> **Note**
>
> A `PyMutex` currently occupies one byte, but the size should be considered unstable. The size may change in future Python releases without a deprecation period.

Ajouté dans la version 3.13.

void **PyMutex_Lock**(*PyMutex* *m)

Lock mutex *m*. If another thread has already locked it, the calling thread will block until the mutex is unlocked. While blocked, the thread will temporarily detach the *thread state* if one exists.

Ajouté dans la version 3.13.

void **PyMutex_Unlock**(*PyMutex* *m)

Unlock mutex *m*. The mutex must be locked --- otherwise, the function will issue a fatal error.

Ajouté dans la version 3.13.

int **PyMutex_IsLocked**(*PyMutex* *m)

Returns non-zero if the mutex *m* is currently locked, zero otherwise.

> **Note**
>
> This function is intended for use in assertions and debugging only and should not be used to make concurrency control decisions, as the lock state may change immediately after the check.

Ajouté dans la version 3.14.

10.12.1 Python Critical Section API

The critical section API provides a deadlock avoidance layer on top of per-object locks for *free-threaded* CPython. They are intended to replace reliance on the *global interpreter lock*, and are no-ops in versions of Python with the global interpreter lock.

Critical sections are intended to be used for custom types implemented in C-API extensions. They should generally not be used with built-in types like `list` and `dict` because their public C-APIs already use critical sections internally, with the notable exception of `PyDict_Next()`, which requires critical section to be acquired externally.

Critical sections avoid deadlocks by implicitly suspending active critical sections, hence, they do not provide exclusive access such as provided by traditional locks like `PyMutex`. When a critical section is started, the per-object lock for the object is acquired. If the code executed inside the critical section calls C-API functions then it can suspend the critical section thereby releasing the per-object lock, so other threads can acquire the per-object lock for the same object.

Variants that accept `PyMutex` pointers rather than Python objects are also available. Use these variants to start a critical section in a situation where there is no `PyObject` -- for example, when working with a C type that does not extend or wrap `PyObject` but still needs to call into the C API in a manner that might lead to deadlocks.

The functions and structs used by the macros are exposed for cases where C macros are not available. They should only be used as in the given macro expansions. Note that the sizes and contents of the structures may change in future Python versions.

> **Note**
>
> Operations that need to lock two objects at once must use `Py_BEGIN_CRITICAL_SECTION2`. You *cannot* use nested critical sections to lock more than one object at once, because the inner critical section may suspend the outer critical sections. This API does not provide a way to lock more than two objects at once.

Exemple d'utilisation :

```
static PyObject *
set_field(MyObject *self, PyObject *value)
{
   Py_BEGIN_CRITICAL_SECTION(self);
   Py_SETREF(self->field, Py_XNewRef(value));
   Py_END_CRITICAL_SECTION();
   Py_RETURN_NONE;
}
```

In the above example, *Py_SETREF* calls *Py_DECREF*, which can call arbitrary code through an object's deallocation function. The critical section API avoids potential deadlocks due to reentrancy and lock ordering by allowing the runtime to temporarily suspend the critical section if the code triggered by the finalizer blocks and calls *PyEval_SaveThread()*.

Py_BEGIN_CRITICAL_SECTION(op)

Acquires the per-object lock for the object *op* and begins a critical section.

In the free-threaded build, this macro expands to :

```
{
    PyCriticalSection _py_cs;
    PyCriticalSection_Begin(&_py_cs, (PyObject*)(op))
```

In the default build, this macro expands to `{`.

Ajouté dans la version 3.13.

Py_BEGIN_CRITICAL_SECTION_MUTEX(m)

Locks the mutex *m* and begins a critical section.

In the free-threaded build, this macro expands to :

```
{
     PyCriticalSection _py_cs;
     PyCriticalSection_BeginMutex(&_py_cs, m)
```

Note that unlike *Py_BEGIN_CRITICAL_SECTION*, there is no cast for the argument of the macro - it must be a *PyMutex* pointer.

On the default build, this macro expands to `{`.

Ajouté dans la version 3.14.

Py_END_CRITICAL_SECTION()

Ends the critical section and releases the per-object lock.

In the free-threaded build, this macro expands to :

```
    PyCriticalSection_End(&_py_cs);
}
```

In the default build, this macro expands to `}`.

Ajouté dans la version 3.13.

Py_BEGIN_CRITICAL_SECTION2(a, b)

Acquires the per-objects locks for the objects *a* and *b* and begins a critical section. The locks are acquired in a consistent order (lowest address first) to avoid lock ordering deadlocks.

In the free-threaded build, this macro expands to :

```
{
    PyCriticalSection2 _py_cs2;
    PyCriticalSection2_Begin(&_py_cs2, (PyObject*)(a), (PyObject*)(b))
```

In the default build, this macro expands to `{`.

Ajouté dans la version 3.13.

Py_BEGIN_CRITICAL_SECTION2_MUTEX(m1, m2)

Locks the mutexes *m1* and *m2* and begins a critical section.

In the free-threaded build, this macro expands to :

```
{
     PyCriticalSection2 _py_cs2;
     PyCriticalSection2_BeginMutex(&_py_cs2, m1, m2)
```

Note that unlike *Py_BEGIN_CRITICAL_SECTION2*, there is no cast for the arguments of the macro - they must be *PyMutex* pointers.

On the default build, this macro expands to `{`.

Ajouté dans la version 3.14.

Py_END_CRITICAL_SECTION2()

Ends the critical section and releases the per-object locks.

In the free-threaded build, this macro expands to :

```
    PyCriticalSection2_End(&_py_cs2);
}
```

In the default build, this macro expands to `}`.

Ajouté dans la version 3.13.

CHAPITRE 11

Python Initialization Configuration

11.1 PyInitConfig C API

Ajouté dans la version 3.14.

Python can be initialized with *Py_InitializeFromInitConfig()*.

The *Py_RunMain()* function can be used to write a customized Python program.

See also *Initialization, Finalization, and Threads*.

Voir aussi

PEP 741 ”Python Configuration C API”.

11.1.1 Example

Example of customized Python always running with the Python Development Mode enabled ; return `-1` on error :

```
int init_python(void)
{
    PyInitConfig *config = PyInitConfig_Create();
    if (config == NULL) {
        printf("PYTHON INIT ERROR: memory allocation failed\n");
        return -1;
    }

    // Enable the Python Development Mode
    if (PyInitConfig_SetInt(config, "dev_mode", 1) < 0) {
        goto error;
    }

    // Initialize Python with the configuration
    if (Py_InitializeFromInitConfig(config) < 0) {
        goto error;
    }
    PyInitConfig_Free(config);
```

(suite sur la page suivante)

(suite de la page précédente)

```
    return 0;

error:
    {
        // Display the error message.
        //
        // This uncommon braces style is used, because you cannot make
        // goto targets point to variable declarations.
        const char *err_msg;
        (void)PyInitConfig_GetError(config, &err_msg);
        printf("PYTHON INIT ERROR: %s\n", err_msg);
        PyInitConfig_Free(config);
        return -1;
    }
}
```

11.1.2 Create Config

struct **PyInitConfig**

Opaque structure to configure the Python initialization.

PyInitConfig ***PyInitConfig_Create**(void)

Create a new initialization configuration using *Isolated Configuration* default values.

It must be freed by *PyInitConfig_Free()*.

Return `NULL` on memory allocation failure.

void **PyInitConfig_Free**(*PyInitConfig* *config)

Free memory of the initialization configuration *config*.

If *config* is `NULL`, no operation is performed.

11.1.3 Error Handling

int **PyInitConfig_GetError**(*PyInitConfig* *config, const char **err_msg)

Get the *config* error message.
— Set **err_msg* and return `1` if an error is set.
— Set **err_msg* to `NULL` and return `0` otherwise.
An error message is an UTF-8 encoded string.

If *config* has an exit code, format the exit code as an error message.

The error message remains valid until another `PyInitConfig` function is called with *config*. The caller doesn't have to free the error message.

int **PyInitConfig_GetExitCode**(*PyInitConfig* *config, int *exitcode)

Get the *config* exit code.
— Set **exitcode* and return `1` if *config* has an exit code set.
— Return `0` if *config* has no exit code set.
Only the `Py_InitializeFromInitConfig()` function can set an exit code if the `parse_argv` option is non-zero.

An exit code can be set when parsing the command line failed (exit code `2`) or when a command line option asks to display the command line help (exit code `0`).

11.1.4 Get Options

The configuration option *name* parameter must be a non-NULL null-terminated UTF-8 encoded string. See *Configuration Options*.

int **PyInitConfig_HasOption**(*PyInitConfig* *config, const char *name)

Test if the configuration has an option called *name*.

Return `1` if the option exists, or return `0` otherwise.

int **PyInitConfig_GetInt**(*PyInitConfig* *config, const char *name, int64_t *value)

Get an integer configuration option.
- Set **value*, and return `0` on success.
- Set an error in *config* and return `-1` on error.

int **PyInitConfig_GetStr**(*PyInitConfig* *config, const char *name, char **value)

Get a string configuration option as a null-terminated UTF-8 encoded string.
- Set **value*, and return `0` on success.
- Set an error in *config* and return `-1` on error.

**value* can be set to `NULL` if the option is an optional string and the option is unset.

On success, the string must be released with `free(value)` if it's not `NULL`.

int **PyInitConfig_GetStrList**(*PyInitConfig* *config, const char *name, size_t *length, char ***items)

Get a string list configuration option as an array of null-terminated UTF-8 encoded strings.
- Set **length* and **value*, and return `0` on success.
- Set an error in *config* and return `-1` on error.

On success, the string list must be released with `PyInitConfig_FreeStrList(length, items)`.

void **PyInitConfig_FreeStrList**(size_t length, char **items)

Free memory of a string list created by `PyInitConfig_GetStrList()`.

11.1.5 Set Options

The configuration option *name* parameter must be a non-NULL null-terminated UTF-8 encoded string. See *Configuration Options*.

Some configuration options have side effects on other options. This logic is only implemented when `Py_InitializeFromInitConfig()` is called, not by the "Set" functions below. For example, setting `dev_mode` to `1` does not set `faulthandler` to `1`.

int **PyInitConfig_SetInt**(*PyInitConfig* *config, const char *name, int64_t value)

Set an integer configuration option.
- Return `0` on success.
- Set an error in *config* and return `-1` on error.

int **PyInitConfig_SetStr**(*PyInitConfig* *config, const char *name, const char *value)

Set a string configuration option from a null-terminated UTF-8 encoded string. The string is copied.
- Return `0` on success.
- Set an error in *config* and return `-1` on error.

int **PyInitConfig_SetStrList**(*PyInitConfig* *config, const char *name, size_t length, char *const *items)

Set a string list configuration option from an array of null-terminated UTF-8 encoded strings. The string list is copied.
- Return `0` on success.
- Set an error in *config* and return `-1` on error.

11.1.6 Module

int **PyInitConfig_AddModule**(*PyInitConfig* *config, const char *name, *PyObject* *(*initfunc)(void))

Add a built-in extension module to the table of built-in modules.

The new module can be imported by the name *name*, and uses the function *initfunc* as the initialization function called on the first attempted import.
— Return `0` on success.
— Set an error in *config* and return `-1` on error.
If Python is initialized multiple times, `PyInitConfig_AddModule()` must be called at each Python initialization.

Similar to the *PyImport_AppendInittab()* function.

11.1.7 Initialize Python

int **Py_InitializeFromInitConfig**(*PyInitConfig* *config)

Initialize Python from the initialization configuration.
— Return `0` on success.
— Set an error in *config* and return `-1` on error.
— Set an exit code in *config* and return `-1` if Python wants to exit.
See `PyInitConfig_GetExitcode()` for the exit code case.

11.2 Configuration Options

Option	PyConfig/PyPreConfig member	Type	Visibility
"allocator"	*allocator*	int	Read-only
"argv"	*argv*	list[str]	Public
"base_exec_prefix"	*base_exec_prefix*	str	Public
"base_executable"	*base_executable*	str	Public
"base_prefix"	*base_prefix*	str	Public
"buffered_stdio"	*buffered_stdio*	bool	Read-only
"bytes_warning"	*bytes_warning*	int	Public
"check_hash_pycs_mode"	*check_hash_pycs_mode*	str	Read-only
"code_debug_ranges"	*code_debug_ranges*	bool	Read-only
"coerce_c_locale"	*coerce_c_locale*	bool	Read-only
"coerce_c_locale_warn"	*coerce_c_locale_warn*	bool	Read-only
"configure_c_stdio"	*configure_c_stdio*	bool	Read-only
"configure_locale"	*configure_locale*	bool	Read-only
"cpu_count"	*cpu_count*	int	Public
"dev_mode"	*dev_mode*	bool	Read-only
"dump_refs"	*dump_refs*	bool	Read-only
"dump_refs_file"	*dump_refs_file*	str	Read-only
"exec_prefix"	*exec_prefix*	str	Public
"executable"	*executable*	str	Public
"faulthandler"	*faulthandler*	bool	Read-only
"filesystem_encoding"	*filesystem_encoding*	str	Read-only
"filesystem_errors"	*filesystem_errors*	str	Read-only
"hash_seed"	*hash_seed*	int	Read-only
"home"	*home*	str	Read-only
"import_time"	*import_time*	int	Read-only
"inspect"	*inspect*	bool	Public
"install_signal_handlers"	*install_signal_handlers*	bool	Read-only
"int_max_str_digits"	*int_max_str_digits*	int	Public
"interactive"	*interactive*	bool	Public
"isolated"	*isolated*	bool	Read-only
"legacy_windows_fs_encoding"	*legacy_windows_fs_encoding*	bool	Read-only
"legacy_windows_stdio"	*legacy_windows_stdio*	bool	Read-only
"malloc_stats"	*malloc_stats*	bool	Read-only
"module_search_paths"	*module_search_paths*	list[str]	Public

suite sur la page suivante

Tableau 1 – suite de la page précédente

Option	PyConfig/PyPreConfig member	Type	Visibility
"optimization_level"	*optimization_level*	int	Public
"orig_argv"	*orig_argv*	list[str]	Read-only
"parse_argv"	*parse_argv*	bool	Read-only
"parser_debug"	*parser_debug*	bool	Public
"pathconfig_warnings"	*pathconfig_warnings*	bool	Read-only
"perf_profiling"	*perf_profiling*	bool	Read-only
"platlibdir"	*platlibdir*	str	Public
"prefix"	*prefix*	str	Public
"program_name"	*program_name*	str	Read-only
"pycache_prefix"	*pycache_prefix*	str	Public
"quiet"	*quiet*	bool	Public
"run_command"	*run_command*	str	Read-only
"run_filename"	*run_filename*	str	Read-only
"run_module"	*run_module*	str	Read-only
"run_presite"	*run_presite*	str	Read-only
"safe_path"	*safe_path*	bool	Read-only
"show_ref_count"	*show_ref_count*	bool	Read-only
"site_import"	*site_import*	bool	Read-only
"skip_source_first_line"	*skip_source_first_line*	bool	Read-only
"stdio_encoding"	*stdio_encoding*	str	Read-only
"stdio_errors"	*stdio_errors*	str	Read-only
"stdlib_dir"	*stdlib_dir*	str	Public
"tracemalloc"	*tracemalloc*	int	Read-only
"use_environment"	*use_environment*	bool	Public
"use_frozen_modules"	*use_frozen_modules*	bool	Read-only
"use_hash_seed"	*use_hash_seed*	bool	Read-only
"use_system_logger"	*use_system_logger*	bool	Read-only
"user_site_directory"	*user_site_directory*	bool	Read-only
"utf8_mode"	*utf8_mode*	bool	Read-only
"verbose"	*verbose*	int	Public
"warn_default_encoding"	*warn_default_encoding*	bool	Read-only
"warnoptions"	*warnoptions*	list[str]	Public
"write_bytecode"	*write_bytecode*	bool	Public
"xoptions"	*xoptions*	dict[str, str]	Public
"_pystats"	*_pystats*	bool	Read-only

Visibility :
- Public : Can by get by *PyConfig_Get()* and set by *PyConfig_Set()*.
- Read-only : Can by get by *PyConfig_Get()*, but cannot be set by *PyConfig_Set()*.

11.3 Runtime Python configuration API

At runtime, it's possible to get and set configuration options using *PyConfig_Get()* and *PyConfig_Set()* functions.

The configuration option *name* parameter must be a non-NULL null-terminated UTF-8 encoded string. See *Configuration Options*.

Some options are read from the `sys` attributes. For example, the option `"argv"` is read from `sys.argv`.

PyObject ***PyConfig_Get**(const char *name)

Get the current runtime value of a configuration option as a Python object.
- Return a new reference on success.
- Set an exception and return `NULL` on error.

The object type depends on the configuration option. It can be :
- `bool`

— `int`
— `str`
— `list[str]`
— `dict[str, str]`

The caller must have an *attached thread state*. The function cannot be called before Python initialization nor after Python finalization.

Ajouté dans la version 3.14.

int **PyConfig_GetInt**(const char *name, int *value)

Similar to `PyConfig_Get()`, but get the value as a C int.
— Return `0` on success.
— Set an exception and return `-1` on error.
Ajouté dans la version 3.14.

PyObject ***PyConfig_Names**(void)

Get all configuration option names as a `frozenset`.
— Return a new reference on success.
— Set an exception and return `NULL` on error.
The caller must have an *attached thread state*. The function cannot be called before Python initialization nor after Python finalization.

Ajouté dans la version 3.14.

int **PyConfig_Set**(const char *name, *PyObject* *value)

Set the current runtime value of a configuration option.
— Raise a `ValueError` if there is no option *name*.
— Raise a `ValueError` if *value* is an invalid value.
— Raise a `ValueError` if the option is read-only (cannot be set).
— Raise a `TypeError` if *value* has not the proper type.
The caller must have an *attached thread state*. The function cannot be called before Python initialization nor after Python finalization.

Raises an auditing event `cpython.PyConfig_Set` with arguments `name`, `value`.

Ajouté dans la version 3.14.

11.4 PyConfig C API

Ajouté dans la version 3.8.

Python can be initialized with `Py_InitializeFromConfig()` and the `PyConfig` structure. It can be preinitialized with `Py_PreInitialize()` and the `PyPreConfig` structure.

There are two kinds of configuration :
— The *Python Configuration* can be used to build a customized Python which behaves as the regular Python. For example, environment variables and command line arguments are used to configure Python.
— The *Isolated Configuration* can be used to embed Python into an application. It isolates Python from the system. For example, environment variables are ignored, the LC_CTYPE locale is left unchanged and no signal handler is registered.

The `Py_RunMain()` function can be used to write a customized Python program.

See also *Initialization, Finalization, and Threads*.

Voir aussi

PEP 587 ”Python Initialization Configuration”.

11.4.1 Example

Example of customized Python always running in isolated mode :

```
int main(int argc, char **argv)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config.isolated = 1;

    /* Decode command line arguments.
       Implicitly preinitialize Python (in isolated mode). */
    status = PyConfig_SetBytesArgv(&config, argc, argv);
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);

    return Py_RunMain();

exception:
    PyConfig_Clear(&config);
    if (PyStatus_IsExit(status)) {
        return status.exitcode;
    }
    /* Display the error message and exit the process with
       non-zero exit code */
    Py_ExitStatusException(status);
}
```

11.4.2 PyWideStringList

type **PyWideStringList**

List of `wchar_t*` strings.

If *length* is non-zero, *items* must be non-`NULL` and all strings must be non-`NULL`.

Methods :

PyStatus **PyWideStringList_Append**(*PyWideStringList* *list, const wchar_t *item)

Append *item* to *list*.

Python must be preinitialized to call this function.

PyStatus **PyWideStringList_Insert**(*PyWideStringList* *list, *Py_ssize_t* index, const wchar_t *item)

Insert *item* into *list* at *index*.

If *index* is greater than or equal to *list* length, append *item* to *list*.

index must be greater than or equal to `0`.

Python must be preinitialized to call this function.

Structure fields :

Py_ssize_t **length**

List length.

wchar_t **__items__

List items.

11.4.3 PyStatus

type **PyStatus**

Structure to store an initialization function status : success, error or exit.

For an error, it can store the C function name which created the error.

Structure fields :

int **exitcode**

Exit code. Argument passed to `exit()`.

const char ***err_msg**

Error message.

const char ***func**

Name of the function which created an error, can be `NULL`.

Functions to create a status :

PyStatus **PyStatus_Ok** (void)

Success.

PyStatus **PyStatus_Error** (const char *err_msg)

Initialization error with a message.

err_msg must not be `NULL`.

PyStatus **PyStatus_NoMemory** (void)

Memory allocation failure (out of memory).

PyStatus **PyStatus_Exit** (int exitcode)

Exit Python with the specified exit code.

Functions to handle a status :

int **PyStatus_Exception** (*PyStatus* status)

Is the status an error or an exit? If true, the exception must be handled; by calling *Py_ExitStatusException()* for example.

int **PyStatus_IsError** (*PyStatus* status)

Is the result an error?

int **PyStatus_IsExit** (*PyStatus* status)

Is the result an exit?

void **Py_ExitStatusException** (*PyStatus* status)

Call `exit(exitcode)` if *status* is an exit. Print the error message and exit with a non-zero exit code if *status* is an error. Must only be called if `PyStatus_Exception(status)` is non-zero.

Note

Internally, Python uses macros which set `PyStatus.func`, whereas functions to create a status set `func` to `NULL`.

Exemple :

```
PyStatus alloc(void **ptr, size_t size)
{
    *ptr = PyMem_RawMalloc(size);
    if (*ptr == NULL) {
        return PyStatus_NoMemory();
    }
    return PyStatus_Ok();
}

int main(int argc, char **argv)
{
    void *ptr;
    PyStatus status = alloc(&ptr, 16);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
    PyMem_Free(ptr);
    return 0;
}
```

11.4.4 PyPreConfig

type **PyPreConfig**

Structure used to preinitialize Python.

Function to initialize a preconfiguration :

void **PyPreConfig_InitPythonConfig**(*PyPreConfig* *preconfig)

Initialize the preconfiguration with *Python Configuration*.

void **PyPreConfig_InitIsolatedConfig**(*PyPreConfig* *preconfig)

Initialize the preconfiguration with *Isolated Configuration*.

Structure fields :

int **allocator**

Name of the Python memory allocators :

- `PYMEM_ALLOCATOR_NOT_SET` (`0`) : don't change memory allocators (use defaults).
- `PYMEM_ALLOCATOR_DEFAULT` (`1`) : *default memory allocators*.
- `PYMEM_ALLOCATOR_DEBUG` (`2`) : *default memory allocators* with *debug hooks*.
- `PYMEM_ALLOCATOR_MALLOC` (`3`) : use `malloc()` of the C library.
- `PYMEM_ALLOCATOR_MALLOC_DEBUG` (`4`) : force usage of `malloc()` with *debug hooks*.
- `PYMEM_ALLOCATOR_PYMALLOC` (`5`) : *Python pymalloc memory allocator*.
- `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` (`6`) : *Python pymalloc memory allocator* with *debug hooks*.
- `PYMEM_ALLOCATOR_MIMALLOC` (`6`) : use `mimalloc`, a fast malloc replacement.
- `PYMEM_ALLOCATOR_MIMALLOC_DEBUG` (`7`) : use `mimalloc`, a fast malloc replacement with *debug hooks*.

`PYMEM_ALLOCATOR_PYMALLOC` and `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` are not supported if Python is `configured using --without-pymalloc`.

`PYMEM_ALLOCATOR_MIMALLOC` and `PYMEM_ALLOCATOR_MIMALLOC_DEBUG` are not supported if Python is `configured using --without-mimalloc` or if the underlying atomic support isn't available.

See *Memory Management*.

Default : `PYMEM_ALLOCATOR_NOT_SET`.

int **configure_locale**

Set the LC_CTYPE locale to the user preferred locale.

If equals to 0, set *coerce_c_locale* and *coerce_c_locale_warn* members to 0.

See the *locale encoding*.

Default : 1 in Python config, 0 in isolated config.

int **coerce_c_locale**

If equals to 2, coerce the C locale.

If equals to 1, read the LC_CTYPE locale to decide if it should be coerced.

See the *locale encoding*.

Default : -1 in Python config, 0 in isolated config.

int **coerce_c_locale_warn**

If non-zero, emit a warning if the C locale is coerced.

Default : -1 in Python config, 0 in isolated config.

int **dev_mode**

Python Development Mode : see *PyConfig.dev_mode*.

Default : -1 in Python mode, 0 in isolated mode.

int **isolated**

Isolated mode : see *PyConfig.isolated*.

Default : 0 in Python mode, 1 in isolated mode.

int **legacy_windows_fs_encoding**

If non-zero :
— Set *PyPreConfig.utf8_mode* to 0,
— Set *PyConfig.filesystem_encoding* to "mbcs",
— Set *PyConfig.filesystem_errors* to "replace".
Initialized from the PYTHONLEGACYWINDOWSFSENCODING environment variable value.

Only available on Windows. #ifdef MS_WINDOWS macro can be used for Windows specific code.

Default : 0.

int **parse_argv**

If non-zero, *Py_PreInitializeFromArgs()* and *Py_PreInitializeFromBytesArgs()* parse their argv argument the same way the regular Python parses command line arguments : see Command Line Arguments.

Default : 1 in Python config, 0 in isolated config.

int **use_environment**

Use environment variables ? See *PyConfig.use_environment*.

Default : 1 in Python config and 0 in isolated config.

int **utf8_mode**

If non-zero, enable the Python UTF-8 Mode.

Set to 0 or 1 by the -X utf8 command line option and the PYTHONUTF8 environment variable.

Also set to 1 if the LC_CTYPE locale is C or POSIX.

Default : -1 in Python config and 0 in isolated config.

11.4.5 Preinitialize Python with PyPreConfig

The preinitialization of Python :
— Set the Python memory allocators (*PyPreConfig.allocator*)
— Configure the LC_CTYPE locale (*locale encoding*)
— Set the Python UTF-8 Mode (*PyPreConfig.utf8_mode*)

The current preconfiguration (`PyPreConfig` type) is stored in `_PyRuntime.preconfig`.

Functions to preinitialize Python :

PyStatus **Py_PreInitialize**(const *PyPreConfig* *preconfig)

Preinitialize Python from *preconfig* preconfiguration.

preconfig must not be `NULL`.

PyStatus **Py_PreInitializeFromBytesArgs**(const *PyPreConfig* *preconfig, int argc, char *const *argv)

Preinitialize Python from *preconfig* preconfiguration.

Parse *argv* command line arguments (bytes strings) if *parse_argv* of *preconfig* is non-zero.

preconfig must not be `NULL`.

PyStatus **Py_PreInitializeFromArgs**(const *PyPreConfig* *preconfig, int argc, wchar_t *const *argv)

Preinitialize Python from *preconfig* preconfiguration.

Parse *argv* command line arguments (wide strings) if *parse_argv* of *preconfig* is non-zero.

preconfig must not be `NULL`.

The caller is responsible to handle exceptions (error or exit) using *PyStatus_Exception()* and *Py_ExitStatusException()*.

For *Python Configuration* (*PyPreConfig_InitPythonConfig()*), if Python is initialized with command line arguments, the command line arguments must also be passed to preinitialize Python, since they have an effect on the pre-configuration like encodings. For example, the `-X utf8` command line option enables the Python UTF-8 Mode.

`PyMem_SetAllocator()` can be called after *Py_PreInitialize()* and before *Py_InitializeFromConfig()* to install a custom memory allocator. It can be called before *Py_PreInitialize()* if *PyPreConfig.allocator* is set to `PYMEM_ALLOCATOR_NOT_SET`.

Python memory allocation functions like *PyMem_RawMalloc()* must not be used before the Python preinitialization, whereas calling directly `malloc()` and `free()` is always safe. *Py_DecodeLocale()* must not be called before the Python preinitialization.

Example using the preinitialization to enable the Python UTF-8 Mode :

```
PyStatus status;
PyPreConfig preconfig;
PyPreConfig_InitPythonConfig(&preconfig);

preconfig.utf8_mode = 1;

status = Py_PreInitialize(&preconfig);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}

/* at this point, Python speaks UTF-8 */

Py_Initialize();
/* ... use Python API here ... */
Py_Finalize();
```

11.4.6 PyConfig

type **PyConfig**

Structure containing most parameters to configure Python.

When done, the *PyConfig_Clear()* function must be used to release the configuration memory.

Structure methods :

void **PyConfig_InitPythonConfig**(*PyConfig* *config)

Initialize configuration with the *Python Configuration*.

void **PyConfig_InitIsolatedConfig**(*PyConfig* *config)

Initialize configuration with the *Isolated Configuration*.

PyStatus **PyConfig_SetString**(*PyConfig* *config, wchar_t *const *config_str, const wchar_t *str)

Copy the wide character string *str* into `*config_str`.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesString**(*PyConfig* *config, wchar_t *const *config_str, const char *str)

Decode *str* using *Py_DecodeLocale()* and set the result into `*config_str`.

Preinitialize Python if needed.

PyStatus **PyConfig_SetArgv**(*PyConfig* *config, int argc, wchar_t *const *argv)

Set command line arguments (*argv* member of *config*) from the *argv* list of wide character strings.

Preinitialize Python if needed.

PyStatus **PyConfig_SetBytesArgv**(*PyConfig* *config, int argc, char *const *argv)

Set command line arguments (*argv* member of *config*) from the *argv* list of bytes strings. Decode bytes using *Py_DecodeLocale()*.

Preinitialize Python if needed.

PyStatus **PyConfig_SetWideStringList**(*PyConfig* *config, *PyWideStringList* *list, *Py_ssize_t* length, wchar_t **items)

Set the list of wide strings *list* to *length* and *items*.

Preinitialize Python if needed.

PyStatus **PyConfig_Read**(*PyConfig* *config)

Read all Python configuration.

Fields which are already initialized are left unchanged.

Fields for *path configuration* are no longer calculated or modified when calling this function, as of Python 3.11.

The *PyConfig_Read()* function only parses *PyConfig.argv* arguments once : *PyConfig.parse_argv* is set to `2` after arguments are parsed. Since Python arguments are stripped from *PyConfig.argv*, parsing arguments twice would parse the application options as Python options.

Preinitialize Python if needed.

Modifié dans la version 3.10 : The *PyConfig.argv* arguments are now only parsed once, *PyConfig.parse_argv* is set to `2` after arguments are parsed, and arguments are only parsed if *PyConfig.parse_argv* equals `1`.

Modifié dans la version 3.11 : *PyConfig_Read()* no longer calculates all paths, and so fields listed under *Python Path Configuration* may no longer be updated until *Py_InitializeFromConfig()* is called.

void **PyConfig_Clear**(*PyConfig* *config)

Release configuration memory.

Most `PyConfig` methods *preinitialize Python* if needed. In that case, the Python preinitialization configuration (*PyPreConfig*) in based on the *PyConfig*. If configuration fields which are in common with *PyPreConfig* are tuned, they must be set before calling a *PyConfig* method :

— *PyConfig.dev_mode*
— *PyConfig.isolated*
— *PyConfig.parse_argv*
— *PyConfig.use_environment*

Moreover, if *PyConfig_SetArgv()* or *PyConfig_SetBytesArgv()* is used, this method must be called before other methods, since the preinitialization configuration depends on command line arguments (if

parse_argv is non-zero).

The caller of these methods is responsible to handle exceptions (error or exit) using `PyStatus_Exception()` and `Py_ExitStatusException()`.

Structure fields :

PyWideStringList **argv**

Set `sys.argv` command line arguments based on *argv*. These parameters are similar to those passed to the program's `main()` function with the difference that the first entry should refer to the script file to be executed rather than the executable hosting the Python interpreter. If there isn't a script that will be run, the first entry in *argv* can be an empty string.

Set *parse_argv* to `1` to parse *argv* the same way the regular Python parses Python command line arguments and then to strip Python arguments from *argv*.

If *argv* is empty, an empty string is added to ensure that `sys.argv` always exists and is never empty.

Default : `NULL`.

See also the *orig_argv* member.

int **safe_path**

If equals to zero, `Py_RunMain()` prepends a potentially unsafe path to `sys.path` at startup :
— If *argv[0]* is equal to `L"-m"` (`python -m module`), prepend the current working directory.
— If running a script (`python script.py`), prepend the script's directory. If it's a symbolic link, resolve symbolic links.
— Otherwise (`python -c code` and `python`), prepend an empty string, which means the current working directory.
Set to `1` by the `-P` command line option and the `PYTHONSAFEPATH` environment variable.

Default : `0` in Python config, `1` in isolated config.

Ajouté dans la version 3.11.

wchar_t ***base_exec_prefix**

`sys.base_exec_prefix`.

Default : `NULL`.

Part of the *Python Path Configuration* output.

See also *PyConfig.exec_prefix*.

wchar_t ***base_executable**

Python base executable : `sys._base_executable`.

Set by the `__PYVENV_LAUNCHER__` environment variable.

Set from *PyConfig.executable* if `NULL`.

Default : `NULL`.

Part of the *Python Path Configuration* output.

See also *PyConfig.executable*.

wchar_t ***base_prefix**

`sys.base_prefix`.

Default : `NULL`.

Part of the *Python Path Configuration* output.

See also *PyConfig.prefix*.

int **buffered_stdio**

If equals to `0` and *configure_c_stdio* is non-zero, disable buffering on the C streams stdout and stderr.

Set to `0` by the `-u` command line option and the `PYTHONUNBUFFERED` environment variable.

stdin is always opened in buffered mode.

Default : `1`.

int **bytes_warning**

If equals to `1`, issue a warning when comparing `bytes` or `bytearray` with `str`, or comparing `bytes` with `int`.

If equal or greater to `2`, raise a `BytesWarning` exception in these cases.

Incremented by the `-b` command line option.

Default : `0`.

int **warn_default_encoding**

If non-zero, emit a `EncodingWarning` warning when `io.TextIOWrapper` uses its default encoding. See io-encoding-warning for details.

Default : `0`.

Ajouté dans la version 3.10.

int **code_debug_ranges**

If equals to `0`, disables the inclusion of the end line and column mappings in code objects. Also disables traceback printing carets to specific error locations.

Set to `0` by the `PYTHONNODEBUGRANGES` environment variable and by the `-X no_debug_ranges` command line option.

Default : `1`.

Ajouté dans la version 3.11.

wchar_t ***check_hash_pycs_mode**

Control the validation behavior of hash-based `.pyc` files : value of the `--check-hash-based-pycs` command line option.

Valid values :
— `L"always"` : Hash the source file for invalidation regardless of value of the 'check_source' flag.
— `L"never"` : Assume that hash-based pycs always are valid.
— `L"default"` : The 'check_source' flag in hash-based pycs determines invalidation.
Default : `L"default"`.

See also **PEP 552** "Deterministic pycs".

int **configure_c_stdio**

If non-zero, configure C standard streams :
— On Windows, set the binary mode (`O_BINARY`) on stdin, stdout and stderr.
— If *`buffered_stdio`* equals zero, disable buffering of stdin, stdout and stderr streams.
— If *`interactive`* is non-zero, enable stream buffering on stdin and stdout (only stdout on Windows).
Default : `1` in Python config, `0` in isolated config.

int **dev_mode**

If non-zero, enable the Python Development Mode.

Set to `1` by the `-X dev` option and the `PYTHONDEVMODE` environment variable.

Default : `-1` in Python mode, `0` in isolated mode.

int **dump_refs**

Dump Python references ?

If non-zero, dump all objects which are still alive at exit.

Set to `1` by the `PYTHONDUMPREFS` environment variable.

Needs a special build of Python with the `Py_TRACE_REFS` macro defined : see the `configure`

`--with-trace-refs` option.

Default : `0`.

wchar_t ***dump_refs_file**

Filename where to dump Python references.

Set by the `PYTHONDUMPREFSFILE` environment variable.

Default : `NULL`.

Ajouté dans la version 3.11.

wchar_t ***exec_prefix**

The site-specific directory prefix where the platform-dependent Python files are installed : `sys.exec_prefix`.

Default : `NULL`.

Part of the *Python Path Configuration* output.

See also *`PyConfig.base_exec_prefix`*.

wchar_t ***executable**

The absolute path of the executable binary for the Python interpreter : `sys.executable`.

Default : `NULL`.

Part of the *Python Path Configuration* output.

See also *`PyConfig.base_executable`*.

int **faulthandler**

Enable faulthandler ?

If non-zero, call `faulthandler.enable()` at startup.

Set to `1` by `-X faulthandler` and the `PYTHONFAULTHANDLER` environment variable.

Default : `-1` in Python mode, `0` in isolated mode.

wchar_t ***filesystem_encoding**

Filesystem encoding : `sys.getfilesystemencoding()`.

On macOS, Android and VxWorks : use `"utf-8"` by default.

On Windows : use `"utf-8"` by default, or `"mbcs"` if *`legacy_windows_fs_encoding`* of *`PyPreConfig`* is non-zero.

Default encoding on other platforms :
— `"utf-8"` if *`PyPreConfig.utf8_mode`* is non-zero.
— `"ascii"` if Python detects that `nl_langinfo(CODESET)` announces the ASCII encoding, whereas the `mbstowcs()` function decodes from a different encoding (usually Latin1).
— `"utf-8"` if `nl_langinfo(CODESET)` returns an empty string.
— Otherwise, use the *locale encoding* : `nl_langinfo(CODESET)` result.
At Python startup, the encoding name is normalized to the Python codec name. For example, `"ANSI_X3.4-1968"` is replaced with `"ascii"`.

See also the *`filesystem_errors`* member.

wchar_t ***filesystem_errors**

Filesystem error handler : `sys.getfilesystemencodeerrors()`.

On Windows : use `"surrogatepass"` by default, or `"replace"` if *`legacy_windows_fs_encoding`* of *`PyPreConfig`* is non-zero.

On other platforms : use `"surrogateescape"` by default.

Supported error handlers :
— `"strict"`

— "surrogateescape"
— "surrogatepass" (only supported with the UTF-8 encoding)
See also the *filesystem_encoding* member.

int **use_frozen_modules**

If non-zero, use frozen modules.

Set by the PYTHON_FROZEN_MODULES environment variable.

Default : 1 in a release build, or 0 in a debug build.

unsigned long **hash_seed**

int **use_hash_seed**

Randomized hash function seed.

If *use_hash_seed* is zero, a seed is chosen randomly at Python startup, and *hash_seed* is ignored.

Set by the PYTHONHASHSEED environment variable.

Default *use_hash_seed* value : -1 in Python mode, 0 in isolated mode.

wchar_t ***home**

Set the default Python "home" directory, that is, the location of the standard Python libraries (see PYTHONHOME).

Set by the PYTHONHOME environment variable.

Default : NULL.

Part of the *Python Path Configuration* input.

int **import_time**

If 1, profile import time. If 2, include additional output that indicates when an imported module has already been loaded.

Set by the -X importtime option and the PYTHONPROFILEIMPORTTIME environment variable.

Default : 0.

Modifié dans la version 3.14 : Added support for import_time = 2

int **inspect**

Enter interactive mode after executing a script or a command.

If greater than 0, enable inspect : when a script is passed as first argument or the -c option is used, enter interactive mode after executing the script or the command, even when sys.stdin does not appear to be a terminal.

Incremented by the -i command line option. Set to 1 if the PYTHONINSPECT environment variable is non-empty.

Default : 0.

int **install_signal_handlers**

Install Python signal handlers ?

Default : 1 in Python mode, 0 in isolated mode.

int **interactive**

If greater than 0, enable the interactive mode (REPL).

Incremented by the -i command line option.

Default : 0.

int **int_max_str_digits**

Configures the integer string conversion length limitation. An initial value of -1 means the value

will be taken from the command line or environment or otherwise default to 4300 (`sys.int_info.default_max_str_digits`). A value of `0` disables the limitation. Values greater than zero but less than 640 (`sys.int_info.str_digits_check_threshold`) are unsupported and will produce an error.

Configured by the `-X int_max_str_digits` command line flag or the `PYTHONINTMAXSTRDIGITS` environment variable.

Default : `-1` in Python mode. 4300 (`sys.int_info.default_max_str_digits`) in isolated mode.

Ajouté dans la version 3.12.

int **cpu_count**

If the value of *cpu_count* is not `-1` then it will override the return values of `os.cpu_count()`, `os.process_cpu_count()`, and `multiprocessing.cpu_count()`.

Configured by the `-X cpu_count=n|default` command line flag or the `PYTHON_CPU_COUNT` environment variable.

Default : `-1`.

Ajouté dans la version 3.13.

int **isolated**

If greater than `0`, enable isolated mode :
- Set *safe_path* to `1` : don't prepend a potentially unsafe path to `sys.path` at Python startup, such as the current directory, the script's directory or an empty string.
- Set *use_environment* to `0` : ignore `PYTHON` environment variables.
- Set *user_site_directory* to `0` : don't add the user site directory to `sys.path`.
- Python REPL doesn't import `readline` nor enable default readline configuration on interactive prompts.

Set to `1` by the `-I` command line option.

Default : `0` in Python mode, `1` in isolated mode.

See also the *Isolated Configuration* and *PyPreConfig.isolated*.

int **legacy_windows_stdio**

If non-zero, use `io.FileIO` instead of `io._WindowsConsoleIO` for `sys.stdin`, `sys.stdout` and `sys.stderr`.

Set to `1` if the `PYTHONLEGACYWINDOWSSTDIO` environment variable is set to a non-empty string.

Only available on Windows. `#ifdef MS_WINDOWS` macro can be used for Windows specific code.

Default : `0`.

See also the **PEP 528** (Change Windows console encoding to UTF-8).

int **malloc_stats**

If non-zero, dump statistics on *Python pymalloc memory allocator* at exit.

Set to `1` by the `PYTHONMALLOCSTATS` environment variable.

The option is ignored if Python is `configured using the --without-pymalloc option`.

Default : `0`.

wchar_t ***platlibdir**

Platform library directory name : `sys.platlibdir`.

Set by the `PYTHONPLATLIBDIR` environment variable.

Default : value of the `PLATLIBDIR` macro which is set by the `configure --with-platlibdir option` (default : `"lib"`, or `"DLLs"` on Windows).

Part of the *Python Path Configuration* input.

Ajouté dans la version 3.9.

Modifié dans la version 3.11 : This macro is now used on Windows to locate the standard library extension modules, typically under `DLLs`. However, for compatibility, note that this value is ignored for any non-standard layouts, including in-tree builds and virtual environments.

wchar_t ***pythonpath_env**

Module search paths (`sys.path`) as a string separated by `DELIM` (`os.pathsep`).

Set by the `PYTHONPATH` environment variable.

Default : `NULL`.

Part of the *Python Path Configuration* input.

PyWideStringList **module_search_paths**

int **module_search_paths_set**

Module search paths : `sys.path`.

If *module_search_paths_set* is equal to `0`, *Py_InitializeFromConfig()* will replace *module_search_paths* and sets *module_search_paths_set* to `1`.

Default : empty list (`module_search_paths`) and `0` (`module_search_paths_set`).

Part of the *Python Path Configuration* output.

int **optimization_level**

Compilation optimization level :
— `0` : Peephole optimizer, set `__debug__` to `True`.
— `1` : Level 0, remove assertions, set `__debug__` to `False`.
— `2` : Level 1, strip docstrings.
Incremented by the `-O` command line option. Set to the `PYTHONOPTIMIZE` environment variable value.

Default : `0`.

PyWideStringList **orig_argv**

The list of the original command line arguments passed to the Python executable : `sys.orig_argv`.

If *orig_argv* list is empty and *argv* is not a list only containing an empty string, *PyConfig_Read()* copies *argv* into *orig_argv* before modifying *argv* (if *parse_argv* is non-zero).

See also the *argv* member and the *Py_GetArgcArgv()* function.

Default : empty list.

Ajouté dans la version 3.10.

int **parse_argv**

Parse command line arguments ?

If equals to `1`, parse *argv* the same way the regular Python parses command line arguments, and strip Python arguments from *argv*.

The *PyConfig_Read()* function only parses *PyConfig.argv* arguments once : *PyConfig.parse_argv* is set to `2` after arguments are parsed. Since Python arguments are stripped from *PyConfig.argv*, parsing arguments twice would parse the application options as Python options.

Default : `1` in Python mode, `0` in isolated mode.

Modifié dans la version 3.10 : The *PyConfig.argv* arguments are now only parsed if *PyConfig.parse_argv* equals to `1`.

int **parser_debug**

Parser debug mode. If greater than `0`, turn on parser debugging output (for expert only, depending on compilation options).

Incremented by the `-d` command line option. Set to the `PYTHONDEBUG` environment variable value.

Needs a debug build of Python (the `Py_DEBUG` macro must be defined).

Default : `0`.

int **pathconfig_warnings**

If non-zero, calculation of path configuration is allowed to log warnings into `stderr`. If equals to `0`, suppress these warnings.

Default : `1` in Python mode, `0` in isolated mode.

Part of the *Python Path Configuration* input.

Modifié dans la version 3.11 : Now also applies on Windows.

wchar_t ***prefix**

The site-specific directory prefix where the platform independent Python files are installed : `sys.prefix`.

Default : `NULL`.

Part of the *Python Path Configuration* output.

See also *`PyConfig.base_prefix`*.

wchar_t ***program_name**

Program name used to initialize *`executable`* and in early error messages during Python initialization.
- On macOS, use `PYTHONEXECUTABLE` environment variable if set.
- If the `WITH_NEXT_FRAMEWORK` macro is defined, use `__PYVENV_LAUNCHER__` environment variable if set.
- Use `argv[0]` of *`argv`* if available and non-empty.
- Otherwise, use `L"python"` on Windows, or `L"python3"` on other platforms.

Default : `NULL`.

Part of the *Python Path Configuration* input.

wchar_t ***pycache_prefix**

Directory where cached `.pyc` files are written : `sys.pycache_prefix`.

Set by the `-X pycache_prefix=PATH` command line option and the `PYTHONPYCACHEPREFIX` environment variable. The command-line option takes precedence.

If `NULL`, `sys.pycache_prefix` is set to `None`.

Default : `NULL`.

int **quiet**

Quiet mode. If greater than `0`, don't display the copyright and version at Python startup in interactive mode.

Incremented by the `-q` command line option.

Default : `0`.

wchar_t ***run_command**

Value of the `-c` command line option.

Used by *`Py_RunMain()`*.

Default : `NULL`.

wchar_t ***run_filename**

Filename passed on the command line : trailing command line argument without `-c` or `-m`. It is used by the *`Py_RunMain()`* function.

For example, it is set to `script.py` by the `python3 script.py arg` command line.

See also the *`PyConfig.skip_source_first_line`* option.

Default : `NULL`.

wchar_t ***run_module**

Value of the `-m` command line option.

Used by *Py_RunMain()*.

Default : `NULL`.

wchar_t ***run_presite**

`package.module` path to module that should be imported before `site.py` is run.

Set by the `-X presite=package.module` command-line option and the `PYTHON_PRESITE` environment variable. The command-line option takes precedence.

Needs a debug build of Python (the `Py_DEBUG` macro must be defined).

Default : `NULL`.

int **show_ref_count**

Show total reference count at exit (excluding *immortal* objects) ?

Set to `1` by `-X showrefcount` command line option.

Needs a debug build of Python (the `Py_REF_DEBUG` macro must be defined).

Default : `0`.

int **site_import**

Import the `site` module at startup ?

If equal to zero, disable the import of the module site and the site-dependent manipulations of `sys.path` that it entails.

Also disable these manipulations if the `site` module is explicitly imported later (call `site.main()` if you want them to be triggered).

Set to `0` by the `-S` command line option.

`sys.flags.no_site` is set to the inverted value of *site_import*.

Default : `1`.

int **skip_source_first_line**

If non-zero, skip the first line of the *PyConfig.run_filename* source.

It allows the usage of non-Unix forms of `#!cmd`. This is intended for a DOS specific hack only.

Set to `1` by the `-x` command line option.

Default : `0`.

wchar_t ***stdio_encoding**

wchar_t ***stdio_errors**

Encoding and encoding errors of `sys.stdin`, `sys.stdout` and `sys.stderr` (but `sys.stderr` always uses `"backslashreplace"` error handler).

Use the `PYTHONIOENCODING` environment variable if it is non-empty.

Default encoding :
— `"UTF-8"` if *PyPreConfig.utf8_mode* is non-zero.
— Otherwise, use the *locale encoding*.
Default error handler :
— On Windows : use `"surrogateescape"`.
— `"surrogateescape"` if *PyPreConfig.utf8_mode* is non-zero, or if the LC_CTYPE locale is "C" or "POSIX".
— `"strict"` otherwise.
See also *PyConfig.legacy_windows_stdio*.

int **tracemalloc**

Enable tracemalloc ?

If non-zero, call `tracemalloc.start()` at startup.

Set by `-X tracemalloc=N` command line option and by the `PYTHONTRACEMALLOC` environment variable.

Default : `-1` in Python mode, `0` in isolated mode.

int **perf_profiling**

Enable the Linux `perf` profiler support ?

If equals to `1`, enable support for the Linux `perf` profiler.

If equals to `2`, enable support for the Linux `perf` profiler with DWARF JIT support.

Set to `1` by `-X perf` command-line option and the `PYTHONPERFSUPPORT` environment variable.

Set to `2` by the `-X perf_jit` command-line option and the `PYTHON_PERF_JIT_SUPPORT` environment variable.

Default : `-1`.

Voir aussi

See perf_profiling for more information.

Ajouté dans la version 3.12.

wchar_t ***stdlib_dir**

Directory of the Python standard library.

Default : `NULL`.

Ajouté dans la version 3.11.

int **use_environment**

Use environment variables ?

