The Python/C API

リリース *3.12.11*

Guido van Rossum and the Python development team

9 月 29, 2025

目次

第 1 章　はじめに 3
1.1 コーディング基準 . . . 3
1.2 インクルードファイル . . . 3
1.3 便利なマクロ . . . 4
1.4 オブジェクト、型および参照カウント . . . 8
1.5 例外 . . . 13
1.6 Python の埋め込み . . . 15
1.7 デバッグ版ビルド (Debugging Builds) . . . 16

第 2 章　C API の安定性 17
2.1 Unstable C API . . . 17
2.2 安定 ABI (Stable Appliction Binary Interface) . . . 18
2.3 プラットフォームで考慮すべき点 . . . 20
2.4 限定版 API の内容 . . . 20

第 3 章　超高水準レイヤ 53

第 4 章　参照カウント 59

第 5 章　例外処理 63
5.1 出力とクリア . . . 63
5.2 例外の送出 . . . 64
5.3 警告 . . . 68
5.4 エラーインジケータの問い合わせ . . . 69
5.5 シグナルハンドリング . . . 73
5.6 例外クラス . . . 75
5.7 例外オブジェクト . . . 75
5.8 Unicode 例外オブジェクト . . . 76
5.9 再帰の管理 . . . 78
5.10 標準例外 . . . 79
5.11 標準警告カテゴリ . . . 81

第 6 章　ユーティリティ 83
6.1 オペレーティングシステム関連のユーティリティ . . . 83

6.2 システム関数 . . . 87
6.3 プロセス制御 . . . 90
6.4 モジュールのインポート . . . 90
6.5 データ整列化 (data marshalling) のサポート . . . 95
6.6 引数の解釈と値の構築 . . . 96
6.7 文字列の変換と書式化 . . . 108
6.8 PyHash API . . . 110
6.9 リフレクション . . . 111
6.10 codec レジストリとサポート関数 . . . 111
6.11 Support for Perf Maps . . . 114

第 7 章 抽象オブジェクトレイヤ (Abstract Objects Layer) 117
7.1 オブジェクトプロトコル (object protocol) . . . 117
7.2 Call プロトコル . . . 124
7.3 数値型プロトコル (number protocol) . . . 130
7.4 シーケンス型プロトコル (sequence protocol) . . . 134
7.5 マップ型プロトコル (mapping protocol) . . . 137
7.6 イテレータプロトコル (iterator protocol) . . . 138
7.7 バッファプロトコル (buffer Protocol) . . . 140
7.8 古いバッファプロトコル . . . 149

第 8 章 具象オブジェクト (concrete object) レイヤ 151
8.1 基本オブジェクト (fundamental object) . . . 151
8.2 数値型オブジェクト (numeric object) . . . 159
8.3 シーケンスオブジェクト (sequence object) . . . 170
8.4 Container オブジェクト . . . 204
8.5 Function オブジェクト . . . 211
8.6 その他のオブジェクト . . . 221

第 9 章 初期化 (initialization)、終了処理 (finalization)、スレッド 255
9.1 Python 初期化以前 . . . 255
9.2 グローバルな設定変数 . . . 256
9.3 インタープリタの初期化と終了処理 . . . 260
9.4 プロセスワイドのパラメータ . . . 262
9.5 スレッド状態 (thread state) とグローバルインタプリタロック (global interpreter lock) . . . 267
9.6 サブインタプリタサポート . . . 277
9.7 非同期通知 . . . 282
9.8 プロファイルとトレース (profiling and tracing) . . . 283
9.9 高度なデバッガサポート (advanced debugger support) . . . 285
9.10 スレッドローカルストレージのサポート . . . 285

第 10 章 Python 初期化設定 289
10.1 使用例 . . . 289
10.2 PyWideStringList . . . 290
10.3 PyStatus . . . 291

10.4 PyPreConfig . . . 293
10.5 Preinitialize Python with PyPreConfig . . . 295
10.6 PyConfig . . . 296
10.7 Initialization with PyConfig . . . 310
10.8 Isolated Configuration . . . 312
10.9 Python Configuration . . . 313
10.10 Python Path Configuration . . . 313
10.11 Py_RunMain() . . . 315
10.12 Py_GetArgcArgv() . . . 315
10.13 Multi-Phase Initialization Private Provisional API . . . 315

第 11 章　メモリ管理 319
11.1 概要 . . . 319
11.2 Allocator Domains . . . 320
11.3 生メモリインターフェース . . . 321
11.4 メモリインターフェース . . . 322
11.5 オブジェクトアロケータ . . . 323
11.6 Default Memory Allocators . . . 325
11.7 メモリアロケータをカスタマイズする . . . 325
11.8 Debug hooks on the Python memory allocators . . . 327
11.9 pymalloc アロケータ . . . 329
11.10 tracemalloc C API . . . 330
11.11 使用例 . . . 331

第 12 章　オブジェクト実装サポート (object implementation support) 333
12.1 オブジェクトをヒープ上にメモリ確保する . . . 333
12.2 共通のオブジェクト構造体 (common object structure) . . . 334
12.3 Type Object Structures . . . 346
12.4 循環参照ガベージコレクションをサポートする . . . 394

第 13 章　API と ABI のバージョニング 399

付録 A 章 用語集 401

付録 B 章 このドキュメントについて 423
B.1 Python ドキュメントへの貢献者 . . . 423

付録 C 章 歴史とライセンス 425
C.1 Python の歴史 . . . 425
C.2 Terms and conditions for accessing or otherwise using Python . . . 426
C.3 Licenses and Acknowledgements for Incorporated Software . . . 431