If equals to zero, ignore the environment variables.

Set to `0` by the `-E` environment variable.

Default : `1` in Python config and `0` in isolated config.

int **use_system_logger**

If non-zero, `stdout` and `stderr` will be redirected to the system log.

Only available on macOS 10.12 and later, and on iOS.

Default : `0` (don't use the system log) on macOS ; `1` on iOS (use the system log).

Ajouté dans la version 3.14.

int **user_site_directory**

If non-zero, add the user site directory to `sys.path`.

Set to `0` by the `-s` and `-I` command line options.

Set to `0` by the `PYTHONNOUSERSITE` environment variable.

Default : `1` in Python mode, `0` in isolated mode.

int **verbose**

Verbose mode. If greater than `0`, print a message each time a module is imported, showing the place (filename or built-in module) from which it is loaded.

If greater than or equal to `2`, print a message for each file that is checked for when searching for a module. Also provides information on module cleanup at exit.

Incremented by the `-v` command line option.

Set by the PYTHONVERBOSE environment variable value.

Default : 0.

PyWideStringList **warnoptions**

Options of the warnings module to build warnings filters, lowest to highest priority : sys.warnoptions.

The warnings module adds sys.warnoptions in the reverse order : the last *PyConfig.warnoptions* item becomes the first item of warnings.filters which is checked first (highest priority).

The -W command line options adds its value to *warnoptions*, it can be used multiple times.

The PYTHONWARNINGS environment variable can also be used to add warning options. Multiple options can be specified, separated by commas (,).

Default : empty list.

int **write_bytecode**

If equal to 0, Python won't try to write .pyc files on the import of source modules.

Set to 0 by the -B command line option and the PYTHONDONTWRITEBYTECODE environment variable.

sys.dont_write_bytecode is initialized to the inverted value of *write_bytecode*.

Default : 1.

PyWideStringList **xoptions**

Values of the -X command line options : sys._xoptions.

Default : empty list.

int **_pystats**

If non-zero, write performance statistics at Python exit.

Need a special build with the Py_STATS macro : see --enable-pystats.

Default : 0.

If *parse_argv* is non-zero, *argv* arguments are parsed the same way the regular Python parses command line arguments, and Python arguments are stripped from *argv*.

The *xoptions* options are parsed to set other options : see the -X command line option.

Modifié dans la version 3.9 : The show_alloc_count field has been removed.

11.4.7 Initialization with PyConfig

Initializing the interpreter from a populated configuration struct is handled by calling *Py_InitializeFromConfig()*.

The caller is responsible to handle exceptions (error or exit) using *PyStatus_Exception()* and *Py_ExitStatusException()*.

If *PyImport_FrozenModules()*, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* are used, they must be set or called after Python preinitialization and before the Python initialization. If Python is initialized multiple times, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* must be called before each Python initialization.

The current configuration (PyConfig type) is stored in PyInterpreterState.config.

Example setting the program name :

```
void init_python(void)
{
    PyStatus status;

```

(suite sur la page suivante)

(suite de la page précédente)

```
    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name. Implicitly preinitialize Python. */
    status = PyConfig_SetString(&config, &config.program_name,
                                L"/path/to/my_program");
    if (PyStatus_Exception(status)) {
        goto exception;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto exception;
    }
    PyConfig_Clear(&config);
    return;

exception:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}
```

More complete example modifying the default configuration, read the configuration, and then override some parameters. Note that since 3.11, many parameters are not calculated until initialization, and so values cannot be read from the configuration structure. Any values set before initialize is called will be left unchanged by initialization :

```
PyStatus init_python(const char *program_name)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name before reading the configuration
       (decode byte string from the locale encoding).

       Implicitly preinitialize Python. */
    status = PyConfig_SetBytesString(&config, &config.program_name,
                                     program_name);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Read all configuration at once */
    status = PyConfig_Read(&config);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Specify sys.path explicitly */
    /* If you want to modify the default set of paths, finish
       initialization first and then use PySys_GetObject("path") */
    config.module_search_paths_set = 1;
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/stdlib");
    if (PyStatus_Exception(status)) {
```

(suite sur la page suivante)

(suite de la page précédente)

```
        goto done;
    }
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/more/modules");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Override executable computed by PyConfig_Read() */
    status = PyConfig_SetString(&config, &config.executable,
                                L"/path/to/my_executable");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    status = Py_InitializeFromConfig(&config);

done:
    PyConfig_Clear(&config);
    return status;
}
```

11.4.8 Isolated Configuration

PyPreConfig_InitIsolatedConfig() and *PyConfig_InitIsolatedConfig()* functions create a configuration to isolate Python from the system. For example, to embed Python into an application.

This configuration ignores global configuration variables, environment variables, command line arguments (*PyConfig.argv* is not parsed) and user site directory. The C standard streams (ex : `stdout`) and the LC_CTYPE locale are left unchanged. Signal handlers are not installed.

Configuration files are still used with this configuration to determine paths that are unspecified. Ensure *PyConfig.home* is specified to avoid computing the default path configuration.

11.4.9 Python Configuration

PyPreConfig_InitPythonConfig() and *PyConfig_InitPythonConfig()* functions create a configuration to build a customized Python which behaves as the regular Python.

Environments variables and command line arguments are used to configure Python, whereas global configuration variables are ignored.

This function enables C locale coercion (**PEP 538**) and Python UTF-8 Mode (**PEP 540**) depending on the LC_CTYPE locale, `PYTHONUTF8` and `PYTHONCOERCECLOCALE` environment variables.

11.4.10 Python Path Configuration

PyConfig contains multiple fields for the path configuration :

- Path configuration inputs :
  - *PyConfig.home*
  - *PyConfig.platlibdir*
  - *PyConfig.pathconfig_warnings*
  - *PyConfig.program_name*
  - *PyConfig.pythonpath_env*
  - current working directory : to get absolute paths
  - `PATH` environment variable to get the program full path (from *PyConfig.program_name*)
  - `__PYVENV_LAUNCHER__` environment variable
  - (Windows only) Application paths in the registry under "SoftwarePythonPythonCoreX.YPythonPath" of HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE (where X.Y is the Python version).

- Path configuration output fields :
  - *PyConfig.base_exec_prefix*
  - *PyConfig.base_executable*
  - *PyConfig.base_prefix*
  - *PyConfig.exec_prefix*
  - *PyConfig.executable*
  - *PyConfig.module_search_paths_set*, *PyConfig.module_search_paths*
  - *PyConfig.prefix*

If at least one "output field" is not set, Python calculates the path configuration to fill unset fields. If *module_search_paths_set* is equal to `0`, *module_search_paths* is overridden and *module_search_paths_set* is set to `1`.

It is possible to completely ignore the function calculating the default path configuration by setting explicitly all path configuration output fields listed above. A string is considered as set even if it is non-empty. `module_search_paths` is considered as set if `module_search_paths_set` is set to `1`. In this case, `module_search_paths` will be used without modification.

Set *pathconfig_warnings* to `0` to suppress warnings when calculating the path configuration (Unix only, Windows does not log any warning).

If *base_prefix* or *base_exec_prefix* fields are not set, they inherit their value from *prefix* and *exec_prefix* respectively.

Py_RunMain() and *Py_Main()* modify `sys.path` :

- If *run_filename* is set and is a directory which contains a `__main__.py` script, prepend *run_filename* to `sys.path`.
- If *isolated* is zero :
  - If *run_module* is set, prepend the current directory to `sys.path`. Do nothing if the current directory cannot be read.
  - If *run_filename* is set, prepend the directory of the filename to `sys.path`.
  - Otherwise, prepend an empty string to `sys.path`.

If *site_import* is non-zero, `sys.path` can be modified by the `site` module. If *user_site_directory* is non-zero and the user's site-package directory exists, the `site` module appends the user's site-package directory to `sys.path`.

The following configuration files are used by the path configuration :

- `pyvenv.cfg`
- `._pth` file (ex : `python._pth`)
- `pybuilddir.txt` (Unix only)

If a `._pth` file is present :

- Set *isolated* to `1`.
- Set *use_environment* to `0`.
- Set *site_import* to `0`.
- Set *safe_path* to `1`.

If *home* is not set and a `pyvenv.cfg` file is present in the same directory as *executable*, or its parent, *prefix* and *exec_prefix* are set that location. When this happens, *base_prefix* and *base_exec_prefix* still keep their value, pointing to the base installation. See sys-path-init-virtual-environments for more information.

The `__PYVENV_LAUNCHER__` environment variable is used to set *PyConfig.base_executable*.

Modifié dans la version 3.14 : *prefix*, and *exec_prefix*, are now set to the `pyvenv.cfg` directory. This was previously done by `site`, therefore affected by `-S`.

11.5 Py_GetArgcArgv()

void **Py_GetArgcArgv**(int *argc, wchar_t ***argv)

Get the original command line arguments, before Python modified them.

See also *PyConfig.orig_argv* member.

11.6 Delaying main module execution

In some embedding use cases, it may be desirable to separate interpreter initialization from the execution of the main module.

This separation can be achieved by setting `PyConfig.run_command` to the empty string during initialization (to prevent the interpreter from dropping into the interactive prompt), and then subsequently executing the desired main module code using `__main__.__dict__` as the global namespace.

CHAPITRE 12

Memory Management

12.1 Aperçu

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the *Python memory manager*. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if they regularly manipulate object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

To avoid memory corruption, extension writers should never try to operate on Python objects with the functions exported by the C library : `malloc()`, `calloc()`, `realloc()` and `free()`. This will result in mixed calls between the C allocator and the Python memory manager with fatal consequences, because they implement different algorithms and operate on different heaps. However, one may safely allocate and release memory blocks with the C library allocator for individual purposes, as shown in the following example :

```
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyBytes_FromString(buf);
free(buf); /* malloc'ed */
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the bytes object returned as a result.

In most situations, however, it is recommended to allocate memory from the Python heap specifically because the latter is under control of the Python memory manager. For example, this is required when the interpreter is extended with new object types written in C. Another reason for using the Python heap is the desire to *inform* the Python memory manager about the memory needs of the extension module. Even when the requested memory is used exclusively for internal, highly specific purposes, delegating all memory requests to the Python memory manager causes the interpreter to have a more accurate image of its memory footprint as a whole. Consequently, under certain circumstances, the Python memory manager may or may not trigger appropriate actions, like garbage collection, memory compaction or other preventive procedures. Note that by using the C library allocator as shown in the previous example, the allocated memory for the I/O buffer escapes completely the Python memory manager.

Voir aussi

The `PYTHONMALLOC` environment variable can be used to configure the memory allocators used by Python.

The `PYTHONMALLOCSTATS` environment variable can be used to print statistics of the *pymalloc memory allocator* every time a new pymalloc object arena is created, and on shutdown.

12.2 Allocator Domains

All allocating functions belong to one of three different ”domains” (see also `PyMemAllocatorDomain`). These domains represent different allocation strategies and are optimized for different purposes. The specific details on how every domain allocates memory or what internal functions each domain calls is considered an implementation detail, but for debugging purposes a simplified table can be found at *here*. The APIs used to allocate and free a block of memory must be from the same domain. For example, `PyMem_Free()` must be used to free memory allocated using `PyMem_Malloc()`.

The three allocation domains are :

- Raw domain : intended for allocating memory for general-purpose memory buffers where the allocation *must* go to the system allocator or where the allocator can operate without an *attached thread state*. The memory is requested directly from the system. See *Raw Memory Interface*.
- ”Mem” domain : intended for allocating memory for Python buffers and general-purpose memory buffers where the allocation must be performed with an *attached thread state*. The memory is taken from the Python private heap. See *Memory Interface*.
- Object domain : intended for allocating memory for Python objects. The memory is taken from the Python private heap. See *Object allocators*.

Note

The *free-threaded* build requires that only Python objects are allocated using the ”object” domain and that all Python objects are allocated using that domain. This differs from the prior Python versions, where this was only a best practice and not a hard requirement.

For example, buffers (non-Python objects) should be allocated using `PyMem_Malloc()`, `PyMem_RawMalloc()`, or `malloc()`, but not `PyObject_Malloc()`.

See Memory Allocation APIs.

12.3 Raw Memory Interface

The following function sets are wrappers to the system allocator. These functions are thread-safe, so a *thread state* does not need to be *attached*.

The *default raw memory allocator* uses the following functions : `malloc()`, `calloc()`, `realloc()` and `free()` ; call `malloc(1)` (or `calloc(1, 1)`) when requesting zero bytes.

Ajouté dans la version 3.4.

void *__PyMem_RawMalloc__(size_t n)

Part of the Stable ABI *Depuis la version 3.13.* Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawMalloc(1)` had been called instead. The memory will not have been initialized in any way.

void *__PyMem_RawCalloc__(size_t nelem, size_t elsize)

Part of the Stable ABI *Depuis la version 3.13.* Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_RawCalloc(1, 1)` had been called instead.

Ajouté dans la version 3.5.

void *__PyMem_RawRealloc__(void *p, size_t n)

Part of the Stable ABI *Depuis la version 3.13.* Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyMem_RawMalloc(n)` ; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to *`PyMem_RawMalloc()`*, *`PyMem_RawRealloc()`* or *`PyMem_RawCalloc()`*.

If the request fails, *`PyMem_RawRealloc()`* returns `NULL` and *p* remains a valid pointer to the previous memory area.

void __PyMem_RawFree__(void *p)

Part of the Stable ABI *Depuis la version 3.13.* Frees the memory block pointed to by *p*, which must have been returned by a previous call to *`PyMem_RawMalloc()`*, *`PyMem_RawRealloc()`* or *`PyMem_RawCalloc()`*. Otherwise, or if `PyMem_RawFree(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

12.4 Memory Interface

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

The *default memory allocator* uses the *pymalloc memory allocator*.

Avertissement

There must be an *attached thread state* when using these functions.

Modifié dans la version 3.6 : The default allocator is now pymalloc instead of system `malloc()`.

void *__PyMem_Malloc__(size_t n)

Part of the Stable ABI. Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void *__PyMem_Calloc__(size_t nelem, size_t elsize)

Part of the Stable ABI *Depuis la version 3.7.* Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyMem_Calloc(1, 1)` had been called instead.

Ajouté dans la version 3.5.

void ***PyMem_Realloc**(void *p, size_t n)

Part of the Stable ABI. Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyMem_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to *PyMem_Malloc()*, *PyMem_Realloc()* or *PyMem_Calloc()*.

If the request fails, *PyMem_Realloc()* returns `NULL` and *p* remains a valid pointer to the previous memory area.

void **PyMem_Free**(void *p)

Part of the Stable ABI. Frees the memory block pointed to by *p*, which must have been returned by a previous call to *PyMem_Malloc()*, *PyMem_Realloc()* or *PyMem_Calloc()*. Otherwise, or if `PyMem_Free(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

The following type-oriented macros are provided for convenience. Note that *TYPE* refers to any C type.

PyMem_New(TYPE, n)

Same as *PyMem_Malloc()*, but allocates `(n * sizeof(TYPE))` bytes of memory. Returns a pointer cast to `TYPE*`. The memory will not have been initialized in any way.

PyMem_Resize(p, TYPE, n)

Same as *PyMem_Realloc()*, but the memory block is resized to `(n * sizeof(TYPE))` bytes. Returns a pointer cast to `TYPE*`. On return, *p* will be a pointer to the new memory area, or `NULL` in the event of failure.

This is a C preprocessor macro; *p* is always reassigned. Save the original value of *p* to avoid losing memory when handling errors.

void **PyMem_Del**(void *p)

Same as *PyMem_Free()*.

In addition, the following macro sets are provided for calling the Python memory allocator directly, without involving the C API functions listed above. However, note that their use does not preserve binary compatibility across Python versions and is therefore deprecated in extension modules.

- `PyMem_MALLOC(size)`
- `PyMem_NEW(type, size)`
- `PyMem_REALLOC(ptr, size)`
- `PyMem_RESIZE(ptr, type, size)`
- `PyMem_FREE(ptr)`
- `PyMem_DEL(ptr)`

12.5 Object allocators

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap.

Note

There is no guarantee that the memory returned by these allocators can be successfully cast to a Python object when intercepting the allocating functions in this domain by the methods described in the *Customize Memory Allocators* section.

The *default object allocator* uses the *pymalloc memory allocator*.

Avertissement

There must be an *attached thread state* when using these functions.

void *`PyObject_Malloc`(size_t n)

Part of the Stable ABI. Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

void *`PyObject_Calloc`(size_t nelem, size_t elsize)

Part of the Stable ABI *Depuis la version 3.7.* Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

Requesting zero elements or elements of size zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Calloc(1, 1)` had been called instead.

Ajouté dans la version 3.5.

void *`PyObject_Realloc`(void *p, size_t n)

Part of the Stable ABI. Resizes the memory block pointed to by *p* to *n* bytes. The contents will be unchanged to the minimum of the old and the new sizes.

If *p* is `NULL`, the call is equivalent to `PyObject_Malloc(n)` ; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-`NULL`.

Unless *p* is `NULL`, it must have been returned by a previous call to *`PyObject_Malloc()`*, *`PyObject_Realloc()`* or *`PyObject_Calloc()`*.

If the request fails, *`PyObject_Realloc()`* returns `NULL` and *p* remains a valid pointer to the previous memory area.

void `PyObject_Free`(void *p)

Part of the Stable ABI. Frees the memory block pointed to by *p*, which must have been returned by a previous call to *`PyObject_Malloc()`*, *`PyObject_Realloc()`* or *`PyObject_Calloc()`*. Otherwise, or if `PyObject_Free(p)` has been called before, undefined behavior occurs.

If *p* is `NULL`, no operation is performed.

Do not call this directly to free an object's memory; call the type's *`tp_free`* slot instead.

Do not use this for memory allocated by *`PyObject_GC_New`* or *`PyObject_GC_NewVar`*; use *`PyObject_GC_Del()`* instead.

Voir aussi

- *`PyObject_GC_Del()`* is the equivalent of this function for memory allocated by types that support garbage collection.
- *`PyObject_Malloc()`*
- *`PyObject_Realloc()`*
- *`PyObject_Calloc()`*
- *`PyObject_New`*
- *`PyObject_NewVar`*
- *`PyType_GenericAlloc()`*
- *`tp_free`*

12.6 Default Memory Allocators

Default memory allocators :

Configuration	Nom	PyMem_RawMalloc	PyMem_Malloc	PyObject_Malloc
Release build	`"pymalloc"`	`malloc`	`pymalloc`	`pymalloc`
Debug build	`"pymalloc_debug`	`malloc` + debug	`pymalloc` + debug	`pymalloc` + debug
Release build, without pymalloc	`"malloc"`	`malloc`	`malloc`	`malloc`
Debug build, without pymalloc	`"malloc_debug"`	`malloc` + debug	`malloc` + debug	`malloc` + debug

Legend :

- Name : value for `PYTHONMALLOC` environment variable.
- `malloc` : system allocators from the standard C library, C functions : `malloc()`, `calloc()`, `realloc()` and `free()`.
- `pymalloc` : *pymalloc memory allocator*.
- `mimalloc` : *mimalloc memory allocator*. The pymalloc allocator will be used if mimalloc support isn't available.
- "+ debug" : with *debug hooks on the Python memory allocators*.
- "Debug build" : Python build in debug mode.

12.7 Customize Memory Allocators

Ajouté dans la version 3.4.

type **PyMemAllocatorEx**

Structure used to describe a memory block allocator. The structure has the following fields :

Champ	Signification
`void *ctx`	user context passed as first argument
`void* malloc(void *ctx, size_t size)`	allocate a memory block
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	allocate a memory block initialized with zeros
`void* realloc(void *ctx, void *ptr, size_t new_size)`	allocate or resize a memory block
`void free(void *ctx, void *ptr)`	free a memory block

Modifié dans la version 3.5 : The `PyMemAllocator` structure was renamed to *PyMemAllocatorEx* and a new `calloc` field was added.

type **PyMemAllocatorDomain**

Enum used to identify an allocator domain. Domains :

PYMEM_DOMAIN_RAW

Functions :

- *PyMem_RawMalloc()*
- *PyMem_RawRealloc()*
- *PyMem_RawCalloc()*
- *PyMem_RawFree()*

PYMEM_DOMAIN_MEM

Functions :

— *PyMem_Malloc()*,
— *PyMem_Realloc()*
— *PyMem_Calloc()*
— *PyMem_Free()*

PYMEM_DOMAIN_OBJ

Functions :
— *PyObject_Malloc()*
— *PyObject_Realloc()*
— *PyObject_Calloc()*
— *PyObject_Free()*

void **PyMem_GetAllocator**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

Get the memory block allocator of the specified domain.

void **PyMem_SetAllocator**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

Set the memory block allocator of the specified domain.

The new allocator must return a distinct non-`NULL` pointer when requesting zero bytes.

For the *PYMEM_DOMAIN_RAW* domain, the allocator must be thread-safe : a *thread state* is not *attached* when the allocator is called.

For the remaining domains, the allocator must also be thread-safe : the allocator may be called in different interpreters that do not share a *GIL*.

If the new allocator is not a hook (does not call the previous allocator), the *PyMem_SetupDebugHooks()* function must be called to reinstall the debug hooks on top on the new allocator.

See also *PyPreConfig.allocator* and *Preinitialize Python with PyPreConfig*.

Avertissement

PyMem_SetAllocator() does have the following contract :
— It can be called after *Py_PreInitialize()* and before *Py_InitializeFromConfig()* to install a custom memory allocator. There are no restrictions over the installed allocator other than the ones imposed by the domain (for instance, the Raw Domain allows the allocator to be called without an *attached thread state*). See *the section on allocator domains* for more information.
— If called after Python has finish initializing (after *Py_InitializeFromConfig()* has been called) the allocator **must** wrap the existing allocator. Substituting the current allocator for some other arbitrary one is **not supported**.

Modifié dans la version 3.12 : All allocators must be thread-safe.

void **PyMem_SetupDebugHooks**(void)

Setup *debug hooks in the Python memory allocators* to detect memory errors.

12.8 Debug hooks on the Python memory allocators

When Python is built in debug mode, the *PyMem_SetupDebugHooks()* function is called at the *Python preinitialization* to setup debug hooks on Python memory allocators to detect memory errors.

The `PYTHONMALLOC` environment variable can be used to install debug hooks on a Python compiled in release mode (ex : `PYTHONMALLOC=debug`).

The *PyMem_SetupDebugHooks()* function can be used to set debug hooks after calling *PyMem_SetAllocator()*.

These debug hooks fill dynamically allocated memory blocks with special, recognizable bit patterns. Newly allocated memory is filled with the byte `0xCD` (`PYMEM_CLEANBYTE`), freed memory is filled with the byte `0xDD` (`PYMEM_DEADBYTE`). Memory blocks are surrounded by ”forbidden bytes” filled with the byte `0xFD` (`PYMEM_FORBIDDENBYTE`). Strings of these bytes are unlikely to be valid addresses, floats, or ASCII strings.

Runtime checks :

- Detect API violations. For example, detect if *PyObject_Free()* is called on a memory block allocated by *PyMem_Malloc()*.
- Detect write before the start of the buffer (buffer underflow).
- Detect write after the end of the buffer (buffer overflow).
- Check that there is an *attached thread state* when allocator functions of *PYMEM_DOMAIN_OBJ* (ex : *PyObject_Malloc()*) and *PYMEM_DOMAIN_MEM* (ex : *PyMem_Malloc()*) domains are called.

On error, the debug hooks use the `tracemalloc` module to get the traceback where a memory block was allocated. The traceback is only displayed if `tracemalloc` is tracing Python memory allocations and the memory block was traced.

Let *S* = `sizeof(size_t).2*S` bytes are added at each end of each block of *N* bytes requested. The memory layout is like so, where p represents the address returned by a malloc-like or realloc-like function (`p[i:j]` means the slice of bytes from `*(p+i)` inclusive up to `*(p+j)` exclusive; note that the treatment of negative indices differs from a Python slice) :

p[-2*S:-S]
: Number of bytes originally asked for. This is a size_t, big-endian (easier to read in a memory dump).

p[-S]
: API identifier (ASCII character) :
  - `'r'` for *PYMEM_DOMAIN_RAW*.
  - `'m'` for *PYMEM_DOMAIN_MEM*.
  - `'o'` for *PYMEM_DOMAIN_OBJ*.

p[-S+1:0]
: Copies of PYMEM_FORBIDDENBYTE. Used to catch under- writes and reads.

p[0:N]
: The requested memory, filled with copies of PYMEM_CLEANBYTE, used to catch reference to uninitialized memory. When a realloc-like function is called requesting a larger memory block, the new excess bytes are also filled with PYMEM_CLEANBYTE. When a free-like function is called, these are overwritten with PYMEM_DEADBYTE, to catch reference to freed memory. When a realloc- like function is called requesting a smaller memory block, the excess old bytes are also filled with PYMEM_DEADBYTE.

p[N:N+S]
: Copies of PYMEM_FORBIDDENBYTE. Used to catch over- writes and reads.

p[N+S:N+2*S]
: Only used if the `PYMEM_DEBUG_SERIALNO` macro is defined (not defined by default).

  A serial number, incremented by 1 on each call to a malloc-like or realloc-like function. Big-endian `size_t`. If "bad memory" is detected later, the serial number gives an excellent way to set a breakpoint on the next run, to capture the instant at which this block was passed out. The static function bumpserialno() in obmalloc.c is the only place the serial number is incremented, and exists so you can set such a breakpoint easily.

A realloc-like or free-like function first checks that the PYMEM_FORBIDDENBYTE bytes at each end are intact. If they've been altered, diagnostic output is written to stderr, and the program is aborted via Py_FatalError(). The other main failure mode is provoking a memory error when a program reads up one of the special bit patterns and tries to use it as an address. If you get in a debugger then and look at the object, you're likely to see that it's entirely filled with PYMEM_DEADBYTE (meaning freed memory is getting used) or PYMEM_CLEANBYTE (meaning uninitialized memory is getting used).

Modifié dans la version 3.6 : The *PyMem_SetupDebugHooks()* function now also works on Python compiled in release mode. On error, the debug hooks now use `tracemalloc` to get the traceback where a memory block was allocated. The debug hooks now also check if there is an *attached thread state* when functions of *PYMEM_DOMAIN_OBJ* and *PYMEM_DOMAIN_MEM* domains are called.

Modifié dans la version 3.8 : Byte patterns `0xCB` (`PYMEM_CLEANBYTE`), `0xDB` (`PYMEM_DEADBYTE`) and `0xFB` (`PYMEM_FORBIDDENBYTE`) have been replaced with `0xCD`, `0xDD` and `0xFD` to use the same values than Windows CRT debug `malloc()` and `free()`.

12.9 The pymalloc allocator

Python has a *pymalloc* allocator optimized for small objects (smaller or equal to 512 bytes) with a short lifetime. It uses memory mappings called ”arenas” with a fixed size of either 256 KiB on 32-bit platforms or 1 MiB on 64-bit platforms. It falls back to *PyMem_RawMalloc()* and *PyMem_RawRealloc()* for allocations larger than 512 bytes.

pymalloc is the *default allocator* of the *PYMEM_DOMAIN_MEM* (ex : *PyMem_Malloc()*) and *PYMEM_DOMAIN_OBJ* (ex : *PyObject_Malloc()*) domains.

The arena allocator uses the following functions :

- `VirtualAlloc()` and `VirtualFree()` on Windows,
- `mmap()` and `munmap()` if available,
- `malloc()` and `free()` otherwise.

This allocator is disabled if Python is configured with the `--without-pymalloc` option. It can also be disabled at runtime using the `PYTHONMALLOC` environment variable (ex : `PYTHONMALLOC=malloc`).

Typically, it makes sense to disable the pymalloc allocator when building Python with AddressSanitizer (`--with-address-sanitizer`) which helps uncover low level bugs within the C code.

12.9.1 Customize pymalloc Arena Allocator

Ajouté dans la version 3.4.

type **PyObjectArenaAllocator**

Structure used to describe an arena allocator. The structure has three fields :

Champ	Signification
`void *ctx`	user context passed as first argument
`void* alloc(void *ctx, size_t size)`	allocate an arena of size bytes
`void free(void *ctx, void *ptr, size_t size)`	free an arena

void **PyObject_GetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

Get the arena allocator.

void **PyObject_SetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

Set the arena allocator.

12.10 The mimalloc allocator

Ajouté dans la version 3.13.

Python supports the mimalloc allocator when the underlying platform support is available. mimalloc ”is a general purpose allocator with excellent performance characteristics. Initially developed by Daan Leijen for the runtime systems of the Koka and Lean languages.”

12.11 tracemalloc C API

Ajouté dans la version 3.7.

int **PyTraceMalloc_Track**(unsigned int domain, uintptr_t ptr, size_t size)

Track an allocated memory block in the `tracemalloc` module.

Return `0` on success, return `-1` on error (failed to allocate memory to store the trace). Return `-2` if tracemalloc is disabled.

If memory block is already tracked, update the existing trace.

int **PyTraceMalloc_Untrack**(unsigned int domain, uintptr_t ptr)

Untrack an allocated memory block in the `tracemalloc` module. Do nothing if the block was not tracked.

Return `-2` if tracemalloc is disabled, otherwise return `0`.

12.12 Exemples

Here is the example from section *Aperçu*, rewritten so that the I/O buffer is allocated from the Python heap by using the first function set :

```
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```

The same code using the type-oriented function set :

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_New */
return res;
```

Note that in the two examples above, the buffer is always manipulated via functions belonging to the same set. Indeed, it is required to use the same memory API family for a given memory block, so that the risk of mixing different allocators is reduced to a minimum. The following code sequence contains two errors, one of which is labeled as *fatal* because it mixes two different allocators operating on different heaps.

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);  /* Wrong -- should be PyMem_Free() */
free(buf2);       /* Right -- allocated via malloc() */
free(buf1);       /* Fatal -- should be PyMem_Free()  */
```

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with *PyObject_New*, *PyObject_NewVar* and *PyObject_Free()*.

These will be explained in the next chapter on defining and implementing new object types in C.

CHAPITRE 13

Implémentation d'objets

Ce chapitre décrit les fonctions, types, et macros utilisées pour définir de nouveaux types d'objets.

13.1 Allouer des objets dans le tas

PyObject ***_PyObject_New**(*PyTypeObject* *type)

Valeur de retour : nouvelle référence.

PyVarObject ***_PyObject_NewVar**(*PyTypeObject* *type, *Py_ssize_t* size)

Valeur de retour : nouvelle référence.

PyObject ***PyObject_Init**(*PyObject* *op, *PyTypeObject* *type)

Valeur de retour : référence empruntée. Part of the Stable ABI. Initialize a newly allocated object *op* with its type and initial reference. Returns the initialized object. Other fields of the object are not initialized. Despite its name, this function is unrelated to the object's `__init__()` method (`tp_init` slot). Specifically, this function does **not** call the object's `__init__()` method.

In general, consider this function to be a low-level routine. Use `tp_alloc` where possible. For implementing `tp_alloc` for your type, prefer `PyType_GenericAlloc()` or `PyObject_New()`.

> **Note**
>
> This function only initializes the object's memory corresponding to the initial `PyObject` structure. It does not zero the rest.

PyVarObject ***PyObject_InitVar**(*PyVarObject* *op, *PyTypeObject* *type, *Py_ssize_t* size)

Valeur de retour : référence empruntée. Part of the Stable ABI. Effectue les mêmes opérations que `PyObject_Init()` fait, et initialise également l'information de la longueur pour un objet de taille variable.

> **Note**
>
> This function only initializes some of the object's memory. It does not zero the rest.

PyObject_New(TYPE, typeobj)

Allocates a new Python object using the C structure type *TYPE* and the Python type object *ty-*

peobj (`PyTypeObject*`) by calling *PyObject_Malloc()* to allocate memory and initializing it like *PyObject_Init()*. The caller will own the only reference to the object (i.e. its reference count will be one).

Avoid calling this directly to allocate memory for an object; call the type's *tp_alloc* slot instead.

When populating a type's *tp_alloc* slot, *PyType_GenericAlloc()* is preferred over a custom function that simply calls this macro.

This macro does not call *tp_alloc*, *tp_new* (`__new__()`), or *tp_init* (`__init__()`).

This cannot be used for objects with *Py_TPFLAGS_HAVE_GC* set in *tp_flags*; use *PyObject_GC_New* instead.

Memory allocated by this macro must be freed with *PyObject_Free()* (usually called via the object's *tp_free* slot).

Note

The returned memory is not guaranteed to have been completely zeroed before it was initialized.

Note

This macro does not construct a fully initialized object of the given type; it merely allocates memory and prepares it for further initialization by *tp_init*. To construct a fully initialized object, call *typeobj* instead. For example :

```
PyObject *foo = PyObject_CallNoArgs((PyObject *)&PyFoo_Type);
```

Voir aussi

- *PyObject_Free()*
- *PyObject_GC_New*
- *PyType_GenericAlloc()*
- *tp_alloc*

PyObject_NewVar(TYPE, typeobj, size)

Like *PyObject_New* except :
- It allocates enough memory for the *TYPE* structure plus *size* (`Py_ssize_t`) fields of the size given by the *tp_itemsize* field of *typeobj*.
- The memory is initialized like *PyObject_InitVar()*.

This is useful for implementing objects like tuples, which are able to determine their size at construction time. Embedding the array of fields into the same allocation decreases the number of allocations, improving the memory management efficiency.

Avoid calling this directly to allocate memory for an object; call the type's *tp_alloc* slot instead.

When populating a type's *tp_alloc* slot, *PyType_GenericAlloc()* is preferred over a custom function that simply calls this macro.

This cannot be used for objects with *Py_TPFLAGS_HAVE_GC* set in *tp_flags*; use *PyObject_GC_NewVar* instead.

Memory allocated by this function must be freed with *PyObject_Free()* (usually called via the object's *tp_free* slot).

Note

The returned memory is not guaranteed to have been completely zeroed before it was initialized.

Note

This macro does not construct a fully initialized object of the given type ; it merely allocates memory and prepares it for further initialization by `tp_init`. To construct a fully initialized object, call *typeobj* instead. For example :

```
PyObject *list_instance = PyObject_CallNoArgs((PyObject *)&PyList_Type);
```

Voir aussi

- `PyObject_Free()`
- `PyObject_GC_NewVar`
- `PyType_GenericAlloc()`
- `tp_alloc`

void **PyObject_Del**(void *op)

Same as `PyObject_Free()`.

PyObject **_Py_NoneStruct**

Objet qui est visible en tant que `None` dans Python. Ne devrait être accessible uniquement en utilisant la macro `Py_None`, qui évalue cet objet à un pointeur.

Voir aussi

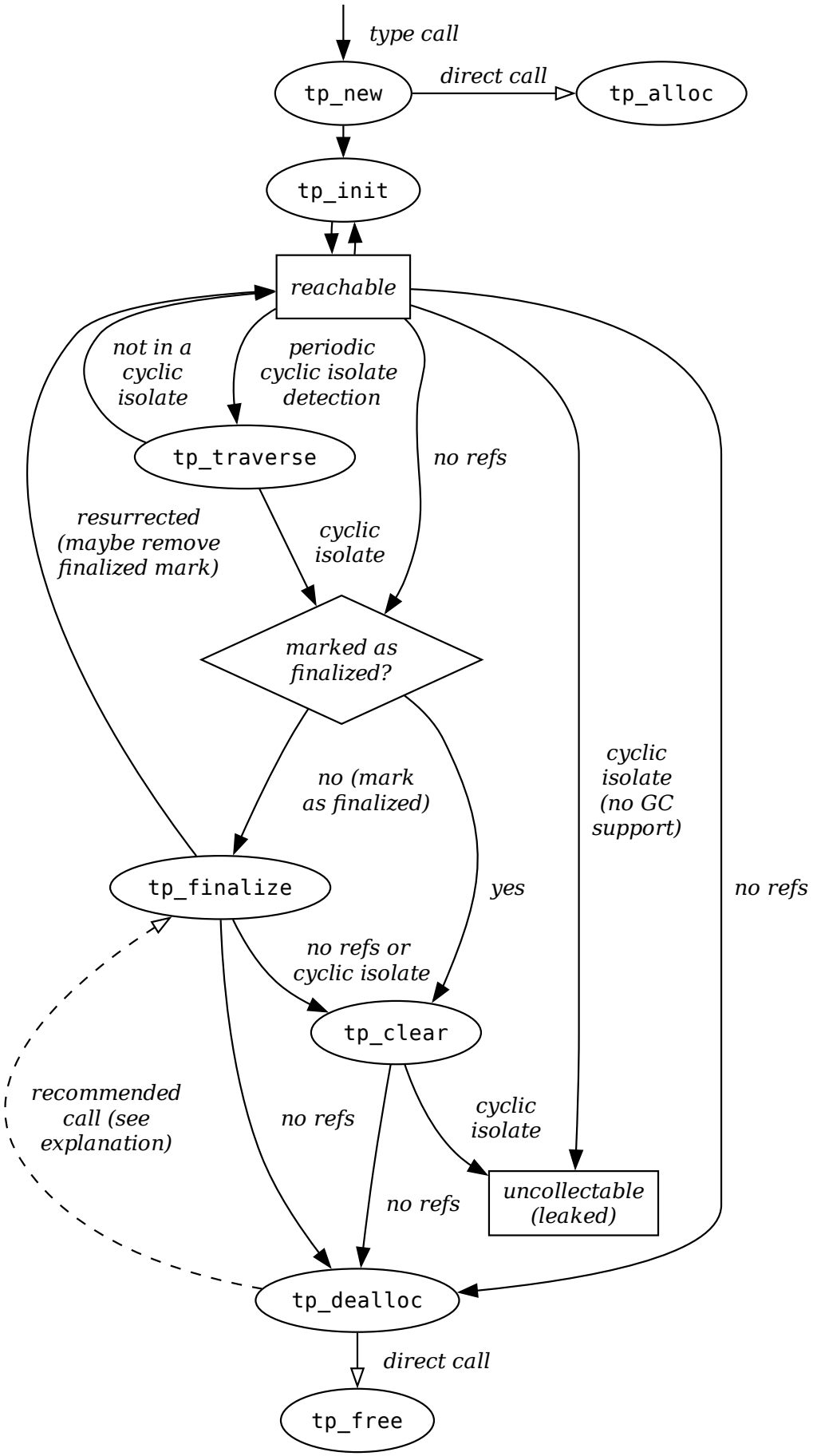
Module Objects
Allouer et créer des modules d'extension.

13.2 Object Life Cycle

This section explains how a type's slots relate to each other throughout the life of an object. It is not intended to be a complete canonical reference for the slots ; instead, refer to the slot-specific documentation in *Type Object Structures* for details about a particular slot.

13.2.1 Life Events

The figure below illustrates the order of events that can occur throughout an object's life. An arrow from *A* to *B* indicates that event *B* can occur after event *A* has occurred, with the arrow's label indicating the condition that must be true for *B* to occur after *A*.


type call
tp_new
direct call
tp_alloc
tp_init
reachable
not in a cyclic isolate
periodic cyclic isolate detection
tp_traverse
no refs
resurrected (maybe remove finalized mark)
cyclic isolate
marked as finalized?
no (mark as finalized)
cyclic isolate (no GC support)
tp_finalize
yes
no refs
no refs or cyclic isolate
tp_clear
recommended call (see explanation)
no refs
cyclic isolate
uncollectable (leaked)
no refs
tp_dealloc
direct call
tp_free

Explanation :

— When a new object is constructed by calling its type :

1. *tp_new* is called to create a new object.
2. *tp_alloc* is directly called by *tp_new* to allocate the memory for the new object.
3. *tp_init* initializes the newly created object. tp_init can be called again to re-initialize an object, if desired. The tp_init call can also be skipped entirely, for example by Python code calling __new__().

— After tp_init completes, the object is ready to use.

— Some time after the last reference to an object is removed :

1. If an object is not marked as *finalized*, it might be finalized by marking it as *finalized* and calling its *tp_finalize* function. Python does *not* finalize an object when the last reference to it is deleted; use *PyObject_CallFinalizerFromDealloc()* to ensure that *tp_finalize* is always called.
2. If the object is marked as finalized, *tp_clear* might be called by the garbage collector to clear references held by the object. It is *not* called when the object's reference count reaches zero.
3. *tp_dealloc* is called to destroy the object. To avoid code duplication, *tp_dealloc* typically calls into *tp_clear* to free up the object's references.
4. When *tp_dealloc* finishes object destruction, it directly calls *tp_free* (usually set to *PyObject_Free()* or *PyObject_GC_Del()* automatically as appropriate for the type) to deallocate the memory.

— The *tp_finalize* function is permitted to add a reference to the object if desired. If it does, the object is *resurrected*, preventing its pending destruction. (Only tp_finalize is allowed to resurrect an object; *tp_clear* and *tp_dealloc* cannot without calling into tp_finalize.) Resurrecting an object may or may not cause the object's *finalized* mark to be removed. Currently, Python does not remove the *finalized* mark from a resurrected object if it supports garbage collection (i.e., the *Py_TPFLAGS_HAVE_GC* flag is set) but does remove the mark if the object does not support garbage collection; either or both of these behaviors may change in the future.

— *tp_dealloc* can optionally call *tp_finalize* via *PyObject_CallFinalizerFromDealloc()* if it wishes to reuse that code to help with object destruction. This is recommended because it guarantees that tp_finalize is always called before destruction. See the *tp_dealloc* documentation for example code.

— If the object is a member of a *cyclic isolate* and either *tp_clear* fails to break the reference cycle or the cyclic isolate is not detected (perhaps gc.disable() was called, or the *Py_TPFLAGS_HAVE_GC* flag was erroneously omitted in one of the involved types), the objects remain indefinitely uncollectable (they "leak"). See gc.garbage.

If the object is marked as supporting garbage collection (the *Py_TPFLAGS_HAVE_GC* flag is set in *tp_flags*), the following events are also possible :

— The garbage collector occasionally calls *tp_traverse* to identify *cyclic isolates*.

— When the garbage collector discovers a *cyclic isolate*, it finalizes one of the objects in the group by marking it as *finalized* and calling its *tp_finalize* function, if it has one. This repeats until the cyclic isolate doesn't exist or all of the objects have been finalized.

— *tp_finalize* is permitted to resurrect the object by adding a reference from outside the *cyclic isolate*. The new reference causes the group of objects to no longer form a cyclic isolate (the reference cycle may still exist, but if it does the objects are no longer isolated).

— When the garbage collector discovers a *cyclic isolate* and all of the objects in the group have already been marked as *finalized*, the garbage collector clears one or more of the uncleared objects in the group (possibly concurrently) by calling each's *tp_clear* function. This repeats as long as the cyclic isolate still exists and not all of the objects have been cleared.

13.2.2 Cyclic Isolate Destruction

Listed below are the stages of life of a hypothetical *cyclic isolate* that continues to exist after each member object is finalized or cleared. It is a memory leak if a cyclic isolate progresses through all of these stages; it should vanish once all objects are cleared, if not sooner. A cyclic isolate can vanish either because the reference cycle is broken or because the objects are no longer isolated due to finalizer resurrection (see *tp_finalize*).

0. **Reachable** (not yet a cyclic isolate) : All objects are in their normal, reachable state. A reference cycle could exist, but an external reference means the objects are not yet isolated.
1. **Unreachable but consistent :** The final reference from outside the cyclic group of objects has been removed, causing the objects to become isolated (thus a cyclic isolate is born). None of the group's objects have been

finalized or cleared yet. The cyclic isolate remains at this stage until some future run of the garbage collector (not necessarily the next run because the next run might not scan every object).

2. **Mix of finalized and not finalized :** Objects in a cyclic isolate are finalized one at a time, which means that there is a period of time when the cyclic isolate is composed of a mix of finalized and non-finalized objects. Finalization order is unspecified, so it can appear random. A finalized object must behave in a sane manner when non-finalized objects interact with it, and a non-finalized object must be able to tolerate the finalization of an arbitrary subset of its referents.
3. **All finalized :** All objects in a cyclic isolate are finalized before any of them are cleared.
4. **Mix of finalized and cleared :** The objects can be cleared serially or concurrently (but with the *GIL* held) ; either way, some will finish before others. A finalized object must be able to tolerate the clearing of a subset of its referents. **PEP 442** calls this stage "cyclic trash".
5. **Leaked :** If a cyclic isolate still exists after all objects in the group have been finalized and cleared, then the objects remain indefinitely uncollectable (see `gc.garbage`). It is a bug if a cyclic isolate reaches this stage---it means the *tp_clear* methods of the participating objects have failed to break the reference cycle as required.

If *tp_clear* did not exist, then Python would have no way to safely break a reference cycle. Simply destroying an object in a cyclic isolate would result in a dangling pointer, triggering undefined behavior when an object referencing the destroyed object is itself destroyed. The clearing step makes object destruction a two-phase process : first *tp_clear* is called to partially destroy the objects enough to detangle them from each other, then *tp_dealloc* is called to complete the destruction.

Unlike clearing, finalization is not a phase of destruction. A finalized object must still behave properly by continuing to fulfill its design contracts. An object's finalizer is allowed to execute arbitrary Python code, and is even allowed to prevent the impending destruction by adding a reference. The finalizer is only related to destruction by call order---if it runs, it runs before destruction, which starts with *tp_clear* (if called) and concludes with *tp_dealloc*.

The finalization step is not necessary to safely reclaim the objects in a cyclic isolate, but its existence makes it easier to design types that behave in a sane manner when objects are cleared. Clearing an object might necessarily leave it in a broken, partially destroyed state---it might be unsafe to call any of the cleared object's methods or access any of its attributes. With finalization, only finalized objects can possibly interact with cleared objects ; non-finalized objects are guaranteed to interact with only non-cleared (but potentially finalized) objects.

To summarize the possible interactions :

— A non-finalized object might have references to or from non-finalized and finalized objects, but not to or from cleared objects.
— A finalized object might have references to or from non-finalized, finalized, and cleared objects.
— A cleared object might have references to or from finalized and cleared objects, but not to or from non-finalized objects.

Without any reference cycles, an object can be simply destroyed once its last reference is deleted ; the finalization and clearing steps are not necessary to safely reclaim unused objects. However, it can be useful to automatically call *tp_finalize* and *tp_clear* before destruction anyway because type design is simplified when all objects always experience the same series of events regardless of whether they participated in a cyclic isolate. Python currently only calls *tp_finalize* and *tp_clear* as needed to destroy a cyclic isolate ; this may change in a future version.

13.2.3 Functions

To allocate and free memory, see *Allouer des objets dans le tas*.

void **PyObject_CallFinalizer**(*PyObject* *op)

Finalizes the object as described in *tp_finalize*. Call this function (or *PyObject_CallFinalizerFromDealloc()*) instead of calling *tp_finalize* directly because this function may deduplicate multiple calls to `tp_finalize`. Currently, calls are only deduplicated if the type supports garbage collection (i.e., the *Py_TPFLAGS_HAVE_GC* flag is set) ; this may change in the future.

int **PyObject_CallFinalizerFromDealloc**(*PyObject* *op)

Same as *PyObject_CallFinalizer()* but meant to be called at the beginning of the object's destructor (*tp_dealloc*). There must not be any references to the object. If the object's finalizer resurrects the object, this function returns -1 ; no further destruction should happen. Otherwise, this function returns 0 and destruction can continue normally.

Voir aussi

tp_dealloc for example code.

13.3 Common Object Structures

There are a large number of structures which are used in the definition of object types for Python. This section describes these structures and how they are used.

13.3.1 Base object types and macros

All Python objects ultimately share a small number of fields at the beginning of the object's representation in memory. These are represented by the *PyObject* and *PyVarObject* types, which are defined, in turn, by the expansions of some macros also used, whether directly or indirectly, in the definition of all other Python objects. Additional macros can be found under *reference counting*.

type **PyObject**

Part of the Limited API. *(Only some members are part of the stable ABI.)* All object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In a normal "release" build, it contains only the object's reference count and a pointer to the corresponding type object. Nothing is actually declared to be a *PyObject*, but every pointer to a Python object can be cast to a *PyObject**.

The members must not be accessed directly; instead use macros such as *Py_REFCNT* and *Py_TYPE*.

Py_ssize_t **ob_refcnt**

Part of the Stable ABI. The object's reference count, as returned by *Py_REFCNT*. Do not use this field directly; instead use functions and macros such as `Py_REFCNT`, *Py_INCREF()* and *Py_DecRef()*.

The field type may be different from `Py_ssize_t`, depending on build configuration and platform.

PyTypeObject ***ob_type**

Part of the Stable ABI. The object's type. Do not use this field directly; use *Py_TYPE* and *Py_SET_TYPE()* instead.

type **PyVarObject**

Part of the Limited API. *(Only some members are part of the stable ABI.)* An extension of *PyObject* that adds the *ob_size* field. This is intended for objects that have some notion of *length*.

As with `PyObject`, the members must not be accessed directly; instead use macros such as *Py_SIZE*, *Py_REFCNT* and *Py_TYPE*.

Py_ssize_t **ob_size**

Part of the Stable ABI. A size field, whose contents should be considered an object's internal implementation detail.

Do not use this field directly; use *Py_SIZE* instead.

Object creation functions such as *PyObject_NewVar()* will generally set this field to the requested size (number of items). After creation, arbitrary values can be stored in `ob_size` using *Py_SET_SIZE*.

To get an object's publicly exposed length, as returned by the Python function `len()`, use *PyObject_Length()* instead.

PyObject_HEAD

This is a macro used when declaring new types which represent objects without a varying length. The PyObject_HEAD macro expands to :

```
PyObject ob_base;
```

See documentation of *PyObject* above.