付録 D 章 Copyright 449

索引 451

索引 451

このマニュアルでは、拡張モジュールを書いたり Python インタプリタをアプリケーションに埋め込んだりしたい C/C++ プログラマが利用できる API について述べています。extending-index は拡張モジュールを書く際の一般的な決まりごとについて記述していますが、API の詳細までは記述していないので、このドキュメントが手引きになります。

第

ONE

はじめに

Python のアプリケーションプログラマ用インタフェース (Application Programmer's Interface, API) は、Python インタプリタに対する様々なレベルでのアクセス手段を C や C++ のプログラマに提供しています。この API は通常 C++ からも全く同じように利用できるのですが、簡潔な呼び名にするために Python/C API と名づけられています。根本的に異なる二つの目的から、Python/C API が用いられます。第一は、特定用途の **拡張モジュール** *(extension module)* 、すなわち Python インタプリタを拡張する C で書かれたモジュールを記述する、という目的です。第二は、より大規模なアプリケーション内で Python を構成要素 (component) として利用するという目的です; このテクニックは、一般的にはアプリケーションへの Python の埋め込み (*embedding*) と呼びます。

拡張モジュールの作成は比較的わかりやすいプロセスで、" 手引書 (cookbook)" 的なアプローチでうまく実現できます。作業をある程度まで自動化してくれるツールもいくつかあります。一方、他のアプリケーションへの Python の埋め込みは、Python ができてから早い時期から行われてきましたが、拡張モジュールの作成に比べるとやや難解です。

多くの API 関数は、Python の埋め込みであるか拡張であるかに関わらず役立ちます; とはいえ、Python を埋め込んでいるほとんどのアプリケーションは、同時に自作の拡張モジュールも提供する必要が生じることになるでしょうから、Python を実際にアプリケーションに埋め込んでみる前に拡張モジュールの書き方に詳しくなっておくのはよい考えだと思います。

1.1 コーディング基準

CPython に含める C コードを書いている場合は、**PEP 7** のガイドラインと基準に従わなければ **なりません** 。このガイドラインは、コントリビュート対象の Python のバージョンに関係無く適用されます。自身のサードパーティーのモジュールでは、それをいつか Python にコントリビュートするつもりでなければ、この慣習に従う必要はありません。

1.2 インクルードファイル

Python/C API を使うために必要な、関数、型およびマクロの全ての定義をインクルードするには、以下の行:

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

をソースコードに記述します。この行を記述すると、標準ヘッダ: `<stdio.h>`, `<string.h>`, `<errno.h>`,

<limits.h>, <assert.h>, <stdlib.h> を (利用できれば) インクルードします。

> **注釈**
>
> Python は、システムによっては標準ヘッダの定義に影響するようなプリプロセッサ定義を行っているので、Python.h をいずれの標準ヘッダよりも前にインクルード **せねばなりません** 。
>
> Python.h をインクルードする前に、常に PY_SSIZE_T_CLEAN を定義することが推奨されます。このマクロの解説については 引数の解釈と値の構築 を参照してください。

Python.h で定義されている、ユーザから見える名前全て (Python.h がインクルードしている標準ヘッダの名前は除きます) には、接頭文字列 Py または _Py が付きます。_Py で始まる名前は Python 実装で内部使用するための名前で、拡張モジュールの作者は使ってはなりません。構造体のメンバには予約済みの接頭文字列はありません。

> **注釈**
>
> API のユーザは、Py や _Py で始まる名前を定義するコードを絶対に書いてはなりません。後からコードを読む人を混乱させたり、将来の Python のバージョンで同じ名前が定義されて、ユーザの書いたコードの可搬性を危うくする可能性があります。

ヘッダファイル群は通常 Python と共にインストールされます。Unix では *prefix*/include/pythonversion/ および *exec_prefix*/include/pythonversion/ に置かれます。prefix と exec_prefix は Python をビルドする際の **configure** スクリプトに与えたパラメタに対応し、*version* は '%d.%d' % sys.version_info[:2] に対応します。Windows では、ヘッダは *prefix*/include に置かれます。prefix はインストーラに指定したインストールディレクトリです。

ヘッダをインクルードするには、各ヘッダの入ったディレクトリ (別々のディレクトリの場合は両方) を、コンパイラがインクルードファイルを検索するためのパスに入れます。親ディレクトリをサーチパスに入れて、#include <pythonX.Y/Python.h> のようにしては **なりません** ; prefix 内のプラットフォームに依存しないヘッダは、exec_prefix からプラットフォーム依存のヘッダをインクルードしているので、このような操作を行うと複数のプラットフォームでのビルドができなくなります。

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++.

1.3 便利なマクロ

Python のヘッダーファイルには便利なマクロがいくつか定義されています。多くのマクロは、それが役に立つところ (例えば、*Py_RETURN_NONE*) の近くに定義があります。より一般的な使われかたをする他のマクロはこれらのヘッダーファイルに定義されています。ただし、ここで完全に列挙されているとは限りません。

PyMODINIT_FUNC

Declare an extension module PyInit initialization function. The function return type is

*PyObject**. The macro declares any special linkage declarations required by the platform, and for C++ declares the function as `extern "C"`.

The initialization function must be named `PyInit_`*name*, where *name* is the name of the module, and should be the only non-`static` item defined in the module file. Example:

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

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

`Py_ABS`(x)

x の絶対値を返します。

Added in version 3.3.

`Py_ALWAYS_INLINE`

Ask the compiler to always inline a static inline function. The compiler can ignore it and decides 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; }
```

Added in version 3.11.

`Py_CHARMASK`(c)

引数は文字か、[-128, 127] あるいは [0, 255] の範囲の整数でなければなりません。このマクロは **符号なし文字** にキャストした c を返します。

Py_DEPRECATED(version)

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

以下はプログラム例です:

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

バージョン 3.8 で変更: MSVC サポートが追加されました。

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)

x と y の最大値を返します。

Added in version 3.3.

Py_MEMBER_SIZE(type, member)

(type) 構造体の member のサイズをバイト単位で返します。

Added in version 3.6.

Py_MIN(x, y)

x と y の最小値を返します。

Added in 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).

使い方:

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

Added in version 3.11.

Py_STRINGIFY(x)

x を C 文字列へ変換します。例えば、Py_STRINGIFY(123) は "123" を返します。

Added in 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.

Added in 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; }`.

Added in 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**.

以下はプログラム例です:

```
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**.

以下はプログラム例です:

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

(次のページに続く)

(前のページからの続き)

```
    {NULL, NULL}
};
```

1.4 オブジェクト、型および参照カウント

Python/C API 関数は、*PyObject** 型の一つ以上の引数と戻り値を持ちます。この型は、任意の Python オブジェクトを表現する不透明 (opaque) なデータ型へのポインタです。Python 言語は、全ての Python オブジェクト型をほとんどの状況 (例えば代入、スコープ規則 (scope rule)、引数渡し) で同様に扱います。ほとんど全ての Python オブジェクトはヒープ (heap) 上に置かれます: このため、*PyObject* 型のオブジェクトは、自動記憶 (automatic) としても静的記憶 (static) としても宣言できません。*PyObject** 型のポインタ変数のみ宣言できます。唯一の例外は、型オブジェクトです; 型オブジェクトはメモリ解放 (deallocate) してはならないので、通常は静的記憶の *PyTypeObject* オブジェクトにします。

全ての Python オブジェクトには (Python 整数型ですら) 型 (*type*) と参照カウント (*reference count*) があります。あるオブジェクトの型は、そのオブジェクトがどの種類のオブジェクトか (例えば整数、リスト、ユーザ定義関数、など; その他多数については types で説明しています) を決定します。よく知られている型については、各々マクロが存在して、あるオブジェクトがその型かどうか調べられます; 例えば、`PyList_Check(a)` は、*a* で示されたオブジェクトが Python リスト型のとき (かつそのときに限り) 真値を返します。

1.4.1 参照カウント法

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.

参照カウントの詳細

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*.

逆に、ある関数呼び出しで、あるオブジェクトへの参照を呼び出される関数に渡す際には、二つの可能性: 関数がオブジェクトへの参照を **盗み取る** (steal) 場合と、そうでない場合があります。**参照を盗む** とは、関数に参照を渡したときに、参照の所有者がその関数になったと仮定し、関数の呼び出し元には所有権がなくなるということです。

参照を盗み取る関数はほとんどありません; 例外としてよく知られているのは、*PyList_SetItem()* と *PyTuple_SetItem()* で、これらはシーケンスに入れる要素に対する参照を盗み取ります (しかし、要素の入る先のタプルやリストの参照は盗み取りません!)。これらの関数は、リストやタプルの中に新たに作成されたオブジェクトを入れていく際の常套的な書き方をしやすくするために、参照を盗み取るように設計されています; 例えば、`(1, 2, "three")` というタプルを生成するコードは以下のようになります (とりあえず例外処理のことは忘れておきます; もっとよい書き方を後で示します):

```
PyObject *t;

t = PyTuple_New(3);
```

(次のページに続く)

(前のページからの続き)

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

ここで、*PyLong_FromLong()* は新しい参照を返し、すぐに *PyTuple_SetItem()* に盗まれます。参照が盗まれた後もそのオブジェクトを利用したい場合は、参照盗む関数を呼び出す前に、*Py_INCREF()* を利用してもう一つの参照を取得してください。

ちなみに、*PyTuple_SetItem()* はタプルに値をセットするための **唯一の** 方法です; タプルは変更不能なデータ型なので、*PySequence_SetItem()* や *PyObject_SetItem()* を使うと上の操作は拒否されてしまいます。自分でタプルの値を入れていくつもりなら、*PyTuple_SetItem()* だけしか使えません。

同じく、リストに値を入れていくコードは *PyList_New()* と *PyList_SetItem()* で書けます。

しかし実際には、タプルやリストを生成して値を入れる際には、上記のような方法はほとんど使いません。より汎用性のある関数、*Py_BuildValue()* があり、ほとんどの主要なオブジェクトをフォーマット文字列 *format string* の指定に基づいて C の値から生成できます。例えば、上の二種類のコードブロックは、以下のように置き換えられます (エラーチェックにも配慮しています):

```
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);
```

(次のページに続く)

(前のページからの続き)

```
            return -1;
        }
        Py_DECREF(index);
    }
    return 0;
}
```

関数の戻り値の場合には、状況は少し異なります。ほとんどの関数については、参照を渡してもその参照に対する所有権が変わることがない一方で、あるオブジェクトに対する参照を返すような多くの関数は、参照に対する所有権を呼び出し側に与えます。理由は簡単です: 多くの場合、関数が返すオブジェクトはその場で (on the fly) 生成されるため、呼び出し側が得る参照は生成されたオブジェクトに対する唯一の参照になるからです。従って、*PyObject_GetItem()* や *PySequence_GetItem()* のように、オブジェクトに対する参照を返す汎用の関数は、常に新たな参照を返します (呼び出し側が参照の所有者になります)。