PyObject_VAR_HEAD

This is a macro used when declaring new types which represent objects with a length that varies from instance to instance. The PyObject_VAR_HEAD macro expands to :

```
PyVarObject ob_base;
```

See documentation of *PyVarObject* above.

PyTypeObject **PyBaseObject_Type**

Part of the Stable ABI. The base class of all other objects, the same as `object` in Python.

int **Py_Is**(*PyObject* *x, *PyObject* *y)

Part of the Stable ABI *Depuis la version 3.10.* Test if the *x* object is the *y* object, the same as `x is y` in Python.

Ajouté dans la version 3.10.

int **Py_IsNone**(*PyObject* *x)

Part of the Stable ABI *Depuis la version 3.10.* Test if an object is the `None` singleton, the same as `x is None` in Python.

Ajouté dans la version 3.10.

int **Py_IsTrue**(*PyObject* *x)

Part of the Stable ABI *Depuis la version 3.10.* Test if an object is the `True` singleton, the same as `x is True` in Python.

Ajouté dans la version 3.10.

int **Py_IsFalse**(*PyObject* *x)

Part of the Stable ABI *Depuis la version 3.10.* Test if an object is the `False` singleton, the same as `x is False` in Python.

Ajouté dans la version 3.10.

PyTypeObject ***Py_TYPE**(*PyObject* *o)

Valeur de retour : référence empruntée. Part of the Stable ABI *Depuis la version 3.14.* Get the type of the Python object *o*.

The returned reference is *borrowed* from *o*. Do not release it with *Py_DECREF()* or similar.

Modifié dans la version 3.11 : *Py_TYPE()* is changed to an inline static function. The parameter type is no longer `const PyObject*`.

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

Return non-zero if the object *o* type is *type*. Return zero otherwise. Equivalent to : `Py_TYPE(o) == type`.

Ajouté dans la version 3.9.

void **Py_SET_TYPE**(*PyObject* *o, *PyTypeObject* *type)

Set the type of object *o* to *type*, without any checking or reference counting.

This is a very low-level operation. Consider instead setting the Python attribute `__class__` using *PyObject_SetAttrString()* or similar.

Note that assigning an incompatible type can lead to undefined behavior.

If *type* is a *heap type*, the caller must create a new reference to it. Similarly, if the old type of *o* is a heap type, the caller must release a reference to that type.

Ajouté dans la version 3.9.

Py_ssize_t **Py_SIZE**(*PyVarObject* *o)

Get the *ob_size* field of *o*.

Modifié dans la version 3.11 : *Py_SIZE()* is changed to an inline static function. The parameter type is no longer `const PyVarObject*`.

void **Py_SET_SIZE**(*PyVarObject* *o, *Py_ssize_t* size)

Set the `ob_size` field of *o* to *size*.

Ajouté dans la version 3.9.

PyObject_HEAD_INIT(type)

This is a macro which expands to initialization values for a new `PyObject` type. This macro expands to :

```
_PyObject_EXTRA_INIT
1, type,
```

PyVarObject_HEAD_INIT(type, size)

This is a macro which expands to initialization values for a new `PyVarObject` type, including the `ob_size` field. This macro expands to :

```
_PyObject_EXTRA_INIT
1, type, size,
```

13.3.2 Implementing functions and methods

type **PyCFunction**

Part of the Stable ABI. Type of the functions used to implement most Python callables in C. Functions of this type take two `PyObject*` parameters and return one such value. If the return value is `NULL`, an exception shall have been set. If not `NULL`, the return value is interpreted as the return value of the function as exposed in Python. The function must return a new reference.

The function signature is :

```
PyObject *PyCFunction(PyObject *self,
                      PyObject *args);
```

type **PyCFunctionWithKeywords**

Part of the Stable ABI. Type of the functions used to implement Python callables in C with signature *METH_VARARGS | METH_KEYWORDS*. The function signature is :

```
PyObject *PyCFunctionWithKeywords(PyObject *self,
                                  PyObject *args,
                                  PyObject *kwargs);
```

type **PyCFunctionFast**

Part of the Stable ABI *Depuis la version 3.13.* Type of the functions used to implement Python callables in C with signature `METH_FASTCALL`. The function signature is :

```
PyObject *PyCFunctionFast(PyObject *self,
                          PyObject *const *args,
                          Py_ssize_t nargs);
```

type **PyCFunctionFastWithKeywords**

Part of the Stable ABI *Depuis la version 3.13.* Type of the functions used to implement Python callables in C with signature *METH_FASTCALL | METH_KEYWORDS*. The function signature is :

```
PyObject *PyCFunctionFastWithKeywords(PyObject *self,
                                      PyObject *const *args,
                                      Py_ssize_t nargs,
                                      PyObject *kwnames);
```

type **PyCMethod**

Type of the functions used to implement Python callables in C with signature *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*. The function signature is :

```
PyObject *PyCMethod(PyObject *self,
                    PyTypeObject *defining_class,
                    PyObject *const *args,
                    Py_ssize_t nargs,
                    PyObject *kwnames)
```

Ajouté dans la version 3.9.

type **PyMethodDef**

Part of the Stable ABI *(including all members).* Structure used to describe a method of an extension type. This structure has four fields :

const char ***ml_name**

Name of the method.

PyCFunction **ml_meth**

Pointer to the C implementation.

int **ml_flags**

Flags bits indicating how the call should be constructed.

const char ***ml_doc**

Points to the contents of the docstring.

The *ml_meth* is a C function pointer. The functions may be of different types, but they always return *PyObject**. If the function is not of the *PyCFunction*, the compiler will require a cast in the method table. Even though *PyCFunction* defines the first parameter as *PyObject**, it is common that the method implementation uses the specific C type of the *self* object.

The *ml_flags* field is a bitfield which can include the following flags. The individual flags indicate either a calling convention or a binding convention.

There are these calling conventions :

METH_VARARGS

This is the typical calling convention, where the methods have the type *PyCFunction*. The function expects two *PyObject** values. The first one is the *self* object for methods ; for module functions, it is the module object. The second parameter (often called *args*) is a tuple object representing all arguments. This parameter is typically processed using *PyArg_ParseTuple()* or *PyArg_UnpackTuple()*.

METH_KEYWORDS

Can only be used in certain combinations with other flags : *METH_VARARGS | METH_KEYWORDS*, *METH_FASTCALL | METH_KEYWORDS* and *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*.

METH_VARARGS* | *METH_KEYWORDS

Methods with these flags must be of type *PyCFunctionWithKeywords*. The function expects three parameters : *self*, *args*, *kwargs* where *kwargs* is a dictionary of all the keyword arguments or possibly `NULL` if there are no keyword arguments. The parameters are typically processed using *PyArg_ParseTupleAndKeywords()*.

METH_FASTCALL

Fast calling convention supporting only positional arguments. The methods have the type *PyCFunctionFast*. The first parameter is *self*, the second parameter is a C array of *PyObject** values indicating the arguments and the third parameter is the number of arguments (the length of the array).

Ajouté dans la version 3.7.

Modifié dans la version 3.10 : `METH_FASTCALL` is now part of the *stable ABI*.

METH_FASTCALL* | *METH_KEYWORDS

Extension of *METH_FASTCALL* supporting also keyword arguments, with methods of type

PyCFunctionFastWithKeywords. Keyword arguments are passed the same way as in the *vectorcall protocol* : there is an additional fourth *PyObject** parameter which is a tuple representing the names of the keyword arguments (which are guaranteed to be strings) or possibly NULL if there are no keywords. The values of the keyword arguments are stored in the *args* array, after the positional arguments.

Ajouté dans la version 3.7.

METH_METHOD

Can only be used in the combination with other flags : *METH_METHOD | METH_FASTCALL | METH_KEYWORDS*.

METH_METHOD* | *METH_FASTCALL* | *METH_KEYWORDS

Extension of *METH_FASTCALL | METH_KEYWORDS* supporting the *defining class*, that is, the class that contains the method in question. The defining class might be a superclass of Py_TYPE(self).

The method needs to be of type *PyCMethod*, the same as for METH_FASTCALL | METH_KEYWORDS with defining_class argument added after self.

Ajouté dans la version 3.9.

METH_NOARGS

Methods without parameters don't need to check whether arguments are given if they are listed with the *METH_NOARGS* flag. They need to be of type *PyCFunction*. The first parameter is typically named *self* and will hold a reference to the module or object instance. In all cases the second parameter will be NULL.

The function must have 2 parameters. Since the second parameter is unused, *Py_UNUSED* can be used to prevent a compiler warning.

METH_O

Methods with a single object argument can be listed with the *METH_O* flag, instead of invoking *PyArg_ParseTuple()* with a "O" argument. They have the type *PyCFunction*, with the *self* parameter, and a *PyObject** parameter representing the single argument.

These two constants are not used to indicate the calling convention but the binding when use with methods of classes. These may not be used for functions defined for modules. At most one of these flags may be set for any given method.

METH_CLASS

The method will be passed the type object as the first parameter rather than an instance of the type. This is used to create *class methods*, similar to what is created when using the classmethod() built-in function.

METH_STATIC

The method will be passed NULL as the first parameter rather than an instance of the type. This is used to create *static methods*, similar to what is created when using the staticmethod() built-in function.

One other constant controls whether a method is loaded in place of another definition with the same method name.

METH_COEXIST

The method will be loaded in place of existing definitions. Without *METH_COEXIST*, the default is to skip repeated definitions. Since slot wrappers are loaded before the method table, the existence of a *sq_contains* slot, for example, would generate a wrapped method named __contains__() and preclude the loading of a corresponding PyCFunction with the same name. With the flag defined, the PyCFunction will be loaded in place of the wrapper object and will co-exist with the slot. This is helpful because calls to PyCFunctions are optimized more than wrapper object calls.

PyObject ***PyCMethod_New**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module, *PyTypeObject* *cls)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.9.* Turn *ml* into a Python *callable* object. The caller must ensure that *ml* outlives the *callable*. Typically, *ml* is defined as a static variable.

The *self* parameter will be passed as the *self* argument to the C function in ml->ml_meth when invoked. *self* can be NULL.

The *callable* object's `__module__` attribute can be set from the given *module* argument. *module* should be a Python string, which will be used as name of the module the function is defined in. If unavailable, it can be set to `None` or `NULL`.

Voir aussi

`function.__module__`

The *cls* parameter will be passed as the *defining_class* argument to the C function. Must be set if *METH_METHOD* is set on `ml->ml_flags`.

Ajouté dans la version 3.9.

PyObject ***PyCFunction_NewEx**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module)

Valeur de retour : nouvelle référence. Part of the Stable ABI. Equivalent to `PyCMethod_New(ml, self, module, NULL)`.

PyObject ***PyCFunction_New**(*PyMethodDef* *ml, *PyObject* *self)

Valeur de retour : nouvelle référence. Part of the Stable ABI *Depuis la version 3.4.* Equivalent to `PyCMethod_New(ml, self, NULL, NULL)`.

13.3.3 Accessing attributes of extension types

type **PyMemberDef**

Part of the Stable ABI *(including all members).* Structure which describes an attribute of a type which corresponds to a C struct member. When defining a class, put a NULL-terminated array of these structures in the *tp_members* slot.

Its fields are, in order :

const char ***name**

Name of the member. A NULL value marks the end of a `PyMemberDef[]` array.

The string should be static, no copy is made of it.

int **type**

The type of the member in the C struct. See *Member types* for the possible values.

Py_ssize_t **offset**

The offset in bytes that the member is located on the type's object struct.

int **flags**

Zero or more of the *Member flags*, combined using bitwise OR.

const char ***doc**

The docstring, or NULL. The string should be static, no copy is made of it. Typically, it is defined using *PyDoc_STR*.

By default (when *flags* is `0`), members allow both read and write access. Use the *Py_READONLY* flag for read-only access. Certain types, like *Py_T_STRING*, imply *Py_READONLY*. Only *Py_T_OBJECT_EX* (and legacy *T_OBJECT*) members can be deleted.

For heap-allocated types (created using *PyType_FromSpec()* or similar), `PyMemberDef` may contain a definition for the special member `"__vectorcalloffset__"`, corresponding to *tp_vectorcall_offset* in type objects. This member must be defined with `Py_T_PYSSIZET`, and either `Py_READONLY` or `Py_READONLY | Py_RELATIVE_OFFSET`. For example :

```
static PyMemberDef spam_type_members[] = {
    {"__vectorcalloffset__", Py_T_PYSSIZET,
     offsetof(Spam_object, vectorcall), Py_READONLY},
    {NULL}  /* Sentinel */
};
```

(You may need to `#include <stddef.h>` for `offsetof()`.)

The legacy offsets *tp_dictoffset* and *tp_weaklistoffset* can be defined similarly using `"__dictoffset__"` and `"__weaklistoffset__"` members, but extensions are strongly encouraged to use *Py_TPFLAGS_MANAGED_DICT* and *Py_TPFLAGS_MANAGED_WEAKREF* instead.

Modifié dans la version 3.12 : `PyMemberDef` is always available. Previously, it required including `"structmember.h"`.

Modifié dans la version 3.14 : *Py_RELATIVE_OFFSET* is now allowed for `"__vectorcalloffset__"`, `"__dictoffset__"` and `"__weaklistoffset__"`.

PyObject ***PyMember_GetOne**(const char *obj_addr, struct *PyMemberDef* *m)

Part of the Stable ABI. Get an attribute belonging to the object at address *obj_addr*. The attribute is described by `PyMemberDef` *m*. Returns `NULL` on error.

Modifié dans la version 3.12 : `PyMember_GetOne` is always available. Previously, it required including `"structmember.h"`.

int **PyMember_SetOne**(char *obj_addr, struct *PyMemberDef* *m, *PyObject* *o)

Part of the Stable ABI. Set an attribute belonging to the object at address *obj_addr* to object *o*. The attribute to set is described by `PyMemberDef` *m*. Returns `0` if successful and a negative value on failure.

Modifié dans la version 3.12 : `PyMember_SetOne` is always available. Previously, it required including `"structmember.h"`.

Member flags

The following flags can be used with *PyMemberDef.flags* :

Py_READONLY

Not writable.

Py_AUDIT_READ

Emit an `object.__getattr__` audit event before reading.

Py_RELATIVE_OFFSET

Indicates that the *offset* of this `PyMemberDef` entry indicates an offset from the subclass-specific data, rather than from `PyObject`.

Can only be used as part of *Py_tp_members slot* when creating a class using negative *basicsize*. It is mandatory in that case.

This flag is only used in *PyType_Slot*. When setting *tp_members* during class creation, Python clears it and sets *PyMemberDef.offset* to the offset from the `PyObject` struct.

Modifié dans la version 3.10 : The `RESTRICTED`, `READ_RESTRICTED` and `WRITE_RESTRICTED` macros available with `#include "structmember.h"` are deprecated. `READ_RESTRICTED` and `RESTRICTED` are equivalent to *Py_AUDIT_READ*; `WRITE_RESTRICTED` does nothing.

Modifié dans la version 3.12 : The `READONLY` macro was renamed to *Py_READONLY*. The `PY_AUDIT_READ` macro was renamed with the `Py_` prefix. The new names are now always available. Previously, these required `#include "structmember.h"`. The header is still available and it provides the old names.

Member types

PyMemberDef.type can be one of the following macros corresponding to various C types. When the member is accessed in Python, it will be converted to the equivalent Python type. When it is set from Python, it will be converted back to the C type. If that is not possible, an exception such as `TypeError` or `ValueError` is raised.

Unless marked (D), attributes defined this way cannot be deleted using e.g. `del` or `delattr()`.

Macro name	Type C	Type Python
Py_T_BYTE	`char`	`int`
Py_T_SHORT	`short`	`int`
Py_T_INT	`int`	`int`
Py_T_LONG	`long`	`int`
Py_T_LONGLONG	`long long`	`int`
Py_T_UBYTE	`unsigned char`	`int`
Py_T_UINT	`unsigned int`	`int`
Py_T_USHORT	`unsigned short`	`int`
Py_T_ULONG	`unsigned long`	`int`
Py_T_ULONGLONG	`unsigned long long`	`int`
Py_T_PYSSIZET	*Py_ssize_t*	`int`
Py_T_FLOAT	`float`	`float`
Py_T_DOUBLE	`double`	`float`
Py_T_BOOL	`char` (written as 0 or 1)	`bool`
Py_T_STRING	`const char*` (*)	`str` (RO)
Py_T_STRING_INPLACE	`const char[]` (*)	`str` (RO)
Py_T_CHAR	`char` (0-127)	`str` (**)
Py_T_OBJECT_EX	*PyObject*`*`	`object` (D)

(*) : Zero-terminated, UTF8-encoded C string. With `Py_T_STRING` the C representation is a pointer ;

with `Py_T_STRING_INPLACE` the string is stored directly in the structure.

(**) : String of length 1. Only ASCII is accepted.

(RO) : Implies *`Py_READONLY`*.

(D) : Can be deleted, in which case the pointer is set to `NULL`. Reading a `NULL` pointer raises `AttributeError`.

Ajouté dans la version 3.12 : In previous versions, the macros were only available with `#include "structmember.h"` and were named without the `Py_` prefix (e.g. as `T_INT`). The header is still available and contains the old names, along with the following deprecated types :

`T_OBJECT`

Like `Py_T_OBJECT_EX`, but `NULL` is converted to `None`. This results in surprising behavior in Python : deleting the attribute effectively sets it to `None`.

`T_NONE`

Always `None`. Must be used with *`Py_READONLY`*.

Defining Getters and Setters

type **`PyGetSetDef`**

Part of the Stable ABI *(including all members).* Structure to define property-like access for a type. See also description of the *`PyTypeObject.tp_getset`* slot.

const char ***`name`**

attribute name

getter **`get`**

C function to get the attribute.

setter **`set`**

Optional C function to set or delete the attribute. If `NULL`, the attribute is read-only.

const char ***`doc`**

optional docstring

void ***`closure`**

Optional user data pointer, providing additional data for getter and setter.

typedef *PyObject* *(***`getter`**)(*PyObject**, void*)

Part of the Stable ABI. The `get` function takes one *`PyObject*`* parameter (the instance) and a user data pointer (the associated `closure`) :

It should return a new reference on success or `NULL` with a set exception on failure.

typedef int (***`setter`**)(*PyObject**, *PyObject**, void*)

Part of the Stable ABI. `set` functions take two *`PyObject*`* parameters (the instance and the value to be set) and a user data pointer (the associated `closure`) :

In case the attribute should be deleted the second parameter is `NULL`. Should return `0` on success or `-1` with a set exception on failure.

13.4 Type Object Structures

Perhaps one of the most important structures of the Python object system is the structure that defines a new type : the *`PyTypeObject`* structure. Type objects can be handled using any of the `PyObject_*` or `PyType_*` functions, but do not offer much that's interesting to most Python applications. These objects are fundamental to how objects behave, so they are very important to the interpreter itself and to any extension module that implements new types.

Type objects are fairly large compared to most of the standard types. The reason for the size is that each type object stores a large number of values, mostly C function pointers, each of which implements a small part of the type's

functionality. The fields of the type object are examined in detail in this section. The fields will be described in the order in which they occur in the structure.

In addition to the following quick reference, the *Exemples* section provides at-a-glance insight into the meaning and use of `PyTypeObject`.

13.4.1 Quick Reference

"tp slots"

PyTypeObject Slot[page 299, 1]	*Type*	special methods/attrs	Info[page 299, 2] C	T	D	I
<R> `tp_name`	const char *	__name__	X	X		
`tp_basicsize`	`Py_ssize_t`		X	X		X
`tp_itemsize`	`Py_ssize_t`			X		X
`tp_dealloc`	`destructor`		X	X		X
`tp_vectorcall_offset`	`Py_ssize_t`			X		X
(`tp_getattr`)	`getattrfunc`	__getattribute__, __getattr__				G
(`tp_setattr`)	`setattrfunc`	__setattr__, __delattr__				G
`tp_as_async`	`PyAsyncMethods` *	*sub-slots*				%
`tp_repr`	`reprfunc`	__repr__	X	X		X
`tp_as_number`	`PyNumberMethods` *	*sub-slots*				%
`tp_as_sequence`	`PySequenceMethods` *	*sub-slots*				%
`tp_as_mapping`	`PyMappingMethods` *	*sub-slots*				%
`tp_hash`	`hashfunc`	__hash__	X			G
`tp_call`	`ternaryfunc`	__call__		X		X
`tp_str`	`reprfunc`	__str__	X			X
`tp_getattro`	`getattrofunc`	__getattribute__, __getattr__	X	X		G
`tp_setattro`	`setattrofunc`	__setattr__, __delattr__	X	X		G
`tp_as_buffer`	`PyBufferProcs` *	*sub-slots*				%
`tp_flags`	`unsigned long`		X	X		?
`tp_doc`	const char *	__doc__	X	X		
`tp_traverse`	`traverseproc`			X		G
`tp_clear`	`inquiry`			X		G
`tp_richcompare`	`richcmpfunc`	__lt__, __le__, __eq__, __ne__, __gt__, __ge__	X			G
(`tp_weaklistoffset`)	`Py_ssize_t`			X		?
`tp_iter`	`getiterfunc`	__iter__				X
`tp_iternext`	`iternextfunc`	__next__				X
`tp_methods`	`PyMethodDef` []		X	X		
`tp_members`	`PyMemberDef` []			X		
`tp_getset`	`PyGetSetDef` []		X	X		
`tp_base`	`PyTypeObject` *	__base__			X	
`tp_dict`	`PyObject` *	__dict__			?	
`tp_descr_get`	`descrgetfunc`	__get__				X
`tp_descr_set`	`descrsetfunc`	__set__, __delete__				X
(`tp_dictoffset`)	`Py_ssize_t`			X		?
`tp_init`	`initproc`	__init__	X	X		X
`tp_alloc`	`allocfunc`		X		?	?
`tp_new`	`newfunc`	__new__	X	X	?	?
`tp_free`	`freefunc`		X	X	?	?
`tp_is_gc`	`inquiry`			X		X
<`tp_bases`>	`PyObject` *	__bases__			~	
<`tp_mro`>	`PyObject` *	__mro__			~	
[`tp_cache`]	`PyObject` *					
[`tp_subclasses`]	void *	__subclasses__				
[`tp_weaklist`]	`PyObject` *					

suite sur la page suivante

Tableau 1 – suite de la page précédente

PyTypeObject Slot[1]	*Type*	special methods/attrs	Info[2] C T D I
(*tp_del*)	*destructor*		
[*tp_version_tag*]	unsigned int		
tp_finalize	*destructor*	__del__	X
tp_vectorcall	*vectorcallfunc*		
[*tp_watched*]	unsigned char		

sub-slots

Slot	*Type*	special methods
am_await	*unaryfunc*	__await__
am_aiter	*unaryfunc*	__aiter__
am_anext	*unaryfunc*	__anext__
am_send	*sendfunc*	
nb_add	*binaryfunc*	__add__ __radd__
nb_inplace_add	*binaryfunc*	__iadd__
nb_subtract	*binaryfunc*	__sub__ __rsub__
nb_inplace_subtract	*binaryfunc*	__isub__
nb_multiply	*binaryfunc*	__mul__ __rmul__
nb_inplace_multiply	*binaryfunc*	__imul__
nb_remainder	*binaryfunc*	__mod__ __rmod__
nb_inplace_remainder	*binaryfunc*	__imod__
nb_divmod	*binaryfunc*	__divmod__ __rdivmod__
nb_power	*ternaryfunc*	__pow__ __rpow__
nb_inplace_power	*ternaryfunc*	__ipow__
nb_negative	*unaryfunc*	__neg__
nb_positive	*unaryfunc*	__pos__
nb_absolute	*unaryfunc*	__abs__
nb_bool	*inquiry*	__bool__
nb_invert	*unaryfunc*	__invert__
nb_lshift	*binaryfunc*	__lshift__ __rlshift__
nb_inplace_lshift	*binaryfunc*	__ilshift__

suite sur la page suivante

1. () : A slot name in parentheses indicates it is (effectively) deprecated.
   <> : Names in angle brackets should be initially set to `NULL` and treated as read-only.
   [] : Names in square brackets are for internal use only.
   <R> (as a prefix) means the field is required (must be non-`NULL`).
2. Columns :
   "O" : set on *PyBaseObject_Type*
   "T" : set on *PyType_Type*
   "D" : default (if slot is set to `NULL`)

```
X - PyType_Ready sets this value if it is NULL
~ - PyType_Ready always sets this value (it should be NULL)
? - PyType_Ready may set this value depending on other slots

Also see the inheritance column ("I").
```

"I" : inheritance

```
X - type slot is inherited via *PyType_Ready* if defined with a *NULL* value
% - the slots of the sub-struct are inherited individually
G - inherited, but only in combination with other slots; see the slot's description
? - it's complicated; see the slot's description
```

Note that some slots are effectively inherited through the normal attribute lookup chain.

Tableau 2 – suite de la page précédente

Slot	*Type*	special methods
nb_rshift	*binaryfunc*	__rshift__ __rrshift__
nb_inplace_rshift	*binaryfunc*	__irshift__
nb_and	*binaryfunc*	__and__ __rand__
nb_inplace_and	*binaryfunc*	__iand__
nb_xor	*binaryfunc*	__xor__ __rxor__
nb_inplace_xor	*binaryfunc*	__ixor__
nb_or	*binaryfunc*	__or__ __ror__
nb_inplace_or	*binaryfunc*	__ior__
nb_int	*unaryfunc*	__int__
nb_reserved	void *	
nb_float	*unaryfunc*	__float__
nb_floor_divide	*binaryfunc*	__floordiv__
nb_inplace_floor_divide	*binaryfunc*	__ifloordiv__
nb_true_divide	*binaryfunc*	__truediv__
nb_inplace_true_divide	*binaryfunc*	__itruediv__
nb_index	*unaryfunc*	__index__
nb_matrix_multiply	*binaryfunc*	__matmul__ __rmatmul__
nb_inplace_matrix_multiply	*binaryfunc*	__imatmul__
mp_length	*lenfunc*	__len__
mp_subscript	*binaryfunc*	__getitem__
mp_ass_subscript	*objobjargproc*	__setitem__, __delitem__
sq_length	*lenfunc*	__len__
sq_concat	*binaryfunc*	__add__
sq_repeat	*ssizeargfunc*	__mul__
sq_item	*ssizeargfunc*	__getitem__
sq_ass_item	*ssizeobjargproc*	__setitem__ __delitem__
sq_contains	*objobjproc*	__contains__
sq_inplace_concat	*binaryfunc*	__iadd__
sq_inplace_repeat	*ssizeargfunc*	__imul__
bf_getbuffer	*getbufferproc()*	__buffer__
bf_releasebuffer	*releasebufferproc()*	__release_buffer__

slot typedefs

typedef	Parameter Types	Return Type
allocfunc	*PyTypeObject* * *Py_ssize_t*	*PyObject* *
destructor	*PyObject* *	void
freefunc	void *	void
traverseproc	*PyObject* * *visitproc* void *	*int*
newfunc	*PyTypeObject* * *PyObject* * *PyObject* *	*PyObject* *
initproc	*PyObject* * *PyObject* * *PyObject* *	*int*
reprfunc	*PyObject* *	*PyObject* *
getattrfunc	*PyObject* * const char *	*PyObject* *
setattrfunc	*PyObject* * const char * *PyObject* *	*int*
getattrofunc	*PyObject* * *PyObject* *	*PyObject* *
setattrofunc	*PyObject* * *PyObject* * *PyObject* *	*int*
descrgetfunc	*PyObject* * *PyObject* * *PyObject* *	*PyObject* *
descrsetfunc	*PyObject* * *PyObject* * *PyObject* *	*int*
hashfunc	*PyObject* *	Py_hash_t
richcmpfunc		*PyObject* *

See *Slot Type typedefs* below for more detail.

13.4.2 PyTypeObject Definition

The structure definition for *PyTypeObject* can be found in `Include/cpython/object.h`. For convenience of reference, this repeats the definition found there :

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                    or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

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

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    Py_ssize_t tp_weaklistoffset;
```

(suite sur la page suivante)

(suite de la page précédente)

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

    /* Attribute descriptor and subclassing stuff */
    PyMethodDef *tp_methods;
    PyMemberDef *tp_members;
    PyGetSetDef *tp_getset;
    // Strong reference on a heap type, borrowed reference on a static type
    PyTypeObject *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; /* no longer used */
    void *tp_subclasses;  /* for static builtin types this is an index */
    PyObject *tp_weaklist; /* not used for static builtin types */
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6.
     * If zero, the cache is invalid and must be initialized.
     */
    unsigned int tp_version_tag;

    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;

    /* bitset of which type-watchers care about this type */
    unsigned char tp_watched;

    /* Number of tp_version_tag values used.
     * Set to _Py_ATTR_CACHE_UNUSED if the attribute cache is
     * disabled for this type (e.g. due to custom MRO entries).
     * Otherwise, limited to MAX_VERSIONS_PER_CLASS (defined elsewhere).
     */
    uint16_t tp_versions_used;
} PyTypeObject;
```

13.4.3 PyObject Slots

The type object structure extends the *PyVarObject* structure. The *ob_size* field is used for dynamic types (created by `type_new()`, usually called from a class statement). Note that *PyType_Type* (the metatype) initializes *tp_itemsize*, which means that its instances (i.e. type objects) *must* have the *ob_size* field.

PyObject.ob_refcnt

The type object's reference count is initialized to `1` by the `PyObject_HEAD_INIT` macro. Note that for *statically allocated type objects*, the type's instances (objects whose *ob_type* points back to the type) do *not* count as references. But for *dynamically allocated type objects*, the instances *do* count as references.

Inheritance :

This field is not inherited by subtypes.

PyObject.ob_type

This is the type's type, in other words its metatype. It is initialized by the argument to the PyObject_HEAD_INIT macro, and its value should normally be &PyType_Type. However, for dynamically loadable extension modules that must be usable on Windows (at least), the compiler complains that this is not a valid initializer. Therefore, the convention is to pass NULL to the PyObject_HEAD_INIT macro and to initialize this field explicitly at the start of the module's initialization function, before doing anything else. This is typically done like this :

```
Foo_Type.ob_type = &PyType_Type;
```

This should be done before any instances of the type are created. *PyType_Ready()* checks if *ob_type* is NULL, and if so, initializes it to the *ob_type* field of the base class. *PyType_Ready()* will not change this field if it is non-zero.

Inheritance :

This field is inherited by subtypes.

13.4.4 PyVarObject Slots

PyVarObject.ob_size

For *statically allocated type objects*, this should be initialized to zero. For *dynamically allocated type objects*, this field has a special internal meaning.

This field should be accessed using the *Py_SIZE()* macro.

Inheritance :

This field is not inherited by subtypes.

13.4.5 PyTypeObject Slots

Each slot has a section describing inheritance. If *PyType_Ready()* may set a value when the field is set to NULL then there will also be a "Default" section. (Note that many fields set on *PyBaseObject_Type* and *PyType_Type* effectively act as defaults.)

const char **PyTypeObject*.**tp_name**

Pointer to a NUL-terminated string containing the name of the type. For types that are accessible as module globals, the string should be the full module name, followed by a dot, followed by the type name; for built-in types, it should be just the type name. If the module is a submodule of a package, the full package name is part of the full module name. For example, a type named T defined in module M in subpackage Q in package P should have the *tp_name* initializer "P.Q.M.T".

For *dynamically allocated type objects*, this should just be the type name, and the module name explicitly stored in the type dict as the value for key '__module__'.

For *statically allocated type objects*, the *tp_name* field should contain a dot. Everything before the last dot is made accessible as the __module__ attribute, and everything after the last dot is made accessible as the __name__ attribute.

If no dot is present, the entire *tp_name* field is made accessible as the __name__ attribute, and the __module__ attribute is undefined (unless explicitly set in the dictionary, as explained above). This means your type will be impossible to pickle. Additionally, it will not be listed in module documentations created with pydoc.

This field must not be NULL. It is the only required field in *PyTypeObject()* (other than potentially *tp_itemsize*).

Inheritance :

This field is not inherited by subtypes.

Py_ssize_t *PyTypeObject*.**tp_basicsize**

Py_ssize_t *PyTypeObject*.**tp_itemsize**

These fields allow calculating the size in bytes of instances of the type.

There are two kinds of types : types with fixed-length instances have a zero `tp_itemsize` field, types with variable-length instances have a non-zero `tp_itemsize` field. For a type with fixed-length instances, all instances have the same size, given in `tp_basicsize`. (Exceptions to this rule can be made using *PyUnstable_Object_GC_NewWithExtraData()*.)

For a type with variable-length instances, the instances must have an *ob_size* field, and the instance size is `tp_basicsize` plus N times `tp_itemsize`, where N is the "length" of the object.

Functions like *PyObject_NewVar()* will take the value of N as an argument, and store in the instance's *ob_size* field. Note that the *ob_size* field may later be used for other purposes. For example, `int` instances use the bits of *ob_size* in an implementation-defined way; the underlying storage and its size should be accessed using *PyLong_Export()*.

> **Note**
>
> The *ob_size* field should be accessed using the *Py_SIZE()* and *Py_SET_SIZE()* macros.

Also, the presence of an *ob_size* field in the instance layout doesn't mean that the instance structure is variable-length. For example, the `list` type has fixed-length instances, yet those instances have a *ob_size* field. (As with `int`, avoid reading lists' `ob_size` directly. Call *PyList_Size()* instead.)

The `tp_basicsize` includes size needed for data of the type's *tp_base*, plus any extra data needed by each instance.

The correct way to set `tp_basicsize` is to use the `sizeof` operator on the struct used to declare the instance layout. This struct must include the struct used to declare the base type. In other words, `tp_basicsize` must be greater than or equal to the base's `tp_basicsize`.

Since every type is a subtype of `object`, this struct must include *PyObject* or *PyVarObject* (depending on whether *ob_size* should be included). These are usually defined by the macro *PyObject_HEAD* or *PyObject_VAR_HEAD*, respectively.

The basic size does not include the GC header size, as that header is not part of *PyObject_HEAD*.

For cases where struct used to declare the base type is unknown, see *PyType_Spec.basicsize* and *PyType_FromMetaclass()*.

Notes about alignment :

— `tp_basicsize` must be a multiple of `_Alignof(PyObject)`. When using `sizeof` on a `struct` that includes *PyObject_HEAD*, as recommended, the compiler ensures this. When not using a C `struct`, or when using compiler extensions like `__attribute__((packed))`, it is up to you.

— If the variable items require a particular alignment, `tp_basicsize` and `tp_itemsize` must each be a multiple of that alignment. For example, if a type's variable part stores a `double`, it is your responsibility that both fields are a multiple of `_Alignof(double)`.

Inheritance :

These fields are inherited separately by subtypes. (That is, if the field is set to zero, *PyType_Ready()* will copy the value from the base type, indicating that the instances do not need additional storage.)

If the base type has a non-zero *tp_itemsize*, it is generally not safe to set *tp_itemsize* to a different non-zero value in a subtype (though this depends on the implementation of the base type).

destructor *PyTypeObject*.**tp_dealloc**

A pointer to the instance destructor function. The function signature is :

```
void tp_dealloc(PyObject *self);
```

The destructor function should remove all references which the instance owns (e.g., call *Py_CLEAR()*), free all memory buffers owned by the instance, and call the type's *tp_free* function to free the object itself.

If you may call functions that may set the error indicator, you must use *PyErr_GetRaisedException()* and *PyErr_SetRaisedException()* to ensure you don't clobber a preexisting error indicator (the deallocation could have occurred while processing a different error) :

```
static void
foo_dealloc(foo_object *self)
{
    PyObject *et, *ev, *etb;
    PyObject *exc = PyErr_GetRaisedException();
    ...
    PyErr_SetRaisedException(exc);
}
```

The dealloc handler itself must not raise an exception; if it hits an error case it should call *PyErr_FormatUnraisable()* to log (and clear) an unraisable exception.

No guarantees are made about when an object is destroyed, except :
— Python will destroy an object immediately or some time after the final reference to the object is deleted, unless its finalizer (*tp_finalize*) subsequently resurrects the object.
— An object will not be destroyed while it is being automatically finalized (*tp_finalize*) or automatically cleared (*tp_clear*).
CPython currently destroys an object immediately from *Py_DECREF()* when the new reference count is zero, but this may change in a future version.

It is recommended to call *PyObject_CallFinalizerFromDealloc()* at the beginning of `tp_dealloc` to guarantee that the object is always finalized before destruction.

If the type supports garbage collection (the *Py_TPFLAGS_HAVE_GC* flag is set), the destructor should call *PyObject_GC_UnTrack()* before clearing any member fields.

It is permissible to call *tp_clear* from `tp_dealloc` to reduce code duplication and to guarantee that the object is always cleared before destruction. Beware that `tp_clear` might have already been called.

If the type is heap allocated (*Py_TPFLAGS_HEAPTYPE*), the deallocator should release the owned reference to its type object (via *Py_DECREF()*) after calling the type deallocator. See the example code below. :

```
static void
foo_dealloc(PyObject *op)
{
   foo_object *self = (foo_object *) op;
   PyObject_GC_UnTrack(self);
   Py_CLEAR(self->ref);
   Py_TYPE(self)->tp_free(self);
}
```

`tp_dealloc` must leave the exception status unchanged. If it needs to call something that might raise an exception, the exception state must be backed up first and restored later (after logging any exceptions with *PyErr_WriteUnraisable()*).

Example :

```
static void
foo_dealloc(PyObject *self)
{
    PyObject *exc = PyErr_GetRaisedException();

    if (PyObject_CallFinalizerFromDealloc(self) < 0) {
        // self was resurrected.
        goto done;
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    PyTypeObject *tp = Py_TYPE(self);

    if (tp->tp_flags & Py_TPFLAGS_HAVE_GC) {
        PyObject_GC_UnTrack(self);
    }

    // Optional, but convenient to avoid code duplication.
    if (tp->tp_clear && tp->tp_clear(self) < 0) {
        PyErr_WriteUnraisable(self);
    }

    // Any additional destruction goes here.

    tp->tp_free(self);
    self = NULL;  // In case PyErr_WriteUnraisable() is called below.

    if (tp->tp_flags & Py_TPFLAGS_HEAPTYPE) {
        Py_CLEAR(tp);
    }

done:
    // Optional, if something was called that might have raised an
    // exception.
    if (PyErr_Occurred()) {
        PyErr_WriteUnraisable(self);
    }
    PyErr_SetRaisedException(exc);
}
```

`tp_dealloc` may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a garbage collection on any thread). This is not a problem for Python API calls, since the thread on which `tp_dealloc` is called with an *attached thread state*. However, if the object being destroyed in turn destroys objects from some other C library, care should be taken to ensure that destroying those objects on the thread which called `tp_dealloc` will not violate any assumptions of the library.

Inheritance :

This field is inherited by subtypes.

Voir aussi

Object Life Cycle for details about how this slot relates to other slots.

Py_ssize_t *PyTypeObject*.**tp_vectorcall_offset**

An optional offset to a per-instance function that implements calling the object using the *vectorcall protocol*, a more efficient alternative of the simpler *tp_call*.

This field is only used if the flag *Py_TPFLAGS_HAVE_VECTORCALL* is set. If so, this must be a positive integer containing the offset in the instance of a *vectorcallfunc* pointer.

The *vectorcallfunc* pointer may be `NULL`, in which case the instance behaves as if *Py_TPFLAGS_HAVE_VECTORCALL* was not set : calling the instance falls back to *tp_call*.

Any class that sets `Py_TPFLAGS_HAVE_VECTORCALL` must also set *tp_call* and make sure its behaviour is consistent with the *vectorcallfunc* function. This can be done by setting *tp_call* to *PyVectorcall_Call()*.

Modifié dans la version 3.8 : Before version 3.8, this slot was named `tp_print`. In Python 2.x, it was used for printing to a file. In Python 3.0 to 3.7, it was unused.

Modifié dans la version 3.12 : Before version 3.12, it was not recommended for *mutable heap types* to implement the vectorcall protocol. When a user sets `__call__` in Python code, only *tp_call* is updated, likely making it inconsistent with the vectorcall function. Since 3.12, setting `__call__` will disable vectorcall optimization by clearing the `Py_TPFLAGS_HAVE_VECTORCALL` flag.

Inheritance :

This field is always inherited. However, the `Py_TPFLAGS_HAVE_VECTORCALL` flag is not always inherited. If it's not set, then the subclass won't use *vectorcall*, except when `PyVectorcall_Call()` is explicitly called.

getattrfunc `PyTypeObject`**.tp_getattr**

An optional pointer to the get-attribute-string function.

This field is deprecated. When it is defined, it should point to a function that acts the same as the `tp_getattro` function, but taking a C string instead of a Python string object to give the attribute name.

Inheritance :

Group : `tp_getattr`, `tp_getattro`

This field is inherited by subtypes together with `tp_getattro` : a subtype inherits both `tp_getattr` and `tp_getattro` from its base type when the subtype's `tp_getattr` and `tp_getattro` are both `NULL`.

setattrfunc `PyTypeObject`**.tp_setattr**

An optional pointer to the function for setting and deleting attributes.

This field is deprecated. When it is defined, it should point to a function that acts the same as the `tp_setattro` function, but taking a C string instead of a Python string object to give the attribute name.

Inheritance :

Group : `tp_setattr`, `tp_setattro`

This field is inherited by subtypes together with `tp_setattro` : a subtype inherits both `tp_setattr` and `tp_setattro` from its base type when the subtype's `tp_setattr` and `tp_setattro` are both `NULL`.

PyAsyncMethods *`PyTypeObject`**.tp_as_async**

Pointer to an additional structure that contains fields relevant only to objects which implement *awaitable* and *asynchronous iterator* protocols at the C-level. See *Async Object Structures* for details.

Ajouté dans la version 3.5 : Formerly known as `tp_compare` and `tp_reserved`.

Inheritance :

The `tp_as_async` field is not inherited, but the contained fields are inherited individually.

reprfunc `PyTypeObject`**.tp_repr**

An optional pointer to a function that implements the built-in function `repr()`.

The signature is the same as for `PyObject_Repr()` :

```
PyObject *tp_repr(PyObject *self);
```

The function must return a string or a Unicode object. Ideally, this function should return a string that, when passed to `eval()`, given a suitable environment, returns an object with the same value. If this is not feasible, it should return a string starting with `'<'` and ending with `'>'` from which both the type and the value of the object can be deduced.

Inheritance :

This field is inherited by subtypes.

Default :

When this field is not set, a string of the form `<%s object at %p>` is returned, where `%s` is replaced by the type name, and `%p` by the object's memory address.

PyNumberMethods *`PyTypeObject`**.tp_as_number**

Pointer to an additional structure that contains fields relevant only to objects which implement the number

protocol. These fields are documented in *Number Object Structures*.

Inheritance :

The *tp_as_number* field is not inherited, but the contained fields are inherited individually.

PySequenceMethods **PyTypeObject*.**tp_as_sequence**

Pointer to an additional structure that contains fields relevant only to objects which implement the sequence protocol. These fields are documented in *Sequence Object Structures*.

Inheritance :

The *tp_as_sequence* field is not inherited, but the contained fields are inherited individually.

PyMappingMethods **PyTypeObject*.**tp_as_mapping**

Pointer to an additional structure that contains fields relevant only to objects which implement the mapping protocol. These fields are documented in *Mapping Object Structures*.

Inheritance :

The *tp_as_mapping* field is not inherited, but the contained fields are inherited individually.

hashfunc *PyTypeObject*.**tp_hash**

An optional pointer to a function that implements the built-in function `hash()`.

The signature is the same as for *PyObject_Hash()* :

```
Py_hash_t tp_hash(PyObject *);
```

The value `-1` should not be returned as a normal return value; when an error occurs during the computation of the hash value, the function should set an exception and return `-1`.

When this field is not set (*and* *tp_richcompare* is not set), an attempt to take the hash of the object raises `TypeError`. This is the same as setting it to *PyObject_HashNotImplemented()*.

This field can be set explicitly to *PyObject_HashNotImplemented()* to block inheritance of the hash method from a parent type. This is interpreted as the equivalent of `__hash__ = None` at the Python level, causing `isinstance(o, collections.Hashable)` to correctly return `False`. Note that the converse is also true - setting `__hash__ = None` on a class at the Python level will result in the `tp_hash` slot being set to *PyObject_HashNotImplemented()*.

Inheritance :

Group : *tp_hash*, *tp_richcompare*

This field is inherited by subtypes together with *tp_richcompare* : a subtype inherits both of *tp_richcompare* and *tp_hash*, when the subtype's *tp_richcompare* and *tp_hash* are both `NULL`.

Default :

PyBaseObject_Type uses *PyObject_GenericHash()*.

ternaryfunc *PyTypeObject*.**tp_call**

An optional pointer to a function that implements calling the object. This should be `NULL` if the object is not callable. The signature is the same as for *PyObject_Call()* :

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

Inheritance :

This field is inherited by subtypes.

reprfunc *PyTypeObject*.**tp_str**

An optional pointer to a function that implements the built-in operation `str()`. (Note that `str` is a type now, and `str()` calls the constructor for that type. This constructor calls *PyObject_Str()* to do the actual work, and *PyObject_Str()* will call this handler.)

The signature is the same as for *PyObject_Str()* :

```
PyObject *tp_str(PyObject *self);
```

The function must return a string or a Unicode object. It should be a ”friendly” string representation of the object, as this is the representation that will be used, among other things, by the `print()` function.

Inheritance :

This field is inherited by subtypes.

Default :

When this field is not set, *PyObject_Repr()* is called to return a string representation.

getattrofunc *PyTypeObject*.**tp_getattro**

An optional pointer to the get-attribute function.

The signature is the same as for *PyObject_GetAttr()* :

```
PyObject *tp_getattro(PyObject *self, PyObject *attr);
```

It is usually convenient to set this field to *PyObject_GenericGetAttr()*, which implements the normal way of looking for object attributes.

Inheritance :

Group : *tp_getattr*, *tp_getattro*

This field is inherited by subtypes together with *tp_getattr* : a subtype inherits both *tp_getattr* and *tp_getattro* from its base type when the subtype's *tp_getattr* and *tp_getattro* are both `NULL`.

Default :

PyBaseObject_Type uses *PyObject_GenericGetAttr()*.

setattrofunc *PyTypeObject*.**tp_setattro**

An optional pointer to the function for setting and deleting attributes.

The signature is the same as for *PyObject_SetAttr()* :

```
int tp_setattro(PyObject *self, PyObject *attr, PyObject *value);
```

In addition, setting *value* to `NULL` to delete an attribute must be supported. It is usually convenient to set this field to *PyObject_GenericSetAttr()*, which implements the normal way of setting object attributes.

Inheritance :

Group : *tp_setattr*, *tp_setattro*

This field is inherited by subtypes together with *tp_setattr* : a subtype inherits both *tp_setattr* and *tp_setattro* from its base type when the subtype's *tp_setattr* and *tp_setattro* are both `NULL`.

Default :

PyBaseObject_Type uses *PyObject_GenericSetAttr()*.

PyBufferProcs **PyTypeObject*.**tp_as_buffer**

Pointer to an additional structure that contains fields relevant only to objects which implement the buffer interface. These fields are documented in *Buffer Object Structures*.

Inheritance :

The *tp_as_buffer* field is not inherited, but the contained fields are inherited individually.

unsigned long *PyTypeObject*.**tp_flags**

This field is a bit mask of various flags. Some flags indicate variant semantics for certain situations; others are used to indicate that certain fields in the type object (or in the extension structures referenced via *tp_as_number*, *tp_as_sequence*, *tp_as_mapping*, and *tp_as_buffer*) that were historically not always present are valid; if such a flag bit is clear, the type fields it guards must not be accessed and must be

considered to have a zero or NULL value instead.

Inheritance :

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type's value of the flag bit is copied into the subtype together with a pointer to the extension structure. The *Py_TPFLAGS_HAVE_GC* flag bit is inherited together with the *tp_traverse* and *tp_clear* fields, i.e. if the *Py_TPFLAGS_HAVE_GC* flag bit is clear in the subtype and the *tp_traverse* and *tp_clear* fields in the subtype exist and have NULL values.

Default :

PyBaseObject_Type uses Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE.

Bit Masks :

The following bit masks are currently defined; these can be ORed together using the | operator to form the value of the *tp_flags* field. The macro *PyType_HasFeature()* takes a type and a flags value, *tp* and *f*, and checks whether tp->tp_flags & f is non-zero.

Py_TPFLAGS_HEAPTYPE

This bit is set when the type object itself is allocated on the heap, for example, types created dynamically using *PyType_FromSpec()*. In this case, the *ob_type* field of its instances is considered a reference to the type, and the type object is INCREF'ed when a new instance is created, and DECREF'ed when an instance is destroyed (this does not apply to instances of subtypes; only the type referenced by the instance's ob_type gets INCREF'ed or DECREF'ed). Heap types should also *support garbage collection* as they can form a reference cycle with their own module object.

Inheritance :

???

Py_TPFLAGS_BASETYPE

This bit is set when the type can be used as the base type of another type. If this bit is clear, the type cannot be subtyped (similar to a "final" class in Java).

Inheritance :

???

Py_TPFLAGS_READY

This bit is set when the type object has been fully initialized by *PyType_Ready()*.

Inheritance :

???

Py_TPFLAGS_READYING

This bit is set while *PyType_Ready()* is in the process of initializing the type object.

Inheritance :

???

Py_TPFLAGS_HAVE_GC

This bit is set when the object supports garbage collection. If this bit is set, memory for new instances (see *tp_alloc*) must be allocated using *PyObject_GC_New* or *PyType_GenericAlloc()* and deallocated (see *tp_free*) using *PyObject_GC_Del()*. More information in section *Supporting Cyclic Garbage Collection*.