重要なのは、関数が返す参照の所有権を持てるかどうかは、どの関数を呼び出すかだけによる、と理解することです --- 関数呼び出し時の **お飾り** (関数に引数として渡したオブジェクトの型) は **この問題には関係ありません!** 従って、*PyList_GetItem()* を使ってリスト内の要素を得た場合には、参照の所有者にはなりません --- が、同じ要素を同じリストから *PySequence_GetItem()* (図らずもこの関数は全く同じ引数をとります) を使って取り出すと、返されたオブジェクトに対する参照を得ます。

以下は、整数からなるリストに対して各要素の合計を計算する関数をどのようにして書けるかを示した例です; 一つは *PyList_GetItem()* を使っていて、もう一つは *PySequence_GetItem()* を使っています。

```
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;
```

(次のページに続く)

(前のページからの続き)

```
}
```

```
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;
}
```

1.4.2 型

他にも Python/C API において重要な役割を持つデータ型がいくつかあります; ほとんどは `int`, `long`, `double`, および `char*` といった、単なる C のデータ型です。また、モジュールで公開している関数を列挙する際に用いられる静的なテーブルや、新しいオブジェクト型におけるデータ属性を記述したり、複素数の値を記述したりするために構造体をいくつか使っています。これらの型については、その型を使う関数とともに説明してゆきます。

type **Py_ssize_t**

> **次に属します:** 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*.

1.5 例外

Python プログラマは、特定のエラー処理が必要なときだけしか例外を扱う必要はありません; 処理しなかった例外は、処理の呼び出し側、そのまた呼び出し側、といった具合に、トップレベルのインタプリタ層まで自動的に伝播します。インタプリタ層は、スタックトレースバックと合わせて例外をユーザに報告します。

ところが、C プログラマの場合、エラーチェックは常に明示的に行わねばなりません。Python/C API の全ての関数は、関数のドキュメントで明確に説明がない限り例外を発行する可能性があります。一般的な話として、ある関数が何らかのエラーに遭遇すると、関数は例外を設定して、関数内における参照の所有権を全て放棄し、エラー値 (error indicator) を返します。ドキュメントに書かれてない場合、このエラー値は関数の戻り値の型によって、`NULL` か `-1` のどちらかになります。いくつかの関数ではブール型で真/偽を返し、偽はエラーを示します。きわめて少数の関数では明確なエラー指標を返さなかったり、あいまいな戻り値を返したりするので、*PyErr_Occurred()* で明示的にエラーテストを行う必要があります。これらの例外は常に明示的にドキュメント化されます。

例外時の状態情報 (exception state) は、スレッド単位に用意された記憶領域 (per-thread storage) 内で管理されます (この記憶領域は、スレッドを使わないアプリケーションではグローバルな記憶領域と同じです)。一つのスレッドは二つの状態のどちらか: 例外が発生したか、まだ発生していないか、をとります。関数 *PyErr_Occurred()* を使うと、この状態を調べられます: この関数は例外が発生した際にはその例外型オブジェクトに対する借用参照 (borrowed reference) を返し、そうでないときには `NULL` を返します。例外状態を設定する関数は数多くあります: *PyErr_SetString()* はもっともよく知られている (が、もっとも汎用性のない) 例外を設定するための関数で、*PyErr_Clear()* は例外状態情報を消し去る関数です。

完全な例外状態情報は、3 つのオブジェクト: 例外の型、例外の値、そしてトレースバック、からなります (どのオブジェクトも `NULL` を取り得ます)。これらの情報は、Python の `sys.exc_info()` の結果と同じ意味を持ちます; とはいえ、C と Python の例外状態情報は全く同じではありません: Python における例外オブジェクトは、Python の `try` ... `except` 文で最近処理したオブジェクトを表す一方、C レベルの例外状態情報が存続するのは、渡された例外情報を `sys.exc_info()` その他に転送するよう取り計らう Python のバイトコードインタプリタのメインループに到達するまで、例外が関数の間で受け渡しされている間だけです。

Python 1.5 からは、Python で書かれたコードから例外状態情報にアクセスする方法として、推奨されていてスレッドセーフな方法は `sys.exc_info()` になっているので注意してください。この関数は Python コードの実行されているスレッドにおける例外状態情報を返します。また、これらの例外状態情報に対するアクセス手段は、両方とも意味づけ (semantics) が変更され、ある関数が例外を捕捉すると、その関数を実行しているスレッドの例外状態情報を保存して、呼び出し側の例外状態情報を維持するようになりました。この変更によって、無害そうに見える関数が現在扱っている例外を上書きすることで引き起こされる、例外処理コードでよくおきていたバグを抑止しています; また、トレースバック内のスタックフレームで参照されているオブジェクトがしばしば不必要に寿命を永らえていたのをなくしています。

一般的な原理として、ある関数が別の関数を呼び出して何らかの作業をさせるとき、呼び出し先の関数が例外を送出していないか調べなくてはならず、もし送出していれば、その例外状態情報は呼び出し側に渡されなければなりません。呼び出し元の関数はオブジェクト参照の所有権をすべて放棄し、エラー指標を返さなくてはなりませんが、余計に例外を設定する必要は **ありません** --- そんなことをすれば、たった今送出されたばかりの例外を上書きしてしまい、エラーの原因そのものに関する重要な情報を失うことになります。

A simple example of detecting exceptions and passing them on is shown in the `sum_sequence()` example above. It so happens that this example doesn't need to clean up any owned references when it detects

an error. The following example function shows some error cleanup. First, to remind you why you like Python, we show the equivalent Python code:

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

以下は対応するコードを C で完璧に書いたものです:

```
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 */
```

(次のページに続く)

(前のページからの続き)

```
 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 */
}
```

なんとこの例は C で goto 文を使うお勧めの方法まで示していますね! この例では、特定の例外を処理するために *PyErr_ExceptionMatches()* および *PyErr_Clear()* をどう使うかを示しています。また、所有権を持っている参照で、値が NULL になるかもしれないものを捨てるために *Py_XDECREF()* をどう使うかも示しています (関数名に 'X' が付いていることに注意してください; *Py_DECREF()* は NULL 参照に出くわすとクラッシュします)。正しく動作させるためには、所有権を持つ参照を保持するための変数を NULL で初期化することが重要です; 同様に、あらかじめ戻り値を定義する際には値を -1 (失敗) で初期化しておいて、最後の関数呼び出しまでうまくいった場合にのみ 0 (成功) に設定します。

1.6 Python の埋め込み

Python インタプリタの埋め込みを行う人 (いわば拡張モジュールの書き手の対極) が気にかけなければならない重要なタスクは、Python インタプリタの初期化処理 (initialization)、そしておそらくは終了処理 (finalization) です。インタプリタのほとんどの機能は、インタプリタの起動後しか使えません。

基本的な初期化処理を行う関数は *Py_Initialize()* です。この関数はロード済みのモジュールからなるテーブルを作成し、土台となるモジュール builtins, __main__, および sys を作成します。また、モジュール検索パス (sys.path) の初期化も行います。

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*.

ほとんどのシステムでは (特に Unix と Windows は、詳細がわずかに異なりはしますが)、*Py_Initialize()* は標準の Python インタプリタ実行形式の場所に対する推定結果に基づいて、Python のライブラリが Python インタプリタ実行形式からの相対パスで見つかるという仮定の下にモジュール検索パスを計算します。とりわけこの検索では、シェルコマンド検索パス (環境変数 PATH) 上に見つかった python という名前の実行ファイルの置かれているディレクトリの親ディレクトリからの相対で、lib/python*X.Y* という名前のディレクトリを探します。

例えば、Python 実行形式が /usr/local/bin/python で見つかったとすると、ライブラリが /usr/local/lib/python*X.Y* にあるものと仮定します。(実際には、このパスは "フォールバック (fallback)" のライブラリ位置でもあり、python が PATH 上に無い場合に使われます。) ユーザは PYTHONHOME を設定することでこの動作をオーバーライドしたり、PYTHONPATH を設定して追加のディレクトリを標準モジュール検索パスの前

に挿入したりできます。

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

たまに、Python を初期化前の状態にもどしたいことがあります。例えば、あるアプリケーションでは実行を最初からやりなおし (start over) させる (*Py_Initialize()* をもう一度呼び出させる) ようにしたいかもしれません。あるいは、アプリケーションが Python を一旦使い終えて、Python が確保したメモリを解放させたいかもしれません。*Py_FinalizeEx()* を使うとこうした処理を実現できます。また、関数 *Py_IsInitialized()* は、Python が現在初期化済みの状態にある場合に真を返します。これらの関数についてのさらなる情報は、後の章で説明します。*Py_FinalizeEx()* が Python インタプリタに確保された全てのメモリを **解放するわけではない** ことに注意してください。例えば、拡張モジュールによって確保されたメモリは、現在のところ解放する事ができません。

1.7 デバッグ版ビルド (Debugging Builds)

インタプリタと拡張モジュールに対しての追加チェックをするためのいくつかのマクロを有効にして Python をビルドすることができます。これらのチェックは、実行時に大きなオーバーヘッドを生じる傾向があります。なので、デフォルトでは有効にされていません。

Python デバッグ版ビルドの全ての種類のリストが、Python ソース配布 (source distribution) の中の `Misc/SpecialBuilds.txt` にあります。参照カウントのトレース、メモリアロケータのデバッグ、インタプリタのメインループの低レベルプロファイリングが利用可能です。よく使われるビルドについてのみ、この節の残りの部分で説明します。

`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.

`Py_TRACE_REFS` を宣言すると、参照トレースが有効になります (`configure --with-trace-refs` **オプション** を参照してください)。全ての *PyObject* に二つのフィールドを追加することで、使用中のオブジェクトの循環二重連結リストが管理されます。全ての割り当て (allocation) がトレースされます。終了時に、全ての残っているオブジェクトが表示されます。(インタラクティブモードでは、インタプリタによる文の実行のたびに表示されます。)

より詳しい情報については、Python のソース配布 (source distribution) の中の `Misc/SpecialBuilds.txt` を参照してください。

第

TWO

C API の安定性

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 **プラットフォームで考慮すべき点** 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 安定 ABI (Stable Appliction Binary Interface)

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`

このマクロを `Python.h` をインクルードする前に定義することで、Limited API のみを使用することを選択し、Limited API バージョンを選択することができます。

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.

`PY_VERSION_HEX` マクロを直接使うのではなく、将来の Python のバージョンでコンパイルするときの安定性のために、最小のマイナーバージョン (例えば、Python 3.10 なら `0x030A0000`) をハードコードします。

また、`Py_LIMITED_API` を `3` に定義することができます。これは `0x03020000` (Python 3.2, Limited API が導入されたバージョン) と同じように動作します。

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.

> **注釈**
>
> 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.