Inheritance :

Group : *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

The *Py_TPFLAGS_HAVE_GC* flag bit is inherited together with the *tp_traverse* and *tp_clear* fields, i.e. if the *Py_TPFLAGS_HAVE_GC* flag bit is clear in the subtype and the *tp_traverse* and *tp_clear* fields in the subtype exist and have NULL values.

Py_TPFLAGS_DEFAULT

This is a bitmask of all the bits that pertain to the existence of certain fields in the type object and its extension structures. Currently, it includes the following bits : `Py_TPFLAGS_HAVE_STACKLESS_EXTENSION`.

Inheritance :

???

Py_TPFLAGS_METHOD_DESCRIPTOR

This bit indicates that objects behave like unbound methods.

If this flag is set for `type(meth)`, then :
— `meth.__get__(obj, cls)(*args, **kwds)` (with `obj` not None) must be equivalent to `meth(obj, *args, **kwds)`.
— `meth.__get__(None, cls)(*args, **kwds)` must be equivalent to `meth(*args, **kwds)`.
This flag enables an optimization for typical method calls like `obj.meth()` : it avoids creating a temporary "bound method" object for `obj.meth`.

Ajouté dans la version 3.8.

Inheritance :

This flag is never inherited by types without the *Py_TPFLAGS_IMMUTABLETYPE* flag set. For extension types, it is inherited whenever *tp_descr_get* is inherited.

Py_TPFLAGS_MANAGED_DICT

This bit indicates that instances of the class have a `__dict__` attribute, and that the space for the dictionary is managed by the VM.

If this flag is set, *Py_TPFLAGS_HAVE_GC* should also be set.

The type traverse function must call *PyObject_VisitManagedDict()* and its clear function must call *PyObject_ClearManagedDict()*.

Ajouté dans la version 3.12.

Inheritance :

This flag is inherited unless the *tp_dictoffset* field is set in a superclass.

Py_TPFLAGS_MANAGED_WEAKREF

This bit indicates that instances of the class should be weakly referenceable.

Ajouté dans la version 3.12.

Inheritance :

This flag is inherited unless the *tp_weaklistoffset* field is set in a superclass.

Py_TPFLAGS_ITEMS_AT_END

Only usable with variable-size types, i.e. ones with non-zero *tp_itemsize*.

Indicates that the variable-sized portion of an instance of this type is at the end of the instance's memory area, at an offset of `Py_TYPE(obj)->tp_basicsize` (which may be different in each subclass).

When setting this flag, be sure that all superclasses either use this memory layout, or are not variable-sized. Python does not check this.

Ajouté dans la version 3.12.

Inheritance :

This flag is inherited.

Py_TPFLAGS_LONG_SUBCLASS

Py_TPFLAGS_LIST_SUBCLASS

Py_TPFLAGS_TUPLE_SUBCLASS

Py_TPFLAGS_BYTES_SUBCLASS

Py_TPFLAGS_UNICODE_SUBCLASS

Py_TPFLAGS_DICT_SUBCLASS

Py_TPFLAGS_BASE_EXC_SUBCLASS

Py_TPFLAGS_TYPE_SUBCLASS

These flags are used by functions such as *PyLong_Check()* to quickly determine if a type is a subclass of a built-in type ; such specific checks are faster than a generic check, like *PyObject_IsInstance()*. Custom types that inherit from built-ins should have their *tp_flags* set appropriately, or the code that interacts with such types will behave differently depending on what kind of check is used.

Py_TPFLAGS_HAVE_FINALIZE

This bit is set when the *tp_finalize* slot is present in the type structure.

Ajouté dans la version 3.4.

Obsolète depuis la version 3.8 : This flag isn't necessary anymore, as the interpreter assumes the *tp_finalize* slot is always present in the type structure.

Py_TPFLAGS_HAVE_VECTORCALL

This bit is set when the class implements the *vectorcall protocol*. See *tp_vectorcall_offset* for details.

Inheritance :

This bit is inherited if *tp_call* is also inherited.

Ajouté dans la version 3.9.

Modifié dans la version 3.12 : This flag is now removed from a class when the class's `__call__()` method is reassigned.

This flag can now be inherited by mutable classes.

Py_TPFLAGS_IMMUTABLETYPE

This bit is set for type objects that are immutable : type attributes cannot be set nor deleted.

PyType_Ready() automatically applies this flag to *static types*.

Inheritance :

This flag is not inherited.

Ajouté dans la version 3.10.

Py_TPFLAGS_DISALLOW_INSTANTIATION

Disallow creating instances of the type : set *tp_new* to NULL and don't create the `__new__` key in the type dictionary.

The flag must be set before creating the type, not after. For example, it must be set before *PyType_Ready()* is called on the type.

The flag is set automatically on *static types* if *tp_base* is NULL or `&PyBaseObject_Type` and *tp_new* is NULL.

Inheritance :

This flag is not inherited. However, subclasses will not be instantiable unless they provide a non-NULL *tp_new* (which is only possible via the C API).

Note

To disallow instantiating a class directly but allow instantiating its subclasses (e.g. for an *abstract base class*), do not use this flag. Instead, make `tp_new` only succeed for subclasses.

Ajouté dans la version 3.10.

Py_TPFLAGS_MAPPING

This bit indicates that instances of the class may match mapping patterns when used as the subject of a `match` block. It is automatically set when registering or subclassing `collections.abc.Mapping`, and unset when registering `collections.abc.Sequence`.

Note

`Py_TPFLAGS_MAPPING` and `Py_TPFLAGS_SEQUENCE` are mutually exclusive ; it is an error to enable both flags simultaneously.

Inheritance :

This flag is inherited by types that do not already set `Py_TPFLAGS_SEQUENCE`.

Voir aussi

PEP 634 — Spécifications pour le filtrage par motif

Ajouté dans la version 3.10.

Py_TPFLAGS_SEQUENCE

This bit indicates that instances of the class may match sequence patterns when used as the subject of a `match` block. It is automatically set when registering or subclassing `collections.abc.Sequence`, and unset when registering `collections.abc.Mapping`.

Note

`Py_TPFLAGS_MAPPING` and `Py_TPFLAGS_SEQUENCE` are mutually exclusive ; it is an error to enable both flags simultaneously.

Inheritance :

This flag is inherited by types that do not already set `Py_TPFLAGS_MAPPING`.

Voir aussi

PEP 634 — Spécifications pour le filtrage par motif

Ajouté dans la version 3.10.

Py_TPFLAGS_VALID_VERSION_TAG

Internal. Do not set or unset this flag. To indicate that a class has changed call `PyType_Modified()`

Avertissement

This flag is present in header files, but is not be used. It will be removed in a future version of CPython

const char *`PyTypeObject`.**tp_doc**

An optional pointer to a NUL-terminated C string giving the docstring for this type object. This is exposed as

the `__doc__` attribute on the type and instances of the type.

Inheritance :

This field is *not* inherited by subtypes.

traverseproc *PyTypeObject*.**tp_traverse**

An optional pointer to a traversal function for the garbage collector. This is only used if the *Py_TPFLAGS_HAVE_GC* flag bit is set. The signature is :

```
int tp_traverse(PyObject *self, visitproc visit, void *arg);
```

More information about Python's garbage collection scheme can be found in section *Supporting Cyclic Garbage Collection*.

The *tp_traverse* pointer is used by the garbage collector to detect reference cycles. A typical implementation of a *tp_traverse* function simply calls *Py_VISIT()* on each of the instance's members that are Python objects that the instance owns. For example, this is function `local_traverse()` from the `_thread` extension module :

```
static int
local_traverse(PyObject *op, visitproc visit, void *arg)
{
    localobject *self = (localobject *) op;
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

Note that *Py_VISIT()* is called only on those members that can participate in reference cycles. Although there is also a `self->key` member, it can only be `NULL` or a Python string and therefore cannot be part of a reference cycle.

On the other hand, even if you know a member can never be part of a cycle, as a debugging aid you may want to visit it anyway just so the `gc` module's `get_referents()` function will include it.

Heap types (*Py_TPFLAGS_HEAPTYPE*) must visit their type with :

```
Py_VISIT(Py_TYPE(self));
```

It is only needed since Python 3.9. To support Python 3.8 and older, this line must be conditional :

```
#if PY_VERSION_HEX >= 0x03090000
    Py_VISIT(Py_TYPE(self));
#endif
```

If the *Py_TPFLAGS_MANAGED_DICT* bit is set in the *tp_flags* field, the traverse function must call *PyObject_VisitManagedDict()* like this :

```
PyObject_VisitManagedDict((PyObject*)self, visit, arg);
```

Avertissement

When implementing *tp_traverse*, only the members that the instance *owns* (by having *strong references* to them) must be visited. For instance, if an object supports weak references via the *tp_weaklist* slot, the pointer supporting the linked list (what *tp_weaklist* points to) must **not** be visited as the instance does not directly own the weak references to itself (the weakreference list is there to support the weak reference machinery, but the instance has no strong reference to the elements inside it, as they are allowed to be removed even if the instance is still alive).

Note that *Py_VISIT()* requires the *visit* and *arg* parameters to `local_traverse()` to have these specific names ; don't name them just anything.

Instances of *heap-allocated types* hold a reference to their type. Their traversal function must therefore either visit *Py_TYPE(self)*, or delegate this responsibility by calling `tp_traverse` of another heap-allocated type (such as a heap-allocated superclass). If they do not, the type object may not be garbage-collected.

Note

The *tp_traverse* function can be called from any thread.

Modifié dans la version 3.9 : Heap-allocated types are expected to visit `Py_TYPE(self)` in `tp_traverse`. In earlier versions of Python, due to bug 40217, doing this may lead to crashes in subclasses.

Inheritance :

Group : *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

This field is inherited by subtypes together with *tp_clear* and the *Py_TPFLAGS_HAVE_GC* flag bit : the flag bit, *tp_traverse*, and *tp_clear* are all inherited from the base type if they are all zero in the subtype.

inquiry *PyTypeObject*.**tp_clear**

An optional pointer to a clear function. The signature is :

```
int tp_clear(PyObject *);
```

The purpose of this function is to break reference cycles that are causing a *cyclic isolate* so that the objects can be safely destroyed. A cleared object is a partially destroyed object ; the object is not obligated to satisfy design invariants held during normal use.

`tp_clear` does not need to delete references to objects that can't participate in reference cycles, such as Python strings or Python integers. However, it may be convenient to clear all references, and write the type's *tp_dealloc* function to invoke `tp_clear` to avoid code duplication. (Beware that `tp_clear` might have already been called. Prefer calling idempotent functions like *Py_CLEAR()*.)

Any non-trivial cleanup should be performed in *tp_finalize* instead of `tp_clear`.

Note

If `tp_clear` fails to break a reference cycle then the objects in the *cyclic isolate* may remain indefinitely uncollectable ("leak"). See `gc.garbage`.

Note

Referents (direct and indirect) might have already been cleared ; they are not guaranteed to be in a consistent state.

Note

The *tp_clear* function can be called from any thread.

Note

An object is not guaranteed to be automatically cleared before its destructor (*tp_dealloc*) is called.

This function differs from the destructor (*tp_dealloc*) in the following ways :

— The purpose of clearing an object is to remove references to other objects that might participate in a reference cycle. The purpose of the destructor, on the other hand, is a superset : it must release *all* resources it owns, including references to objects that cannot participate in a reference cycle (e.g., integers) as well as the object's own memory (by calling *tp_free*).

— When `tp_clear` is called, other objects might still hold references to the object being cleared. Because of this, `tp_clear` must not deallocate the object's own memory (*tp_free*). The destructor, on the other hand, is only called when no (strong) references exist, and as such, must safely destroy the object itself by deallocating it.

— `tp_clear` might never be automatically called. An object's destructor, on the other hand, will be automatically called some time after the object becomes unreachable (i.e., either there are no references to the object or the object is a member of a *cyclic isolate*).

No guarantees are made about when, if, or how often Python automatically clears an object, except :

— Python will not automatically clear an object if it is reachable, i.e., there is a reference to it and it is not a member of a *cyclic isolate*.

— Python will not automatically clear an object if it has not been automatically finalized (see *tp_finalize*). (If the finalizer resurrected the object, the object may or may not be automatically finalized again before it is cleared.)

— If an object is a member of a *cyclic isolate*, Python will not automatically clear it if any member of the cyclic isolate has not yet been automatically finalized (*tp_finalize*).

— Python will not destroy an object until after any automatic calls to its `tp_clear` function have returned. This ensures that the act of breaking a reference cycle does not invalidate the `self` pointer while `tp_clear` is still executing.

— Python will not automatically call `tp_clear` multiple times concurrently.

CPython currently only automatically clears objects as needed to break reference cycles in a *cyclic isolate*, but future versions might clear objects regularly before their destruction.

Taken together, all *tp_clear* functions in the system must combine to break all reference cycles. This is subtle, and if in any doubt supply a *tp_clear* function. For example, the tuple type does not implement a *tp_clear* function, because it's possible to prove that no reference cycle can be composed entirely of tuples. Therefore the *tp_clear* functions of other types are responsible for breaking any cycle containing a tuple. This isn't immediately obvious, and there's rarely a good reason to avoid implementing *tp_clear*.

Implementations of *tp_clear* should drop the instance's references to those of its members that may be Python objects, and set its pointers to those members to `NULL`, as in the following example :

```
static int
local_clear(PyObject *op)
{
    localobject *self = (localobject *) op;
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

The *Py_CLEAR()* macro should be used, because clearing references is delicate : the reference to the contained object must not be released (via *Py_DECREF()*) until after the pointer to the contained object is set to `NULL`. This is because releasing the reference may cause the contained object to become trash, triggering a chain of reclamation activity that may include invoking arbitrary Python code (due to finalizers, or weakref callbacks, associated with the contained object). If it's possible for such code to reference *self* again, it's important that the pointer to the contained object be `NULL` at that time, so that *self* knows the contained object can no longer be used. The *Py_CLEAR()* macro performs the operations in a safe order.

If the *Py_TPFLAGS_MANAGED_DICT* bit is set in the *tp_flags* field, the traverse function must call *PyObject_ClearManagedDict()* like this :

```
PyObject_ClearManagedDict((PyObject*)self);
```

More information about Python's garbage collection scheme can be found in section *Supporting Cyclic Garbage Collection*.

Inheritance :

Group : *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

This field is inherited by subtypes together with *tp_traverse* and the *Py_TPFLAGS_HAVE_GC* flag bit : the flag bit, *tp_traverse*, and *tp_clear* are all inherited from the base type if they are all zero in the subtype.

Voir aussi

Object Life Cycle for details about how this slot relates to other slots.

richcmpfunc *PyTypeObject*.**tp_richcompare**

An optional pointer to the rich comparison function, whose signature is :

```
PyObject *tp_richcompare(PyObject *self, PyObject *other, int op);
```

The first parameter is guaranteed to be an instance of the type that is defined by *PyTypeObject*.

The function should return the result of the comparison (usually `Py_True` or `Py_False`). If the comparison is undefined, it must return `Py_NotImplemented`, if another error occurred it must return `NULL` and set an exception condition.

The following constants are defined to be used as the third argument for *tp_richcompare* and for *PyObject_RichCompare()* :

Constante	Comparaison
Py_LT	<
Py_LE	<=
Py_EQ	==
Py_NE	!=
Py_GT	>
Py_GE	>=

The following macro is defined to ease writing rich comparison functions :

Py_RETURN_RICHCOMPARE(VAL_A, VAL_B, op)

Return `Py_True` or `Py_False` from the function, depending on the result of a comparison. VAL_A and VAL_B must be orderable by C comparison operators (for example, they may be C ints or floats). The third argument specifies the requested operation, as for *PyObject_RichCompare()*.

The returned value is a new *strong reference*.

On error, sets an exception and returns `NULL` from the function.

Ajouté dans la version 3.7.

Inheritance :

Group : *tp_hash*, *tp_richcompare*

This field is inherited by subtypes together with *tp_hash* : a subtype inherits *tp_richcompare* and *tp_hash* when the subtype's *tp_richcompare* and *tp_hash* are both `NULL`.

Default :

PyBaseObject_Type provides a *tp_richcompare* implementation, which may be inherited. However, if only *tp_hash* is defined, not even the inherited function is used and instances of the type will not be able to participate in any comparisons.

Py_ssize_t *PyTypeObject*.**tp_weaklistoffset**

While this field is still supported, *Py_TPFLAGS_MANAGED_WEAKREF* should be used instead, if at all possible.

If the instances of this type are weakly referenceable, this field is greater than zero and contains the offset in the instance structure of the weak reference list head (ignoring the GC header, if present) ; this offset is used by *PyObject_ClearWeakRefs()* and the `PyWeakref_*` functions. The instance structure needs to include a field of type *PyObject** which is initialized to `NULL`.

Do not confuse this field with *tp_weaklist* ; that is the list head for weak references to the type object itself.

It is an error to set both the *Py_TPFLAGS_MANAGED_WEAKREF* bit and *tp_weaklistoffset*.

Inheritance :

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset ; this means that the subtype uses a different weak reference list head than the base type. Since the list head is always found via *tp_weaklistoffset*, this should not be a problem.

Default :

If the *Py_TPFLAGS_MANAGED_WEAKREF* bit is set in the *tp_flags* field, then *tp_weaklistoffset* will be set to a negative value, to indicate that it is unsafe to use this field.

getiterfunc *PyTypeObject*.**tp_iter**

An optional pointer to a function that returns an *iterator* for the object. Its presence normally signals that the instances of this type are *iterable* (although sequences may be iterable without this function).

This function has the same signature as *PyObject_GetIter()* :

```
PyObject *tp_iter(PyObject *self);
```

Inheritance :

This field is inherited by subtypes.

iternextfunc *PyTypeObject*.**tp_iternext**

An optional pointer to a function that returns the next item in an *iterator*. The signature is :

```
PyObject *tp_iternext(PyObject *self);
```

When the iterator is exhausted, it must return `NULL` ; a `StopIteration` exception may or may not be set. When another error occurs, it must return `NULL` too. Its presence signals that the instances of this type are iterators.

Iterator types should also define the *tp_iter* function, and that function should return the iterator instance itself (not a new iterator instance).

This function has the same signature as *PyIter_Next()*.

Inheritance :

This field is inherited by subtypes.

struct *PyMethodDef* **PyTypeObject.tp_methods*

An optional pointer to a static NULL-terminated array of *PyMethodDef* structures, declaring regular methods of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a method descriptor.

Inheritance :

This field is not inherited by subtypes (methods are inherited through a different mechanism).

struct *PyMemberDef* **PyTypeObject.tp_members*

An optional pointer to a static NULL-terminated array of *PyMemberDef* structures, declaring regular data members (fields or slots) of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a member descriptor.

Inheritance :

This field is not inherited by subtypes (members are inherited through a different mechanism).

struct *PyGetSetDef* **PyTypeObject.tp_getset*

An optional pointer to a static NULL-terminated array of *PyGetSetDef* structures, declaring computed attributes of instances of this type.

For each entry in the array, an entry is added to the type's dictionary (see *tp_dict* below) containing a getset descriptor.

Inheritance :

This field is not inherited by subtypes (computed attributes are inherited through a different mechanism).

PyTypeObject **PyTypeObject.tp_base*

An optional pointer to a base type from which type properties are inherited. At this level, only single inheritance is supported ; multiple inheritance require dynamically creating a type object by calling the metatype.

> **Note**
>
> Slot initialization is subject to the rules of initializing globals. C99 requires the initializers to be "address constants". Function designators like *PyType_GenericNew()*, with implicit conversion to a pointer, are valid C99 address constants.
>
> However, the unary '&' operator applied to a non-static variable like *PyBaseObject_Type* is not required to produce an address constant. Compilers may support this (gcc does), MSVC does not. Both compilers are strictly standard conforming in this particular behavior.
>
> Consequently, *tp_base* should be set in the extension module's init function.

Inheritance :

This field is not inherited by subtypes (obviously).

Default :

This field defaults to `&PyBaseObject_Type` (which to Python programmers is known as the type `object`).

PyObject **PyTypeObject.tp_dict*

The type's dictionary is stored here by *PyType_Ready()*.

This field should normally be initialized to NULL before PyType_Ready is called ; it may also be initialized to a dictionary containing initial attributes for the type. Once *PyType_Ready()* has initialized the type, extra attributes for the type may be added to this dictionary only if they don't correspond to overloaded operations (like `__add__()`). Once initialization for the type has finished, this field should be treated as read-only.

Some types may not store their dictionary in this slot. Use *PyType_GetDict()* to retrieve the dictionary for

an arbitrary type.

Modifié dans la version 3.12 : Internals detail : For static builtin types, this is always `NULL`. Instead, the dict for such types is stored on `PyInterpreterState`. Use *PyType_GetDict()* to get the dict for an arbitrary type.

Inheritance :

This field is not inherited by subtypes (though the attributes defined in here are inherited through a different mechanism).

Default :

If this field is `NULL`, *PyType_Ready()* will assign a new dictionary to it.

Avertissement

It is not safe to use *PyDict_SetItem()* on or otherwise modify *tp_dict* with the dictionary C-API.

descrgetfunc *PyTypeObject*.**tp_descr_get**

An optional pointer to a ”descriptor get” function.

The function signature is :

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *type);
```

Inheritance :

This field is inherited by subtypes.

descrsetfunc *PyTypeObject*.**tp_descr_set**

An optional pointer to a function for setting and deleting a descriptor's value.

The function signature is :

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

The *value* argument is set to `NULL` to delete the value.

Inheritance :

This field is inherited by subtypes.

Py_ssize_t *PyTypeObject*.**tp_dictoffset**

While this field is still supported, *Py_TPFLAGS_MANAGED_DICT* should be used instead, if at all possible.

If the instances of this type have a dictionary containing instance variables, this field is non-zero and contains the offset in the instances of the type of the instance variable dictionary; this offset is used by *PyObject_GenericGetAttr()*.

Do not confuse this field with *tp_dict*; that is the dictionary for attributes of the type object itself.

The value specifies the offset of the dictionary from the start of the instance structure.

The *tp_dictoffset* should be regarded as write-only. To get the pointer to the dictionary call *PyObject_GenericGetDict()*. Calling *PyObject_GenericGetDict()* may need to allocate memory for the dictionary, so it is may be more efficient to call *PyObject_GetAttr()* when accessing an attribute on the object.

It is an error to set both the *Py_TPFLAGS_MANAGED_DICT* bit and *tp_dictoffset*.

Inheritance :

This field is inherited by subtypes. A subtype should not override this offset; doing so could be unsafe, if C code tries to access the dictionary at the previous offset. To properly support inheritance, use *Py_TPFLAGS_MANAGED_DICT*.

Default :

This slot has no default. For *static types*, if the field is `NULL` then no `__dict__` gets created for instances.

If the *Py_TPFLAGS_MANAGED_DICT* bit is set in the *tp_flags* field, then *tp_dictoffset* will be set to `-1`, to indicate that it is unsafe to use this field.

initproc *PyTypeObject*.**tp_init**

An optional pointer to an instance initialization function.

This function corresponds to the `__init__()` method of classes. Like `__init__()`, it is possible to create an instance without calling `__init__()`, and it is possible to reinitialize an instance by calling its `__init__()` method again.

The function signature is :

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwds);
```

The self argument is the instance to be initialized ; the *args* and *kwds* arguments represent positional and keyword arguments of the call to `__init__()`.

The *tp_init* function, if not `NULL`, is called when an instance is created normally by calling its type, after the type's *tp_new* function has returned an instance of the type. If the *tp_new* function returns an instance of some other type that is not a subtype of the original type, no *tp_init* function is called ; if *tp_new* returns an instance of a subtype of the original type, the subtype's *tp_init* is called.

Returns `0` on success, `-1` and sets an exception on error.

Inheritance :

This field is inherited by subtypes.

Default :

For *static types* this field does not have a default.

allocfunc *PyTypeObject*.**tp_alloc**

An optional pointer to an instance allocation function.

The function signature is :

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems);
```

Inheritance :

Static subtypes inherit this slot, which will be *PyType_GenericAlloc()* if inherited from `object`.

Heap subtypes do not inherit this slot.

Default :

For heap subtypes, this field is always set to *PyType_GenericAlloc()*.

For static subtypes, this slot is inherited (see above).

newfunc *PyTypeObject*.**tp_new**

An optional pointer to an instance creation function.

The function signature is :

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds);
```

The *subtype* argument is the type of the object being created ; the *args* and *kwds* arguments represent positional and keyword arguments of the call to the type. Note that *subtype* doesn't have to equal the type whose *tp_new* function is called ; it may be a subtype of that type (but not an unrelated type).

The *tp_new* function should call `subtype->tp_alloc(subtype, nitems)` to allocate space for the object, and then do only as much further initialization as is absolutely necessary. Initialization that can safely be ignored or repeated should be placed in the *tp_init* handler. A good rule of thumb is that for immutable

types, all initialization should take place in *tp_new*, while for mutable types, most initialization should be deferred to *tp_init*.

Set the *Py_TPFLAGS_DISALLOW_INSTANTIATION* flag to disallow creating instances of the type in Python.

Inheritance :

This field is inherited by subtypes, except it is not inherited by *static types* whose *tp_base* is `NULL` or `&PyBaseObject_Type`.

Default :

For *static types* this field has no default. This means if the slot is defined as `NULL`, the type cannot be called to create new instances; presumably there is some other way to create instances, like a factory function.

freefunc *PyTypeObject*.**tp_free**

An optional pointer to an instance deallocation function. Its signature is :

```
void tp_free(void *self);
```

This function must free the memory allocated by *tp_alloc*.

Inheritance :

Static subtypes inherit this slot, which will be *PyObject_Free()* if inherited from `object`. Exception : If the type supports garbage collection (i.e., the *Py_TPFLAGS_HAVE_GC* flag is set in *tp_flags*) and it would inherit *PyObject_Free()*, then this slot is not inherited but instead defaults to *PyObject_GC_Del()*.

Heap subtypes do not inherit this slot.

Default :

For *heap subtypes*, this slot defaults to a deallocator suitable to match *PyType_GenericAlloc()* and the value of the *Py_TPFLAGS_HAVE_GC* flag.

For static subtypes, this slot is inherited (see above).

inquiry *PyTypeObject*.**tp_is_gc**

An optional pointer to a function called by the garbage collector.

The garbage collector needs to know whether a particular object is collectible or not. Normally, it is sufficient to look at the object's type's *tp_flags* field, and check the *Py_TPFLAGS_HAVE_GC* flag bit. But some types have a mixture of statically and dynamically allocated instances, and the statically allocated instances are not collectible. Such types should define this function; it should return `1` for a collectible instance, and `0` for a non-collectible instance. The signature is :

```
int tp_is_gc(PyObject *self);
```

(The only example of this are types themselves. The metatype, *PyType_Type*, defines this function to distinguish between statically and *dynamically allocated types*.)

Inheritance :

This field is inherited by subtypes.

Default :

This slot has no default. If this field is `NULL`, *Py_TPFLAGS_HAVE_GC* is used as the functional equivalent.

PyObject ***PyTypeObject**.**tp_bases**

Tuple of base types.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is *initialized*.

For dynamically created classes, the `Py_tp_bases` *slot* can be used instead of the *bases* argument of *PyType_FromSpecWithBases()*. The argument form is preferred.

> **Avertissement**
>
> Multiple inheritance does not work well for statically defined types. If you set `tp_bases` to a tuple, Python will not raise an error, but some slots will only be inherited from the first base.

Inheritance :

This field is not inherited.

PyObject **PyTypeObject.tp_mro*

Tuple containing the expanded set of base types, starting with the type itself and ending with `object`, in Method Resolution Order.

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is *initialized*.

Inheritance :

This field is not inherited; it is calculated fresh by *PyType_Ready()*.

PyObject **PyTypeObject.tp_cache*

Unused. Internal use only.

Inheritance :

This field is not inherited.

void **PyTypeObject.tp_subclasses*

A collection of subclasses. Internal use only. May be an invalid pointer.

To get a list of subclasses, call the Python method `__subclasses__()`.

Modifié dans la version 3.12 : For some types, this field does not hold a valid *PyObject**. The type was changed to `void*` to indicate this.

Inheritance :

This field is not inherited.

PyObject **PyTypeObject.tp_weaklist*

Weak reference list head, for weak references to this type object. Not inherited. Internal use only.

Modifié dans la version 3.12 : Internals detail : For the static builtin types this is always `NULL`, even if weakrefs are added. Instead, the weakrefs for each are stored on `PyInterpreterState`. Use the public C-API or the internal `_PyObject_GET_WEAKREFS_LISTPTR()` macro to avoid the distinction.

Inheritance :

This field is not inherited.

destructor *PyTypeObject*.tp_del*

This field is deprecated. Use *tp_finalize* instead.

unsigned int *PyTypeObject*.tp_version_tag*

Used to index into the method cache. Internal use only.

Inheritance :

This field is not inherited.

destructor *PyTypeObject*.tp_finalize*

An optional pointer to an instance finalization function. This is the C implementation of the `__del__()` special method. Its signature is :

```
void tp_finalize(PyObject *self);
```

The primary purpose of finalization is to perform any non-trivial cleanup that must be performed before the object is destroyed, while the object and any other objects it directly or indirectly references are still in a

consistent state. The finalizer is allowed to execute arbitrary Python code.

Before Python automatically finalizes an object, some of the object's direct or indirect referents might have themselves been automatically finalized. However, none of the referents will have been automatically cleared (*tp_clear*) yet.

Other non-finalized objects might still be using a finalized object, so the finalizer must leave the object in a sane state (e.g., invariants are still met).

Note

After Python automatically finalizes an object, Python might start automatically clearing (*tp_clear*) the object and its referents (direct and indirect). Cleared objects are not guaranteed to be in a consistent state ; a finalized object must be able to tolerate cleared referents.

Note

An object is not guaranteed to be automatically finalized before its destructor (*tp_dealloc*) is called. It is recommended to call *PyObject_CallFinalizerFromDealloc()* at the beginning of `tp_dealloc` to guarantee that the object is always finalized before destruction.

Note

The *tp_finalize* function can be called from any thread, although the *GIL* will be held.

Note

The `tp_finalize` function can be called during shutdown, after some global variables have been deleted. See the documentation of the `__del__()` method for details.

When Python finalizes an object, it behaves like the following algorithm :

1. Python might mark the object as *finalized*. Currently, Python always marks objects whose type supports garbage collection (i.e., the *Py_TPFLAGS_HAVE_GC* flag is set in *tp_flags*) and never marks other types of objects ; this might change in a future version.
2. If the object is not marked as *finalized* and its `tp_finalize` finalizer function is non-`NULL`, the finalizer function is called.
3. If the finalizer function was called and the finalizer made the object reachable (i.e., there is a reference to the object and it is not a member of a *cyclic isolate*), then the finalizer is said to have *resurrected* the object. It is unspecified whether the finalizer can also resurrect the object by adding a new reference to the object that does not make it reachable, i.e., the object is (still) a member of a cyclic isolate.
4. If the finalizer resurrected the object, the object's pending destruction is canceled and the object's *finalized* mark might be removed if present. Currently, Python never removes the *finalized* mark ; this might change in a future version.

Automatic finalization refers to any finalization performed by Python except via calls to *PyObject_CallFinalizer()* or *PyObject_CallFinalizerFromDealloc()*. No guarantees are made about when, if, or how often an object is automatically finalized, except :

— Python will not automatically finalize an object if it is reachable, i.e., there is a reference to it and it is not a member of a *cyclic isolate*.

— Python will not automatically finalize an object if finalizing it would not mark the object as *finalized*. Currently, this applies to objects whose type does not support garbage collection, i.e., the *Py_TPFLAGS_HAVE_GC* flag is not set. Such objects can still be manually finalized by calling *PyObject_CallFinalizer()* or *PyObject_CallFinalizerFromDealloc()*.

— Python will not automatically finalize any two members of a *cyclic isolate* concurrently.

— Python will not automatically finalize an object after it has automatically cleared (*tp_clear*) the object.
— If an object is a member of a *cyclic isolate*, Python will not automatically finalize it after automatically clearing (see *tp_clear*) any other member.
— Python will automatically finalize every member of a *cyclic isolate* before it automatically clears (see *tp_clear*) any of them.
— If Python is going to automatically clear an object (*tp_clear*), it will automatically finalize the object first.

Python currently only automatically finalizes objects that are members of a *cyclic isolate*, but future versions might finalize objects regularly before their destruction.

To manually finalize an object, do not call this function directly ; call *PyObject_CallFinalizer()* or *PyObject_CallFinalizerFromDealloc()* instead.

tp_finalize should leave the current exception status unchanged. The recommended way to write a non-trivial finalizer is to back up the exception at the beginning by calling *PyErr_GetRaisedException()* and restore the exception at the end by calling *PyErr_SetRaisedException()*. If an exception is encountered in the middle of the finalizer, log and clear it with *PyErr_WriteUnraisable()* or *PyErr_FormatUnraisable()*. For example :

```
static void
foo_finalize(PyObject *self)
{
    // Save the current exception, if any.
    PyObject *exc = PyErr_GetRaisedException();

    // ...

    if (do_something_that_might_raise() != success_indicator) {
        PyErr_WriteUnraisable(self);
        goto done;
    }

done:
    // Restore the saved exception.  This silently discards any exception
    // raised above, so be sure to call PyErr_WriteUnraisable first if
    // necessary.
    PyErr_SetRaisedException(exc);
}
```

Inheritance :

This field is inherited by subtypes.

Ajouté dans la version 3.4.

Modifié dans la version 3.8 : Before version 3.8 it was necessary to set the *Py_TPFLAGS_HAVE_FINALIZE* flags bit in order for this field to be used. This is no longer required.

Voir aussi

— **PEP 442** : "Safe object finalization"
— *Object Life Cycle* for details about how this slot relates to other slots.
— *PyObject_CallFinalizer()*
— *PyObject_CallFinalizerFromDealloc()*

vectorcallfunc *PyTypeObject*.**tp_vectorcall**

A *vectorcall function* to use for calls of this type object (rather than instances). In other words, `tp_vectorcall` can be used to optimize `type.__call__`, which typically returns a new instance of *type*.

As with any vectorcall function, if `tp_vectorcall` is `NULL`, the *tp_call* protocol

(`Py_TYPE(type)->tp_call`) is used instead.

> **Note**
>
> The *vectorcall protocol* requires that the vectorcall function has the same behavior as the corresponding `tp_call`. This means that `type->tp_vectorcall` must match the behavior of `Py_TYPE(type)->tp_call`.
>
> Specifically, if *type* uses the default metaclass, `type->tp_vectorcall` must behave the same as *PyType_Type*`->tp_call`, which :
> — calls `type->tp_new`,
> — if the result is a subclass of *type*, calls `type->tp_init` on the result of `tp_new`, and
> — returns the result of `tp_new`.
>
> Typically, `tp_vectorcall` is overridden to optimize this process for specific *tp_new* and *tp_init*. When doing this for user-subclassable types, note that both can be overridden (using `__new__()` and `__init__()`, respectively).

Inheritance :

This field is never inherited.

Ajouté dans la version 3.9 : (the field exists since 3.8 but it's only used since 3.9)

unsigned char *PyTypeObject*.**tp_watched**

Internal. Do not use.

Ajouté dans la version 3.12.

13.4.6 Static Types

Traditionally, types defined in C code are *static*, that is, a static *PyTypeObject* structure is defined directly in code and initialized using *PyType_Ready()*.

This results in types that are limited relative to types defined in Python :
— Static types are limited to one base, i.e. they cannot use multiple inheritance.
— Static type objects (but not necessarily their instances) are immutable. It is not possible to add or modify the type object's attributes from Python.
— Static type objects are shared across *sub-interpreters*, so they should not include any subinterpreter-specific state.

Also, since *PyTypeObject* is only part of the *Limited API* as an opaque struct, any extension modules using static types must be compiled for a specific Python minor version.

13.4.7 Heap Types

An alternative to *static types* is *heap-allocated types*, or *heap types* for short, which correspond closely to classes created by Python's `class` statement. Heap types have the *Py_TPFLAGS_HEAPTYPE* flag set.

This is done by filling a *PyType_Spec* structure and calling *PyType_FromSpec()*, *PyType_FromSpecWithBases()*, *PyType_FromModuleAndSpec()*, or *PyType_FromMetaclass()*.

13.4.8 Number Object Structures

type **PyNumberMethods**

This structure holds pointers to the functions which an object uses to implement the number protocol. Each function is used by the function of similar name documented in the *Number Protocol* section.

Here is the structure definition :

```
typedef struct {
     binaryfunc nb_add;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    binaryfunc nb_subtract;
    binaryfunc nb_multiply;
    binaryfunc nb_remainder;
    binaryfunc nb_divmod;
    ternaryfunc nb_power;
    unaryfunc nb_negative;
    unaryfunc nb_positive;
    unaryfunc nb_absolute;
    inquiry nb_bool;
    unaryfunc nb_invert;
    binaryfunc nb_lshift;
    binaryfunc nb_rshift;
    binaryfunc nb_and;
    binaryfunc nb_xor;
    binaryfunc nb_or;
    unaryfunc nb_int;
    void *nb_reserved;
    unaryfunc nb_float;

    binaryfunc nb_inplace_add;
    binaryfunc nb_inplace_subtract;
    binaryfunc nb_inplace_multiply;
    binaryfunc nb_inplace_remainder;
    ternaryfunc nb_inplace_power;
    binaryfunc nb_inplace_lshift;
    binaryfunc nb_inplace_rshift;
    binaryfunc nb_inplace_and;
    binaryfunc nb_inplace_xor;
    binaryfunc nb_inplace_or;

    binaryfunc nb_floor_divide;
    binaryfunc nb_true_divide;
    binaryfunc nb_inplace_floor_divide;
    binaryfunc nb_inplace_true_divide;

    unaryfunc nb_index;

    binaryfunc nb_matrix_multiply;
    binaryfunc nb_inplace_matrix_multiply;
} PyNumberMethods;
```

Note

Binary and ternary functions must check the type of all their operands, and implement the necessary conversions (at least one of the operands is an instance of the defined type). If the operation is not defined for the given operands, binary and ternary functions must return `Py_NotImplemented`, if another error occurred they must return `NULL` and set an exception.

Note

The *nb_reserved* field should always be `NULL`. It was previously called `nb_long`, and was renamed in Python 3.0.1.

binaryfunc *PyNumberMethods*.**nb_add**

binaryfunc *PyNumberMethods*.**nb_subtract**

binaryfunc *PyNumberMethods*.**nb_multiply**

binaryfunc *PyNumberMethods*.**nb_remainder**

binaryfunc *PyNumberMethods*.**nb_divmod**

ternaryfunc *PyNumberMethods*.**nb_power**

unaryfunc *PyNumberMethods*.**nb_negative**

unaryfunc *PyNumberMethods*.**nb_positive**

unaryfunc *PyNumberMethods*.**nb_absolute**

inquiry *PyNumberMethods*.**nb_bool**

unaryfunc *PyNumberMethods*.**nb_invert**

binaryfunc *PyNumberMethods*.**nb_lshift**

binaryfunc *PyNumberMethods*.**nb_rshift**

binaryfunc *PyNumberMethods*.**nb_and**

binaryfunc *PyNumberMethods*.**nb_xor**

binaryfunc *PyNumberMethods*.**nb_or**

unaryfunc *PyNumberMethods*.**nb_int**

void **PyNumberMethods*.**nb_reserved**

unaryfunc *PyNumberMethods*.**nb_float**

binaryfunc *PyNumberMethods*.**nb_inplace_add**

binaryfunc *PyNumberMethods*.**nb_inplace_subtract**

binaryfunc *PyNumberMethods*.**nb_inplace_multiply**

binaryfunc *PyNumberMethods*.**nb_inplace_remainder**

ternaryfunc *PyNumberMethods*.**nb_inplace_power**

binaryfunc *PyNumberMethods*.**nb_inplace_lshift**

binaryfunc *PyNumberMethods*.**nb_inplace_rshift**

binaryfunc *PyNumberMethods*.**nb_inplace_and**

binaryfunc *PyNumberMethods*.**nb_inplace_xor**

binaryfunc *PyNumberMethods*.**nb_inplace_or**

binaryfunc *PyNumberMethods*.**nb_floor_divide**

binaryfunc *PyNumberMethods*.**nb_true_divide**

binaryfunc *PyNumberMethods*.**nb_inplace_floor_divide**

binaryfunc *PyNumberMethods*.**nb_inplace_true_divide**

unaryfunc *PyNumberMethods*.**nb_index**

binaryfunc *PyNumberMethods*.**nb_matrix_multiply**

binaryfunc *PyNumberMethods*.**nb_inplace_matrix_multiply**

13.4.9 Mapping Object Structures

type **PyMappingMethods**

This structure holds pointers to the functions which an object uses to implement the mapping protocol. It has three members :

lenfunc *PyMappingMethods*.**mp_length**

This function is used by *PyMapping_Size()* and *PyObject_Size()*, and has the same signature. This slot may be set to NULL if the object has no defined length.

binaryfunc *PyMappingMethods*.**mp_subscript**

This function is used by *PyObject_GetItem()* and *PySequence_GetSlice()*, and has the same signature as PyObject_GetItem(). This slot must be filled for the *PyMapping_Check()* function to return 1, it can be NULL otherwise.

objobjargproc *PyMappingMethods*.**mp_ass_subscript**

This function is used by *PyObject_SetItem()*, *PyObject_DelItem()*, *PySequence_SetSlice()* and *PySequence_DelSlice()*. It has the same signature as PyObject_SetItem(), but *v* can also be set to NULL to delete an item. If this slot is NULL, the object does not support item assignment and deletion.

13.4.10 Sequence Object Structures

type **PySequenceMethods**

This structure holds pointers to the functions which an object uses to implement the sequence protocol.

lenfunc *PySequenceMethods*.**sq_length**

This function is used by *PySequence_Size()* and *PyObject_Size()*, and has the same signature. It is also used for handling negative indices via the *sq_item* and the *sq_ass_item* slots.

binaryfunc *PySequenceMethods*.**sq_concat**

This function is used by *PySequence_Concat()* and has the same signature. It is also used by the + operator, after trying the numeric addition via the *nb_add* slot.

ssizeargfunc *PySequenceMethods*.**sq_repeat**

This function is used by *PySequence_Repeat()* and has the same signature. It is also used by the * operator, after trying numeric multiplication via the *nb_multiply* slot.

ssizeargfunc *PySequenceMethods*.**sq_item**

This function is used by *PySequence_GetItem()* and has the same signature. It is also used by *PyObject_GetItem()*, after trying the subscription via the *mp_subscript* slot. This slot must be filled for the *PySequence_Check()* function to return 1, it can be NULL otherwise.

Negative indexes are handled as follows : if the *sq_length* slot is filled, it is called and the sequence length is used to compute a positive index which is passed to *sq_item*. If sq_length is NULL, the index is passed as is to the function.

ssizeobjargproc *PySequenceMethods*.**sq_ass_item**

This function is used by *PySequence_SetItem()* and has the same signature. It is also used by *PyObject_SetItem()* and *PyObject_DelItem()*, after trying the item assignment and deletion via the *mp_ass_subscript* slot. This slot may be left to NULL if the object does not support item assignment and deletion.

objobjproc *PySequenceMethods*.**sq_contains**

This function may be used by *PySequence_Contains()* and has the same signature. This slot may be left to NULL, in this case PySequence_Contains() simply traverses the sequence until it finds a match.

binaryfunc *PySequenceMethods*.**sq_inplace_concat**

This function is used by *PySequence_InPlaceConcat()* and has the same signature. It should modify its first operand, and return it. This slot may be left to NULL, in this case PySequence_InPlaceConcat() will fall back to *PySequence_Concat()*. It is also used by the augmented assignment +=, after trying numeric in-place addition via the *nb_inplace_add* slot.

ssizeargfunc *PySequenceMethods*.**sq_inplace_repeat**

This function is used by *PySequence_InPlaceRepeat()* and has the same signature. It should modify its first operand, and return it. This slot may be left to NULL, in this case PySequence_InPlaceRepeat() will fall back to *PySequence_Repeat()*. It is also used by the augmented assignment *=, after trying numeric in-place multiplication via the *nb_inplace_multiply* slot.

13.4.11 Buffer Object Structures

type **PyBufferProcs**

This structure holds pointers to the functions required by the *Buffer protocol*. The protocol defines how an exporter object can expose its internal data to consumer objects.

getbufferproc *PyBufferProcs*.**bf_getbuffer**

The signature of this function is :

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

Handle a request to *exporter* to fill in *view* as specified by *flags*. Except for point (3), an implementation of this function MUST take these steps :

(1) Check if the request can be met. If not, raise BufferError, set view->obj to NULL and return -1.

(2) Fill in the requested fields.

(3) Increment an internal counter for the number of exports.

(4) Set view->obj to *exporter* and increment view->obj.

(5) Return 0.

If *exporter* is part of a chain or tree of buffer providers, two main schemes can be used :

— Re-export : Each member of the tree acts as the exporting object and sets view->obj to a new reference to itself.

— Redirect : The buffer request is redirected to the root object of the tree. Here, view->obj will be a new reference to the root object.

The individual fields of *view* are described in section *Buffer structure*, the rules how an exporter must react to specific requests are in section *Buffer request types*.

All memory pointed to in the *Py_buffer* structure belongs to the exporter and must remain valid until there are no consumers left. *format*, *shape*, *strides*, *suboffsets* and *internal* are read-only for the consumer.

PyBuffer_FillInfo() provides an easy way of exposing a simple bytes buffer while dealing correctly with all request types.

PyObject_GetBuffer() is the interface for the consumer that wraps this function.

releasebufferproc *PyBufferProcs*.**bf_releasebuffer**

The signature of this function is :

```
void (PyObject *exporter, Py_buffer *view);
```

Handle a request to release the resources of the buffer. If no resources need to be released, *PyBufferProcs.bf_releasebuffer* may be NULL. Otherwise, a standard implementation of this function will take these optional steps :

(1) Decrement an internal counter for the number of exports.

(2) If the counter is 0, free all memory associated with *view*.

The exporter MUST use the *internal* field to keep track of buffer-specific resources. This field is guaranteed to remain constant, while a consumer MAY pass a copy of the original buffer as the *view* argument.

This function MUST NOT decrement view->obj, since that is done automatically in *PyBuffer_Release()* (this scheme is useful for breaking reference cycles).

PyBuffer_Release() is the interface for the consumer that wraps this function.

13.4.12 Async Object Structures

Ajouté dans la version 3.5.

type **PyAsyncMethods**

This structure holds pointers to the functions required to implement *awaitable* and *asynchronous iterator* objects.