Windows では、Stable ABI を使用する拡張機能は、`python39.dll` のようなバージョン固有のライブラリではなく、`python3.dll` に対してリンクする必要があります。

いくつかのプラットフォームでは、Python は `abi3` タグで名付けられた共有ライブラリファイルを探して読み込みます (例: `mymodule.abi3.so`)。このような拡張モジュールが Stable ABI に適合しているかどうかはチェックされません。ユーザー (またはそのパッケージングツール) は、たとえば 3.10+ Limited API でビルドされた拡張モジュールが、それ以下のバージョンの Python にインストールされないことを確認する必要が

あります。

Stable ABI に含まれるすべての関数は、マクロとしてだけでなく、Python の共有ライブラリの関数として存在します。そのため、C プリプロセッサを使用しない言語から使用することができます。

2.2.3 API スコープとパフォーマンスの制限

Limited API の目標は、フル C API で可能なすべてのことを実現することですが、おそらく性能上の制約があります。

例えば、*PyList_GetItem()* は利用可能ですが、その “unsafe” マクロの変種 *PyList_GET_ITEM()* は利用できません。このマクロは、リストオブジェクトのバージョン固有の実装の詳細に依存することができるため、より高速に処理することができます。

Py_LIMITED_API を定義しないと、いくつかの C API 関数がインライン化されたり、マクロに置き換わったりします。Py_LIMITED_API を定義すると、このインライン化が無効になり、Python のデータ構造が改善されても安定した動作が可能になりますが、性能が低下する可能性があります。

Py_LIMITED_API の定義を省くことで、Limited API 拡張をバージョン固有の ABI でコンパイルすることが可能です。これにより、その Python のバージョンでパフォーマンスを向上させることができますが、互換性は制限されます。Py_LIMITED_API でコンパイルすると、バージョンに依存しない拡張機能が利用できない場合、例えば、次期 Python バージョンのプレリリースに対応した拡張モジュールを配布することができるようになります。

2.2.4 制限付き API の注意点

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.

Py_LIMITED_API が防げない問題の 1 つは、Python の下位バージョンでは無効な引数を持つ関数を呼び出すことです。例えば、引数に NULL を受け取る関数を考えてみましょう。Python 3.9 では NULL はデフォルトの挙動を選択しますが、Python 3.8 ではこの引数は直接使用され、NULL の参照外れを起こしクラッシュします。同様の引数は、構造体のフィールドに対しても機能します。

もう一つの問題は、一部の構造体フィールドが Limited API の一部であるにもかかわらず、Py_LIMITED_API が定義されたときに現在非表示になっていないことです。

これらの理由から、私たちは拡張モジュールがサポートする **すべての** マイナーな Python バージョンでテストすること、そしてできれば **最も低い** バージョンでビルドすることを推奨します。

また、使用するすべての API のドキュメントを確認し、それが明示的に Limited API の一部であるかどうかをチェックすることをお勧めします。Py_LIMITED_API が定義されていても、技術的な理由で (あるいはバグとして意図せず) いくつかのプライベート宣言が公開されることがあります。

Python 3.8 で Py_LIMITED_API をコンパイルすると、その拡張モジュールは Python 3.12 で動作しますが、必ずしも Python 3.12 で **コンパイル** できるとは限らないことに注意してください。特に、Limited API の一部は、Stable ABI が安定している限り、非推奨で削除されるかもしれません。

2.3 プラットフォームで考慮すべき点

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

特定のプラットフォーム上のすべての Python バージョンが安定版 ABI を破壊しない方法でビルドされていることを保証するのは、Python の各特定配布者の責任です。これは `python.org` や多くのサードパーティーの配布元からの Windows と macOS のリリースの場合です。

2.4 限定版 API の内容

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*
- *PyCFunctionWithKeywords*
- PyCFunction_Call()
- 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_GetItemString()*
- *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_AcquireLock()*
- *PyEval_AcquireThread()*
- PyEval_CallFunction()
- PyEval_CallMethod()
- PyEval_CallObjectWithKeywords()
- *PyEval_EvalCode()*
- *PyEval_EvalCodeEx()*
- *PyEval_EvalFrame()*
- *PyEval_EvalFrameEx()*
- *PyEval_GetBuiltins()*
- *PyEval_GetFrame()*
- *PyEval_GetFuncDesc()*
- *PyEval_GetFuncName()*
- *PyEval_GetGlobals()*
- *PyEval_GetLocals()*
- *PyEval_InitThreads()*

- *PyEval_ReleaseLock()*
- *PyEval_ReleaseThread()*
- *PyEval_RestoreThread()*
- *PyEval_SaveThread()*
- *PyEval_ThreadsInitialized()*
- 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_AppendInittab()*
- *PyImport_ExecCodeModule()*
- *PyImport_ExecCodeModuleEx()*
- *PyImport_ExecCodeModuleObject()*

- *PyImport_ExecCodeModuleWithPathnames()*
- *PyImport_GetImporter()*
- *PyImport_GetMagicNumber()*
- *PyImport_GetMagicTag()*
- *PyImport_GetModule()*
- *PyImport_GetModuleDict()*
- *PyImport_Import()*
- *PyImport_ImportFrozenModule()*
- *PyImport_ImportFrozenModuleObject()*
- *PyImport_ImportModule()*
- *PyImport_ImportModuleLevel()*
- *PyImport_ImportModuleLevelObject()*
- *PyImport_ImportModuleNoBlock()*
- *PyImport_ReloadModule()*
- *PyIndex_Check()*
- *PyInterpreterState*
- *PyInterpreterState_Clear()*
- *PyInterpreterState_Delete()*
- *PyInterpreterState_Get()*
- *PyInterpreterState_GetDict()*
- *PyInterpreterState_GetID()*
- *PyInterpreterState_New()*
- *PyIter_Check()*
- *PyIter_Next()*
- *PyIter_Send()*
- PyListIter_Type
- PyListRevIter_Type
- *PyList_Append()*
- *PyList_AsTuple()*