Here is the structure definition :

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
    sendfunc am_send;
} PyAsyncMethods;
```

unaryfunc *PyAsyncMethods*.**am_await**

The signature of this function is :

```
PyObject *am_await(PyObject *self);
```

The returned object must be an *iterator*, i.e. *PyIter_Check()* must return `1` for it.

This slot may be set to `NULL` if an object is not an *awaitable*.

unaryfunc *PyAsyncMethods*.**am_aiter**

The signature of this function is :

```
PyObject *am_aiter(PyObject *self);
```

Must return an *asynchronous iterator* object. See `__anext__()` for details.

This slot may be set to `NULL` if an object does not implement asynchronous iteration protocol.

unaryfunc *PyAsyncMethods*.**am_anext**

The signature of this function is :

```
PyObject *am_anext(PyObject *self);
```

Must return an *awaitable* object. See `__anext__()` for details. This slot may be set to `NULL`.

sendfunc *PyAsyncMethods*.**am_send**

The signature of this function is :

```
PySendResult am_send(PyObject *self, PyObject *arg, PyObject **result);
```

See *PyIter_Send()* for details. This slot may be set to `NULL`.

Ajouté dans la version 3.10.

13.4.13 Slot Type typedefs

typedef *PyObject* *(***allocfunc**)(*PyTypeObject* *cls, *Py_ssize_t* nitems)

Part of the Stable ABI. The purpose of this function is to separate memory allocation from memory initialization. It should return a pointer to a block of memory of adequate length for the instance, suitably aligned, and initialized to zeros, but with *ob_refcnt* set to `1` and *ob_type* set to the type argument. If the type's *tp_itemsize* is non-zero, the object's *ob_size* field should be initialized to *nitems* and the length of the allocated memory block should be `tp_basicsize + nitems*tp_itemsize`, rounded up to a multiple of `sizeof(void*)` ; otherwise, *nitems* is not used and the length of the block should be *tp_basicsize*.

This function should not do any other instance initialization, not even to allocate additional memory ; that should be done by *tp_new*.

typedef void (***destructor**)(*PyObject**)

Part of the Stable ABI.

typedef void (***freefunc**)(void*)

See *tp_free*.

typedef *PyObject* *(***newfunc**)(*PyTypeObject**, *PyObject**, *PyObject**)

Part of the Stable ABI. See *tp_new*.

typedef int (***initproc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the Stable ABI. See *tp_init*.

typedef *PyObject* *(***reprfunc**)(*PyObject**)

Part of the Stable ABI. See *tp_repr*.

typedef *PyObject* *(***getattrfunc**)(*PyObject* *self, char *attr)

Part of the Stable ABI. Return the value of the named attribute for the object.

typedef int (***setattrfunc**)(*PyObject* *self, char *attr, *PyObject* *value)

Part of the Stable ABI. Set the value of the named attribute for the object. The value argument is set to `NULL` to delete the attribute.

typedef *PyObject* *(***getattrofunc**)(*PyObject* *self, *PyObject* *attr)

Part of the Stable ABI. Return the value of the named attribute for the object.

See *tp_getattro*.

typedef int (***setattrofunc**)(*PyObject* *self, *PyObject* *attr, *PyObject* *value)

Part of the Stable ABI. Set the value of the named attribute for the object. The value argument is set to `NULL` to delete the attribute.

See *tp_setattro*.

typedef *PyObject* *(***descrgetfunc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the Stable ABI. See *tp_descr_get*.

typedef int (***descrsetfunc**)(*PyObject**, *PyObject**, *PyObject**)

Part of the Stable ABI. See *tp_descr_set*.

typedef *Py_hash_t* (***hashfunc**)(*PyObject**)

Part of the Stable ABI. See *tp_hash*.

typedef *PyObject* *(***richcmpfunc**)(*PyObject**, *PyObject**, int)

Part of the Stable ABI. See *tp_richcompare*.

typedef *PyObject* *(***getiterfunc**)(*PyObject**)

Part of the Stable ABI. See *tp_iter*.

typedef *PyObject* *(***iternextfunc**)(*PyObject**)

Part of the Stable ABI. See *tp_iternext*.

typedef *Py_ssize_t* (***lenfunc**)(*PyObject**)

Part of the Stable ABI.

typedef int (***getbufferproc**)(*PyObject**, *Py_buffer**, int)

Part of the Stable ABI *Depuis la version 3.12.*

typedef void (***releasebufferproc**)(*PyObject**, *Py_buffer**)

Part of the Stable ABI *Depuis la version 3.12.*

typedef *PyObject* *(***unaryfunc**)(*PyObject**)

Part of the Stable ABI.

typedef *PyObject* *(***binaryfunc**)(*PyObject**, *PyObject**)
Part of the Stable ABI.

typedef *PySendResult* (***sendfunc**)(*PyObject**, *PyObject**, *PyObject*****)
See *am_send*.

typedef *PyObject* *(***ternaryfunc**)(*PyObject**, *PyObject**, *PyObject**)
Part of the Stable ABI.

typedef *PyObject* *(***ssizeargfunc**)(*PyObject**, *Py_ssize_t*)
Part of the Stable ABI.

typedef int (***ssizeobjargproc**)(*PyObject**, *Py_ssize_t*, *PyObject**)
Part of the Stable ABI.

typedef int (***objobjproc**)(*PyObject**, *PyObject**)
Part of the Stable ABI.

typedef int (***objobjargproc**)(*PyObject**, *PyObject**, *PyObject**)
Part of the Stable ABI.

13.4.14 Exemples

The following are simple examples of Python type definitions. They include common usage you may encounter. Some demonstrate tricky corner cases. For more examples, practical info, and a tutorial, see defining-new-types and new-types-topics.

A basic *static type* :

```
typedef struct {
    PyObject_HEAD
    const char *data;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_new = myobj_new,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
};
```

You may also find older code (especially in the CPython code base) with a more verbose initializer :

```
static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "mymod.MyObject",               /* tp_name */
    sizeof(MyObject),               /* tp_basicsize */
    0,                              /* tp_itemsize */
    (destructor)myobj_dealloc,      /* tp_dealloc */
    0,                              /* tp_vectorcall_offset */
    0,                              /* tp_getattr */
    0,                              /* tp_setattr */
    0,                              /* tp_as_async */
    (reprfunc)myobj_repr,           /* tp_repr */
    0,                              /* tp_as_number */
    0,                              /* tp_as_sequence */
    0,                              /* tp_as_mapping */
```

(suite sur la page suivante)

(suite de la page précédente)

```
    0,                              /* tp_hash */
    0,                              /* tp_call */
    0,                              /* tp_str */
    0,                              /* tp_getattro */
    0,                              /* tp_setattro */
    0,                              /* tp_as_buffer */
    0,                              /* tp_flags */
    PyDoc_STR("My objects"),        /* tp_doc */
    0,                              /* tp_traverse */
    0,                              /* tp_clear */
    0,                              /* tp_richcompare */
    0,                              /* tp_weaklistoffset */
    0,                              /* tp_iter */
    0,                              /* tp_iternext */
    0,                              /* tp_methods */
    0,                              /* tp_members */
    0,                              /* tp_getset */
    0,                              /* tp_base */
    0,                              /* tp_dict */
    0,                              /* tp_descr_get */
    0,                              /* tp_descr_set */
    0,                              /* tp_dictoffset */
    0,                              /* tp_init */
    0,                              /* tp_alloc */
    myobj_new,                      /* tp_new */
};
```

A type that supports weakrefs, instance dicts, and hashing :

```
typedef struct {
    PyObject_HEAD
    const char *data;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
         Py_TPFLAGS_HAVE_GC | Py_TPFLAGS_MANAGED_DICT |
         Py_TPFLAGS_MANAGED_WEAKREF,
    .tp_new = myobj_new,
    .tp_traverse = (traverseproc)myobj_traverse,
    .tp_clear = (inquiry)myobj_clear,
    .tp_alloc = PyType_GenericNew,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
    .tp_hash = (hashfunc)myobj_hash,
    .tp_richcompare = PyBaseObject_Type.tp_richcompare,
};
```

A str subclass that cannot be subclassed and cannot be called to create instances (e.g. uses a separate factory func) using *Py_TPFLAGS_DISALLOW_INSTANTIATION* flag :

```
typedef struct {
    PyUnicodeObject raw;
```

(suite sur la page suivante)

(suite de la page précédente)

```
    char *extra;
} MyStr;

static PyTypeObject MyStr_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyStr",
    .tp_basicsize = sizeof(MyStr),
    .tp_base = NULL,  // set to &PyUnicode_Type in module init
    .tp_doc = PyDoc_STR("my custom str"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_DISALLOW_INSTANTIATION,
    .tp_repr = (reprfunc)myobj_repr,
};
```

The simplest *static type* with fixed-length instances :

```
typedef struct {
    PyObject_HEAD
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
};
```

The simplest *static type* with variable-length instances :

```
typedef struct {
    PyObject_VAR_HEAD
    const char *data[1];
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject) - sizeof(char *),
    .tp_itemsize = sizeof(char *),
};
```

13.5 Supporting Cyclic Garbage Collection

Python's support for detecting and collecting garbage which involves circular references requires support from object types which are "containers" for other objects which may also be containers. Types which do not store references to other objects, or which only store references to atomic types (such as numbers or strings), do not need to provide any explicit support for garbage collection.

To create a container type, the `tp_flags` field of the type object must include the `Py_TPFLAGS_HAVE_GC` and provide an implementation of the `tp_traverse` handler. If instances of the type are mutable, a `tp_clear` implementation must also be provided.

Py_TPFLAGS_HAVE_GC

Objects with a type with this flag set must conform with the rules documented here. For convenience these objects will be referred to as container objects.

Constructors for container types must conform to two rules :

1. The memory for the object must be allocated using `PyObject_GC_New` or `PyObject_GC_NewVar`.
2. Once all the fields which may contain references to other containers are initialized, it must call `PyObject_GC_Track()`.

Similarly, the deallocator for the object must conform to a similar pair of rules :

1. Before fields which refer to other containers are invalidated, `PyObject_GC_UnTrack()` must be called.
2. The object's memory must be deallocated using `PyObject_GC_Del()`.

> **Avertissement**
>
> If a type adds the Py_TPFLAGS_HAVE_GC, then it *must* implement at least a `tp_traverse` handler or explicitly use one from its subclass or subclasses.
>
> When calling `PyType_Ready()` or some of the APIs that indirectly call it like `PyType_FromSpecWithBases()` or `PyType_FromSpec()` the interpreter will automatically populate the `tp_flags`, `tp_traverse` and `tp_clear` fields if the type inherits from a class that implements the garbage collector protocol and the child class does *not* include the `Py_TPFLAGS_HAVE_GC` flag.

PyObject_GC_New(TYPE, typeobj)

Analogous to `PyObject_New` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

Do not call this directly to allocate memory for an object; call the type's `tp_alloc` slot instead.

When populating a type's `tp_alloc` slot, `PyType_GenericAlloc()` is preferred over a custom function that simply calls this macro.

Memory allocated by this macro must be freed with `PyObject_GC_Del()` (usually called via the object's `tp_free` slot).

> **Voir aussi**
>
> — `PyObject_GC_Del()`
> — `PyObject_New`
> — `PyType_GenericAlloc()`
> — `tp_alloc`

PyObject_GC_NewVar(TYPE, typeobj, size)

Analogous to `PyObject_NewVar` but for container objects with the `Py_TPFLAGS_HAVE_GC` flag set.

Do not call this directly to allocate memory for an object; call the type's `tp_alloc` slot instead.

When populating a type's `tp_alloc` slot, `PyType_GenericAlloc()` is preferred over a custom function that simply calls this macro.

Memory allocated by this macro must be freed with `PyObject_GC_Del()` (usually called via the object's `tp_free` slot).

> **Voir aussi**
>
> — `PyObject_GC_Del()`
> — `PyObject_NewVar`
> — `PyType_GenericAlloc()`
> — `tp_alloc`

PyObject ***PyUnstable_Object_GC_NewWithExtraData**(*PyTypeObject* *type, size_t extra_size)

> This is *Unstable API*. It may change without warning in minor releases.

Analogous to *PyObject_GC_New* but allocates *extra_size* bytes at the end of the object (at offset *tp_basicsize*). The allocated memory is initialized to zeros, except for the *Python object header*.

The extra data will be deallocated with the object, but otherwise it is not managed by Python.

Memory allocated by this function must be freed with *PyObject_GC_Del()* (usually called via the object's *tp_free* slot).

Avertissement

The function is marked as unstable because the final mechanism for reserving extra data after an instance is not yet decided. For allocating a variable number of fields, prefer using *PyVarObject* and *tp_itemsize* instead.

Ajouté dans la version 3.12.

PyObject_GC_Resize(TYPE, op, newsize)

Resize an object allocated by *PyObject_NewVar*. Returns the resized object of type `TYPE*` (refers to any C type) or `NULL` on failure.

op must be of type *PyVarObject** and must not be tracked by the collector yet. *newsize* must be of type *Py_ssize_t*.

void **PyObject_GC_Track**(*PyObject* *op)

Part of the Stable ABI. Adds the object *op* to the set of container objects tracked by the collector. The collector can run at unexpected times so objects must be valid while being tracked. This should be called once all the fields followed by the *tp_traverse* handler become valid, usually near the end of the constructor.

int **PyObject_IS_GC**(*PyObject* *obj)

Returns non-zero if the object implements the garbage collector protocol, otherwise returns 0.

The object cannot be tracked by the garbage collector if this function returns 0.

int **PyObject_GC_IsTracked**(*PyObject* *op)

Part of the Stable ABI *Depuis la version 3.9.* Returns 1 if the object type of *op* implements the GC protocol and *op* is being currently tracked by the garbage collector and 0 otherwise.

This is analogous to the Python function `gc.is_tracked()`.

Ajouté dans la version 3.9.

int **PyObject_GC_IsFinalized**(*PyObject* *op)

Part of the Stable ABI *Depuis la version 3.9.* Returns 1 if the object type of *op* implements the GC protocol and *op* has been already finalized by the garbage collector and 0 otherwise.

This is analogous to the Python function `gc.is_finalized()`.

Ajouté dans la version 3.9.

void **PyObject_GC_Del**(void *op)

Part of the Stable ABI. Releases memory allocated to an object using *PyObject_GC_New* or *PyObject_GC_NewVar*.

Do not call this directly to free an object's memory; call the type's *tp_free* slot instead.

Do not use this for memory allocated by *PyObject_New*, *PyObject_NewVar*, or related allocation functions; use *PyObject_Free()* instead.

Voir aussi

— *PyObject_Free()* is the non-GC equivalent of this function.
— *PyObject_GC_New*
— *PyObject_GC_NewVar*

```
    — PyType_GenericAlloc()
    — tp_free
```

void **PyObject_GC_UnTrack**(void *op)

Part of the Stable ABI. Remove the object *op* from the set of container objects tracked by the collector. Note that `PyObject_GC_Track()` can be called again on this object to add it back to the set of tracked objects. The deallocator (`tp_dealloc` handler) should call this for the object before any of the fields used by the `tp_traverse` handler become invalid.

Modifié dans la version 3.8 : The `_PyObject_GC_TRACK()` and `_PyObject_GC_UNTRACK()` macros have been removed from the public C API.

The `tp_traverse` handler accepts a function parameter of this type :

typedef int (***visitproc**)(*PyObject* *object, void *arg)

Part of the Stable ABI. Type of the visitor function passed to the `tp_traverse` handler. The function should be called with an object to traverse as *object* and the third parameter to the `tp_traverse` handler as *arg*. The Python core uses several visitor functions to implement cyclic garbage detection ; it's not expected that users will need to write their own visitor functions.

The `tp_traverse` handler must have the following type :

typedef int (***traverseproc**)(*PyObject* *self, *visitproc* visit, void *arg)

Part of the Stable ABI. Traversal function for a container object. Implementations must call the *visit* function for each object directly contained by *self*, with the parameters to *visit* being the contained object and the *arg* value passed to the handler. The *visit* function must not be called with a `NULL` object argument. If *visit* returns a non-zero value that value should be returned immediately.

To simplify writing `tp_traverse` handlers, a `Py_VISIT()` macro is provided. In order to use this macro, the `tp_traverse` implementation must name its arguments exactly *visit* and *arg* :

Py_VISIT(o)

If the `PyObject*` *o* is not `NULL`, call the *visit* callback, with arguments *o* and *arg*. If *visit* returns a non-zero value, then return it. Using this macro, `tp_traverse` handlers look like :

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    Py_VISIT(self->bar);
    return 0;
}
```

The `tp_clear` handler must be of the `inquiry` type, or `NULL` if the object is immutable.

typedef int (***inquiry**)(*PyObject* *self)

Part of the Stable ABI. Drop references that may have created reference cycles. Immutable objects do not have to define this method since they can never directly create reference cycles. Note that the object must still be valid after calling this method (don't just call `Py_DECREF()` on a reference). The collector will call this method if it detects that this object is involved in a reference cycle.

13.5.1 Controlling the Garbage Collector State

The C-API provides the following functions for controlling garbage collection runs.

Py_ssize_t **PyGC_Collect**(void)

Part of the Stable ABI. Perform a full garbage collection, if the garbage collector is enabled. (Note that `gc.collect()` runs it unconditionally.)

Returns the number of collected + unreachable objects which cannot be collected. If the garbage collector is disabled or already collecting, returns `0` immediately. Errors during garbage collection are passed to `sys.unraisablehook`. This function does not raise exceptions.

int **PyGC_Enable**(void)

Part of the Stable ABI *Depuis la version 3.10.* Enable the garbage collector : similar to `gc.enable()`. Returns the previous state, 0 for disabled and 1 for enabled.

Ajouté dans la version 3.10.

int **PyGC_Disable**(void)

Part of the Stable ABI *Depuis la version 3.10.* Disable the garbage collector : similar to `gc.disable()`. Returns the previous state, 0 for disabled and 1 for enabled.

Ajouté dans la version 3.10.

int **PyGC_IsEnabled**(void)

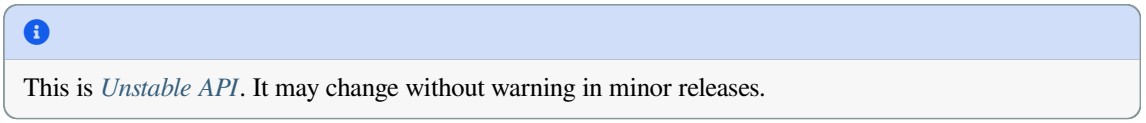
Part of the Stable ABI *Depuis la version 3.10.* Query the state of the garbage collector : similar to `gc.isenabled()`. Returns the current state, 0 for disabled and 1 for enabled.

Ajouté dans la version 3.10.

13.5.2 Querying Garbage Collector State

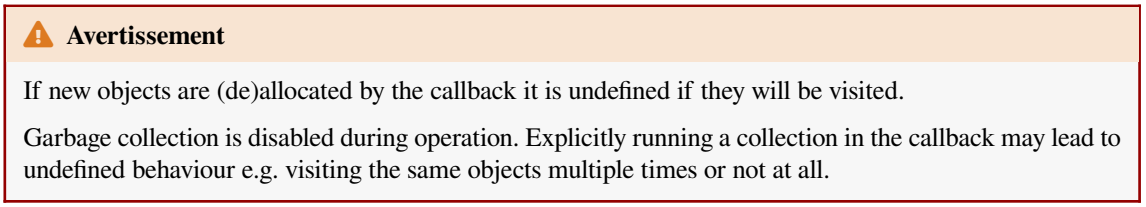
The C-API provides the following interface for querying information about the garbage collector.

void **PyUnstable_GC_VisitObjects**(*gcvisitobjects_t* callback, void *arg)



This is *Unstable API*. It may change without warning in minor releases.

Run supplied *callback* on all live GC-capable objects. *arg* is passed through to all invocations of *callback*.



Avertissement

If new objects are (de)allocated by the callback it is undefined if they will be visited.

Garbage collection is disabled during operation. Explicitly running a collection in the callback may lead to undefined behaviour e.g. visiting the same objects multiple times or not at all.

Ajouté dans la version 3.12.

typedef int (***gcvisitobjects_t**)(*PyObject* *object, void *arg)

Type of the visitor function to be passed to *`PyUnstable_GC_VisitObjects()`*. *arg* is the same as the *arg* passed to `PyUnstable_GC_VisitObjects`. Return `1` to continue iteration, return `0` to stop iteration. Other return values are reserved for now so behavior on returning anything else is undefined.

Ajouté dans la version 3.12.

CHAPITRE 14

Version des API et ABI

14.1 Build-time version constants

CPython exposes its version number in the following macros. Note that these correspond to the version code is **built** with. See *Py_Version* for the version used at **run time**.

Voir *Stabilité de l'API C* pour une discussion sur la stabilité des API et ABI entre (ou en fonction) de la version.

PY_MAJOR_VERSION

Le `3` dans `3.4.1a2`.)

PY_MINOR_VERSION

Le `4` dans `3.4.1a2`.

PY_MICRO_VERSION

Le `1` dans `3.4.1a2`.)

PY_RELEASE_LEVEL

The `a` in `3.4.1a2`. This can be `0xA` for alpha, `0xB` for beta, `0xC` for release candidate or `0xF` for final.

PY_RELEASE_SERIAL

Le `2` dans `3.4.1a2`. Nul pour des versions finales.

PY_VERSION_HEX

The Python version number encoded in a single integer. See *Py_PACK_FULL_VERSION()* for the encoding details.

Use this for numeric comparisons, for example, `#if PY_VERSION_HEX >= ...`.

14.2 Run-time version

const unsigned long **Py_Version**

Part of the Stable ABI *Depuis la version 3.11.* The Python runtime version number encoded in a single constant integer. See *Py_PACK_FULL_VERSION()* for the encoding details. This contains the Python version used at run time.

Use this for numeric comparisons, for example, `if (Py_Version >= ...)`.

Ajouté dans la version 3.11.

14.3 Bit-packing macros

uint32_t **Py_PACK_FULL_VERSION**(int major, int minor, int micro, int release_level, int release_serial)

Part of the Stable ABI *Depuis la version 3.14.* Return the given version, encoded as a single 32-bit integer with the following structure :

Argument	No. of bits	Bit mask	Bit shift	Example values	
				`3.4.1a2`	`3.10.0`
major	8	`0xFF000000`	24	`0x03`	`0x03`
minor	8	`0x00FF0000`	16	`0x04`	`0x0A`
micro	8	`0x0000FF00`	8	`0x01`	`0x00`
release_level	4	`0x000000F0`	4	`0xA`	`0xF`
release_serial	4	`0x0000000F`	0	`0x2`	`0x0`

For example :

Version	`Py_PACK_FULL_VERSION` arguments	Encoded version
`3.4.1a2`	`(3, 4, 1, 0xA, 2)`	`0x030401a2`
`3.10.0`	`(3, 10, 0, 0xF, 0)`	`0x030a00f0`

Out-of range bits in the arguments are ignored. That is, the macro can be defined as :

```
#ifndef Py_PACK_FULL_VERSION
#define Py_PACK_FULL_VERSION(X, Y, Z, LEVEL, SERIAL) ( \
   (((X) & 0xff) << 24) |                              \
   (((Y) & 0xff) << 16) |                              \
   (((Z) & 0xff) << 8) |                               \
   (((LEVEL) & 0xf) << 4) |                            \
   (((SERIAL) & 0xf) << 0))
#endif
```

`Py_PACK_FULL_VERSION` is primarily a macro, intended for use in `#if` directives, but it is also available as an exported function.

Ajouté dans la version 3.14.

uint32_t **Py_PACK_VERSION**(int major, int minor)

Part of the Stable ABI *Depuis la version 3.14.* Equivalent to `Py_PACK_FULL_VERSION(major, minor, 0, 0, 0)`. The result does not correspond to any Python release, but is useful in numeric comparisons.

Ajouté dans la version 3.14.

CHAPITRE 15

Monitoring C API

Added in version 3.13.

An extension may need to interact with the event monitoring system. Subscribing to events and registering callbacks can be done via the Python API exposed in `sys.monitoring`.

CHAPITRE 16

Generating Execution Events

The functions below make it possible for an extension to fire monitoring events as it emulates the execution of Python code. Each of these functions accepts a `PyMonitoringState` struct which contains concise information about the activation state of events, as well as the event arguments, which include a `PyObject*` representing the code object, the instruction offset and sometimes additional, event-specific arguments (see `sys.monitoring` for details about the signatures of the different event callbacks). The `codelike` argument should be an instance of `types.CodeType` or of a type that emulates it.

The VM disables tracing when firing an event, so there is no need for user code to do that.

Monitoring functions should not be called with an exception set, except those listed below as working with the current exception.

type **PyMonitoringState**

Representation of the state of an event type. It is allocated by the user while its contents are maintained by the monitoring API functions described below.

All of the functions below return 0 on success and -1 (with an exception set) on error.

See `sys.monitoring` for descriptions of the events.

int **PyMonitoring_FirePyStartEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_START` event.

int **PyMonitoring_FirePyResumeEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_RESUME` event.

int **PyMonitoring_FirePyReturnEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Fire a `PY_RETURN` event.

int **PyMonitoring_FirePyYieldEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Fire a `PY_YIELD` event.

int **PyMonitoring_FireCallEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *callable, *PyObject* *arg0)

Fire a `CALL` event.

int **PyMonitoring_FireLineEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, int lineno)

Fire a `LINE` event.

int **PyMonitoring_FireJumpEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

Fire a `JUMP` event.

int **PyMonitoring_FireBranchLeftEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

Fire a `BRANCH_LEFT` event.

int **PyMonitoring_FireBranchRightEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *target_offset)

Fire a `BRANCH_RIGHT` event.

int **PyMonitoring_FireCReturnEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *retval)

Fire a `C_RETURN` event.

int **PyMonitoring_FirePyThrowEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_THROW` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireRaiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `RAISE` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireCRaiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `C_RAISE` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireReraiseEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `RERAISE` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireExceptionHandledEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire an `EXCEPTION_HANDLED` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FirePyUnwindEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset)

Fire a `PY_UNWIND` event with the current exception (as returned by *`PyErr_GetRaisedException()`*).

int **PyMonitoring_FireStopIterationEvent**(*PyMonitoringState* *state, *PyObject* *codelike, int32_t offset, *PyObject* *value)

Fire a `STOP_ITERATION` event. If `value` is an instance of `StopIteration`, it is used. Otherwise, a new `StopIteration` instance is created with `value` as its argument.

16.1 Managing the Monitoring State

Monitoring states can be managed with the help of monitoring scopes. A scope would typically correspond to a python function.

int **PyMonitoring_EnterScope**(*PyMonitoringState* *state_array, uint64_t *version, const uint8_t *event_types, *Py_ssize_t* length)

Enter a monitored scope. `event_types` is an array of the event IDs for events that may be fired from the scope. For example, the ID of a `PY_START` event is the value `PY_MONITORING_EVENT_PY_START`, which is numerically equal to the base-2 logarithm of `sys.monitoring.events.PY_START`. `state_array` is an array with a monitoring state entry for each event in `event_types`, it is allocated by the user but populated by `PyMonitoring_EnterScope()` with information about the activation state of the event. The size of `event_types` (and hence also of `state_array`) is given in `length`.

The `version` argument is a pointer to a value which should be allocated by the user together with `state_array` and initialized to 0, and then set only by `PyMonitoring_EnterScope()` itself. It allows this function to determine whether event states have changed since the previous call, and to return quickly if they have not.

The scopes referred to here are lexical scopes : a function, class or method. `PyMonitoring_EnterScope()` should be called whenever the lexical scope is entered. Scopes can be reentered, reusing the same *state_array* and *version*, in situations like when emulating a recursive Python function. When a code-like's execution is paused, such as when emulating a generator, the scope needs to be exited and re-entered.

The macros for *event_types* are :

Macro	Événement
PY_MONITORING_EVENT_BRANCH_LEFT	BRANCH_LEFT
PY_MONITORING_EVENT_BRANCH_RIGHT	BRANCH_RIGHT
PY_MONITORING_EVENT_CALL	CALL
PY_MONITORING_EVENT_C_RAISE	C_RAISE
PY_MONITORING_EVENT_C_RETURN	C_RETURN
PY_MONITORING_EVENT_EXCEPTION_HANDLED	EXCEPTION_HANDLED
PY_MONITORING_EVENT_INSTRUCTION	INSTRUCTION
PY_MONITORING_EVENT_JUMP	JUMP
PY_MONITORING_EVENT_LINE	LINE
PY_MONITORING_EVENT_PY_RESUME	PY_RESUME
PY_MONITORING_EVENT_PY_RETURN	PY_RETURN
PY_MONITORING_EVENT_PY_START	PY_START
PY_MONITORING_EVENT_PY_THROW	PY_THROW
PY_MONITORING_EVENT_PY_UNWIND	PY_UNWIND
PY_MONITORING_EVENT_PY_YIELD	PY_YIELD
PY_MONITORING_EVENT_RAISE	RAISE
PY_MONITORING_EVENT_RERAISE	RERAISE
PY_MONITORING_EVENT_STOP_ITERATION	STOP_ITERATION

int **PyMonitoring_ExitScope**(void)

Exit the last scope that was entered with `PyMonitoring_EnterScope()`.

int **PY_MONITORING_IS_INSTRUMENTED_EVENT**(uint8_t ev)

Return true if the event corresponding to the event ID *ev* is a local event.

Ajouté dans la version 3.13.

Obsolète depuis la version 3.14 : This function is *soft deprecated*.

ANNEXE A

Glossaire

>>>
: The default Python prompt of the *interactive* shell. Often seen for code examples which can be executed interactively in the interpreter.

...
: Peut faire référence à :
— The default Python prompt of the *interactive* shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.
— The three dots form of the Ellipsis object.

classe mère abstraite
: Les classes mères abstraites (ABC, suivant l'abréviation anglaise *Abstract Base Class*) complètent le *duck-typing* en fournissant un moyen de définir des interfaces pour les cas où d'autres techniques comme `hasattr()` seraient inélégantes ou subtilement fausses (par exemple avec les méthodes magiques). Les ABC introduisent des sous-classes virtuelles qui n'héritent pas d'une classe mais qui sont quand même reconnues par `isinstance()` ou `issubclass()` (voir la documentation du module `abc`). Python contient de nombreuses ABC pour les structures de données (dans le module `collections.abc`), les nombres (dans le module `numbers`), les flux (dans le module `io`) et les chercheurs-chargeurs du système d'importation (dans le module `importlib.abc`). Vous pouvez créer vos propres ABC avec le module `abc`.

annotate function
: A function that can be called to retrieve the *annotations* of an object. This function is accessible as the `__annotate__` attribute of functions, classes, and modules. Annotate functions are a subset of *evaluate functions*.

annotation
: Étiquette associée à une variable, un attribut de classe, un paramètre de fonction ou une valeur de retour. Elle est utilisée par convention comme *type hint*.

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions can be retrieved by calling `annotationlib.get_annotations()` on modules, classes, and functions, respectively.

See *variable annotation*, *function annotation*, **PEP 484**, **PEP 526**, and **PEP 649**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

argument
: Valeur, donnée à une *fonction* ou à une *méthode* lors de son appel. Il existe deux types d'arguments :
— *argument nommé* : un argument précédé d'un identifiant (comme `name=`) ou un dictionnaire précédé de `**`, lors d'un appel de fonction. Par exemple, `3` et `5` sont tous les deux des arguments nommés dans l'appel à `complex()` ici :

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

— *argument positionnel* : un argument qui n'est pas nommé. Les arguments positionnels apparaissent au début de la liste des arguments, ou donnés sous forme d'un *itérable* précédé par `*`. Par exemple, `3` et `5` sont tous les deux des arguments positionnels dans les appels suivants :

```
complex(3, 5)
complex(*(3, 5))
```

Les arguments se retrouvent dans le corps de la fonction appelée parmi les variables locales. Voir la section calls à propos des règles dictant cette affectation. Syntaxiquement, toute expression est acceptée comme argument, et c'est la valeur résultante de l'expression qui sera affectée à la variable locale.

Voir aussi *paramètre* dans le glossaire, la question Différence entre argument et paramètre de la FAQ et la **PEP 362**.

gestionnaire de contexte asynchrone
(*asynchronous context manager* en anglais) Objet contrôlant l'environnement à l'intérieur d'une instruction `async with` en définissant les méthodes `__aenter__()` et `__aexit__()`. A été Introduit par la **PEP 492**.

générateur asynchrone
Fonction qui renvoie un *itérateur de générateur asynchrone*. Cela ressemble à une coroutine définie par `async def`, sauf qu'elle contient une ou des expressions `yield` produisant ainsi uns série de valeurs utilisables dans une boucle `async for`.

Générateur asynchrone fait généralement référence à une fonction, mais peut faire référence à un *itérateur de générateur asynchrone* dans certains contextes. Dans les cas où le sens voulu n'est pas clair, utiliser l'ensemble des termes lève l'ambiguïté.

Un générateur asynchrone peut contenir des expressions `await` ainsi que des instructions `async for`, et `async with`.

itérateur de générateur asynchrone
An object created by an *asynchronous generator* function.

C'est un *asynchronous iterator* qui, lorsqu'il est appelé via la méthode `__anext__()` renvoie un objet *awaitable* qui exécute le corps de la fonction du générateur asynchrone jusqu'au prochain `yield`.

Each `yield` temporarily suspends processing, remembering the execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See **PEP 492** and **PEP 525**.

itérable asynchrone
Objet qui peut être utilisé dans une instruction `async for`. Sa méthode `__aiter__()` doit renvoyer un *asynchronous iterator*. A été introduit par la **PEP 492**.

itérateur asynchrone
Objet qui implémente les méthodes `__aiter__()` et `__anext__()`. `__anext__()` doit renvoyer un objet *awaitable*. Tant que la méthode `__anext__()` produit des objets *awaitable*, le `async for` appelant les consomme. L'itérateur asynchrone lève une exception `StopAsyncIteration` pour signifier la fin de l'itération. A été introduit par la **PEP 492**.

attached thread state
A *thread state* that is active for the current OS thread.

When a *thread state* is attached, the OS thread has access to the full Python C API and can safely invoke the bytecode interpreter.

Unless a function explicitly notes otherwise, attempting to call the C API without an attached thread state will result in a fatal error or undefined behavior. A thread state can be attached and detached explicitly by the user through the C API, or implicitly by the runtime, including during blocking C calls and by the bytecode interpreter in between calls.

On most builds of Python, having an attached thread state implies that the caller holds the *GIL* for the current interpreter, so only one OS thread can have an attached thread state at a given moment. In *free-threaded* builds of Python, threads can concurrently hold an attached thread state, allowing for true parallelism of the

bytecode interpreter.

attribut
: Valeur associée à un objet et habituellement désignée par son nom *via* une notation utilisant des points. Par exemple, si un objet *o* possède un attribut *a*, cet attribut est référencé par *o.a*.

  Il est possible de donner à un objet un attribut dont le nom n'est pas un identifiant tel que défini pour les identifiers, par exemple en utilisant `setattr()`, si l'objet le permet. Un tel attribut ne sera pas accessible à l'aide d'une expression pointée et on devra y accéder avec `getattr()`.

attendable (*awaitable*)
: Objet pouvant être utilisé dans une expression `await`. Ce peut être une *coroutine* ou un objet avec une méthode `__await__()`. Voir aussi la **PEP 492**.

BDFL
: Dictateur bienveillant à vie (*Benevolent Dictator For Life* en anglais). Pseudonyme de Guido van Rossum, le créateur de Python.

fichier binaire
: A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

  Consultez *fichier texte*, un objet fichier capable de lire et d'écrire des objets `str`.

référence empruntée
: In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

  Il est recommandé d'appeler `Py_INCREF()` sur la *référence empruntée*, ce qui la transforme *in situ* en une *référence forte*. Vous pouvez faire une exception si vous êtes certain que l'objet ne peut pas être supprimé avant la dernière utilisation de la référence empruntée. Voir aussi la fonction `Py_NewRef()`, qui crée une nouvelle *référence forte*.

objet octet-compatible
: Un objet gérant le *protocole tampon* et pouvant exporter un tampon (*buffer* en anglais) C-*contigu*. Cela inclut les objets `bytes`, `bytearray` et `array.array`, ainsi que beaucoup d'objets `memoryview`. Les objets octets-compatibles peuvent être utilisés pour diverses opérations sur des données binaires, comme la compression, la sauvegarde dans un fichier binaire ou l'envoi sur le réseau.

  Certaines opérations nécessitent de travailler sur des données binaires variables. La documentation parle de ceux-ci comme des *read-write bytes-like objects*. Par exemple, `bytearray` ou une `memoryview` d'un `bytearray` en font partie. D'autres opérations nécessitent de travailler sur des données binaires stockées dans des objets immuables (« *objets octets-compatibles en lecture seule* »), par exemple des `bytes` ou des `memoryview` d'un objet `bytes`.

code intermédiaire (*bytecode*)
: Le code source, en Python, est compilé en un code intermédiaire (*bytecode* en anglais), la représentation interne à CPython d'un programme Python. Le code intermédiaire est mis en cache dans un fichier `.pyc` de manière à ce qu'une seconde exécution soit plus rapide (la compilation en code intermédiaire a déjà été faite). On dit que ce *langage intermédiaire* est exécuté sur une *virtual machine* qui exécute des instructions machine pour chaque instruction du code intermédiaire. Notez que le code intermédiaire n'a pas vocation à fonctionner sur différentes machines virtuelles Python ou à être stable entre différentes versions de Python.

  La documentation du module dis fournit une liste des instructions du code intermédiaire.

appelable (*callable*)
: Un appelable est un objet qui peut être appelé, éventuellement avec un ensemble d'arguments (voir *argument*), avec la syntaxe suivante :

```
callable(argument1, argument2, argumentN)
```

  Une *fonction*, et par extension une *méthode*, est un appelable. Une instance d'une classe qui implémente la méthode `__call__()` est également un appelable.

fonction de rappel (*callback*)
: Une fonction (classique, par opposition à une coroutine) passée en argument pour être exécutée plus tard.

classe
: Modèle pour créer des objets définis par l'utilisateur. Une définition de classe (*class*) contient normalement des définitions de méthodes qui agissent sur les instances de la classe.

variable de classe
: Une variable définie dans une classe et destinée à être modifiée uniquement au niveau de la classe (c'est-à-dire, pas dans une instance de la classe).

closure variable
: A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

  For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable* :

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

  Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

nombre complexe
: Extension des nombres réels familiers, dans laquelle tous les nombres sont exprimés sous la forme d'une somme d'une partie réelle et d'une partie imaginaire. Les nombres imaginaires sont les nombres réels multipliés par l'unité imaginaire (la racine carrée de `-1`, souvent écrite `i` en mathématiques ou `j` par les ingénieurs). Python comprend nativement les nombres complexes, écrits avec cette dernière notation : la partie imaginaire est écrite avec un suffixe `j`, exemple, `3+1j`. Pour utiliser les équivalents complexes de `math`, utilisez `cmath`. Les nombres complexes sont un concept assez avancé en mathématiques. Si vous ne connaissez pas ce concept, vous pouvez tranquillement les ignorer.

context
: This term has different meanings depending on where and how it is used. Some common meanings :
  — The temporary state or environment established by a *context manager* via a `with` statement.
  — The collection of keyvalue bindings associated with a particular `contextvars.Context` object and accessed via `ContextVar` objects. Also see *context variable*.
  — A `contextvars.Context` object. Also see *current context*.

context management protocol
: The `__enter__()` and `__exit__()` methods called by the `with` statement. See **PEP 343**.

gestionnaire de contexte
: An object which implements the *context management protocol* and controls the environment seen in a `with` statement. See **PEP 343**.

variable de contexte
: A variable whose value depends on which context is the *current context*. Values are accessed via `contextvars.ContextVar` objects. Context variables are primarily used to isolate state between concurrent asynchronous tasks.

contigu
: Un tampon (*buffer* en anglais) est considéré comme contigu s'il est soit *C-contigu* soit *Fortran-contigu*. Les tampons de dimension zéro sont C-contigus et Fortran-contigus. Pour un tableau à une dimension, ses éléments doivent être placés en mémoire l'un à côté de l'autre, dans l'ordre croissant de leur indice, en commençant à zéro. Pour qu'un tableau multidimensionnel soit C-contigu, le dernier indice doit être celui qui varie le plus rapidement lors du parcours de ses éléments dans l'ordre de leur adresse mémoire. À l'inverse, dans les tableaux Fortran-contigu, c'est le premier indice qui doit varier le plus rapidement.

coroutine
: Les coroutines sont une forme généralisée des fonctions. On entre dans une fonction en un point et on en sort en un autre point. On peut entrer, sortir et reprendre l'exécution d'une coroutine en plusieurs points. Elles

peuvent être implémentées en utilisant l'instruction `async def`. Voir aussi la **PEP 492**.

fonction coroutine
Fonction qui renvoie un objet *coroutine*. Une fonction coroutine peut être définie par l'instruction `async def` et peut contenir les mots clés `await`, `async for` ainsi que `async with`. A été introduit par la **PEP 492**.

CPython
L'implémentation canonique du langage de programmation Python, tel que distribué sur python.org. Le terme "CPython" est utilisé dans certains contextes lorsqu'il est nécessaire de distinguer cette implémentation des autres comme *Jython* ou *IronPython*.

current context
The *context* (`contextvars.Context` object) that is currently used by `ContextVar` objects to access (get or set) the values of *context variables*. Each thread has its own current context. Frameworks for executing asynchronous tasks (see `asyncio`) associate each task with a context which becomes the current context whenever the task starts or resumes execution.

cyclic isolate
A subgroup of one or more objects that reference each other in a reference cycle, but are not referenced by objects outside the group. The goal of the *cyclic garbage collector* is to identify these groups and break the reference cycles so that the memory can be reclaimed.

décorateur
Fonction dont la valeur de retour est une autre fonction. Un décorateur est habituellement utilisé pour transformer une fonction via la syntaxe `@wrapper`, dont les exemples typiques sont : `classmethod()` et `staticmethod()`.

La syntaxe des décorateurs est simplement du sucre syntaxique, les définitions des deux fonctions suivantes sont sémantiquement équivalentes :

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

Quoique moins fréquemment utilisé, le même concept existe pour les classes. Consultez la documentation définitions de fonctions et définitions de classes pour en savoir plus sur les décorateurs.

descripteur
Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

Pour plus d'informations sur les méthodes des descripteurs, consultez descriptors ou le guide pour l'utilisation des descripteurs.

dictionnaire
An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

dictionnaire en compréhension (ou dictionnaire en intension)
Écriture concise pour traiter tout ou partie des éléments d'un itérable et renvoyer un dictionnaire contenant les résultats. `results = {n: n ** 2 for n in range(10)}` génère un dictionnaire contenant des clés `n` liées à leurs valeurs `n ** 2`. Voir compréhensions.

vue de dictionnaire
Objets retournés par les méthodes `dict.keys()`, `dict.values()` et `dict.items()`. Ils fournissent des vues dynamiques des entrées du dictionnaire, ce qui signifie que lorsque le dictionnaire change, la vue change. Pour transformer une vue en vraie liste, utilisez `list(dictview)`. Voir dict-views.

chaîne de documentation (*docstring*)
A string literal which appears as the first expression in a class, function or module. While ignored when the

suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

typage canard (*duck-typing*)
Style de programmation qui ne prend pas en compte le type d'un objet pour déterminer s'il respecte une interface, mais qui appelle simplement la méthode ou l'attribut (*Si ça a un bec et que ça cancane, ça doit être un canard*, *duck* signifie canard en anglais). En se concentrant sur les interfaces plutôt que les types, du code bien construit améliore sa flexibilité en autorisant des substitutions polymorphiques. Le *duck-typing* évite de vérifier les types via `type()` ou `isinstance()`, Notez cependant que le *duck-typing* peut travailler de pair avec les *classes mère abstraites*. À la place, le *duck-typing* utilise plutôt `hasattr()` ou la programmation *EAFP*.

dunder
An informal short-hand for "double underscore", used when talking about a *special method*. For example, `__init__` is often pronounced "dunder init".

EAFP
Il est plus simple de demander pardon que demander la permission (*Easier to Ask for Forgiveness than Permission* en anglais). Ce style de développement Python fait l'hypothèse que le code est valide et traite les exceptions si cette hypothèse s'avère fausse. Ce style, propre et efficace, est caractérisé par la présence de beaucoup de mots clés `try` et `except`. Cette technique de programmation contraste avec le style *LBYL* utilisé couramment dans les langages tels que C.

evaluate function
A function that can be called to evaluate a lazily evaluated attribute of an object, such as the value of type aliases created with the `type` statement.

expression
Suite logique de termes et chiffres conformes à la syntaxe Python dont l'évaluation fournit une valeur. En d'autres termes, une expression est une suite d'éléments tels que des noms, opérateurs, littéraux, accès d'attributs, méthodes ou fonctions qui aboutissent à une valeur. Contrairement à beaucoup d'autres langages, les différentes constructions du langage ne sont pas toutes des expressions. On trouve également des *instructions* qui ne peuvent pas être utilisées comme expressions, tel que `while`. Les affectations sont également des instructions et non des expressions.

module d'extension
Module écrit en C ou C++, utilisant l'API C de Python pour interagir avec Python et le code de l'utilisateur.

f-string
f-strings
String literals prefixed with `f` or `F` are commonly called "f-strings" which is short for formatted string literals. See also **PEP 498**.

objet fichier
An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

Il existe en réalité trois catégories de fichiers objets : les *fichiers binaires* bruts, les *fichiers binaires* avec tampon (*buffer*) et les *fichiers textes*. Leurs interfaces sont définies dans le module `io`. Le moyen le plus simple et direct de créer un objet fichier est d'utiliser la fonction `open()`.

objet fichier-compatible
Synonyme de *objet fichier*.

encodage du système de fichiers et gestionnaire d'erreurs associé
Encodage et gestionnaire d'erreurs utilisés par Python pour décoder les octets fournis par le système d'exploitation et encoder les chaînes de caractères Unicode afin de les passer au système.

L'encodage du système de fichiers doit impérativement pouvoir décoder tous les octets jusqu'à 128. Si ce n'est pas le cas, certaines fonctions de l'API lèvent `UnicodeError`.

Cet encodage et son gestionnaire d'erreur peuvent être obtenus à l'aide des fonctions `sys.getfilesystemencoding()` et `sys.getfilesystemencodeerrors()`.

L'*encodage du système de fichiers et gestionnaire d'erreurs associé* sont configurés au démarrage de Python par la fonction *`PyConfig_Read()`* : regardez *`filesystem_encoding`* et *`filesystem_errors`* dans les

membres de *PyConfig*.

Voir aussi *encodage régional*.

chercheur
Objet qui essaie de trouver un *chargeur* pour le module en cours d'importation.

There are two types of finder : *meta path finders* for use with `sys.meta_path`, and *path entry finders* for use with `sys.path_hooks`.

See finders-and-loaders and `importlib` for much more detail.

division entière
Division mathématique arrondissant à l'entier inférieur. L'opérateur de la division entière est `//`. Par exemple l'expression `11 // 4` vaut `2`, contrairement à `11 / 4` qui vaut `2.75`. Notez que `(-11) // 4` vaut `-3` car l'arrondi se fait à l'entier inférieur. Voir la **PEP 328**.

free threading
A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

free variable
Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

fonction
Suite d'instructions qui renvoie une valeur à son appelant. On peut lui passer des *arguments* qui pourront être utilisés dans le corps de la fonction. Voir aussi *paramètre*, *méthode* et function.

annotation de fonction
annotation d'un paramètre de fonction ou valeur de retour.

Les annotations de fonctions sont généralement utilisées pour des *indications de types* : par exemple, cette fonction devrait prendre deux arguments `int` et devrait également avoir une valeur de retour de type `int` :

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

L'annotation syntaxique de la fonction est expliquée dans la section function.

Voir *annotation de variable* et la **PEP 484**, qui décrivent cette fonctionnalité. Voir aussi annotations-howto sur les bonnes pratiques concernant les annotations.

__future__
Une importation depuis le futur s'écrit `from __future__ import <fonctionnalité>`. Lorsqu'une importation du futur est active dans un module, Python compile ce module avec une certaine modification de la syntaxe ou du comportement qui est vouée à devenir standard dans une version ultérieure. Le module `__future__` documente les possibilités pour *fonctionnalité*. L'importation a aussi l'effet normal d'importer une variable du module. Cette variable contient des informations utiles sur la fonctionnalité en question, notamment la version de Python dans laquelle elle a été ajoutée, et celle dans laquelle elle deviendra standard :

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

ramasse-miettes
(*garbage collection* en anglais) Mécanisme permettant de libérer de la mémoire lorsqu'elle n'est plus utilisée. Python utilise un ramasse-miettes par comptage de référence et un ramasse-miettes cyclique capable de détecter et casser les références circulaires. Le ramasse-miettes peut être contrôlé en utilisant le module `gc`.

générateur
Fonction qui renvoie un *itérateur de générateur*. Cela ressemble à une fonction normale, en dehors du fait qu'elle contient une ou des expressions `yield` produisant une série de valeurs utilisable dans une boucle *for* ou récupérées une à une via la fonction `next()`.

Fait généralement référence à une fonction génératrice mais peut faire référence à un *itérateur de généra-*

teur dans certains contextes. Dans les cas où le sens voulu n'est pas clair, utiliser les termes complets lève l'ambiguïté.

itérateur de générateur
Objet créé par une fonction *générateur*.

Each `yield` temporarily suspends processing, remembering the execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

expression génératrice
An *expression* that returns an *iterator*. It looks like a normal expression followed by a `for` clause defining a loop variable, range, and an optional `if` clause. The combined expression generates values for an enclosing function :

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

fonction générique
Fonction composée de plusieurs fonctions implémentant les mêmes opérations pour différents types. L'implémentation à utiliser est déterminée lors de l'appel par l'algorithme de répartition.

Voir aussi *single dispatch*, le décorateur `functools.singledispatch()` et la **PEP 443**.

type générique
Un *type* qui peut être paramétré ; généralement un conteneur comme `list` ou `dict`. Utilisé pour les *indications de type* et les *annotations*.

Pour plus de détails, voir types alias génériques et le module `typing`. On trouvera l'historique de cette fonctionnalité dans les **PEP 483**, **PEP 484** et **PEP 585**.

GIL
Voir *global interpreter lock*.

verrou global de l'interpréteur
(*global interpreter lock* en anglais) Mécanisme utilisé par l'interpréteur *CPython* pour s'assurer qu'un seul fil d'exécution (*thread* en anglais) n'exécute le *bytecode* à la fois. Cela simplifie l'implémentation de CPython en rendant le modèle objet (incluant des parties critiques comme la classe native `dict`) implicitement protégé contre les accès concourants. Verrouiller l'interpréteur entier rend plus facile l'implémentation de multiples fils d'exécution (*multi-thread* en anglais), au détriment malheureusement de beaucoup du parallélisme possible sur les machines ayant plusieurs processeurs.

Cependant, certains modules d'extension, standards ou non, sont conçus de manière à libérer le GIL lorsqu'ils effectuent des tâches lourdes tel que la compression ou le hachage. De la même manière, le GIL est toujours libéré lors des entrées-sorties.

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

In prior versions of Python's C API, a function might declare that it requires the GIL to be held in order to use it. This refers to having an *attached thread state*.

***pyc* utilisant le hachage**
Un fichier de cache de code intermédiaire (*bytecode* en anglais) qui utilise le hachage plutôt que l'heure de dernière modification du fichier source correspondant pour déterminer sa validité. Voir pyc-invalidation.

hachable
An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

La hachabilité permet à un objet d'être utilisé comme clé de dictionnaire ou en tant que membre d'un ensemble (type *set*), car ces structures de données utilisent ce *hash*.

La plupart des types immuables natifs de Python sont hachables, mais les conteneurs mutables (comme les listes ou les dictionnaires) ne le sont pas ; les conteneurs immuables (comme les n-uplets ou les ensembles figés) ne sont hachables que si leurs éléments sont hachables. Les instances de classes définies par les utilisateurs

sont hachables par défaut. Elles sont toutes considérées différentes (sauf avec elles-mêmes) et leur valeur de hachage est calculée à partir de leur `id()`.