- *PyList_GetItem()*
- *PyList_GetSlice()*
- *PyList_Insert()*
- *PyList_New()*
- *PyList_Reverse()*
- *PyList_SetItem()*
- *PyList_SetSlice()*
- *PyList_Size()*
- *PyList_Sort()*
- *PyList_Type*
- *PyLongObject*
- PyLongRangeIter_Type
- *PyLong_AsDouble()*
- *PyLong_AsLong()*
- *PyLong_AsLongAndOverflow()*
- *PyLong_AsLongLong()*
- *PyLong_AsLongLongAndOverflow()*
- *PyLong_AsSize_t()*
- *PyLong_AsSsize_t()*
- *PyLong_AsUnsignedLong()*
- *PyLong_AsUnsignedLongLong()*
- *PyLong_AsUnsignedLongLongMask()*
- *PyLong_AsUnsignedLongMask()*
- *PyLong_AsVoidPtr()*
- *PyLong_FromDouble()*
- *PyLong_FromLong()*
- *PyLong_FromLongLong()*
- *PyLong_FromSize_t()*
- *PyLong_FromSsize_t()*

- *PyLong_FromString()*
- *PyLong_FromUnsignedLong()*
- *PyLong_FromUnsignedLongLong()*
- *PyLong_FromVoidPtr()*
- *PyLong_GetInfo()*
- *PyLong_Type*
- PyMap_Type
- *PyMapping_Check()*
- *PyMapping_GetItemString()*
- *PyMapping_HasKey()*
- *PyMapping_HasKeyString()*
- *PyMapping_Items()*
- *PyMapping_Keys()*
- *PyMapping_Length()*
- *PyMapping_SetItemString()*
- *PyMapping_Size()*
- *PyMapping_Values()*
- *PyMem_Calloc()*
- *PyMem_Free()*
- *PyMem_Malloc()*
- *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_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_AsCharBuffer()*
- *PyObject_AsFileDescriptor()*
- *PyObject_AsReadBuffer()*
- *PyObject_AsWriteBuffer()*
- *PyObject_Bytes()*
- *PyObject_Call()*
- *PyObject_CallFunction()*
- *PyObject_CallFunctionObjArgs()*
- *PyObject_CallMethod()*
- *PyObject_CallMethodObjArgs()*
- *PyObject_CallNoArgs()*
- *PyObject_CallObject()*
- *PyObject_Calloc()*
- *PyObject_CheckBuffer()*
- *PyObject_CheckReadBuffer()*
- *PyObject_ClearWeakRefs()*
- *PyObject_CopyData()*
- *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_GetTypeData()*
- *PyObject_HasAttr()*
- *PyObject_HasAttrString()*
- *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_AddWarnOption()*

- *PySys_AddWarnOptionUnicode()*
- *PySys_AddXOption()*
- *PySys_FormatStderr()*
- *PySys_FormatStdout()*
- *PySys_GetObject()*
- *PySys_GetXOptions()*
- PySys_HasWarnOptions()
- *PySys_ResetWarnOptions()*
- *PySys_SetArgv()*
- *PySys_SetArgvEx()*
- *PySys_SetObject()*
- *PySys_SetPath()*
- *PySys_WriteStderr()*
- *PySys_WriteStdout()*
- *PyThreadState*
- *PyThreadState_Clear()*
- *PyThreadState_Delete()*
- *PyThreadState_Get()*
- *PyThreadState_GetDict()*
- *PyThreadState_GetFrame()*
- *PyThreadState_GetID()*
- *PyThreadState_GetInterpreter()*
- *PyThreadState_New()*
- *PyThreadState_SetAsyncExc()*
- *PyThreadState_Swap()*
- PyThread_GetInfo()
- *PyThread_ReInitTLS()*
- PyThread_acquire_lock()
- PyThread_acquire_lock_timed()

- PyThread_allocate_lock()
- *PyThread_create_key()*
- *PyThread_delete_key()*
- *PyThread_delete_key_value()*
- PyThread_exit_thread()
- PyThread_free_lock()
- *PyThread_get_key_value()*
- PyThread_get_stacksize()
- PyThread_get_thread_ident()
- PyThread_get_thread_native_id()
- PyThread_init_thread()
- PyThread_release_lock()
- *PyThread_set_key_value()*
- PyThread_set_stacksize()
- PyThread_start_new_thread()
- *PyThread_tss_alloc()*
- *PyThread_tss_create()*
- *PyThread_tss_delete()*
- *PyThread_tss_free()*
- *PyThread_tss_get()*
- *PyThread_tss_is_created()*
- *PyThread_tss_set()*
- PyTraceBack_Here()
- PyTraceBack_Print()
- PyTraceBack_Type
- PyTupleIter_Type
- *PyTuple_GetItem()*
- *PyTuple_GetSlice()*
- *PyTuple_New()*

- *PyTuple_Pack()*
- *PyTuple_SetItem()*
- *PyTuple_Size()*
- *PyTuple_Type*
- *PyTypeObject*
- *PyType_ClearCache()*
- *PyType_FromMetaclass()*
- *PyType_FromModuleAndSpec()*
- *PyType_FromSpec()*
- *PyType_FromSpecWithBases()*
- *PyType_GenericAlloc()*
- *PyType_GenericNew()*
- *PyType_GetFlags()*
- *PyType_GetModule()*
- *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_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_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_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_IsInitialized()*
- *Py_IsNone()*
- *Py_IsTrue()*
- *Py_LeaveRecursiveCall()*
- *Py_Main()*
- Py_MakePendingCalls()
- *Py_NewInterpreter()*
- *Py_NewRef()*
- *Py_ReprEnter()*
- *Py_ReprLeave()*
- *Py_SetPath()*
- *Py_SetProgramName()*
- *Py_SetPythonHome()*
- Py_SetRecursionLimit()
- *Py_UCS4*
- *Py_UNBLOCK_THREADS*
- Py_UTF8Mode
- *Py_VaBuildValue()*