IDLE
Environnement d'apprentissage et de développement intégré pour Python. IDLE est un éditeur basique et un interpréteur livré avec la distribution standard de Python.

immortal
Immortal objects are a CPython implementation detail introduced in **PEP 683**.

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

Immortal objects can be identified via `sys._is_immortal()`, or via *`PyUnstable_IsImmortal()`* in the C API.

immuable
Objet dont la valeur ne change pas. Les nombres, les chaînes et les *n*-uplets sont immuables. Ils ne peuvent être modifiés. Un nouvel objet doit être créé si une valeur différente doit être stockée. Ils jouent un rôle important quand une valeur de *hash* constante est requise, typiquement en clé de dictionnaire.

chemin des importations
Liste de *entrées* dans lesquelles le *chercheur basé sur les chemins* cherche les modules à importer. Typiquement, lors d'une importation, cette liste vient de `sys.path` ; pour les sous-paquets, elle peut aussi venir de l'attribut `__path__` du paquet parent.

importation
Processus rendant le code Python d'un module disponible dans un autre.

importateur
Objet qui trouve et charge un module, en même temps un *chercheur* et un *chargeur*.

interactif
Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`). For more on interactive mode, see tut-interac.

interprété
Python est un langage interprété, en opposition aux langages compilés, bien que la frontière soit floue en raison de la présence d'un compilateur en code intermédiaire. Cela signifie que les fichiers sources peuvent être exécutés directement, sans avoir à compiler un fichier exécutable intermédiaire. Les langages interprétés ont généralement un cycle de développement / débogage plus court que les langages compilés. Cependant, ils s'exécutent généralement plus lentement. Voir aussi *interactif*.

arrêt de l'interpréteur
Lorsqu'on lui demande de s'arrêter, l'interpréteur Python entre dans une phase spéciale où il libère graduellement les ressources allouées, comme les modules ou quelques structures de données internes. Il fait aussi quelques appels au *ramasse-miettes*. Cela peut déclencher l'exécution de code dans des destructeurs ou des fonctions de rappels de *weakrefs*. Le code exécuté lors de l'arrêt peut rencontrer des exceptions puisque les ressources auxquelles il fait appel sont susceptibles de ne plus fonctionner, (typiquement les modules des bibliothèques ou le mécanisme de *warning*).

La principale raison d'arrêt de l'interpréteur est que le module `__main__` ou le script en cours d'exécution a terminé de s'exécuter.

itérable
An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *sequence* semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

itérateur

An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

Vous trouverez davantage d'informations dans typeiter.

Particularité de l'implémentation CPython : CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

fonction clé

Une fonction clé est un objet appelable qui renvoie une valeur à fins de tri ou de classement. Par exemple, la fonction `locale.strxfrm()` est utilisée pour générer une clé de classement prenant en compte les conventions de classement spécifiques aux paramètres régionaux courants.

Plusieurs outils dans Python acceptent des fonctions clés pour déterminer comment les éléments sont classés ou groupés. On peut citer les fonctions `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()` et `itertools.groupby()`.

Il existe plusieurs moyens de créer une fonction clé. Par exemple, la méthode `str.lower()` peut servir de fonction clé pour effectuer des recherches insensibles à la casse. Aussi, il est possible de créer des fonctions clés avec des expressions `lambda`, comme `lambda r: (r[0], r[2])`. Par ailleurs `attrgetter()`, `itemgetter()` et `methodcaller()` permettent de créer des fonctions clés. Voir le guide pour le tri pour des exemples de création et d'utilisation de fonctions clefs.

argument nommé

Voir *argument*.

lambda

Fonction anonyme sous la forme d'une *expression* et ne contenant qu'une seule expression, exécutée lorsque la fonction est appelée. La syntaxe pour créer des fonctions lambda est : `lambda [parameters]: expression`

LBYL

Regarde avant de sauter, (*Look before you leap* en anglais). Ce style de programmation consiste à vérifier des conditions avant d'effectuer des appels ou des accès. Ce style contraste avec le style *EAFP* et se caractérise par la présence de beaucoup d'instructions `if`.

Dans un environnement avec plusieurs fils d'exécution (*multi-threaded* en anglais), le style *LBYL* peut engendrer un séquencement critique (*race condition* en anglais) entre le "regarde" et le "sauter". Par exemple, le code `if key in mapping: return mapping[key]` peut échouer si un autre fil d'exécution supprime la clé *key* du *mapping* après le test mais avant l'accès. Ce problème peut être résolu avec des verrous (*locks*) ou avec l'approche EAFP.

lexical analyzer

Formal name for the *tokenizer*; see *token*.

liste

A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is *O*(1).

liste en compréhension (ou liste en intension)

Écriture concise pour manipuler tout ou partie des éléments d'une séquence et renvoyer une liste contenant les résultats. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` génère la liste composée des nombres pairs de 0 à 255 écrits sous formes de chaînes de caractères et en hexadécimal (`0x…`). La clause `if` est optionnelle. Si elle est omise, tous les éléments du `range(256)` seront utilisés.

chargeur

An object that loads a module. It must define the `exec_module()` and `create_module()` methods to implement the `Loader` interface. A loader is typically returned by a *finder*. See also :

— finders-and-loaders

— `importlib.abc.Loader`

— PEP 302

encodage régional
Sous Unix, il est défini par la variable régionale LC_CTYPE. Il peut être modifié par `locale.setlocale(locale.LC_CTYPE, new_locale)`.

Sous Windows, c'est un encodage ANSI (par ex. : `"cp1252"`).

Sous Android et VxWorks, Python utilise `"utf-8"` comme encodage régional.

`locale.getencoding()` can be used to get the locale encoding.

Voir aussi l'*encodage du systèmes de fichiers et gestionnaire d'erreurs associé*.

méthode magique
Un synonyme informel de *special method*.

tableau de correspondances (*mapping* en anglais)
Conteneur permettant de rechercher des éléments à partir de clés et implémentant les méthodes spécifiées dans les classes mères abstraites des `tableaux de correspondances` (immuables) ou `tableaux de correspondances mutables` (voir les classes mères abstraites). Les classes suivantes sont des exemples de tableaux de correspondances : `dict`, `collections.defaultdict`, `collections.OrderedDict` et `collections.Counter`.

chercheur dans les méta-chemins
Un *chercheur* renvoyé par une recherche dans `sys.meta_path`. Les chercheurs dans les méta-chemins ressemblent, mais sont différents des *chercheurs d'entrée dans path*.

Voir `importlib.abc.MetaPathFinder` pour les méthodes que les chercheurs dans les méta-chemins doivent implémenter.

métaclasse
Classe d'une classe. Les définitions de classe créent un nom pour la classe, un dictionnaire de classe et une liste de classes parentes. La métaclasse a pour rôle de réunir ces trois paramètres pour construire la classe. La plupart des langages orientés objet fournissent une implémentation par défaut. La particularité de Python est la possibilité de créer des métaclasses personnalisées. La plupart des utilisateurs n'auront jamais besoin de cet outil, mais lorsque le besoin survient, les métaclasses offrent des solutions élégantes et puissantes. Elles sont utilisées pour journaliser les accès à des propriétés, rendre sûrs les environnements *multi-threads*, suivre la création d'objets, implémenter des singletons et bien d'autres tâches.

Plus d'informations sont disponibles dans : metaclasses.

méthode
Fonction définie à l'intérieur d'une classe. Lorsqu'elle est appelée comme un attribut d'une instance de cette classe, la méthode reçoit l'instance en premier *argument* (qui, par convention, est habituellement nommé `self`). Voir *function* et *nested scope*.

ordre de résolution des méthodes
Method Resolution Order is the order in which base classes are searched for a member during lookup. See python_2.3_mro for details of the algorithm used by the Python interpreter since the 2.3 release.

module
Objet utilisé pour organiser une portion unitaire de code en Python. Les modules ont un espace de nommage et peuvent contenir n'importe quels objets Python. Charger des modules est appelé *importer*.

Voir aussi *paquet*.

spécificateur de module
Espace de nommage contenant les informations, relatives à l'importation, utilisées pour charger un module. C'est une instance de la classe `importlib.machinery.ModuleSpec`.

See also module-specs.

MRO
Voir *ordre de résolution des méthodes*.

mutable
Un objet mutable peut changer de valeur tout en gardant le même `id()`. Voir aussi *immuable*.

***n*-uplet nommé**
Le terme "n-uplet nommé" s'applique à tous les types ou classes qui héritent de la classe `tuple` et dont les éléments indexables sont aussi accessibles en utilisant des attributs nommés. Les types et classes peuvent avoir aussi d'autres caractéristiques.

Plusieurs types natifs sont appelés n-uplets, y compris les valeurs retournées par `time.localtime()` et `os.stat()`. Un autre exemple est `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
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand, or it can be created by inheriting `typing.NamedTuple`, or with the factory function `collections.namedtuple()`. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

espace de nommage
L'endroit où une variable est stockée. Les espaces de nommage sont implémentés avec des dictionnaires. Il existe des espaces de nommage globaux, natifs ou imbriqués dans les objets (dans les méthodes). Les espaces de nommage favorisent la modularité car ils permettent d'éviter les conflits de noms. Par exemple, les fonctions `builtins.open` et `os.open()` sont différenciées par leurs espaces de nom. Les espaces de nommage aident aussi à la lisibilité et la maintenabilité en rendant clair quel module implémente une fonction. Par exemple, écrire `random.seed()` ou `itertools.islice()` affiche clairement que ces fonctions sont implémentées respectivement dans les modules `random` et `itertools`.

paquet-espace de nommage
A *package* which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a *regular package* because they have no `__init__.py` file.

Namespace packages allow several individually installable packages to have a common parent package. Otherwise, it is recommended to use a *regular package*.

For more information, see **PEP 420** and reference-namespace-package.

Voir aussi *module*.

portée imbriquée
Possibilité de faire référence à une variable déclarée dans une définition englobante. Typiquement, une fonction définie à l'intérieur d'une autre fonction a accès aux variables de cette dernière. Souvenez-vous cependant que cela ne fonctionne que pour accéder à des variables, pas pour les assigner. Les variables locales sont lues et assignées dans l'espace de nommage le plus proche. Tout comme les variables globales qui sont stockés dans l'espace de nommage global, le mot clef `nonlocal` permet d'écrire dans l'espace de nommage dans lequel est déclarée la variable.

nouvelle classe
Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

objet
N'importe quelle donnée comportant des états (sous forme d'attributs ou d'une valeur) et un comportement (des méthodes). C'est aussi (`object`) l'ancêtre commun à absolument toutes les *nouvelles classes*.

optimized scope
A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note : most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

paquet
module Python qui peut contenir des sous-modules ou des sous-paquets. Techniquement, un paquet est un module qui possède un attribut `__path__`.

Voir aussi *paquet classique* et *namespace package*.

paramètre
Entité nommée dans la définition d'une *fonction* (ou méthode), décrivant un *argument* (ou dans certains cas

des arguments) que la fonction accepte. Il existe cinq sortes de paramètres :

— *positional-or-keyword* : l'argument peut être passé soit par sa *position*, soit en tant que *argument nommé*. C'est le type de paramètre par défaut. Par exemple, *foo* et *bar* dans l'exemple suivant :

```
def func(foo, bar=None): ...
```

— *positional-only* : définit un argument qui ne peut être fourni que par position. Les paramètres *positional-only* peuvent être définis en insérant un caractère "/" dans la liste de paramètres de la définition de fonction après eux. Par exemple : *posonly1* et *posonly2* dans le code suivant :

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

— *keyword-only* : l'argument ne peut être fourni que nommé. Les paramètres *keyword-only* peuvent être définis en utilisant un seul paramètre *var-positional*, ou en ajoutant une étoile (`*`) seule dans la liste des paramètres avant eux. Par exemple, *kw_only1* et *kw_only2* dans le code suivant :

```
def func(arg, *, kw_only1, kw_only2): ...
```

— *var-positional* : une séquence d'arguments positionnels peut être fournie (en plus de tous les arguments positionnels déjà acceptés par d'autres paramètres). Un tel paramètre peut être défini en préfixant son nom par une `*`. Par exemple *args* ci-après :

```
def func(*args, **kwargs): ...
```

— *var-keyword* : une quantité arbitraire d'arguments peut être passée, chacun étant nommé (en plus de tous les arguments nommés déjà acceptés par d'autres paramètres). Un tel paramètre est défini en préfixant le nom du paramètre par `**`. Par exemple, *kwargs* ci-dessus.

Les paramètres peuvent spécifier des arguments obligatoires ou optionnels, ainsi que des valeurs par défaut pour les arguments optionnels.

Voir aussi *argument* dans le glossaire, la question sur la différence entre les arguments et les paramètres dans la FAQ, la classe `inspect.Parameter`, la section function et la **PEP 362**.

entrée de chemin
: Emplacement dans le *chemin des importations* (*import path* en anglais, d'où le *path*) que le *chercheur basé sur les chemins* consulte pour trouver des modules à importer.

chercheur de chemins
: *chercheur* renvoyé par un appelable sur un `sys.path_hooks` (c'est-à-dire un *point d'entrée pour la recherche dans path*) qui sait où trouver des modules lorsqu'on lui donne une *entrée de path*.

Voir `importlib.abc.PathEntryFinder` pour les méthodes qu'un chercheur d'entrée dans *path* doit implémenter.

point d'entrée pour la recherche dans *path*
: A callable on the `sys.path_hooks` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

chercheur basé sur les chemins
: L'un des *chercheurs dans les méta-chemins* par défaut qui cherche des modules dans un *chemin des importations*.

objet simili-chemin
: Objet représentant un chemin du système de fichiers. Un objet simili-chemin est un objet `str` ou un objet `bytes` représentant un chemin ou un objet implémentant le protocole `os.PathLike`. Un objet qui accepte le protocole `os.PathLike` peut être converti en un chemin `str` ou `bytes` du système de fichiers en appelant la fonction `os.fspath()`. `os.fsdecode()` et `os.fsencode()` peuvent être utilisées, respectivement, pour garantir un résultat de type `str` ou `bytes` à la place. A été Introduit par la **PEP 519**.

PEP
: *Python Enhancement Proposal* (Proposition d'amélioration de Python). Une PEP est un document de conception fournissant des informations à la communauté Python ou décrivant une nouvelle fonctionnalité pour Python, ses processus ou son environnement. Les PEP doivent fournir une spécification technique concise et une justification des fonctionnalités proposées.

Les PEP sont censées être les principaux mécanismes pour proposer de nouvelles fonctionnalités majeures, pour recueillir les commentaires de la communauté sur une question et pour documenter les décisions de

conception qui sont intégrées en Python. L'auteur du PEP est responsable de l'établissement d'un consensus au sein de la communauté et de documenter les opinions contradictoires.

Voir la **PEP 1**.

portion
Jeu de fichiers dans un seul dossier (pouvant être stocké sous forme de fichier zip) qui contribue à l'espace de nommage d'un paquet, tel que défini dans la **PEP 420**.

argument positionnel
Voir *argument*.

API provisoire
Une API provisoire est une API qui n'offre aucune garantie de rétrocompatibilité (la bibliothèque standard exige la rétrocompatibilité). Bien que des changements majeurs d'une telle interface ne soient pas attendus, tant qu'elle est étiquetée provisoire, des changements cassant la rétrocompatibilité (y compris sa suppression complète) peuvent survenir si les développeurs principaux le jugent nécessaire. Ces modifications ne surviendront que si de sérieux problèmes sont découverts et qu'ils n'avaient pas été identifiés avant l'ajout de l'API.

Même pour les API provisoires, les changements cassant la rétrocompatibilité sont considérés comme des "solutions de dernier recours". Tout ce qui est possible sera fait pour tenter de résoudre les problèmes en conservant la rétrocompatibilité.

Ce processus permet à la bibliothèque standard de continuer à évoluer avec le temps, sans se bloquer longtemps sur des erreurs d'architecture. Voir la **PEP 411** pour plus de détails.

paquet provisoire
Voir *provisional API*.

Python 3000
Surnom donné à la série des Python 3.x (très vieux surnom donné à l'époque où Python 3 représentait un futur lointain). Aussi abrégé *Py3k*.

Pythonique
Idée, ou bout de code, qui colle aux idiomes de Python plutôt qu'aux concepts communs rencontrés dans d'autres langages. Par exemple, il est idiomatique en Python de parcourir les éléments d'un itérable en utilisant `for`. Beaucoup d'autres langages n'ont pas cette possibilité, donc les gens qui ne sont pas habitués à Python utilisent parfois un compteur numérique à la place :

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

Plutôt qu'utiliser la méthode, plus propre et élégante, donc *Pythonique* :

```
for piece in food:
    print(piece)
```

nom qualifié
Nom, comprenant des points, montrant le "chemin" de l'espace de nommage global d'un module vers une classe, fonction ou méthode définie dans ce module, tel que défini dans la **PEP 3155**. Pour les fonctions et classes de premier niveau, le nom qualifié est le même que le nom de l'objet :

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Lorsqu'il est utilisé pour nommer des modules, le *nom qualifié complet* (*fully qualified name* - *FQN* en anglais)

signifie le chemin complet (séparé par des points) vers le module, incluant tous les paquets parents. Par exemple : `email.mime.text` :

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

nombre de références
The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Some objects are *immortal* and have reference counts that are never modified, and therefore the objects are never deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. Programmers can call the `sys.getrefcount()` function to return the reference count for a particular object.

In *CPython*, reference counts are not considered to be stable or well-defined values ; the number of references to an object, and how that number is affected by Python code, may be different between versions.

paquet classique
paquet traditionnel, tel qu'un dossier contenant un fichier `__init__.py`.

Voir aussi *paquet-espace de nommage*.

REPL
An acronym for the "read–eval–print loop", another name for the *interactive* interpreter shell.

__slots__
Déclaration dans une classe qui économise de la mémoire en pré-allouant de l'espace pour les attributs des instances et qui élimine le dictionnaire (des attributs) des instances. Bien que populaire, cette technique est difficile à maîtriser et devrait être réservée à de rares cas où un grand nombre d'instances dans une application devient un sujet critique pour la mémoire.

séquence
An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *hashable* keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see Common Sequence Operations.

ensemble en compréhension (ou ensemble en intension)
Une façon compacte de traiter tout ou partie des éléments d'un itérable et de renvoyer un *set* avec les résultats. `results = {c for c in 'abracadabra' if c not in 'abc'}` génère l'ensemble contenant les lettres « r » et « d » `{'r', 'd'}`. Voir comprehensions.

distribution simple
Forme de distribution, comme les *fonction génériques*, où l'implémentation est choisie en fonction du type d'un seul argument.

tranche
(*slice* en anglais), un objet contenant habituellement une portion de *séquence*. Une tranche est créée en utilisant la notation `[]` avec des `:` entre les nombres lorsque plusieurs sont fournis, comme dans `variable_name[1:3:5]`. Cette notation utilise des objets `slice` en interne.

soft deprecated
A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See PEP 387 : Soft Deprecation.

méthode spéciale
(*special method* en anglais) Méthode appelée implicitement par Python pour exécuter une opération sur un type, comme une addition. De telles méthodes ont des noms commençant et terminant par des doubles tirets bas. Les méthodes spéciales sont documentées dans specialnames.

standard library
The collection of *packages*, *modules* and *extension modules* distributed as a part of the official Python interpreter package. The exact membership of the collection may vary based on platform, available system libraries, or other criteria. Documentation can be found at library-index.

See also `sys.stdlib_module_names` for a list of all possible standard library module names.

instruction
Une instruction (*statement* en anglais) est un composant d'un "bloc" de code. Une instruction est soit une *expression*, soit une ou plusieurs constructions basées sur un mot-clé, comme `if`, `while` ou `for`.

static type checker
An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the `typing` module.

stdlib
An abbreviation of *standard library*.

référence forte
In Python's C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling *`Py_INCREF()`* when the reference is created and released with *`Py_DECREF()`* when the reference is deleted.

Une référence forte est créée à l'aide de la fonction *`Py_NewRef()`*. Il faut normalement appeler *`Py_DECREF()`* dessus avant de sortir de sa portée lexicale, sans quoi il y a une fuite de référence.

Voir aussi *référence empruntée*.

t-string
t-strings
String literals prefixed with `t` or `T` are commonly called "t-strings" which is short for template string literals.

encodages de texte
Une chaîne de caractères en Python est une suite de points de code Unicode (dans l'intervalle `U+0000`--`U+10FFFF`). Pour stocker ou transmettre une chaîne, il est nécessaire de la sérialiser en suite d'octets.

Sérialiser une chaîne de caractères en une suite d'octets s'appelle « encoder » et recréer la chaîne à partir de la suite d'octets s'appelle « décoder ».

Il existe de multiples codecs pour la sérialisation de texte, que l'on regroupe sous l'expression « encodages de texte ».

fichier texte
Objet fichier capable de lire et d'écrire des objets `str`. Souvent, un fichier texte (*text file* en anglais) accède en fait à un flux de donnée en octets et gère l'*encodage de texte* automatiquement. Des exemples de fichiers textes sont les fichiers ouverts en mode texte (`'r'` ou `'w'`), `sys.stdin`, `sys.stdout` et les instances de `io.StringIO`.

Voir aussi *fichier binaire* pour un objet fichier capable de lire et d'écrire des *objets octets-compatibles*.

thread state
The information used by the *CPython* runtime to run in an OS thread. For example, this includes the current exception, if any, and the state of the bytecode interpreter.

Each thread state is bound to a single OS thread, but threads may have many thread states available. At most, one of them may be *attached* at once.

An *attached thread state* is required to call most of Python's C API, unless a function explicitly documents otherwise. The bytecode interpreter only runs under an attached thread state.

Each thread state belongs to a single interpreter, but each interpreter may have many thread states, including multiple for the same OS thread. Thread states from multiple interpreters may be bound to the same thread, but only one can be *attached* in that thread at any given moment.

See *Thread State and the Global Interpreter Lock* for more information.

token
A small unit of source code, generated by the lexical analyzer (also called the *tokenizer*). Names, numbers, strings, operators, newlines and similar are represented by tokens.

The `tokenize` module exposes Python's lexical analyzer. The `token` module contains information on the various types of tokens.

chaîne entre triple guillemets
: Chaîne qui est délimitée par trois guillemets simples (`'`) ou trois guillemets doubles (`"`). Bien qu'elle ne fournisse aucune fonctionnalité qui ne soit pas disponible avec une chaîne entre guillemets, elle est utile pour de nombreuses raisons. Elle vous autorise à insérer des guillemets simples et doubles dans une chaîne sans avoir à les protéger et elle peut s'étendre sur plusieurs lignes sans avoir à terminer chaque ligne par un `\`. Elle est ainsi particulièrement utile pour les chaînes de documentation (*docstrings*).

type
: The type of a Python object determines what kind of object it is ; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

alias de type
: Synonyme d'un type, créé en affectant le type à un identifiant.

  Les alias de types sont utiles pour simplifier les *indications de types*. Par exemple :

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

  pourrait être rendu plus lisible comme ceci :

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

  Voir `typing` et la **PEP 484**, qui décrivent cette fonctionnalité.

indication de type
: L'*annotation* qui spécifie le type attendu pour une variable, un attribut de classe, un paramètre de fonction ou une valeur de retour.

  Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

  Les indications de type de variables globales, d'attributs de classe et de fonctions, mais pas de variables locales, peuvent être consultées en utilisant `typing.get_type_hints()`.

  Voir `typing` et la **PEP 484**, qui décrivent cette fonctionnalité.

retours à la ligne universels
: Une manière d'interpréter des flux de texte dans lesquels sont reconnues toutes les fins de ligne suivantes : la convention Unix `'\n'`, la convention Windows `'\r\n'` et l'ancienne convention Macintosh `'\r'`. Voir la **PEP 278** et la **PEP 3116**, ainsi que la fonction `bytes.splitlines()` pour d'autres usages.

annotation de variable
: *annotation* d'une variable ou d'un attribut de classe.

  Lorsque vous annotez une variable ou un attribut de classe, l'affectation est facultative :

```
class C:
    field: 'annotation'
```

  Les annotations de variables sont généralement utilisées pour des *indications de types* : par exemple, cette variable devrait prendre des valeurs de type `int` :

```
count: int = 0
```

  La syntaxe d'annotation de variable est expliquée dans la section annassign.

  Reportez-vous à *annotation de fonction*, à la **PEP 484** et à la **PEP 526** qui décrivent cette fonctionnalité. Voir aussi annotations-howto sur les bonnes pratiques concernant les annotations.

environnement virtuel
: Environnement d'exécution isolé (en mode coopératif) qui permet aux utilisateurs de Python et aux applications d'installer et de mettre à jour des paquets sans interférer avec d'autres applications Python fonctionnant sur le même système.

Voir aussi `venv`.

machine virtuelle
: Ordinateur défini entièrement par du logiciel. La machine virtuelle (*virtual machine*) de Python exécute le *code intermédiaire* produit par le compilateur de *bytecode*.

walrus operator
: A light-hearted way to refer to the assignment expression operator `:=` because it looks a bit like a walrus if you turn your head.

Le zen de Python
: Liste de principes et de préceptes utiles pour comprendre et utiliser le langage. Cette liste peut être obtenue en tapant "`import this`" dans une invite Python interactive.

ANNEXE B

About this documentation

Python's documentation is generated from reStructuredText sources using Sphinx, a documentation generator originally created for Python and now maintained as an independent project.

Le développement de la documentation et de ses outils est entièrement basé sur le volontariat, tout comme Python. Si vous voulez contribuer, allez voir la page reporting-bugs qui contient des informations pour vous y aider. Les nouveaux volontaires sont toujours les bienvenus !

Merci beaucoup à :

— Fred L. Drake, Jr., the creator of the original Python documentation toolset and author of much of the content ;
— le projet Docutils pour avoir créé *reStructuredText* et la suite d'outils *Docutils* ;
— Fredrik Lundh pour son projet *Alternative Python Reference*, dont Sphinx a pris beaucoup de bonnes idées.

B.1 Contributors to the Python documentation

De nombreuses personnes ont contribué au langage Python, à sa bibliothèque standard et à sa documentation. Consultez Misc/ACKS dans les sources de la distribution Python pour avoir une liste partielle des contributeurs.

Ce n'est que grâce aux suggestions et contributions de la communauté Python que Python a une documentation si merveilleuse — Merci !

ANNEXE C

Histoire et licence

C.1 Histoire du logiciel

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations, which became Zope Corporation. In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation was a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible ; the table below summarizes the various releases.

Version	Dérivé de	Année	Propriétaire	GPL-compatible ? (1)
0.9.0 à 1.2	n/a	1991-1995	CWI	oui
1.3 à 1.5.2	1.2	1995-1999	CNRI	oui
1.6	1.5.2	2000	CNRI	non
2.0	1.6	2000	BeOpen.com	non
1.6.1	1.6	2001	CNRI	yes (2)
2.1	2.0+1.6.1	2001	PSF	non
2.0.1	2.0+1.6.1	2001	PSF	oui
2.1.1	2.1+2.0.1	2001	PSF	oui
2.1.2	2.1.1	2002	PSF	oui
2.1.3	2.1.2	2002	PSF	oui
2.2 et ultérieure	2.1.1	2001-maintenant	PSF	oui

Note

(1) GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL ; the others don't.

(2) According to Richard Stallman, 1.6.1 is not GPL-compatible, because its license has a choice of law clause. According to CNRI, however, Stallman's lawyer has told CNRI's lawyer that 1.6.1 is "not incompatible" with the GPL.

Merci aux nombreux bénévoles qui ont travaillé sous la direction de Guido pour rendre ces versions possibles.

C.2 Conditions générales pour accéder à, ou utiliser, Python

Python software and documentation are licensed under the Python Software Foundation License Version 2.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Version 2 and the *Zero-Clause BSD license*.

Certains logiciels faisant partie de Python sont soumis à d'autres licences. Ces licences sont incluses avec le code lié à celles-ci. Voir *Licences et remerciements pour les logiciels tiers* pour une liste non exhaustive de ces licences.

C.2.1 PYTHON SOFTWARE FOUNDATION LICENSE VERSION 2

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using this
   software ("Python") in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001 Python Software Foundation; All Rights
   Reserved" are retained in Python alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee hereby
   agrees to include in any such work a brief summary of the changes made to␣
↪Python.

4. PSF is making Python available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF PYTHON WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship
```

(suite sur la page suivante)

(suite de la page précédente)

```
   of agency, partnership, or joint venture between PSF and Licensee.  This License
   Agreement does not grant permission to use PSF trademarks or trade name in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.

8. By copying, installing or otherwise using Python, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 LICENCE D'UTILISATION BEOPEN.COM POUR PYTHON 2.0

LICENCE D'UTILISATION LIBRE BEOPEN PYTHON VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 LICENCE D'UTILISATION CNRI POUR 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
```

(suite sur la page suivante)

(suite de la page précédente)

```
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013".

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 LICENCE D'UTILISATION CWI POUR PYTHON 0.9.0 à 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 Licences et remerciements pour les logiciels tiers

Cette section est une liste incomplète mais grandissante de licences et remerciements pour les logiciels tiers incorporés dans la distribution de Python.

C.3.1 Mersenne twister

The `_random` C extension underlying the `random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code :

```
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
```

(suite sur la page suivante)

(suite de la page précédente)

```
modification, are permitted provided that the following conditions
are met:

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

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

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Interfaces de connexion (*sockets*)

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, https://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

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
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 Interfaces de connexion asynchrones

The `test.support.asynchat` and `test.support.asyncore` modules contain the following notice :

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Gestion de témoin (*cookie*)

Le module `http.cookies` contient la note suivante :

```
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
```

(suite sur la page suivante)

(suite de la page précédente)

```
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Traçage d'exécution

Le module `trace` contient la note suivante :

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 Les fonctions UUencode et UUdecode

The `uu` codec contains the following notice :

```
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.
```

(suite sur la page suivante)

(suite de la page précédente)

```
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 Appel de procédures distantes en XML (*RPC*, pour *Remote Procedure Call*)

Le module `xmlrpc.client` contient la note suivante :

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

The `test.test_epoll` module contains the following notice :

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(suite sur la page suivante)

(suite de la page précédente)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

Le module `select` contient la note suivante pour l'interface *kqueue* :

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

Le fichier `Python/pyhash.c` contient une implémentation par Marek Majkowski de l'algorithme *SipHash24* de Dan Bernstein. Il contient la note suivante :

```
<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/
```

(suite sur la page suivante)

(suite de la page précédente)

```
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* et *dtoa*

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice :

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies :

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
```

(suite sur la page suivante)

(suite de la page précédente)

```
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
```

(suite sur la page suivante)

(suite de la page précédente)

```
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
       attribution notices from the Source form of the Work,
       excluding those notices that do not pertain to any part of
       the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
       distribution, then any Derivative Works that You distribute must
       include a readable copy of the attribution notices contained
       within such NOTICE file, excluding those notices that do not
       pertain to any part of the Derivative Works, in at least one
       of the following places: within a NOTICE text file distributed
       as part of the Derivative Works; within the Source form or
       documentation, if provided along with the Derivative Works; or,
       within a display generated by the Derivative Works, if and
       wherever such third-party notices normally appear. The contents
       of the NOTICE file are for informational purposes only and
       do not modify the License. You may add Your own attribution
       notices within Derivative Works that You distribute, alongside
       or as an addendum to the NOTICE text from the Work, provided
       that such additional attribution notices cannot be construed
       as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
```

(suite sur la page suivante)

(suite de la page précédente)

```
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--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
```

(suite sur la page suivante)

(suite de la page précédente)

```
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

The `_ctypes` C extension underlying the `ctypes` module is built using an included copy of the libffi sources unless the build is configured `--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

Le module `zlib` est compilé en utilisant une copie du code source de *zlib* si la version de *zlib* trouvée sur le système est trop vieille pour être utilisée :

```
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.
```

(suite sur la page suivante)

(suite de la page précédente)

```
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

L'implémentation des dictionnaires, utilisée par le module `tracemalloc` est basée sur le projet *cfuhash* :

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

This code is released under the BSD license:

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

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

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

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

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
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

The `_decimal` C extension underlying the `decimal` module is built using an included copy of the libmpdec library unless the build is configured `--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 Ensemble de tests C14N du W3C

Les tests de C14N version 2.0 du module `test` (`Lib/test/xmltestdata/c14n-20/`) proviennent du site du W3C à l'adresse https://www.w3.org/TR/xml-c14n2-testcases/ et sont distribués sous licence BSD modifiée :

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

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

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
```

(suite sur la page suivante)

(suite de la page précédente)

```
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT License :

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

C.3.20 *asyncio*

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license :

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's "Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License :

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR "AS IS" AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.22 Zstandard bindings

Zstandard bindings in `Modules/_zstd` and `Lib/compression/zstd` are based on code from the pyzstd library, copyright Ma Lin and contributors. The pyzstd code is distributed under the 3-Clause BSD License :

```
Copyright (c) 2020-present, Ma Lin and contributors.
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 copyright holder nor the names of its
   contributors may be used to endorse or promote products derived from
   this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER 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,
```

(suite sur la page suivante)

(suite de la page précédente)

```
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.
```

ANNEXE D

Copyright

Python et cette documentation sont :

Copyright © 2001 Python Software Foundation. All rights reserved.

Copyright © 2000 *BeOpen.com*. Tous droits réservés.

Copyright © 1995-2000 *Corporation for National Research Initiatives*. Tous droits réservés.

Copyright © 1991-1995 *Stichting Mathematisch Centrum*. Tous droits réservés.

Voir *Histoire et licence* pour des informations complètes concernant la licence et les permissions.

Bibliographie

[win] `PyExc_WindowsError` is only defined on Windows; protect code that uses this by testing that the preprocessor macro `MS_WINDOWS` is defined.

Index

Non alphabétique

. . ., **351**
    ellipsis literal, 351
>>>, **351**
__all__ (*package variable*), 74
__dict__ (*module attribute*), 193
__doc__ (*module attribute*), 193
__file__ (*module attribute*), 193, 194
__future__, **357**
__import__
    built-in function, 75
__loader__ (*module attribute*), 193
__main__
    module, 12, 221, 236, 237
__name__ (*module attribute*), 193
__package__ (*module attribute*), 193
__PYVENV_LAUNCHER__, 259, 265
__slots__, **365**
_frozen (*C struct*), 77
_inittab (*C struct*), 78
_inittab.initfunc (*C member*), 78
_inittab.name (*C member*), 78
_Py_c_diff (*C function*), 146
_Py_c_neg (*C function*), 146
_Py_c_pow (*C function*), 146
_Py_c_prod (*C function*), 146
_Py_c_quot (*C function*), 146
_Py_c_sum (*C function*), 146
_Py_NoneStruct (*C var*), 285
_PyBytes_Resize (*C function*), 149
_PyCode_GetExtra (*C function*), 191
_PyCode_SetExtra (*C function*), 191
_PyEval_RequestCodeExtraIndex (*C function*), 191
_PyFrameEvalFunction (*C type*), 234
_PyInterpreterFrame (*C struct*), 209
_PyInterpreterState_GetEvalFrameFunc (*C function*), 234
_PyInterpreterState_SetEvalFrameFunc (*C function*), 234
_PyObject_GetDictPtr (*C function*), 101
_PyObject_New (*C function*), 283
_PyObject_NewVar (*C function*), 283
_PyTuple_Resize (*C function*), 173
_thread
    module, 230

A

abort (*C function*), 74
abs
    built-in function, 112
alias de type, **367**
allocfunc (*C type*), 332
annotate function, **351**
annotation, **351**
annotation de fonction, **357**
annotation de variable, **367**
API provisoire, **364**
appelable (*callable*), **353**
argument, **351**
argument nommé, **360**
argument positionnel, **364**
argv (*in module sys*), 226, 259
arrêt de l'interpréteur, **359**
ascii
    built-in function, 101
attached thread state, **352**
attendable (*awaitable*), **353**
attribut, **353**

B

BDFL, **353**
binaryfunc (*C type*), 333
buffer interface
    (see buffer protocol), 119
buffer object
    (see buffer protocol), 119
buffer protocol, 119
built-in function
    __import__, 75
    abs, 112
    ascii, 101
    bytes, 102
    classmethod, 293
    compile, 76
    divmod, 112
    float, 114

hash, 102, 309
int, 114
len, 103, 115, 117, 175, 179, 182
pow, 112, 113
repr, 101, 308
staticmethod, 293
tuple, 116, 176
type, 102
builtins
module, 12, 221, 236, 237
bytearray
object, 150
bytes
built-in function, 102
object, 148

C

calloc (*C function*), 273
Capsule
object, 205
C-contiguous, 122, 354
chaîne de documentation (*docstring*), **355**
chaîne entre triple guillemets, **367**
chargeur, **360**
chemin des importations, **359**
chercheur, **357**
chercheur basé sur les chemins, **363**
chercheur dans les méta-chemins, **361**
chercheur de chemins, **363**
classe, **354**
classe mère abstraite, **351**
classmethod
built-in function, 293
cleanup functions, 74
close (*in module os*), 237
closure variable, **354**
CO_ASYNC_GENERATOR (*C macro*), 190
CO_COROUTINE (*C macro*), 190
CO_FUTURE_ABSOLUTE_IMPORT (*C macro*), 190
CO_FUTURE_ANNOTATIONS (*C macro*), 190
CO_FUTURE_DIVISION (*C macro*), 190
CO_FUTURE_GENERATOR_STOP (*C macro*), 190
CO_FUTURE_PRINT_FUNCTION (*C macro*), 190
CO_FUTURE_UNICODE_LITERALS (*C macro*), 190
CO_FUTURE_WITH_STATEMENT (*C macro*), 190
CO_GENERATOR (*C macro*), 190
CO_HAS_DOCSTRING (*C macro*), 190
CO_ITERABLE_COROUTINE (*C macro*), 190
CO_METHOD (*C macro*), 190
CO_NESTED (*C macro*), 190
CO_NEWLOCALS (*C macro*), 190
CO_OPTIMIZED (*C macro*), 190
CO_VARARGS (*C macro*), 190
CO_VARKEYWORDS (*C macro*), 190
code intermédiaire (*bytecode*), **353**
code object, 187
compile
built-in function, 76
complex number
object, 146
context, **354**
context management protocol, **354**
contigu, **354**
contiguous, 122
copyright (*in module sys*), 225
coroutine, **354**
CPython, **355**
current context, **355**
cyclic isolate, **355**

D

décorateur, **355**
descrgetfunc (*C type*), 333
descripteur, **355**
descrsetfunc (*C type*), 333
destructor (*C type*), 333
dictionary
object, 176
dictionnaire, **355**
dictionnaire en compréhension (*ou dictionnaire en intension*), **355**
distribution simple, **365**
division entière, **357**
divmod
built-in function, 112
dunder, **356**

E

EAFP, **356**
encodage du système de fichiers et gestionnaire d'erreurs associé, **356**
encodage régional, **361**
encodages de texte, **366**
ensemble en compréhension (*ou ensemble en intension*), **365**
entrée de chemin, **363**
environnement virtuel, **367**
EOFError (*built-in exception*), 192
espace de nommage, **362**
evaluate function, **356**
exc_info (*in module sys*), 10
executable (*in module sys*), 224
exit (*C function*), 74
expression, **356**
expression génératrice, **358**

F

f-string, **356**
f-strings, **356**
fichier binaire, **353**
fichier texte, **366**
file
object, 192
float
built-in function, 114

floating-point
    object, 144
fonction, **357**
fonction clé, **360**
fonction coroutine, **355**
fonction de rappel (*callback*), **353**
fonction générique, **358**
Fortran contiguous, 122, 354
free (*C function*), 273
free threading, **357**
free variable, **357**
freefunc (*C type*), 333
freeze utility, 77
frozenset
    object, 181
function
    object, 183

G

gcvisitobjects_t (*C type*), 340
générateur, **357**
générateur asynchrone, **352**
gestionnaire de contexte, **354**
gestionnaire de contexte asynchrone, **352**
getattrfunc (*C type*), 333
getattrofunc (*C type*), 333
getbufferproc (*C type*), 333
getiterfunc (*C type*), 333
getter (*C type*), 297
GIL, **358**
global interpreter lock, 227

H

hachable, **358**
hash
    built-in function, 102, 309
hashfunc (*C type*), 333

I

IDLE, **359**
immortal, **359**
immuable, **359**
importateur, **359**
importation, **359**
incr_item(), 11
indication de type, **367**
initproc (*C type*), 333
inquiry (*C type*), 339
instancemethod
    object, 185
instruction, **366**
int
    built-in function, 114
integer
    object, 134
interactif, **359**
interprété, **359**
interpreter lock, 227
itérable, **359**
itérable asynchrone, **352**
itérateur, **359**
itérateur asynchrone, **352**
itérateur de générateur, **358**
itérateur de générateur asynchrone, **352**
iternextfunc (*C type*), 333

K

KeyboardInterrupt (*built-in exception*), 53

L

lambda, **360**
LBYL, **360**
Le zen de Python, **368**
len
    built-in function, 103, 115, 117, 175, 179, 182
lenfunc (*C type*), 333
lexical analyzer, **360**
list
    object, 174
liste, **360**
liste en compréhension (*ou liste en intension*), **360**
lock, interpreter, 227
long integer
    object, 134
LONG_MAX (*C macro*), 136

M

machine virtuelle, **368**
magic
    méthode, 361
main(), 223, 226, 259
malloc (*C function*), 273
mapping
    object, 176
memoryview
    object, 203
métaclasse, **361**
METH_CLASS (*C macro*), 293
METH_COEXIST (*C macro*), 293
METH_FASTCALL (*C macro*), 292
METH_KEYWORDS (*C macro*), 292
METH_METHOD (*C macro*), 293
METH_NOARGS (*C macro*), 293
METH_O (*C macro*), 293
METH_STATIC (*C macro*), 293
METH_VARARGS (*C macro*), 292
method
    object, 185
méthode, **361**
    magic, 361
    special, 365
méthode magique, **361**
méthode spéciale, **365**
MethodType (*in module types*), 183, 186
module, **361**

__main__, 12, 221, 236, 237
_thread, 230
builtins, 12, 221, 236, 237
object, 193
search path, 12, 221, 225
signal, 53
sys, 12, 221, 236, 237
module d'extension, **356**
modules (*in module sys*), 74, 221
ModuleType (*in module types*), 193
MRO, **361**
mutable, **361**

N

n-uplet nommé, **361**
newfunc (*C type*), 333
nom qualifié, **364**
nombre complexe, **354**
nombre de références, **365**
None
object, 134
nouvelle classe, **362**
numeric
object, 134

O

object
bytearray, 150
bytes, 148
Capsule, 205
code, 187
complex number, 146
dictionary, 176
file, 192
floating-point, 144
frozenset, 181
function, 183
instancemethod, 185
integer, 134
list, 174
long integer, 134
mapping, 176
memoryview, 203
method, 185
module, 193
None, 134
numeric, 134
sequence, 148
set, 181
tuple, 172
type, 6, 127
objet, **362**
objet fichier, **356**
objet fichier-compatible, **356**
objet octet-compatible, **353**
objet simili-chemin, **363**
objobjargproc (*C type*), 334
objobjproc (*C type*), 334
optimized scope, **362**
ordre de résolution des méthodes, **361**
OverflowError (*built-in exception*), 136, 137

P

package variable
__all__, 74
paquet, **362**
paquet classique, **365**
paquet provisoire, **364**
paquet-espace de nommage, **362**
paramètre, **362**
PATH, 12
path
module search, 12, 221, 225
path (*in module sys*), 12, 221, 225
PEP, **363**
platform (*in module sys*), 225
point d'entrée pour la recherche dans path, **363**
portée imbriquée, **362**
portion, **364**
pow
built-in function, 112, 113
Py_ABS (*C macro*), 4
Py_AddPendingCall (*C function*), 238
Py_ALWAYS_INLINE (*C macro*), 4
Py_ASNATIVEBYTES_ALLOW_INDEX (*C macro*), 140
Py_ASNATIVEBYTES_BIG_ENDIAN (*C macro*), 140
Py_ASNATIVEBYTES_DEFAULTS (*C macro*), 140
Py_ASNATIVEBYTES_LITTLE_ENDIAN (*C macro*), 140
Py_ASNATIVEBYTES_NATIVE_ENDIAN (*C macro*), 140
Py_ASNATIVEBYTES_REJECT_NEGATIVE (*C macro*), 140
Py_ASNATIVEBYTES_UNSIGNED_BUFFER (*C macro*), 140
Py_AtExit (*C function*), 74
Py_AUDIT_READ (*C macro*), 295
Py_AuditHookFunction (*C type*), 74
Py_BEGIN_ALLOW_THREADS (*C macro*), 227, 232
Py_BEGIN_CRITICAL_SECTION (*C macro*), 245
Py_BEGIN_CRITICAL_SECTION2 (*C macro*), 245
Py_BEGIN_CRITICAL_SECTION2_MUTEX (*C macro*), 246
Py_BEGIN_CRITICAL_SECTION_MUTEX (*C macro*), 245
Py_BLOCK_THREADS (*C macro*), 232
Py_buffer (*C type*), 119
Py_buffer.buf (*C member*), 119
Py_buffer.format (*C member*), 120
Py_buffer.internal (*C member*), 121
Py_buffer.itemsize (*C member*), 120
Py_buffer.len (*C member*), 120
Py_buffer.ndim (*C member*), 120
Py_buffer.obj (*C member*), 120
Py_buffer.readonly (*C member*), 120

Py_buffer.shape (*C member*), 120
Py_buffer.strides (*C member*), 120
Py_buffer.suboffsets (*C member*), 121
Py_BuildValue (*C function*), 85
Py_BytesMain (*C function*), 222
Py_BytesWarningFlag (*C var*), 218
Py_CHARMASK (*C macro*), 5
Py_CLEANUP_SUPPORTED (*C macro*), 83
Py_CLEAR (*C function*), 43
Py_CompileString (*C function*), 37, 38
Py_CompileStringExFlags (*C function*), 37
Py_CompileStringFlags (*C function*), 37
Py_CompileStringObject (*C function*), 37
Py_complex (*C type*), 146
Py_complex.imag (*C member*), 146
Py_complex.real (*C member*), 146
Py_CONSTANT_ELLIPSIS (*C macro*), 98
Py_CONSTANT_EMPTY_BYTES (*C macro*), 98
Py_CONSTANT_EMPTY_STR (*C macro*), 98
Py_CONSTANT_EMPTY_TUPLE (*C macro*), 98
Py_CONSTANT_FALSE (*C macro*), 98
Py_CONSTANT_NONE (*C macro*), 98
Py_CONSTANT_NOT_IMPLEMENTED (*C macro*), 98
Py_CONSTANT_ONE (*C macro*), 98
Py_CONSTANT_TRUE (*C macro*), 98
Py_CONSTANT_ZERO (*C macro*), 98
PY_CXX_CONST (*C macro*), 85
Py_DEBUG (*C macro*), 12
Py_DebugFlag (*C var*), 218
Py_DecodeLocale (*C function*), 70
Py_DECREF (*C function*), 7, 42
Py_DecRef (*C function*), 43
Py_DEPRECATED (*C macro*), 5
Py_DontWriteBytecodeFlag (*C var*), 218
Py_Ellipsis (*C var*), 203
Py_EncodeLocale (*C function*), 71
Py_END_ALLOW_THREADS (*C macro*), 227, 232
Py_END_CRITICAL_SECTION (*C macro*), 245
Py_END_CRITICAL_SECTION2 (*C macro*), 246
Py_EndInterpreter (*C function*), 237
Py_EnterRecursiveCall (*C function*), 56
Py_EQ (*C macro*), 318
Py_eval_input (*C var*), 38
Py_Exit (*C function*), 74
Py_ExitStatusException (*C function*), 254
Py_False (*C var*), 144
Py_FatalError (*C function*), 74
Py_FatalError(), 226
Py_fclose (*C function*), 72
Py_FdIsInteractive (*C function*), 69
Py_file_input (*C var*), 38
Py_Finalize (*C function*), 222
Py_FinalizeEx (*C function*), 74, 221, 237
Py_fopen (*C function*), 72
Py_FrozenFlag (*C var*), 218
Py_GE (*C macro*), 318
Py_GenericAlias (*C function*), 216
Py_GenericAliasType (*C var*), 216
Py_GetArgcArgv (*C function*), 271
Py_GetBuildInfo (*C function*), 225
Py_GetCompiler (*C function*), 225
Py_GetConstant (*C function*), 97
Py_GetConstantBorrowed (*C function*), 98
Py_GetCopyright (*C function*), 225
Py_GETENV (*C macro*), 5
Py_GetExecPrefix (*C function*), 12, 224
Py_GetPath (*C function*), 12, 225
Py_GetPath(), 223
Py_GetPlatform (*C function*), 225
Py_GetPrefix (*C function*), 12, 224
Py_GetProgramFullPath (*C function*), 12, 224
Py_GetProgramName (*C function*), 223
Py_GetPythonHome (*C function*), 226
Py_GetVersion (*C function*), 225
Py_GT (*C macro*), 318
Py_hash_t (*C type*), 89
Py_HashBuffer (*C function*), 90
Py_HashPointer (*C function*), 90
Py_HashRandomizationFlag (*C var*), 219
Py_IgnoreEnvironmentFlag (*C var*), 219
Py_INCREF (*C function*), 7, 41
Py_IncRef (*C function*), 43
Py_Initialize (*C function*), 12, 221, 237
Py_Initialize(), 223
Py_InitializeEx (*C function*), 221
Py_InitializeFromConfig (*C function*), 221
Py_InitializeFromInitConfig (*C function*), 250
Py_InspectFlag (*C var*), 219
Py_InteractiveFlag (*C var*), 219
Py_Is (*C function*), 290
Py_IS_TYPE (*C function*), 290
Py_IsFalse (*C function*), 290
Py_IsFinalizing (*C function*), 221
Py_IsInitialized (*C function*), 12, 221
Py_IsNone (*C function*), 290
Py_IsolatedFlag (*C var*), 219
Py_IsTrue (*C function*), 290
Py_LE (*C macro*), 318
Py_LeaveRecursiveCall (*C function*), 57
Py_LegacyWindowsFSEncodingFlag (*C var*), 219
Py_LegacyWindowsStdioFlag (*C var*), 219
Py_LIMITED_API (*C macro*), 16
Py_LT (*C macro*), 318
Py_Main (*C function*), 222
PY_MAJOR_VERSION (*C macro*), 341
Py_MARSHAL_VERSION (*C macro*), 78
Py_MAX (*C macro*), 5
Py_MEMBER_SIZE (*C macro*), 5
PY_MICRO_VERSION (*C macro*), 341
Py_MIN (*C macro*), 5
PY_MINOR_VERSION (*C macro*), 341
Py_mod_create (*C macro*), 195
Py_mod_exec (*C macro*), 196
Py_mod_gil (*C macro*), 196
Py_MOD_GIL_NOT_USED (*C macro*), 196
Py_MOD_GIL_USED (*C macro*), 196

Py_mod_multiple_interpreters (*C macro*), 196
Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED (*C macro*), 196
Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED (*C macro*), 196
Py_MOD_PER_INTERPRETER_GIL_SUPPORTED (*C macro*), 196
PY_MONITORING_EVENT_BRANCH_LEFT (*C macro*), 348
PY_MONITORING_EVENT_BRANCH_RIGHT (*C macro*), 348
PY_MONITORING_EVENT_C_RAISE (*C macro*), 348
PY_MONITORING_EVENT_C_RETURN (*C macro*), 348
PY_MONITORING_EVENT_CALL (*C macro*), 348
PY_MONITORING_EVENT_EXCEPTION_HANDLED (*C macro*), 348
PY_MONITORING_EVENT_INSTRUCTION (*C macro*), 348
PY_MONITORING_EVENT_JUMP (*C macro*), 348
PY_MONITORING_EVENT_LINE (*C macro*), 348
PY_MONITORING_EVENT_PY_RESUME (*C macro*), 348
PY_MONITORING_EVENT_PY_RETURN (*C macro*), 348
PY_MONITORING_EVENT_PY_START (*C macro*), 348
PY_MONITORING_EVENT_PY_THROW (*C macro*), 348
PY_MONITORING_EVENT_PY_UNWIND (*C macro*), 348
PY_MONITORING_EVENT_PY_YIELD (*C macro*), 348
PY_MONITORING_EVENT_RAISE (*C macro*), 348
PY_MONITORING_EVENT_RERAISE (*C macro*), 348
PY_MONITORING_EVENT_STOP_ITERATION (*C macro*), 348
PY_MONITORING_IS_INSTRUMENTED_EVENT (*C function*), 349
Py_NE (*C macro*), 318
Py_NewInterpreter (*C function*), 237
Py_NewInterpreterFromConfig (*C function*), 236
Py_NewRef (*C function*), 42
Py_NO_INLINE (*C macro*), 5
Py_None (*C var*), 134
Py_NoSiteFlag (*C var*), 220
Py_NotImplemented (*C var*), 98
Py_NoUserSiteDirectory (*C var*), 220
Py_OpenCodeHookFunction (*C type*), 192
Py_OptimizeFlag (*C var*), 220
Py_PACK_FULL_VERSION (*C function*), 342
Py_PACK_VERSION (*C function*), 342
Py_PreInitialize (*C function*), 257
Py_PreInitializeFromArgs (*C function*), 257
Py_PreInitializeFromBytesArgs (*C function*), 257
Py_PRINT_RAW (*C macro*), 98, 193
Py_QuietFlag (*C var*), 220
Py_READONLY (*C macro*), 295
Py_REFCNT (*C function*), 41
Py_RELATIVE_OFFSET (*C macro*), 295
PY_RELEASE_LEVEL (*C macro*), 341
PY_RELEASE_SERIAL (*C macro*), 341
Py_ReprEnter (*C function*), 57
Py_ReprLeave (*C function*), 57
Py_RETURN_FALSE (*C macro*), 144
Py_RETURN_NONE (*C macro*), 134
Py_RETURN_NOTIMPLEMENTED (*C macro*), 98
Py_RETURN_RICHCOMPARE (*C macro*), 318
Py_RETURN_TRUE (*C macro*), 144
Py_RunMain (*C function*), 223
Py_SET_REFCNT (*C function*), 41
Py_SET_SIZE (*C function*), 290
Py_SET_TYPE (*C function*), 290
Py_SetProgramName (*C function*), 223
Py_SetPythonHome (*C function*), 226
Py_SETREF (*C macro*), 43
Py_single_input (*C var*), 38
Py_SIZE (*C function*), 290
Py_ssize_t (*C type*), 10
PY_SSIZE_T_MAX (*C macro*), 137
Py_STRINGIFY (*C macro*), 5
Py_T_BOOL (*C macro*), 296
Py_T_BYTE (*C macro*), 296
Py_T_CHAR (*C macro*), 296
Py_T_DOUBLE (*C macro*), 296
Py_T_FLOAT (*C macro*), 296
Py_T_INT (*C macro*), 296
Py_T_LONG (*C macro*), 296
Py_T_LONGLONG (*C macro*), 296
Py_T_OBJECT_EX (*C macro*), 296
Py_T_PYSSIZET (*C macro*), 296
Py_T_SHORT (*C macro*), 296
Py_T_STRING (*C macro*), 296
Py_T_STRING_INPLACE (*C macro*), 296
Py_T_UBYTE (*C macro*), 296
Py_T_UINT (*C macro*), 296
Py_T_ULONG (*C macro*), 296
Py_T_ULONGLONG (*C macro*), 296
Py_T_USHORT (*C macro*), 296
Py_tp_token (*C macro*), 133
Py_TP_USE_SPEC (*C macro*), 134
Py_TPFLAGS_BASE_EXC_SUBCLASS (*C macro*), 313
Py_TPFLAGS_BASETYPE (*C macro*), 311
Py_TPFLAGS_BYTES_SUBCLASS (*C macro*), 313
Py_TPFLAGS_DEFAULT (*C macro*), 311
Py_TPFLAGS_DICT_SUBCLASS (*C macro*), 313
Py_TPFLAGS_DISALLOW_INSTANTIATION (*C macro*), 313
Py_TPFLAGS_HAVE_FINALIZE (*C macro*), 313
Py_TPFLAGS_HAVE_GC (*C macro*), 311
Py_TPFLAGS_HAVE_VECTORCALL (*C macro*), 313
Py_TPFLAGS_HEAPTYPE (*C macro*), 311
Py_TPFLAGS_IMMUTABLETYPE (*C macro*), 313
Py_TPFLAGS_ITEMS_AT_END (*C macro*), 312
Py_TPFLAGS_LIST_SUBCLASS (*C macro*), 312
Py_TPFLAGS_LONG_SUBCLASS (*C macro*), 312
Py_TPFLAGS_MANAGED_DICT (*C macro*), 312
Py_TPFLAGS_MANAGED_WEAKREF (*C macro*), 312
Py_TPFLAGS_MAPPING (*C macro*), 314
Py_TPFLAGS_METHOD_DESCRIPTOR (*C macro*), 312
Py_TPFLAGS_READY (*C macro*), 311
Py_TPFLAGS_READYING (*C macro*), 311

Py_TPFLAGS_SEQUENCE (*C macro*), 314
Py_TPFLAGS_TUPLE_SUBCLASS (*C macro*), 312
Py_TPFLAGS_TYPE_SUBCLASS (*C macro*), 313
Py_TPFLAGS_UNICODE_SUBCLASS (*C macro*), 313
Py_TPFLAGS_VALID_VERSION_TAG (*C macro*), 314
Py_tracefunc (*C type*), 239
Py_True (*C var*), 144
Py_tss_NEEDS_INIT (*C macro*), 242
Py_tss_t (*C type*), 242
Py_TYPE (*C function*), 290
Py_UCS1 (*C type*), 151
Py_UCS2 (*C type*), 151
Py_UCS4 (*C type*), 151
Py_uhash_t (*C type*), 89
Py_UNBLOCK_THREADS (*C macro*), 232
Py_UnbufferedStdioFlag (*C var*), 220
Py_UNICODE (*C type*), 171
Py_UNICODE_IS_HIGH_SURROGATE (*C function*), 154
Py_UNICODE_IS_LOW_SURROGATE (*C function*), 154
Py_UNICODE_IS_SURROGATE (*C function*), 154
Py_UNICODE_ISALNUM (*C function*), 153
Py_UNICODE_ISALPHA (*C function*), 153
Py_UNICODE_ISDECIMAL (*C function*), 153
Py_UNICODE_ISDIGIT (*C function*), 153
Py_UNICODE_ISLINEBREAK (*C function*), 153
Py_UNICODE_ISLOWER (*C function*), 153
Py_UNICODE_ISNUMERIC (*C function*), 153
Py_UNICODE_ISPRINTABLE (*C function*), 153
Py_UNICODE_ISSPACE (*C function*), 153
Py_UNICODE_ISTITLE (*C function*), 153
Py_UNICODE_ISUPPER (*C function*), 153
Py_UNICODE_JOIN_SURROGATES (*C function*), 154
Py_UNICODE_REPLACEMENT_CHARACTER (*C macro*), 162
Py_UNICODE_TODECIMAL (*C function*), 154
Py_UNICODE_TODIGIT (*C function*), 154
Py_UNICODE_TOLOWER (*C function*), 153
Py_UNICODE_TONUMERIC (*C function*), 154
Py_UNICODE_TOTITLE (*C function*), 153
Py_UNICODE_TOUPPER (*C function*), 153
Py_UNREACHABLE (*C macro*), 5
Py_UNUSED (*C macro*), 6
Py_VaBuildValue (*C function*), 87
PY_VECTORCALL_ARGUMENTS_OFFSET (*C macro*), 108
Py_VerboseFlag (*C var*), 220
Py_Version (*C var*), 341
PY_VERSION_HEX (*C macro*), 341
Py_VISIT (*C macro*), 339
Py_XDECREF (*C function*), 11, 42
Py_XINCREF (*C function*), 42
Py_XNewRef (*C function*), 42
Py_XSETREF (*C macro*), 43
PyAIter_Check (*C function*), 118
PyAnySet_Check (*C function*), 182
PyAnySet_CheckExact (*C function*), 182
PyArg_Parse (*C function*), 84
PyArg_ParseTuple (*C function*), 84
PyArg_ParseTupleAndKeywords (*C function*), 84
PyArg_UnpackTuple (*C function*), 85
PyArg_ValidateKeywordArguments (*C function*), 84
PyArg_VaParse (*C function*), 84
PyArg_VaParseTupleAndKeywords (*C function*), 84
PyASCIIObject (*C type*), 151
PyAsyncMethods (*C type*), 332
PyAsyncMethods.am_aiter (*C member*), 332
PyAsyncMethods.am_anext (*C member*), 332
PyAsyncMethods.am_await (*C member*), 332
PyAsyncMethods.am_send (*C member*), 332
PyBaseObject_Type (*C var*), 290
PyBool_Check (*C function*), 144
PyBool_FromLong (*C function*), 144
PyBool_Type (*C var*), 144
PyBUF_ANY_CONTIGUOUS (*C macro*), 122
PyBUF_C_CONTIGUOUS (*C macro*), 122
PyBUF_CONTIG (*C macro*), 123
PyBUF_CONTIG_RO (*C macro*), 123
PyBUF_F_CONTIGUOUS (*C macro*), 122
PyBUF_FORMAT (*C macro*), 121
PyBUF_FULL (*C macro*), 123
PyBUF_FULL_RO (*C macro*), 123
PyBUF_INDIRECT (*C macro*), 122
PyBUF_MAX_NDIM (*C macro*), 121
PyBUF_ND (*C macro*), 122
PyBUF_READ (*C macro*), 203
PyBUF_RECORDS (*C macro*), 123
PyBUF_RECORDS_RO (*C macro*), 123
PyBUF_SIMPLE (*C macro*), 122
PyBUF_STRIDED (*C macro*), 123
PyBUF_STRIDED_RO (*C macro*), 123
PyBUF_STRIDES (*C macro*), 122
PyBUF_WRITABLE (*C macro*), 121
PyBUF_WRITE (*C macro*), 203
PyBuffer_FillContiguousStrides (*C function*), 125
PyBuffer_FillInfo (*C function*), 125
PyBuffer_FromContiguous (*C function*), 125
PyBuffer_GetPointer (*C function*), 125
PyBuffer_IsContiguous (*C function*), 125
PyBuffer_Release (*C function*), 124
PyBuffer_SizeFromFormat (*C function*), 125
PyBuffer_ToContiguous (*C function*), 125
PyBufferProcs (*C type*), 119, 331
PyBufferProcs.bf_getbuffer (*C member*), 331
PyBufferProcs.bf_releasebuffer (*C member*), 331
PyByteArray_AS_STRING (*C function*), 151
PyByteArray_AsString (*C function*), 150
PyByteArray_Check (*C function*), 150
PyByteArray_CheckExact (*C function*), 150
PyByteArray_Concat (*C function*), 150
PyByteArray_FromObject (*C function*), 150
PyByteArray_FromStringAndSize (*C function*), 150
PyByteArray_GET_SIZE (*C function*), 151

PyByteArray_Resize (*C function*), 150
PyByteArray_Size (*C function*), 150
PyByteArray_Type (*C var*), 150
PyByteArrayObject (*C type*), 150
PyBytes_AS_STRING (*C function*), 149
PyBytes_AsString (*C function*), 149
PyBytes_AsStringAndSize (*C function*), 149
PyBytes_Check (*C function*), 148
PyBytes_CheckExact (*C function*), 148
PyBytes_Concat (*C function*), 149
PyBytes_ConcatAndDel (*C function*), 149
PyBytes_FromFormat (*C function*), 148
PyBytes_FromFormatV (*C function*), 149
PyBytes_FromObject (*C function*), 149
PyBytes_FromString (*C function*), 148
PyBytes_FromStringAndSize (*C function*), 148
PyBytes_GET_SIZE (*C function*), 149
PyBytes_Join (*C function*), 149
PyBytes_Size (*C function*), 149
PyBytes_Type (*C var*), 148
PyBytesObject (*C type*), 148
pyc utilisant le hachage, **358**
PyCallable_Check (*C function*), 111
PyCallIter_Check (*C function*), 201
PyCallIter_New (*C function*), 201
PyCallIter_Type (*C var*), 201
PyCapsule (*C type*), 205
PyCapsule_CheckExact (*C function*), 206
PyCapsule_Destructor (*C type*), 205
PyCapsule_GetContext (*C function*), 206
PyCapsule_GetDestructor (*C function*), 206
PyCapsule_GetName (*C function*), 206
PyCapsule_GetPointer (*C function*), 206
PyCapsule_Import (*C function*), 206
PyCapsule_IsValid (*C function*), 207
PyCapsule_New (*C function*), 206
PyCapsule_SetContext (*C function*), 207
PyCapsule_SetDestructor (*C function*), 207
PyCapsule_SetName (*C function*), 207
PyCapsule_SetPointer (*C function*), 207
PyCell_Check (*C function*), 186
PyCell_GET (*C function*), 186
PyCell_Get (*C function*), 186
PyCell_New (*C function*), 186
PyCell_SET (*C function*), 186
PyCell_Set (*C function*), 186
PyCell_Type (*C var*), 186
PyCellObject (*C type*), 186
PyCF_ALLOW_TOP_LEVEL_AWAIT (*C macro*), 38
PyCF_ONLY_AST (*C macro*), 38
PyCF_OPTIMIZED_AST (*C macro*), 38
PyCF_TYPE_COMMENTS (*C macro*), 38
PyCFunction (*C type*), 291
PyCFunction_New (*C function*), 294
PyCFunction_NewEx (*C function*), 294
PyCFunctionFast (*C type*), 291
PyCFunctionFastWithKeywords (*C type*), 291
PyCFunctionWithKeywords (*C type*), 291
PyCMethod (*C type*), 291
PyCMethod_New (*C function*), 293
PyCode_Addr2Line (*C function*), 188
PyCode_Addr2Location (*C function*), 188
PyCode_AddWatcher (*C function*), 189
PyCode_Check (*C function*), 187
PyCode_ClearWatcher (*C function*), 189
PyCode_GetCellvars (*C function*), 188
PyCode_GetCode (*C function*), 188
PyCode_GetFreevars (*C function*), 188
PyCode_GetNumFree (*C function*), 187
PyCode_GetVarnames (*C function*), 188
PyCode_New (*C function*), 187
PyCode_NewEmpty (*C function*), 188
PyCode_NewWithPosOnlyArgs (*C function*), 188
PyCode_Type (*C var*), 187
PyCode_WatchCallback (*C type*), 189
PyCodec_BackslashReplaceErrors (*C function*), 93
PyCodec_Decode (*C function*), 92
PyCodec_Decoder (*C function*), 92
PyCodec_Encode (*C function*), 92
PyCodec_Encoder (*C function*), 92
PyCodec_IgnoreErrors (*C function*), 93
PyCodec_IncrementalDecoder (*C function*), 92
PyCodec_IncrementalEncoder (*C function*), 92
PyCodec_KnownEncoding (*C function*), 92
PyCodec_LookupError (*C function*), 93
PyCodec_NameReplaceErrors (*C function*), 93
PyCodec_Register (*C function*), 92
PyCodec_RegisterError (*C function*), 93
PyCodec_ReplaceErrors (*C function*), 93
PyCodec_StreamReader (*C function*), 92
PyCodec_StreamWriter (*C function*), 93
PyCodec_StrictErrors (*C function*), 93
PyCodec_Unregister (*C function*), 92
PyCodec_XMLCharRefReplaceErrors (*C function*), 93
PyCodeEvent (*C type*), 189
PyCodeObject (*C type*), 187
PyCompactUnicodeObject (*C type*), 151
PyCompilerFlags (*C struct*), 38
PyCompilerFlags.cf_feature_version (*C member*), 38
PyCompilerFlags.cf_flags (*C member*), 38
PyComplex_AsCComplex (*C function*), 147
PyComplex_Check (*C function*), 147
PyComplex_CheckExact (*C function*), 147
PyComplex_FromCComplex (*C function*), 147
PyComplex_FromDoubles (*C function*), 147
PyComplex_ImagAsDouble (*C function*), 147
PyComplex_RealAsDouble (*C function*), 147
PyComplex_Type (*C var*), 147
PyComplexObject (*C type*), 147
PyConfig (*C type*), 257
PyConfig_Clear (*C function*), 258
PyConfig_Get (*C function*), 251
PyConfig_GetInt (*C function*), 252

PyConfig_InitIsolatedConfig (*C function*), 258
PyConfig_InitPythonConfig (*C function*), 257
PyConfig_Names (*C function*), 252
PyConfig._pystats (*C member*), 268
PyConfig_Read (*C function*), 258
PyConfig_Set (*C function*), 252
PyConfig_SetArgv (*C function*), 258
PyConfig_SetBytesArgv (*C function*), 258
PyConfig_SetBytesString (*C function*), 258
PyConfig_SetString (*C function*), 258
PyConfig_SetWideStringList (*C function*), 258
PyConfig.argv (*C member*), 259
PyConfig.base_exec_prefix (*C member*), 259
PyConfig.base_executable (*C member*), 259
PyConfig.base_prefix (*C member*), 259
PyConfig.buffered_stdio (*C member*), 259
PyConfig.bytes_warning (*C member*), 260
PyConfig.check_hash_pycs_mode (*C member*), 260
PyConfig.code_debug_ranges (*C member*), 260
PyConfig.configure_c_stdio (*C member*), 260
PyConfig.cpu_count (*C member*), 263
PyConfig.dev_mode (*C member*), 260
PyConfig.dump_refs (*C member*), 260
PyConfig.dump_refs_file (*C member*), 261
PyConfig.exec_prefix (*C member*), 261
PyConfig.executable (*C member*), 261
PyConfig.faulthandler (*C member*), 261
PyConfig.filesystem_encoding (*C member*), 261
PyConfig.filesystem_errors (*C member*), 261
PyConfig.hash_seed (*C member*), 262
PyConfig.home (*C member*), 262
PyConfig.import_time (*C member*), 262
PyConfig.inspect (*C member*), 262
PyConfig.install_signal_handlers (*C member*), 262
PyConfig.int_max_str_digits (*C member*), 262
PyConfig.interactive (*C member*), 262
PyConfig.isolated (*C member*), 263
PyConfig.legacy_windows_stdio (*C member*), 263
PyConfig.malloc_stats (*C member*), 263
PyConfig.module_search_paths (*C member*), 264
PyConfig.module_search_paths_set (*C member*), 264
PyConfig.optimization_level (*C member*), 264
PyConfig.orig_argv (*C member*), 264
PyConfig.parse_argv (*C member*), 264
PyConfig.parser_debug (*C member*), 264
PyConfig.pathconfig_warnings (*C member*), 265
PyConfig.perf_profiling (*C member*), 267
PyConfig.platlibdir (*C member*), 263
PyConfig.prefix (*C member*), 265
PyConfig.program_name (*C member*), 265
PyConfig.pycache_prefix (*C member*), 265
PyConfig.pythonpath_env (*C member*), 264
PyConfig.quiet (*C member*), 265
PyConfig.run_command (*C member*), 265
PyConfig.run_filename (*C member*), 265
PyConfig.run_module (*C member*), 265
PyConfig.run_presite (*C member*), 266
PyConfig.safe_path (*C member*), 259
PyConfig.show_ref_count (*C member*), 266
PyConfig.site_import (*C member*), 266
PyConfig.skip_source_first_line (*C member*), 266
PyConfig.stdio_encoding (*C member*), 266
PyConfig.stdio_errors (*C member*), 266
PyConfig.stdlib_dir (*C member*), 267
PyConfig.tracemalloc (*C member*), 266
PyConfig.use_environment (*C member*), 267
PyConfig.use_frozen_modules (*C member*), 262
PyConfig.use_hash_seed (*C member*), 262
PyConfig.use_system_logger (*C member*), 267
PyConfig.user_site_directory (*C member*), 267
PyConfig.verbose (*C member*), 267
PyConfig.warn_default_encoding (*C member*), 260
PyConfig.warnoptions (*C member*), 268
PyConfig.write_bytecode (*C member*), 268
PyConfig.xoptions (*C member*), 268
PyContext (*C type*), 210
PyContext_AddWatcher (*C function*), 211
PyContext_CheckExact (*C function*), 211
PyContext_ClearWatcher (*C function*), 211
PyContext_Copy (*C function*), 211
PyContext_CopyCurrent (*C function*), 211
PyContext_Enter (*C function*), 211
PyContext_Exit (*C function*), 211
PyContext_New (*C function*), 211
PyContext_Type (*C var*), 211
PyContext_WatchCallback (*C type*), 212
PyContextEvent (*C type*), 211
PyContextToken (*C type*), 211
PyContextToken_CheckExact (*C function*), 211
PyContextToken_Type (*C var*), 211
PyContextVar (*C type*), 211
PyContextVar_CheckExact (*C function*), 211
PyContextVar_Get (*C function*), 212
PyContextVar_New (*C function*), 212
PyContextVar_Reset (*C function*), 212
PyContextVar_Set (*C function*), 212
PyContextVar_Type (*C var*), 211
PyCoro_CheckExact (*C function*), 210
PyCoro_New (*C function*), 210
PyCoro_Type (*C var*), 210
PyCoroObject (*C type*), 210
PyDate_Check (*C function*), 213
PyDate_CheckExact (*C function*), 213
PyDate_FromDate (*C function*), 214
PyDate_FromTimestamp (*C function*), 216
PyDateTime_Check (*C function*), 213
PyDateTime_CheckExact (*C function*), 213
PyDateTime_Date (*C type*), 212
PyDateTime_DATE_GET_FOLD (*C function*), 215
PyDateTime_DATE_GET_HOUR (*C function*), 214

PyDateTime_DATE_GET_MICROSECOND (*C function*), 215
PyDateTime_DATE_GET_MINUTE (*C function*), 215
PyDateTime_DATE_GET_SECOND (*C function*), 215
PyDateTime_DATE_GET_TZINFO (*C function*), 215
PyDateTime_DateTime (*C type*), 212
PyDateTime_DateTimeType (*C var*), 213
PyDateTime_DateType (*C var*), 212
PyDateTime_Delta (*C type*), 212
PyDateTime_DELTA_GET_DAYS (*C function*), 215
PyDateTime_DELTA_GET_MICROSECONDS (*C function*), 215
PyDateTime_DELTA_GET_SECONDS (*C function*), 215
PyDateTime_DeltaType (*C var*), 213
PyDateTime_FromDateAndTime (*C function*), 214
PyDateTime_FromDateAndTimeAndFold (*C function*), 214
PyDateTime_FromTimestamp (*C function*), 215
PyDateTime_GET_DAY (*C function*), 214
PyDateTime_GET_MONTH (*C function*), 214
PyDateTime_GET_YEAR (*C function*), 214
PyDateTime_Time (*C type*), 212
PyDateTime_TIME_GET_FOLD (*C function*), 215
PyDateTime_TIME_GET_HOUR (*C function*), 215
PyDateTime_TIME_GET_MICROSECOND (*C function*), 215
PyDateTime_TIME_GET_MINUTE (*C function*), 215
PyDateTime_TIME_GET_SECOND (*C function*), 215
PyDateTime_TIME_GET_TZINFO (*C function*), 215
PyDateTime_TimeType (*C var*), 213
PyDateTime_TimeZone_UTC (*C var*), 213
PyDateTime_TZInfoType (*C var*), 213
PyDelta_Check (*C function*), 213
PyDelta_CheckExact (*C function*), 213
PyDelta_FromDSU (*C function*), 214
PyDescr_IsData (*C function*), 201
PyDescr_NewClassMethod (*C function*), 201
PyDescr_NewGetSet (*C function*), 201
PyDescr_NewMember (*C function*), 201
PyDescr_NewMethod (*C function*), 201
PyDescr_NewWrapper (*C function*), 201
PyDict_AddWatcher (*C function*), 180
PyDict_Check (*C function*), 176
PyDict_CheckExact (*C function*), 176
PyDict_Clear (*C function*), 177
PyDict_ClearWatcher (*C function*), 180
PyDict_Contains (*C function*), 177
PyDict_ContainsString (*C function*), 177
PyDict_Copy (*C function*), 177
PyDict_DelItem (*C function*), 177
PyDict_DelItemString (*C function*), 177
PyDict_GetItem (*C function*), 177
PyDict_GetItemRef (*C function*), 177
PyDict_GetItemString (*C function*), 178
PyDict_GetItemStringRef (*C function*), 178
PyDict_GetItemWithError (*C function*), 178
PyDict_Items (*C function*), 178
PyDict_Keys (*C function*), 179
PyDict_Merge (*C function*), 180
PyDict_MergeFromSeq2 (*C function*), 180
PyDict_New (*C function*), 176
PyDict_Next (*C function*), 179
PyDict_Pop (*C function*), 178
PyDict_PopString (*C function*), 178
PyDict_SetDefault (*C function*), 178
PyDict_SetDefaultRef (*C function*), 178
PyDict_SetItem (*C function*), 177
PyDict_SetItemString (*C function*), 177
PyDict_Size (*C function*), 179
PyDict_Type (*C var*), 176
PyDict_Unwatch (*C function*), 180
PyDict_Update (*C function*), 180
PyDict_Values (*C function*), 179
PyDict_Watch (*C function*), 180
PyDict_WatchCallback (*C type*), 181
PyDict_WatchEvent (*C type*), 180
PyDictObject (*C type*), 176
PyDictProxy_New (*C function*), 177
PyDoc_STR (*C macro*), 6
PyDoc_STRVAR (*C macro*), 6
PyEllipsis_Type (*C var*), 203
PyErr_BadArgument (*C function*), 47
PyErr_BadInternalCall (*C function*), 48
PyErr_CheckSignals (*C function*), 53
PyErr_Clear (*C function*), 10, 11, 45
PyErr_DisplayException (*C function*), 46
PyErr_ExceptionMatches (*C function*), 11, 50
PyErr_Fetch (*C function*), 50
PyErr_Format (*C function*), 46
PyErr_FormatUnraisable (*C function*), 46
PyErr_FormatV (*C function*), 46
PyErr_GetExcInfo (*C function*), 52
PyErr_GetHandledException (*C function*), 51
PyErr_GetRaisedException (*C function*), 50
PyErr_GivenExceptionMatches (*C function*), 50
PyErr_NewException (*C function*), 54
PyErr_NewExceptionWithDoc (*C function*), 54
PyErr_NoMemory (*C function*), 47
PyErr_NormalizeException (*C function*), 51
PyErr_Occurred (*C function*), 10, 49
PyErr_Print (*C function*), 46
PyErr_PrintEx (*C function*), 45
PyErr_ResourceWarning (*C function*), 49
PyErr_Restore (*C function*), 51
PyErr_SetExcFromWindowsErr (*C function*), 47
PyErr_SetExcFromWindowsErrWithFilename (*C function*), 48
PyErr_SetExcFromWindowsErrWithFilenameObject (*C function*), 48
PyErr_SetExcFromWindowsErrWithFilenameObjects (*C function*), 48
PyErr_SetExcInfo (*C function*), 52
PyErr_SetFromErrno (*C function*), 47
PyErr_SetFromErrnoWithFilename (*C function*), 47

PyErr_SetFromErrnoWithFilenameObject (*C function*), 47
PyErr_SetFromErrnoWithFilenameObjects (*C function*), 47
PyErr_SetFromWindowsErr (*C function*), 47
PyErr_SetFromWindowsErrWithFilename (*C function*), 47
PyErr_SetHandledException (*C function*), 52
PyErr_SetImportError (*C function*), 48
PyErr_SetImportErrorSubclass (*C function*), 48
PyErr_SetInterrupt (*C function*), 53
PyErr_SetInterruptEx (*C function*), 53
PyErr_SetNone (*C function*), 47
PyErr_SetObject (*C function*), 46
PyErr_SetRaisedException (*C function*), 50
PyErr_SetString (*C function*), 10, 46
PyErr_SyntaxLocation (*C function*), 48
PyErr_SyntaxLocationEx (*C function*), 48
PyErr_SyntaxLocationObject (*C function*), 48
PyErr_WarnEx (*C function*), 49
PyErr_WarnExplicit (*C function*), 49
PyErr_WarnExplicitObject (*C function*), 49
PyErr_WarnFormat (*C function*), 49
PyErr_WriteUnraisable (*C function*), 46
PyEval_AcquireThread (*C function*), 234
PyEval_AcquireThread(), 230
PyEval_EvalCode (*C function*), 37
PyEval_EvalCodeEx (*C function*), 38
PyEval_EvalFrame (*C function*), 38
PyEval_EvalFrameEx (*C function*), 38
PyEval_GetBuiltins (*C function*), 91
PyEval_GetFrame (*C function*), 91
PyEval_GetFrameBuiltins (*C function*), 91
PyEval_GetFrameGlobals (*C function*), 91
PyEval_GetFrameLocals (*C function*), 91
PyEval_GetFuncDesc (*C function*), 92
PyEval_GetFuncName (*C function*), 92
PyEval_GetGlobals (*C function*), 91
PyEval_GetLocals (*C function*), 91
PyEval_InitThreads (*C function*), 230
PyEval_InitThreads(), 221
PyEval_MergeCompilerFlags (*C function*), 38
PyEval_ReleaseThread (*C function*), 235
PyEval_ReleaseThread(), 230
PyEval_RestoreThread (*C function*), 227, 230
PyEval_RestoreThread(), 230
PyEval_SaveThread (*C function*), 227, 230
PyEval_SaveThread(), 230
PyEval_SetProfile (*C function*), 240
PyEval_SetProfileAllThreads (*C function*), 240
PyEval_SetTrace (*C function*), 240
PyEval_SetTraceAllThreads (*C function*), 240
PyExc_ArithmeticError (*C var*), 57
PyExc_AssertionError (*C var*), 57
PyExc_AttributeError (*C var*), 58
PyExc_BaseException (*C var*), 57
PyExc_BaseExceptionGroup (*C var*), 57
PyExc_BlockingIOError (*C var*), 58
PyExc_BrokenPipeError (*C var*), 58
PyExc_BufferError (*C var*), 58
PyExc_BytesWarning (*C var*), 63
PyExc_ChildProcessError (*C var*), 58
PyExc_ConnectionAbortedError (*C var*), 58
PyExc_ConnectionError (*C var*), 58
PyExc_ConnectionRefusedError (*C var*), 58
PyExc_ConnectionResetError (*C var*), 58
PyExc_DeprecationWarning (*C var*), 63
PyExc_EncodingWarning (*C var*), 63
PyExc_EnvironmentError (*C var*), 62
PyExc_EOFError (*C var*), 58
PyExc_Exception (*C var*), 57
PyExc_FileExistsError (*C var*), 58
PyExc_FileNotFoundError (*C var*), 58
PyExc_FloatingPointError (*C var*), 58
PyExc_FutureWarning (*C var*), 63
PyExc_GeneratorExit (*C var*), 59
PyExc_ImportError (*C var*), 59
PyExc_ImportWarning (*C var*), 63
PyExc_IndentationError (*C var*), 59
PyExc_IndexError (*C var*), 59
PyExc_InterruptedError (*C var*), 59
PyExc_IOError (*C var*), 62
PyExc_IsADirectoryError (*C var*), 59
PyExc_KeyboardInterrupt (*C var*), 59
PyExc_KeyError (*C var*), 59
PyExc_LookupError (*C var*), 59
PyExc_MemoryError (*C var*), 59
PyExc_ModuleNotFoundError (*C var*), 59
PyExc_NameError (*C var*), 59
PyExc_NotADirectoryError (*C var*), 59
PyExc_NotImplementedError (*C var*), 60
PyExc_OSError (*C var*), 60
PyExc_OverflowError (*C var*), 60
PyExc_PendingDeprecationWarning (*C var*), 63
PyExc_PermissionError (*C var*), 60
PyExc_ProcessLookupError (*C var*), 60
PyExc_PythonFinalizationError (*C var*), 60
PyExc_RecursionError (*C var*), 60
PyExc_ReferenceError (*C var*), 60
PyExc_ResourceWarning (*C var*), 63
PyExc_RuntimeError (*C var*), 60
PyExc_RuntimeWarning (*C var*), 63
PyExc_StopAsyncIteration (*C var*), 60
PyExc_StopIteration (*C var*), 60
PyExc_SyntaxError (*C var*), 60
PyExc_SyntaxWarning (*C var*), 63
PyExc_SystemError (*C var*), 60
PyExc_SystemExit (*C var*), 60
PyExc_TabError (*C var*), 61
PyExc_TimeoutError (*C var*), 61
PyExc_TypeError (*C var*), 61
PyExc_UnboundLocalError (*C var*), 61
PyExc_UnicodeDecodeError (*C var*), 61
PyExc_UnicodeEncodeError (*C var*), 61
PyExc_UnicodeError (*C var*), 61
PyExc_UnicodeTranslateError (*C var*), 61

PyExc_UnicodeWarning (*C var*), 63
PyExc_UserWarning (*C var*), 63
PyExc_ValueError (*C var*), 61
PyExc_Warning (*C var*), 63
PyExc_WindowsError (*C var*), 62
PyExc_ZeroDivisionError (*C var*), 61
PyException_GetArgs (*C function*), 54
PyException_GetCause (*C function*), 54
PyException_GetContext (*C function*), 54
PyException_GetTraceback (*C function*), 54
PyException_SetArgs (*C function*), 55
PyException_SetCause (*C function*), 54
PyException_SetContext (*C function*), 54
PyException_SetTraceback (*C function*), 54
PyExceptionClass_Check (*C function*), 54
PyExceptionClass_Name (*C function*), 54
PyFile_FromFd (*C function*), 192
PyFile_GetLine (*C function*), 192
PyFile_SetOpenCodeHook (*C function*), 192
PyFile_WriteObject (*C function*), 193
PyFile_WriteString (*C function*), 193
PyFloat_AS_DOUBLE (*C function*), 145
PyFloat_AsDouble (*C function*), 144
PyFloat_Check (*C function*), 144
PyFloat_CheckExact (*C function*), 144
PyFloat_FromDouble (*C function*), 144
PyFloat_FromString (*C function*), 144
PyFloat_GetInfo (*C function*), 145
PyFloat_GetMax (*C function*), 145
PyFloat_GetMin (*C function*), 145
PyFloat_Pack2 (*C function*), 145
PyFloat_Pack4 (*C function*), 145
PyFloat_Pack8 (*C function*), 145
PyFloat_Type (*C var*), 144
PyFloat_Unpack2 (*C function*), 146
PyFloat_Unpack4 (*C function*), 146
PyFloat_Unpack8 (*C function*), 146
PyFloatObject (*C type*), 144
PyFrame_Check (*C function*), 207
PyFrame_GetBack (*C function*), 207
PyFrame_GetBuiltins (*C function*), 207
PyFrame_GetCode (*C function*), 208
PyFrame_GetGenerator (*C function*), 208
PyFrame_GetGlobals (*C function*), 208
PyFrame_GetLasti (*C function*), 208
PyFrame_GetLineNumber (*C function*), 208
PyFrame_GetLocals (*C function*), 208
PyFrame_GetVar (*C function*), 208
PyFrame_GetVarString (*C function*), 208
PyFrame_Type (*C var*), 207
PyFrameLocalsProxy_Check (*C function*), 209
PyFrameLocalsProxy_Type (*C var*), 209
PyFrameObject (*C type*), 207
PyFrozenSet_Check (*C function*), 182
PyFrozenSet_CheckExact (*C function*), 182
PyFrozenSet_New (*C function*), 182
PyFrozenSet_Type (*C var*), 181
PyFunction_AddWatcher (*C function*), 184
PyFunction_Check (*C function*), 183
PyFunction_ClearWatcher (*C function*), 184
PyFunction_GET_ANNOTATIONS (*C function*), 184
PyFunction_GET_CLOSURE (*C function*), 184
PyFunction_GET_CODE (*C function*), 184
PyFunction_GET_DEFAULTS (*C function*), 184
PyFunction_GET_GLOBALS (*C function*), 184
PyFunction_GET_KW_DEFAULTS (*C function*), 184
PyFunction_GET_MODULE (*C function*), 184
PyFunction_GetAnnotations (*C function*), 184
PyFunction_GetClosure (*C function*), 184
PyFunction_GetCode (*C function*), 183
PyFunction_GetDefaults (*C function*), 183
PyFunction_GetGlobals (*C function*), 183
PyFunction_GetKwDefaults (*C function*), 184
PyFunction_GetModule (*C function*), 183
PyFunction_New (*C function*), 183
PyFunction_NewWithQualName (*C function*), 183
PyFunction_SetAnnotations (*C function*), 184
PyFunction_SetClosure (*C function*), 184
PyFunction_SetDefaults (*C function*), 183
PyFunction_SetVectorcall (*C function*), 183
PyFunction_Type (*C var*), 183
PyFunction_WatchCallback (*C type*), 185
PyFunction_WatchEvent (*C type*), 184
PyFunctionObject (*C type*), 183
PyGC_Collect (*C function*), 339
PyGC_Disable (*C function*), 340
PyGC_Enable (*C function*), 340
PyGC_IsEnabled (*C function*), 340
PyGen_Check (*C function*), 210
PyGen_CheckExact (*C function*), 210
PyGen_New (*C function*), 210
PyGen_NewWithQualName (*C function*), 210
PyGen_Type (*C var*), 210
PyGenObject (*C type*), 209
PyGetSetDef (*C type*), 297
PyGetSetDef.closure (*C member*), 297
PyGetSetDef.doc (*C member*), 297
PyGetSetDef.get (*C member*), 297
PyGetSetDef.name (*C member*), 297
PyGetSetDef.set (*C member*), 297
PyGILState_Check (*C function*), 231
PyGILState_Ensure (*C function*), 231
PyGILState_GetThisThreadState (*C function*), 231
PyGILState_Release (*C function*), 231
PyHASH_BITS (*C macro*), 89
PyHash_FuncDef (*C type*), 90
PyHash_FuncDef.hash (*C member*), 90
PyHash_FuncDef.hash_bits (*C member*), 90
PyHash_FuncDef.name (*C member*), 90
PyHash_FuncDef.seed_bits (*C member*), 90
PyHash_GetFuncDef (*C function*), 90
PyHASH_IMAG (*C macro*), 89
PyHASH_INF (*C macro*), 89
PyHASH_MODULUS (*C macro*), 89
PyHASH_MULTIPLIER (*C macro*), 89

PyImport_AddModule (*C function*), 75
PyImport_AddModuleObject (*C function*), 75
PyImport_AddModuleRef (*C function*), 75
PyImport_AppendInittab (*C function*), 77
PyImport_ExecCodeModule (*C function*), 76
PyImport_ExecCodeModuleEx (*C function*), 76
PyImport_ExecCodeModuleObject (*C function*), 76
PyImport_ExecCodeModuleWithPathnames (*C function*), 76
PyImport_ExtendInittab (*C function*), 78
PyImport_FrozenModules (*C var*), 77
PyImport_GetImporter (*C function*), 77
PyImport_GetMagicNumber (*C function*), 76
PyImport_GetMagicTag (*C function*), 77
PyImport_GetModule (*C function*), 77
PyImport_GetModuleDict (*C function*), 77
PyImport_Import (*C function*), 75
PyImport_ImportFrozenModule (*C function*), 77
PyImport_ImportFrozenModuleObject (*C function*), 77
PyImport_ImportModule (*C function*), 74
PyImport_ImportModuleAttr (*C function*), 78
PyImport_ImportModuleAttrString (*C function*), 78
PyImport_ImportModuleEx (*C function*), 74
PyImport_ImportModuleLevel (*C function*), 75
PyImport_ImportModuleLevelObject (*C function*), 75
PyImport_ImportModuleNoBlock (*C function*), 74
PyImport_ReloadModule (*C function*), 75
PyIndex_Check (*C function*), 114
PyInit_modulename (*C function*), 66
PyInitConfig (*C struct*), 248
PyInitConfig_AddModule (*C function*), 249
PyInitConfig_Create (*C function*), 248
PyInitConfig_Free (*C function*), 248
PyInitConfig_FreeStrList (*C function*), 249
PyInitConfig_GetError (*C function*), 248
PyInitConfig_GetExitCode (*C function*), 248
PyInitConfig_GetInt (*C function*), 249
PyInitConfig_GetStr (*C function*), 249
PyInitConfig_GetStrList (*C function*), 249
PyInitConfig_HasOption (*C function*), 249
PyInitConfig_SetInt (*C function*), 249
PyInitConfig_SetStr (*C function*), 249
PyInitConfig_SetStrList (*C function*), 249
PyInstanceMethod_Check (*C function*), 185
PyInstanceMethod_Function (*C function*), 185
PyInstanceMethod_GET_FUNCTION (*C function*), 185
PyInstanceMethod_New (*C function*), 185
PyInstanceMethod_Type (*C var*), 185
PyInterpreterConfig (*C type*), 235
PyInterpreterConfig_DEFAULT_GIL (*C macro*), 236
PyInterpreterConfig_OWN_GIL (*C macro*), 236
PyInterpreterConfig_SHARED_GIL (*C macro*), 236
PyInterpreterConfig.allow_daemon_threads (*C member*), 236
PyInterpreterConfig.allow_exec (*C member*), 235
PyInterpreterConfig.allow_fork (*C member*), 235
PyInterpreterConfig.allow_threads (*C member*), 235
PyInterpreterConfig.check_multi_interp_extensions (*C member*), 236
PyInterpreterConfig.gil (*C member*), 236
PyInterpreterConfig.use_main_obmalloc (*C member*), 235
PyInterpreterState (*C type*), 229
PyInterpreterState_Clear (*C function*), 232
PyInterpreterState_Delete (*C function*), 232
PyInterpreterState_Get (*C function*), 233
PyInterpreterState_GetDict (*C function*), 234
PyInterpreterState_GetID (*C function*), 233
PyInterpreterState_Head (*C function*), 241
PyInterpreterState_Main (*C function*), 241
PyInterpreterState_New (*C function*), 232
PyInterpreterState_Next (*C function*), 241
PyInterpreterState_ThreadHead (*C function*), 241
PyIter_Check (*C function*), 118
PyIter_Next (*C function*), 118
PyIter_NextItem (*C function*), 118
PyIter_Send (*C function*), 118
PyList_Append (*C function*), 176
PyList_AsTuple (*C function*), 176
PyList_Check (*C function*), 174
PyList_CheckExact (*C function*), 174
PyList_Clear (*C function*), 176
PyList_Extend (*C function*), 176
PyList_GET_ITEM (*C function*), 175
PyList_GET_SIZE (*C function*), 175
PyList_GetItem (*C function*), 9, 175
PyList_GetItemRef (*C function*), 175
PyList_GetSlice (*C function*), 176
PyList_Insert (*C function*), 175
PyList_New (*C function*), 175
PyList_Reverse (*C function*), 176
PyList_SET_ITEM (*C function*), 175
PyList_SetItem (*C function*), 7, 175
PyList_SetSlice (*C function*), 176
PyList_Size (*C function*), 175
PyList_Sort (*C function*), 176
PyList_Type (*C var*), 174
PyListObject (*C type*), 174
PyLong_AS_LONG (*C function*), 136
PyLong_AsDouble (*C function*), 138
PyLong_AsInt (*C function*), 136
PyLong_AsInt32 (*C function*), 138
PyLong_AsInt64 (*C function*), 138
PyLong_AsLong (*C function*), 136
PyLong_AsLongAndOverflow (*C function*), 136
PyLong_AsLongLong (*C function*), 136

PyLong_AsLongLongAndOverflow (*C function*), 137
PyLong_AsNativeBytes (*C function*), 138
PyLong_AsSize_t (*C function*), 137
PyLong_AsSsize_t (*C function*), 137
PyLong_AsUInt32 (*C function*), 138
PyLong_AsUInt64 (*C function*), 138
PyLong_AsUnsignedLong (*C function*), 137
PyLong_AsUnsignedLongLong (*C function*), 137
PyLong_AsUnsignedLongLongMask (*C function*), 137
PyLong_AsUnsignedLongMask (*C function*), 137
PyLong_AsVoidPtr (*C function*), 138
PyLong_Check (*C function*), 134
PyLong_CheckExact (*C function*), 134
PyLong_Export (*C function*), 143
PyLong_FreeExport (*C function*), 143
PyLong_FromDouble (*C function*), 135
PyLong_FromInt32 (*C function*), 135
PyLong_FromInt64 (*C function*), 135
PyLong_FromLong (*C function*), 134
PyLong_FromLongLong (*C function*), 135
PyLong_FromNativeBytes (*C function*), 135
PyLong_FromSize_t (*C function*), 135
PyLong_FromSsize_t (*C function*), 134
PyLong_FromString (*C function*), 135
PyLong_FromUInt32 (*C function*), 135
PyLong_FromUInt64 (*C function*), 135
PyLong_FromUnicodeObject (*C function*), 135
PyLong_FromUnsignedLong (*C function*), 134
PyLong_FromUnsignedLongLong (*C function*), 135
PyLong_FromUnsignedNativeBytes (*C function*), 136
PyLong_FromVoidPtr (*C function*), 135
PyLong_GetInfo (*C function*), 141
PyLong_GetNativeLayout (*C function*), 142
PyLong_GetSign (*C function*), 141
PyLong_IsNegative (*C function*), 141
PyLong_IsPositive (*C function*), 141
PyLong_IsZero (*C function*), 141
PyLong_Type (*C var*), 134
PyLongExport (*C struct*), 142
PyLongExport.digits (*C member*), 143
PyLongExport.ndigits (*C member*), 142
PyLongExport.negative (*C member*), 142
PyLongExport.value (*C member*), 142
PyLongLayout (*C struct*), 142
PyLongLayout.bits_per_digit (*C member*), 142
PyLongLayout.digit_endianness (*C member*), 142
PyLongLayout.digit_size (*C member*), 142
PyLongLayout.digits_order (*C member*), 142
PyLongObject (*C type*), 134
PyLongWriter (*C struct*), 143
PyLongWriter_Create (*C function*), 143
PyLongWriter_Discard (*C function*), 143
PyLongWriter_Finish (*C function*), 143
PyMapping_Check (*C function*), 116
PyMapping_DelItem (*C function*), 117
PyMapping_DelItemString (*C function*), 117
PyMapping_GetItemString (*C function*), 117
PyMapping_GetOptionalItem (*C function*), 117
PyMapping_GetOptionalItemString (*C function*), 117
PyMapping_HasKey (*C function*), 117
PyMapping_HasKeyString (*C function*), 117
PyMapping_HasKeyStringWithError (*C function*), 117
PyMapping_HasKeyWithError (*C function*), 117
PyMapping_Items (*C function*), 118
PyMapping_Keys (*C function*), 118
PyMapping_Length (*C function*), 116
PyMapping_SetItemString (*C function*), 117
PyMapping_Size (*C function*), 116
PyMapping_Values (*C function*), 118
PyMappingMethods (*C type*), 330
PyMappingMethods.mp_ass_subscript (*C member*), 330
PyMappingMethods.mp_length (*C member*), 330
PyMappingMethods.mp_subscript (*C member*), 330
PyMarshal_ReadLastObjectFromFile (*C function*), 79
PyMarshal_ReadLongFromFile (*C function*), 78
PyMarshal_ReadObjectFromFile (*C function*), 79
PyMarshal_ReadObjectFromString (*C function*), 79
PyMarshal_ReadShortFromFile (*C function*), 79
PyMarshal_WriteLongToFile (*C function*), 78
PyMarshal_WriteObjectToFile (*C function*), 78
PyMarshal_WriteObjectToString (*C function*), 78
PyMem_Calloc (*C function*), 275
PyMem_Del (*C function*), 276
PYMEM_DOMAIN_MEM (*C macro*), 278
PYMEM_DOMAIN_OBJ (*C macro*), 279
PYMEM_DOMAIN_RAW (*C macro*), 278
PyMem_Free (*C function*), 276
PyMem_GetAllocator (*C function*), 279
PyMem_Malloc (*C function*), 275
PyMem_New (*C macro*), 276
PyMem_RawCalloc (*C function*), 275
PyMem_RawFree (*C function*), 275
PyMem_RawMalloc (*C function*), 274
PyMem_RawRealloc (*C function*), 275
PyMem_Realloc (*C function*), 276
PyMem_Resize (*C macro*), 276
PyMem_SetAllocator (*C function*), 279
PyMem_SetupDebugHooks (*C function*), 279
PyMemAllocatorDomain (*C type*), 278
PyMemAllocatorEx (*C type*), 278
PyMember_GetOne (*C function*), 295
PyMember_SetOne (*C function*), 295
PyMemberDef (*C type*), 294
PyMemberDef.doc (*C member*), 294
PyMemberDef.flags (*C member*), 294
PyMemberDef.name (*C member*), 294
PyMemberDef.offset (*C member*), 294

PyMemberDef.type (*C member*), 294
PyMemoryView_Check (*C function*), 204
PyMemoryView_FromBuffer (*C function*), 203
PyMemoryView_FromMemory (*C function*), 203
PyMemoryView_FromObject (*C function*), 203
PyMemoryView_GET_BASE (*C function*), 204
PyMemoryView_GET_BUFFER (*C function*), 204
PyMemoryView_GetContiguous (*C function*), 203
PyMethod_Check (*C function*), 186
PyMethod_Function (*C function*), 186
PyMethod_GET_FUNCTION (*C function*), 186
PyMethod_GET_SELF (*C function*), 186
PyMethod_New (*C function*), 186
PyMethod_Self (*C function*), 186
PyMethod_Type (*C var*), 185
PyMethodDef (*C type*), 292
PyMethodDef.ml_doc (*C member*), 292
PyMethodDef.ml_flags (*C member*), 292
PyMethodDef.ml_meth (*C member*), 292
PyMethodDef.ml_name (*C member*), 292
PyMODINIT_FUNC (*C macro*), 66
PyModule_Add (*C function*), 198
PyModule_AddFunctions (*C function*), 199
PyModule_AddIntConstant (*C function*), 199
PyModule_AddIntMacro (*C macro*), 199
PyModule_AddObject (*C function*), 199
PyModule_AddObjectRef (*C function*), 198
PyModule_AddStringConstant (*C function*), 199
PyModule_AddStringMacro (*C macro*), 199
PyModule_AddType (*C function*), 199
PyModule_Check (*C function*), 193
PyModule_CheckExact (*C function*), 193
PyModule_Create (*C function*), 197
PyModule_Create2 (*C function*), 197
PyModule_ExecDef (*C function*), 197
PyModule_FromDefAndSpec (*C function*), 197
PyModule_FromDefAndSpec2 (*C function*), 197
PyModule_GetDef (*C function*), 193
PyModule_GetDict (*C function*), 193
PyModule_GetFilename (*C function*), 194
PyModule_GetFilenameObject (*C function*), 193
PyModule_GetName (*C function*), 193
PyModule_GetNameObject (*C function*), 193
PyModule_GetState (*C function*), 193
PyModule_New (*C function*), 193
PyModule_NewObject (*C function*), 193
PyModule_SetDocString (*C function*), 200
PyModule_Type (*C var*), 193
PyModuleDef (*C type*), 194
PyModuleDef_Init (*C function*), 67
PyModuleDef_Slot (*C type*), 195
PyModuleDef_Slot.slot (*C member*), 195
PyModuleDef_Slot.value (*C member*), 195
PyModuleDef.m_base (*C member*), 194
PyModuleDef.m_clear (*C member*), 195
PyModuleDef.m_doc (*C member*), 194
PyModuleDef.m_free (*C member*), 195
PyModuleDef.m_methods (*C member*), 194
PyModuleDef.m_name (*C member*), 194
PyModuleDef.m_size (*C member*), 194
PyModuleDef.m_slots (*C member*), 194
PyModuleDef.m_slots.m_reload (*C member*), 195
PyModuleDef.m_traverse (*C member*), 195
PyMonitoring_EnterScope (*C function*), 346
PyMonitoring_ExitScope (*C function*), 348
PyMonitoring_FireBranchLeftEvent (*C function*), 346
PyMonitoring_FireBranchRightEvent (*C function*), 346
PyMonitoring_FireCallEvent (*C function*), 345
PyMonitoring_FireCRaiseEvent (*C function*), 346
PyMonitoring_FireCReturnEvent (*C function*), 346
PyMonitoring_FireExceptionHandledEvent (*C function*), 346
PyMonitoring_FireJumpEvent (*C function*), 345
PyMonitoring_FireLineEvent (*C function*), 345
PyMonitoring_FirePyResumeEvent (*C function*), 345
PyMonitoring_FirePyReturnEvent (*C function*), 345
PyMonitoring_FirePyStartEvent (*C function*), 345
PyMonitoring_FirePyThrowEvent (*C function*), 346
PyMonitoring_FirePyUnwindEvent (*C function*), 346
PyMonitoring_FirePyYieldEvent (*C function*), 345
PyMonitoring_FireRaiseEvent (*C function*), 346
PyMonitoring_FireReraiseEvent (*C function*), 346
PyMonitoring_FireStopIterationEvent (*C function*), 346
PyMonitoringState (*C type*), 345
PyMutex (*C type*), 243
PyMutex_IsLocked (*C function*), 244
PyMutex_Lock (*C function*), 244
PyMutex_Unlock (*C function*), 244
PyNumber_Absolute (*C function*), 112
PyNumber_Add (*C function*), 111
PyNumber_And (*C function*), 112
PyNumber_AsSsize_t (*C function*), 114
PyNumber_Check (*C function*), 111
PyNumber_Divmod (*C function*), 112
PyNumber_Float (*C function*), 114
PyNumber_FloorDivide (*C function*), 112
PyNumber_Index (*C function*), 114
PyNumber_InPlaceAdd (*C function*), 113
PyNumber_InPlaceAnd (*C function*), 114
PyNumber_InPlaceFloorDivide (*C function*), 113
PyNumber_InPlaceLshift (*C function*), 113
PyNumber_InPlaceMatrixMultiply (*C function*), 113
PyNumber_InPlaceMultiply (*C function*), 113
PyNumber_InPlaceOr (*C function*), 114

PyNumber_InPlacePower (*C function*), 113
PyNumber_InPlaceRemainder (*C function*), 113
PyNumber_InPlaceRshift (*C function*), 113
PyNumber_InPlaceSubtract (*C function*), 113
PyNumber_InPlaceTrueDivide (*C function*), 113
PyNumber_InPlaceXor (*C function*), 114
PyNumber_Invert (*C function*), 112
PyNumber_Long (*C function*), 114
PyNumber_Lshift (*C function*), 112
PyNumber_MatrixMultiply (*C function*), 112
PyNumber_Multiply (*C function*), 112
PyNumber_Negative (*C function*), 112
PyNumber_Or (*C function*), 113
PyNumber_Positive (*C function*), 112
PyNumber_Power (*C function*), 112
PyNumber_Remainder (*C function*), 112
PyNumber_Rshift (*C function*), 112
PyNumber_Subtract (*C function*), 111
PyNumber_ToBase (*C function*), 114
PyNumber_TrueDivide (*C function*), 112
PyNumber_Xor (*C function*), 113
PyNumberMethods (*C type*), 327
PyNumberMethods.nb_absolute (*C member*), 329
PyNumberMethods.nb_add (*C member*), 328
PyNumberMethods.nb_and (*C member*), 329
PyNumberMethods.nb_bool (*C member*), 329
PyNumberMethods.nb_divmod (*C member*), 329
PyNumberMethods.nb_float (*C member*), 329
PyNumberMethods.nb_floor_divide (*C member*), 329
PyNumberMethods.nb_index (*C member*), 329
PyNumberMethods.nb_inplace_add (*C member*), 329
PyNumberMethods.nb_inplace_and (*C member*), 329
PyNumberMethods.nb_inplace_floor_divide (*C member*), 329
PyNumberMethods.nb_inplace_lshift (*C member*), 329
PyNumberMethods.nb_inplace_matrix_multiply (*C member*), 329
PyNumberMethods.nb_inplace_multiply (*C member*), 329
PyNumberMethods.nb_inplace_or (*C member*), 329
PyNumberMethods.nb_inplace_power (*C member*), 329
PyNumberMethods.nb_inplace_remainder (*C member*), 329
PyNumberMethods.nb_inplace_rshift (*C member*), 329
PyNumberMethods.nb_inplace_subtract (*C member*), 329
PyNumberMethods.nb_inplace_true_divide (*C member*), 329
PyNumberMethods.nb_inplace_xor (*C member*), 329
PyNumberMethods.nb_int (*C member*), 329
PyNumberMethods.nb_invert (*C member*), 329
PyNumberMethods.nb_lshift (*C member*), 329
PyNumberMethods.nb_matrix_multiply (*C member*), 329
PyNumberMethods.nb_multiply (*C member*), 329
PyNumberMethods.nb_negative (*C member*), 329
PyNumberMethods.nb_or (*C member*), 329
PyNumberMethods.nb_positive (*C member*), 329
PyNumberMethods.nb_power (*C member*), 329
PyNumberMethods.nb_remainder (*C member*), 329
PyNumberMethods.nb_reserved (*C member*), 329
PyNumberMethods.nb_rshift (*C member*), 329
PyNumberMethods.nb_subtract (*C member*), 329
PyNumberMethods.nb_true_divide (*C member*), 329
PyNumberMethods.nb_xor (*C member*), 329
PyObject (*C type*), 289
PyObject_ASCII (*C function*), 101
PyObject_AsFileDescriptor (*C function*), 192
PyObject_Bytes (*C function*), 101
PyObject_Call (*C function*), 109
PyObject_CallFinalizer (*C function*), 288
PyObject_CallFinalizerFromDealloc (*C function*), 288
PyObject_CallFunction (*C function*), 110
PyObject_CallFunctionObjArgs (*C function*), 110
PyObject_CallMethod (*C function*), 110
PyObject_CallMethodNoArgs (*C function*), 110
PyObject_CallMethodObjArgs (*C function*), 110
PyObject_CallMethodOneArg (*C function*), 110
PyObject_CallNoArgs (*C function*), 109
PyObject_CallObject (*C function*), 110
PyObject_Calloc (*C function*), 277
PyObject_CallOneArg (*C function*), 109
PyObject_CheckBuffer (*C function*), 124
PyObject_ClearManagedDict (*C function*), 104
PyObject_ClearWeakRefs (*C function*), 205
PyObject_CopyData (*C function*), 125
PyObject_Del (*C function*), 285
PyObject_DelAttr (*C function*), 100
PyObject_DelAttrString (*C function*), 100
PyObject_DelItem (*C function*), 103
PyObject_DelItemString (*C function*), 103
PyObject_Dir (*C function*), 103
PyObject_Format (*C function*), 101
PyObject_Free (*C function*), 277
PyObject_GC_Del (*C function*), 338
PyObject_GC_IsFinalized (*C function*), 338
PyObject_GC_IsTracked (*C function*), 338
PyObject_GC_New (*C macro*), 337
PyObject_GC_NewVar (*C macro*), 337
PyObject_GC_Resize (*C macro*), 338
PyObject_GC_Track (*C function*), 338
PyObject_GC_UnTrack (*C function*), 339
PyObject_GenericGetAttr (*C function*), 100
PyObject_GenericGetDict (*C function*), 100
PyObject_GenericHash (*C function*), 90
PyObject_GenericSetAttr (*C function*), 100

PyObject_GenericSetDict (*C function*), 101
PyObject_GetAIter (*C function*), 103
PyObject_GetArenaAllocator (*C function*), 281
PyObject_GetAttr (*C function*), 99
PyObject_GetAttrString (*C function*), 99
PyObject_GetBuffer (*C function*), 124
PyObject_GetItem (*C function*), 103
PyObject_GetItemData (*C function*), 104
PyObject_GetIter (*C function*), 103
PyObject_GetOptionalAttr (*C function*), 99
PyObject_GetOptionalAttrString (*C function*), 100
PyObject_GetTypeData (*C function*), 103
PyObject_HasAttr (*C function*), 99
PyObject_HasAttrString (*C function*), 99
PyObject_HasAttrStringWithError (*C function*), 99
PyObject_HasAttrWithError (*C function*), 99
PyObject_Hash (*C function*), 102
PyObject_HashNotImplemented (*C function*), 102
PyObject_HEAD (*C macro*), 289
PyObject_HEAD_INIT (*C macro*), 291
PyObject_Init (*C function*), 283
PyObject_InitVar (*C function*), 283
PyObject_IS_GC (*C function*), 338
PyObject_IsInstance (*C function*), 102
PyObject_IsSubclass (*C function*), 102
PyObject_IsTrue (*C function*), 102
PyObject_Length (*C function*), 103
PyObject_LengthHint (*C function*), 103
PyObject_Malloc (*C function*), 277
PyObject_New (*C macro*), 283
PyObject_NewVar (*C macro*), 284
PyObject_Not (*C function*), 102
PyObject_Print (*C function*), 98
PyObject_Realloc (*C function*), 277
PyObject_Repr (*C function*), 101
PyObject_RichCompare (*C function*), 101
PyObject_RichCompareBool (*C function*), 101
PyObject_SelfIter (*C function*), 103
PyObject_SetArenaAllocator (*C function*), 281
PyObject_SetAttr (*C function*), 100
PyObject_SetAttrString (*C function*), 100
PyObject_SetItem (*C function*), 103
PyObject_Size (*C function*), 103
PyObject_Str (*C function*), 101
PyObject_Type (*C function*), 102
PyObject_TypeCheck (*C function*), 102
PyObject_VAR_HEAD (*C macro*), 289
PyObject_Vectorcall (*C function*), 111
PyObject_VectorcallDict (*C function*), 111
PyObject_VectorcallMethod (*C function*), 111
PyObject_VisitManagedDict (*C function*), 104
PyObjectArenaAllocator (*C type*), 281
PyObject.ob_refcnt (*C member*), 289
PyObject.ob_type (*C member*), 289
PyOS_AfterFork (*C function*), 70
PyOS_AfterFork_Child (*C function*), 70
PyOS_AfterFork_Parent (*C function*), 69
PyOS_BeforeFork (*C function*), 69
PyOS_CheckStack (*C function*), 70
PyOS_double_to_string (*C function*), 89
PyOS_FSPath (*C function*), 69
PyOS_getsig (*C function*), 70
PyOS_InputHook (*C var*), 36
PyOS_ReadlineFunctionPointer (*C var*), 36
PyOS_setsig (*C function*), 70
PyOS_sighandler_t (*C type*), 70
PyOS_snprintf (*C function*), 87
PyOS_stricmp (*C function*), 89
PyOS_string_to_double (*C function*), 88
PyOS_strnicmp (*C function*), 89
PyOS_strtol (*C function*), 88
PyOS_strtoul (*C function*), 88
PyOS_vsnprintf (*C function*), 87
PyPreConfig (*C type*), 255
PyPreConfig_InitIsolatedConfig (*C function*), 255
PyPreConfig_InitPythonConfig (*C function*), 255
PyPreConfig.allocator (*C member*), 255
PyPreConfig.coerce_c_locale (*C member*), 256
PyPreConfig.coerce_c_locale_warn (*C member*), 256
PyPreConfig.configure_locale (*C member*), 255
PyPreConfig.dev_mode (*C member*), 256
PyPreConfig.isolated (*C member*), 256
PyPreConfig.legacy_windows_fs_encoding (*C member*), 256
PyPreConfig.parse_argv (*C member*), 256
PyPreConfig.use_environment (*C member*), 256
PyPreConfig.utf8_mode (*C member*), 256
PyProperty_Type (*C var*), 201
PyRefTracer (*C type*), 240
PyRefTracer_CREATE (*C var*), 240
PyRefTracer_DESTROY (*C var*), 240
PyRefTracer_GetTracer (*C function*), 241
PyRefTracer_SetTracer (*C function*), 240
PyRun_AnyFile (*C function*), 35
PyRun_AnyFileEx (*C function*), 35
PyRun_AnyFileExFlags (*C function*), 35
PyRun_AnyFileFlags (*C function*), 35
PyRun_File (*C function*), 37
PyRun_FileEx (*C function*), 37
PyRun_FileExFlags (*C function*), 37
PyRun_FileFlags (*C function*), 37
PyRun_InteractiveLoop (*C function*), 36
PyRun_InteractiveLoopFlags (*C function*), 36
PyRun_InteractiveOne (*C function*), 36
PyRun_InteractiveOneFlags (*C function*), 36
PyRun_SimpleFile (*C function*), 36
PyRun_SimpleFileEx (*C function*), 36
PyRun_SimpleFileExFlags (*C function*), 36
PyRun_SimpleString (*C function*), 35
PyRun_SimpleStringFlags (*C function*), 35
PyRun_String (*C function*), 37
PyRun_StringFlags (*C function*), 37

PySendResult (*C type*), 118
PySeqIter_Check (*C function*), 201
PySeqIter_New (*C function*), 201
PySeqIter_Type (*C var*), 201
PySequence_Check (*C function*), 114
PySequence_Concat (*C function*), 115
PySequence_Contains (*C function*), 115
PySequence_Count (*C function*), 115
PySequence_DelItem (*C function*), 115
PySequence_DelSlice (*C function*), 115
PySequence_Fast (*C function*), 116
PySequence_Fast_GET_ITEM (*C function*), 116
PySequence_Fast_GET_SIZE (*C function*), 116
PySequence_Fast_ITEMS (*C function*), 116
PySequence_GetItem (*C function*), 9, 115
PySequence_GetSlice (*C function*), 115
PySequence_In (*C function*), 115
PySequence_Index (*C function*), 116
PySequence_InPlaceConcat (*C function*), 115
PySequence_InPlaceRepeat (*C function*), 115
PySequence_ITEM (*C function*), 116
PySequence_Length (*C function*), 114
PySequence_List (*C function*), 116
PySequence_Repeat (*C function*), 115
PySequence_SetItem (*C function*), 115
PySequence_SetSlice (*C function*), 115
PySequence_Size (*C function*), 114
PySequence_Tuple (*C function*), 116
PySequenceMethods (*C type*), 330
PySequenceMethods.sq_ass_item (*C member*), 330
PySequenceMethods.sq_concat (*C member*), 330
PySequenceMethods.sq_contains (*C member*), 330
PySequenceMethods.sq_inplace_concat (*C member*), 330
PySequenceMethods.sq_inplace_repeat (*C member*), 330
PySequenceMethods.sq_item (*C member*), 330
PySequenceMethods.sq_length (*C member*), 330
PySequenceMethods.sq_repeat (*C member*), 330
PySet_Add (*C function*), 182
PySet_Check (*C function*), 181
PySet_CheckExact (*C function*), 182
PySet_Clear (*C function*), 183
PySet_Contains (*C function*), 182
PySet_Discard (*C function*), 182
PySet_GET_SIZE (*C function*), 182
PySet_New (*C function*), 182
PySet_Pop (*C function*), 182
PySet_Size (*C function*), 182
PySet_Type (*C var*), 181
PySetObject (*C type*), 181
PySignal_SetWakeupFd (*C function*), 53
PySlice_AdjustIndices (*C function*), 203
PySlice_Check (*C function*), 202
PySlice_GetIndices (*C function*), 202
PySlice_GetIndicesEx (*C function*), 202
PySlice_New (*C function*), 202
PySlice_Type (*C var*), 202
PySlice_Unpack (*C function*), 203
PyState_AddModule (*C function*), 200
PyState_FindModule (*C function*), 200
PyState_RemoveModule (*C function*), 201
PyStatus (*C type*), 254
PyStatus_Error (*C function*), 254
PyStatus_Exception (*C function*), 254
PyStatus_Exit (*C function*), 254
PyStatus_IsError (*C function*), 254
PyStatus_IsExit (*C function*), 254
PyStatus_NoMemory (*C function*), 254
PyStatus_Ok (*C function*), 254
PyStatus.err_msg (*C member*), 254
PyStatus.exitcode (*C member*), 254
PyStatus.func (*C member*), 254
PyStructSequence_Desc (*C type*), 173
PyStructSequence_Desc.doc (*C member*), 173
PyStructSequence_Desc.fields (*C member*), 173
PyStructSequence_Desc.n_in_sequence (*C member*), 173
PyStructSequence_Desc.name (*C member*), 173
PyStructSequence_Field (*C type*), 173
PyStructSequence_Field.doc (*C member*), 174
PyStructSequence_Field.name (*C member*), 174
PyStructSequence_GET_ITEM (*C function*), 174
PyStructSequence_GetItem (*C function*), 174
PyStructSequence_InitType (*C function*), 173
PyStructSequence_InitType2 (*C function*), 173
PyStructSequence_New (*C function*), 174
PyStructSequence_NewType (*C function*), 173
PyStructSequence_SET_ITEM (*C function*), 174
PyStructSequence_SetItem (*C function*), 174
PyStructSequence_UnnamedField (*C var*), 174
PySys_AddAuditHook (*C function*), 73
PySys_Audit (*C function*), 73
PySys_AuditTuple (*C function*), 73
PySys_FormatStderr (*C function*), 73
PySys_FormatStdout (*C function*), 73
PySys_GetObject (*C function*), 72
PySys_GetXOptions (*C function*), 73
PySys_ResetWarnOptions (*C function*), 72
PySys_SetArgv (*C function*), 226
PySys_SetArgvEx (*C function*), 225
PySys_SetObject (*C function*), 72
PySys_WriteStderr (*C function*), 73
PySys_WriteStdout (*C function*), 72
Python 3000, **364**
Python Enhancement Proposals
    PEP 1, 364
    PEP 7, 3, 6
    PEP 278, 367
    PEP 302, 361
    PEP 328, 357
    PEP 343, 354
    PEP 353, 10
    PEP 362, 352, 363

PEP 383, 159, 160
PEP 387, 15, 16
PEP 393, 151
PEP 411, 364
PEP 420, 362, 364
PEP 442, 288, 326
PEP 443, 358
PEP 446, 72
PEP 451, 195
PEP 456, 90
PEP 483, 358
PEP 484, 351, 357, 358, 367
PEP 489, 65, 66, 236
PEP 492, 352, 353, 355
PEP 498, 356
PEP 519, 363
PEP 523, 209, 234
PEP 525, 352
PEP 526, 351, 367
PEP 528, 220, 263
PEP 529, 160, 219
PEP 538, 270
PEP 539, 242
PEP 540, 270
PEP 552, 260
PEP 554, 237
PEP 578, 73
PEP 585, 358
PEP 587, 252
PEP 590, 107
PEP 623, 151
PEP 0626#out-of-process-debuggers-and-profilers, 188
PEP 634, 314
PEP 649, 351
PEP 667, 91, 208, 209
PEP 0683, 42, 359
PEP 684, 230, 237
PEP 703, 357, 358
PEP 741, 247
PEP 3116, 367
PEP 3119, 102
PEP 3121, 194
PEP 3147, 77
PEP 3151, 61
PEP 3155, 364
PYTHON_ABI_VERSION (*C macro*), 197
PYTHON_API_VERSION (*C macro*), 197
PYTHON_CPU_COUNT, 263
PYTHON_FROZEN_MODULES, 262
PYTHON_GIL, 358
PYTHON_PERF_JIT_SUPPORT, 267
PYTHON_PRESITE, 266
PYTHONCOERCECLOCALE, 270
PYTHONDEBUG, 218, 264
PYTHONDEVMODE, 260
PYTHONDONTWRITEBYTECODE, 218, 268
PYTHONDUMPREFS, 260
PYTHONDUMPREFSFILE, 261
PYTHONEXECUTABLE, 265
PYTHONFAULTHANDLER, 261
PYTHONHASHSEED, 219, 262
PYTHONHOME, 12, 219, 226, 227, 262
PYTHONINSPECT, 219, 262
PYTHONINTMAXSTRDIGITS, 263
PYTHONIOENCODING, 266
Pythonique, **364**
PYTHONLEGACYWINDOWSFSENCODING, 219, 256
PYTHONLEGACYWINDOWSSTDIO, 220, 263
PYTHONMALLOC, 274, 278, 279, 281
PYTHONMALLOCSTATS, 263, 274
PYTHONNODEBUGRANGES, 260
PYTHONNOUSERSITE, 220, 267
PYTHONOPTIMIZE, 220, 264
PYTHONPATH, 12, 219, 264
PYTHONPERFSUPPORT, 267
PYTHONPLATLIBDIR, 263
PYTHONPROFILEIMPORTTIME, 262
PYTHONPYCACHEPREFIX, 265
PYTHONSAFEPATH, 259
PYTHONTRACEMALLOC, 267
PYTHONUNBUFFERED, 220, 260
PYTHONUTF8, 256, 270
PYTHONVERBOSE, 221, 268
PYTHONWARNINGS, 268
PyThread_create_key (*C function*), 243
PyThread_delete_key (*C function*), 243
PyThread_delete_key_value (*C function*), 243
PyThread_get_key_value (*C function*), 243
PyThread_ReInitTLS (*C function*), 243
PyThread_set_key_value (*C function*), 243
PyThread_tss_alloc (*C function*), 242
PyThread_tss_create (*C function*), 242
PyThread_tss_delete (*C function*), 242
PyThread_tss_free (*C function*), 242
PyThread_tss_get (*C function*), 243
PyThread_tss_is_created (*C function*), 242
PyThread_tss_set (*C function*), 243
PyThreadState (*C type*), 227, 230
PyThreadState_Clear (*C function*), 232
PyThreadState_Delete (*C function*), 233
PyThreadState_DeleteCurrent (*C function*), 233
PyThreadState_EnterTracing (*C function*), 233
PyThreadState_Get (*C function*), 230
PyThreadState_GetDict (*C function*), 234
PyThreadState_GetFrame (*C function*), 233
PyThreadState_GetID (*C function*), 233
PyThreadState_GetInterpreter (*C function*), 233
PyThreadState_GetUnchecked (*C function*), 230
PyThreadState_LeaveTracing (*C function*), 233
PyThreadState_New (*C function*), 232
PyThreadState_Next (*C function*), 241
PyThreadState_SetAsyncExc (*C function*), 234
PyThreadState_Swap (*C function*), 230
PyThreadState.interp (*C member*), 230
PyTime_AsSecondsDouble (*C function*), 95

PyTime_Check (*C function*), 213
PyTime_CheckExact (*C function*), 213
PyTime_FromTime (*C function*), 214
PyTime_FromTimeAndFold (*C function*), 214
PyTime_MAX (*C var*), 94
PyTime_MIN (*C var*), 94
PyTime_Monotonic (*C function*), 94
PyTime_MonotonicRaw (*C function*), 94
PyTime_PerfCounter (*C function*), 94
PyTime_PerfCounterRaw (*C function*), 94
PyTime_t (*C type*), 94
PyTime_Time (*C function*), 94
PyTime_TimeRaw (*C function*), 94
PyTimeZone_FromOffset (*C function*), 214
PyTimeZone_FromOffsetAndName (*C function*), 214
PyTrace_C_CALL (*C var*), 239
PyTrace_C_EXCEPTION (*C var*), 239
PyTrace_C_RETURN (*C var*), 239
PyTrace_CALL (*C var*), 239
PyTrace_EXCEPTION (*C var*), 239
PyTrace_LINE (*C var*), 239
PyTrace_OPCODE (*C var*), 239
PyTrace_RETURN (*C var*), 239
PyTraceMalloc_Track (*C function*), 281
PyTraceMalloc_Untrack (*C function*), 281
PyTuple_Check (*C function*), 172
PyTuple_CheckExact (*C function*), 172
PyTuple_GET_ITEM (*C function*), 172
PyTuple_GET_SIZE (*C function*), 172
PyTuple_GetItem (*C function*), 172
PyTuple_GetSlice (*C function*), 172
PyTuple_New (*C function*), 172
PyTuple_Pack (*C function*), 172
PyTuple_SET_ITEM (*C function*), 172
PyTuple_SetItem (*C function*), 7, 172
PyTuple_Size (*C function*), 172
PyTuple_Type (*C var*), 172
PyTupleObject (*C type*), 172
PyType_AddWatcher (*C function*), 128
PyType_Check (*C function*), 127
PyType_CheckExact (*C function*), 127
PyType_ClearCache (*C function*), 127
PyType_ClearWatcher (*C function*), 128
PyType_Freeze (*C function*), 132
PyType_FromMetaclass (*C function*), 131
PyType_FromModuleAndSpec (*C function*), 131
PyType_FromSpec (*C function*), 132
PyType_FromSpecWithBases (*C function*), 131
PyType_GenericAlloc (*C function*), 129
PyType_GenericNew (*C function*), 129
PyType_GetBaseByToken (*C function*), 130
PyType_GetDict (*C function*), 128
PyType_GetFlags (*C function*), 127
PyType_GetFullyQualifiedName (*C function*), 129
PyType_GetModule (*C function*), 130
PyType_GetModuleByDef (*C function*), 130
PyType_GetModuleName (*C function*), 129
PyType_GetModuleState (*C function*), 130
PyType_GetName (*C function*), 129
PyType_GetQualName (*C function*), 129
PyType_GetSlot (*C function*), 130
PyType_GetTypeDataSize (*C function*), 103
PyType_HasFeature (*C function*), 128
PyType_IS_GC (*C function*), 128
PyType_IsSubtype (*C function*), 129
PyType_Modified (*C function*), 128
PyType_Ready (*C function*), 129
PyType_Slot (*C type*), 133
PyType_Slot.pfunc (*C member*), 133
PyType_Slot.slot (*C member*), 133
PyType_Spec (*C type*), 132
PyType_Spec.basicsize (*C member*), 132
PyType_Spec.flags (*C member*), 132
PyType_Spec.itemsize (*C member*), 132
PyType_Spec.name (*C member*), 132
PyType_Spec.slots (*C member*), 133
PyType_Type (*C var*), 127
PyType_Watch (*C function*), 128
PyType_WatchCallback (*C type*), 128
PyTypeObject (*C type*), 127
PyTypeObject.tp_alloc (*C member*), 322
PyTypeObject.tp_as_async (*C member*), 308
PyTypeObject.tp_as_buffer (*C member*), 310
PyTypeObject.tp_as_mapping (*C member*), 309
PyTypeObject.tp_as_number (*C member*), 308
PyTypeObject.tp_as_sequence (*C member*), 309
PyTypeObject.tp_base (*C member*), 320
PyTypeObject.tp_bases (*C member*), 323
PyTypeObject.tp_basicsize (*C member*), 304
PyTypeObject.tp_cache (*C member*), 324
PyTypeObject.tp_call (*C member*), 309
PyTypeObject.tp_clear (*C member*), 316
PyTypeObject.tp_dealloc (*C member*), 305
PyTypeObject.tp_del (*C member*), 324
PyTypeObject.tp_descr_get (*C member*), 321
PyTypeObject.tp_descr_set (*C member*), 321
PyTypeObject.tp_dict (*C member*), 320
PyTypeObject.tp_dictoffset (*C member*), 321
PyTypeObject.tp_doc (*C member*), 314
PyTypeObject.tp_finalize (*C member*), 324
PyTypeObject.tp_flags (*C member*), 310
PyTypeObject.tp_free (*C member*), 323
PyTypeObject.tp_getattr (*C member*), 308
PyTypeObject.tp_getattro (*C member*), 310
PyTypeObject.tp_getset (*C member*), 320
PyTypeObject.tp_hash (*C member*), 309
PyTypeObject.tp_init (*C member*), 322
PyTypeObject.tp_is_gc (*C member*), 323
PyTypeObject.tp_itemsize (*C member*), 304
PyTypeObject.tp_iter (*C member*), 319
PyTypeObject.tp_iternext (*C member*), 319
PyTypeObject.tp_members (*C member*), 320
PyTypeObject.tp_methods (*C member*), 319
PyTypeObject.tp_mro (*C member*), 324
PyTypeObject.tp_name (*C member*), 304
PyTypeObject.tp_new (*C member*), 322

PyTypeObject.tp_repr (*C member*), 308
PyTypeObject.tp_richcompare (*C member*), 318
PyTypeObject.tp_setattr (*C member*), 308
PyTypeObject.tp_setattro (*C member*), 310
PyTypeObject.tp_str (*C member*), 309
PyTypeObject.tp_subclasses (*C member*), 324
PyTypeObject.tp_traverse (*C member*), 315
PyTypeObject.tp_vectorcall (*C member*), 326
PyTypeObject.tp_vectorcall_offset (*C member*), 307
PyTypeObject.tp_version_tag (*C member*), 324
PyTypeObject.tp_watched (*C member*), 327
PyTypeObject.tp_weaklist (*C member*), 324
PyTypeObject.tp_weaklistoffset (*C member*), 319
PyTZInfo_Check (*C function*), 213
PyTZInfo_CheckExact (*C function*), 213
PyUnicode_1BYTE_DATA (*C function*), 152
PyUnicode_1BYTE_KIND (*C macro*), 152
PyUnicode_2BYTE_DATA (*C function*), 152
PyUnicode_2BYTE_KIND (*C macro*), 152
PyUnicode_4BYTE_DATA (*C function*), 152
PyUnicode_4BYTE_KIND (*C macro*), 152
PyUnicode_Append (*C function*), 157
PyUnicode_AppendAndDel (*C function*), 157
PyUnicode_AsASCIIString (*C function*), 165
PyUnicode_AsCharmapString (*C function*), 166
PyUnicode_AsEncodedString (*C function*), 162
PyUnicode_AsLatin1String (*C function*), 165
PyUnicode_AsMBCSString (*C function*), 166
PyUnicode_AsRawUnicodeEscapeString (*C function*), 165
PyUnicode_AsUCS4 (*C function*), 158
PyUnicode_AsUCS4Copy (*C function*), 159
PyUnicode_AsUnicodeEscapeString (*C function*), 164
PyUnicode_AsUTF8 (*C function*), 163
PyUnicode_AsUTF8AndSize (*C function*), 162
PyUnicode_AsUTF8String (*C function*), 162
PyUnicode_AsUTF16String (*C function*), 164
PyUnicode_AsUTF32String (*C function*), 163
PyUnicode_AsWideChar (*C function*), 161
PyUnicode_AsWideCharString (*C function*), 161
PyUnicode_BuildEncodingMap (*C function*), 157
PyUnicode_Check (*C function*), 151
PyUnicode_CHECK_INTERNED (*C function*), 169
PyUnicode_CheckExact (*C function*), 151
PyUnicode_Compare (*C function*), 168
PyUnicode_CompareWithASCIIString (*C function*), 169
PyUnicode_Concat (*C function*), 167
PyUnicode_Contains (*C function*), 169
PyUnicode_CopyCharacters (*C function*), 158
PyUnicode_Count (*C function*), 168
PyUnicode_DATA (*C function*), 152
PyUnicode_Decode (*C function*), 162
PyUnicode_DecodeASCII (*C function*), 165
PyUnicode_DecodeCharmap (*C function*), 165
PyUnicode_DecodeCodePageStateful (*C function*), 166
PyUnicode_DecodeFSDefault (*C function*), 160
PyUnicode_DecodeFSDefaultAndSize (*C function*), 160
PyUnicode_DecodeLatin1 (*C function*), 165
PyUnicode_DecodeLocale (*C function*), 159
PyUnicode_DecodeLocaleAndSize (*C function*), 159
PyUnicode_DecodeMBCS (*C function*), 166
PyUnicode_DecodeMBCSStateful (*C function*), 166
PyUnicode_DecodeRawUnicodeEscape (*C function*), 165
PyUnicode_DecodeUnicodeEscape (*C function*), 164
PyUnicode_DecodeUTF7 (*C function*), 164
PyUnicode_DecodeUTF7Stateful (*C function*), 164
PyUnicode_DecodeUTF8 (*C function*), 162
PyUnicode_DecodeUTF8Stateful (*C function*), 162
PyUnicode_DecodeUTF16 (*C function*), 164
PyUnicode_DecodeUTF16Stateful (*C function*), 164
PyUnicode_DecodeUTF32 (*C function*), 163
PyUnicode_DecodeUTF32Stateful (*C function*), 163
PyUnicode_EncodeCodePage (*C function*), 166
PyUnicode_EncodeFSDefault (*C function*), 160
PyUnicode_EncodeLocale (*C function*), 159
PyUnicode_Equal (*C function*), 168
PyUnicode_EqualToUTF8 (*C function*), 168
PyUnicode_EqualToUTF8AndSize (*C function*), 168
PyUnicode_Fill (*C function*), 158
PyUnicode_Find (*C function*), 167
PyUnicode_FindChar (*C function*), 168
PyUnicode_Format (*C function*), 169
PyUnicode_FromEncodedObject (*C function*), 157
PyUnicode_FromFormat (*C function*), 155
PyUnicode_FromFormatV (*C function*), 157
PyUnicode_FromKindAndData (*C function*), 154
PyUnicode_FromObject (*C function*), 157
PyUnicode_FromOrdinal (*C function*), 157
PyUnicode_FromString (*C function*), 155
PyUnicode_FromStringAndSize (*C function*), 155
PyUnicode_FromWideChar (*C function*), 161
PyUnicode_FSConverter (*C function*), 160
PyUnicode_FSDecoder (*C function*), 160
PyUnicode_GET_LENGTH (*C function*), 152
PyUnicode_GetDefaultEncoding (*C function*), 157
PyUnicode_GetLength (*C function*), 157
PyUnicode_InternFromString (*C function*), 169
PyUnicode_InternInPlace (*C function*), 169
PyUnicode_IS_ASCII (*C function*), 153
PyUnicode_IS_READY (*C function*), 171
PyUnicode_IsIdentifier (*C function*), 153
PyUnicode_Join (*C function*), 167
PyUnicode_KIND (*C function*), 152
PyUnicode_MAX_CHAR_VALUE (*C function*), 152
PyUnicode_New (*C function*), 154

PyUnicode_Partition (*C function*), 167
PyUnicode_READ (*C function*), 152
PyUnicode_READ_CHAR (*C function*), 152
PyUnicode_ReadChar (*C function*), 158
PyUnicode_READY (*C function*), 171
PyUnicode_Replace (*C function*), 168
PyUnicode_Resize (*C function*), 158
PyUnicode_RichCompare (*C function*), 169
PyUnicode_RPartition (*C function*), 167
PyUnicode_RSplit (*C function*), 167
PyUnicode_Split (*C function*), 167
PyUnicode_Splitlines (*C function*), 167
PyUnicode_Substring (*C function*), 158
PyUnicode_Tailmatch (*C function*), 167
PyUnicode_Translate (*C function*), 166
PyUnicode_Type (*C var*), 151
PyUnicode_WRITE (*C function*), 152
PyUnicode_WriteChar (*C function*), 158
PyUnicodeDecodeError_Create (*C function*), 55
PyUnicodeDecodeError_GetEncoding (*C function*), 55
PyUnicodeDecodeError_GetEnd (*C function*), 56
PyUnicodeDecodeError_GetObject (*C function*), 55
PyUnicodeDecodeError_GetReason (*C function*), 56
PyUnicodeDecodeError_GetStart (*C function*), 55
PyUnicodeDecodeError_SetEnd (*C function*), 56
PyUnicodeDecodeError_SetReason (*C function*), 56
PyUnicodeDecodeError_SetStart (*C function*), 55
PyUnicodeEncodeError_GetEncoding (*C function*), 55
PyUnicodeEncodeError_GetEnd (*C function*), 56
PyUnicodeEncodeError_GetObject (*C function*), 55
PyUnicodeEncodeError_GetReason (*C function*), 56
PyUnicodeEncodeError_GetStart (*C function*), 55
PyUnicodeEncodeError_SetEnd (*C function*), 56
PyUnicodeEncodeError_SetReason (*C function*), 56
PyUnicodeEncodeError_SetStart (*C function*), 55
PyUnicodeIter_Type (*C var*), 151
PyUnicodeObject (*C type*), 151
PyUnicodeTranslateError_GetEnd (*C function*), 56
PyUnicodeTranslateError_GetObject (*C function*), 55
PyUnicodeTranslateError_GetReason (*C function*), 56
PyUnicodeTranslateError_GetStart (*C function*), 55
PyUnicodeTranslateError_SetEnd (*C function*), 56
PyUnicodeTranslateError_SetReason (*C function*), 56
PyUnicodeTranslateError_SetStart (*C function*), 55
PyUnicodeWriter (*C type*), 170
PyUnicodeWriter_Create (*C function*), 170
PyUnicodeWriter_DecodeUTF8Stateful (*C function*), 171
PyUnicodeWriter_Discard (*C function*), 170
PyUnicodeWriter_Finish (*C function*), 170
PyUnicodeWriter_Format (*C function*), 171
PyUnicodeWriter_WriteASCII (*C function*), 170
PyUnicodeWriter_WriteChar (*C function*), 170
PyUnicodeWriter_WriteRepr (*C function*), 171
PyUnicodeWriter_WriteStr (*C function*), 171
PyUnicodeWriter_WriteSubstring (*C function*), 171
PyUnicodeWriter_WriteUCS4 (*C function*), 170
PyUnicodeWriter_WriteUTF8 (*C function*), 170
PyUnicodeWriter_WriteWideChar (*C function*), 170
PyUnstable, 15
PyUnstable_AtExit (*C function*), 223
PyUnstable_Code_GetExtra (*C function*), 191
PyUnstable_Code_GetFirstFree (*C function*), 187
PyUnstable_Code_New (*C function*), 187
PyUnstable_Code_NewWithPosOnlyArgs (*C function*), 187
PyUnstable_Code_SetExtra (*C function*), 191
PyUnstable_EnableTryIncRef (*C function*), 106
PyUnstable_Eval_RequestCodeExtraIndex (*C function*), 191
PyUnstable_Exc_PrepReraiseStar (*C function*), 55
PyUnstable_GC_VisitObjects (*C function*), 340
PyUnstable_InterpreterFrame_GetCode (*C function*), 209
PyUnstable_InterpreterFrame_GetLasti (*C function*), 209
PyUnstable_InterpreterFrame_GetLine (*C function*), 209
PyUnstable_IsImmortal (*C function*), 105
PyUnstable_Long_CompactValue (*C function*), 141
PyUnstable_Long_IsCompact (*C function*), 141
PyUnstable_Module_SetGIL (*C function*), 200
PyUnstable_Object_ClearWeakRefsNoCallbacks (*C function*), 205
PyUnstable_Object_EnableDeferredRefcount (*C function*), 104
PyUnstable_Object_GC_NewWithExtraData (*C function*), 337
PyUnstable_Object_IsUniquelyReferenced (*C function*), 107
PyUnstable_Object_IsUniqueReferencedTemporary (*C function*), 104
PyUnstable_PerfMapState_Fini (*C function*), 95
PyUnstable_PerfMapState_Init (*C function*), 95
PyUnstable_TryIncRef (*C function*), 105
PyUnstable_Type_AssignVersionTag (*C function*), 130

PyUnstable_WritePerfMapEntry (*C function*), 95
PyVarObject (*C type*), 289
PyVarObject_HEAD_INIT (*C macro*), 291
PyVarObject.ob_size (*C member*), 289
PyVectorcall_Call (*C function*), 109
PyVectorcall_Function (*C function*), 109
PyVectorcall_NARGS (*C function*), 108
PyWeakref_Check (*C function*), 204
PyWeakref_CheckProxy (*C function*), 204
PyWeakref_CheckRef (*C function*), 204
PyWeakref_GET_OBJECT (*C function*), 205
PyWeakref_GetObject (*C function*), 204
PyWeakref_GetRef (*C function*), 204
PyWeakref_IsDead (*C function*), 205
PyWeakref_NewProxy (*C function*), 204
PyWeakref_NewRef (*C function*), 204
PyWideStringList (*C type*), 253
PyWideStringList_Append (*C function*), 253
PyWideStringList_Insert (*C function*), 253
PyWideStringList.items (*C member*), 254
PyWideStringList.length (*C member*), 253
PyWrapper_New (*C function*), 202

R

ramasse-miettes, **357**
READ_RESTRICTED (*C macro*), 295
READONLY (*C macro*), 295
realloc (*C function*), 273
référence empruntée, **353**
référence forte, **366**
releasebufferproc (*C type*), 333
REPL, **365**
repr
    built-in function, 101, 308
reprfunc (*C type*), 333
RESTRICTED (*C macro*), 295
retours à la ligne universels, **367**
richcmpfunc (*C type*), 333

S

search
    path, module, 12, 221, 225
sendfunc (*C type*), 334
sequence
    object, 148
séquence, **365**
set
    object, 181
set_all(), 8
setattrfunc (*C type*), 333
setattrofunc (*C type*), 333
setswitchinterval (*in module sys*), 227
setter (*C type*), 297
SIGINT (*C macro*), 53
signal
    module, 53
SIZE_MAX (*C macro*), 137
soft deprecated, **365**
special
    méthode, 365
spécificateur de module, **361**
ssizeargfunc (*C type*), 334
ssizeobjargproc (*C type*), 334
standard library, **366**
static type checker, **366**
staticmethod
    built-in function, 293
stderr (*in module sys*), 236, 237
stdin (*in module sys*), 236, 237
stdlib, **366**
stdout (*in module sys*), 236, 237
strerror (*C function*), 47
string
    PyObject_Str (*C function*), 101
structmember.h, 297
sum_list(), 9
sum_sequence(), 9, 10
sys
    module, 12, 221, 236, 237
SystemError (*built-in exception*), 193, 194

T

t-string, **366**
t-strings, **366**
T_BOOL (*C macro*), 297
T_BYTE (*C macro*), 297
T_CHAR (*C macro*), 297
T_DOUBLE (*C macro*), 297
T_FLOAT (*C macro*), 297
T_INT (*C macro*), 297
T_LONG (*C macro*), 297
T_LONGLONG (*C macro*), 297
T_NONE (*C macro*), 297
T_OBJECT (*C macro*), 297
T_OBJECT_EX (*C macro*), 297
T_PYSSIZET (*C macro*), 297
T_SHORT (*C macro*), 297
T_STRING (*C macro*), 297
T_STRING_INPLACE (*C macro*), 297
T_UBYTE (*C macro*), 297
T_UINT (*C macro*), 297
T_ULONG (*C macro*), 297
T_ULONGULONG (*C macro*), 297
T_USHORT (*C macro*), 297
tableau de correspondances (*mapping en anglais*), **361**
ternaryfunc (*C type*), 334
thread state, **366**
token, **366**
tranche, **365**
traverseproc (*C type*), 339
tuple
    built-in function, 116, 176
    object, 172
typage canard (*duck-typing*), **356**
type, **367**

built-in function, 102
object, 6, 127
type générique, **358**

U

ULONG_MAX (*C macro*), 137
unaryfunc (*C type*), 333
USE_STACKCHECK (*C macro*), 70

V

variable de classe, **354**
variable de contexte, **354**
variable d'environnement
__PYVENV_LAUNCHER__, 259, 265
PATH, 12
PYTHON_CPU_COUNT, 263
PYTHON_FROZEN_MODULES, 262
PYTHON_GIL, 358
PYTHON_PERF_JIT_SUPPORT, 267
PYTHON_PRESITE, 266
PYTHONCOERCECLOCALE, 270
PYTHONDEBUG, 218, 264
PYTHONDEVMODE, 260
PYTHONDONTWRITEBYTECODE, 218, 268
PYTHONDUMPREFS, 260
PYTHONDUMPREFSFILE, 261
PYTHONEXECUTABLE, 265
PYTHONFAULTHANDLER, 261
PYTHONHASHSEED, 219, 262
PYTHONHOME, 12, 219, 226, 227, 262
PYTHONINSPECT, 219, 262
PYTHONINTMAXSTRDIGITS, 263
PYTHONIOENCODING, 266
PYTHONLEGACYWINDOWSFSENCODING, 219, 256
PYTHONLEGACYWINDOWSSTDIO, 220, 263
PYTHONMALLOC, 274, 278, 279, 281
PYTHONMALLOCSTATS, 263, 274
PYTHONNODEBUGRANGES, 260
PYTHONNOUSERSITE, 220, 267
PYTHONOPTIMIZE, 220, 264
PYTHONPATH, 12, 219, 264
PYTHONPERFSUPPORT, 267
PYTHONPLATLIBDIR, 263
PYTHONPROFILEIMPORTTIME, 262
PYTHONPYCACHEPREFIX, 265
PYTHONSAFEPATH, 259
PYTHONTRACEMALLOC, 267
PYTHONUNBUFFERED, 220, 260
PYTHONUTF8, 256, 270
PYTHONVERBOSE, 221, 268
PYTHONWARNINGS, 268
vectorcallfunc (*C type*), 108
verrou global de l'interpréteur, **358**
version (*in module sys*), 225
visitproc (*C type*), 339
vue de dictionnaire, **355**
Vulnérabilités et expositions communes
CVE 2008-5983, 226

W

walrus operator, **368**
WRITE_RESTRICTED (*C macro*), 295