- *Py_Version*
- *Py_XNewRef()*
- *Py_buffer*
- Py_intptr_t
- *Py_ssize_t*
- Py_uintptr_t
- *allocfunc*
- *binaryfunc*
- *descrgetfunc*
- *descrsetfunc*
- *destructor*
- *getattrfunc*
- *getattrofunc*
- *getbufferproc*
- *getiterfunc*
- *getter*
- *hashfunc*
- *initproc*
- *inquiry*
- *iternextfunc*
- *lenfunc*
- *newfunc*
- *objobjargproc*
- *objobjproc*
- *releasebufferproc*
- *reprfunc*
- *richcmpfunc*
- *setattrfunc*
- *setattrofunc*

- *setter*
- *ssizeargfunc*
- *ssizeobjargproc*
- ssizessizeargfunc
- ssizessizeobjargproc
- symtable
- *ternaryfunc*
- *traverseproc*
- *unaryfunc*
- *vectorcallfunc*
- *visitproc*

第

THREE

超高水準レイヤ

この章の関数を使うとファイルまたはバッファにある Python ソースコードを実行できますが、より詳細なやり取りをインタプリタとすることはできないでしょう。

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

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

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

> **次に属します**: Stable ABI. The main program for the standard interpreter. This is made available for programs which embed Python. The *argc* and *argv* parameters should be prepared exactly as those which are passed to a C program's `main()` function (converted to wchar_t according to the user's locale). It is important to note that the argument list may be modified (but the contents of the strings pointed to by the argument list are not). The return value will be `0` if the interpreter exits normally (i.e., without an exception), `1` if the interpreter exits due to an exception, or `2` if the parameter list does not represent a valid Python command line.
>
> Note that if an otherwise unhandled `SystemExit` is raised, this function will not return `1`, but exit the process, as long as *PyConfig.inspect* is zero.

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

> **次に属します**: Stable ABI *(バージョン 3.8 より)*. Similar to *Py_Main()* but *argv* is an array of bytes strings.
>
> Added in version 3.8.

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

> 下記の *PyRun_AnyFileExFlags()* の *closeit* を `0` に、*flags* を `NULL` にして単純化したインターフェースです。

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

下記の *PyRun_AnyFileExFlags()* の *closeit* を `0` にして単純化したインターフェースです。

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

下記の *PyRun_AnyFileExFlags()* の *flags* を `NULL` にして単純化したインターフェースです。

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

fp が対話的デバイス (コンソールや端末入力あるいは Unix 仮想端末) と関連づけられたファイルを参照している場合は、*PyRun_InteractiveLoop()* の値を返します。それ以外の場合は、*PyRun_SimpleFile()* の結果を返します。*filename* はファイルシステムのエンコーディング (`sys.getfilesystemencoding()`) でデコードされます。*filename* が `NULL` ならば、この関数はファイル名として `"???"` を使います。*closeit* が真なら、ファイルは `PyRun_SimpleFileExFlags()` が処理を戻す前に閉じられます。

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

下記の *PyRun_SimpleStringFlags()* の *PyCompilerFlags** を `NULL` にして単純化したインタフェースです。

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

`__main__` モジュールの中で *flags* に従って *command* に含まれる Python ソースコードを実行します。`__main__` がまだ存在しない場合は作成されます。正常終了の場合は `0` を返し、また例外が発生した場合は `-1` を返します。エラーがあっても、例外情報を得る方法はありません。*flags* の意味については、後述します。

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)

下記の *PyRun_SimpleFileExFlags()* の *closeit* を `0` に、*flags* を `NULL` にして単純化したインターフェースです。

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

下記の *PyRun_SimpleFileExFlags()* の *flags* を `NULL` にして単純化したインターフェースです。

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

PyRun_SimpleStringFlags() と似ていますが、Python ソースコードをメモリ内の文字列ではなく *fp* から読み込みます。*filename* はそのファイルの名前でなければならず、**ファイルシステムのエンコーディングとエラーハンドラ** でデコードされます。*closeit* に真を指定した場合は、`PyRun_SimpleFileExFlags()` が処理を戻す前にファイルを閉じます。

> **注釈**
>
> Windows では、*fp* はバイナリモードで開くべきです (例えば `fopen(filename, "rb")`)。そうしない場合は、Python は行末が LF のスクリプトを正しく扱えないでしょう。

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

下記の *PyRun_InteractiveOneFlags()* の *flags* を NULL にして単純化したインターフェースです。

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

対話的デバイスに関連付けられたファイルから文を一つ読み込み、*flags* に従って実行します。sys.ps1 と sys.ps2 を使って、ユーザにプロンプトを表示します。*filename* は **ファイルシステムのエンコーディングとエラーハンドラ** でデコードされます。

入力が正常に実行されたときは 0 を返します。例外が発生した場合は -1 を返します。パースエラーの場合は Python の一部として配布されている errcode.h インクルードファイルにあるエラーコードを返します。(Python.h は errcode.h をインクルードしません。従って、必要な場合はその都度インクルードしなければならないことに注意してください。)

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

下記の *PyRun_InteractiveLoopFlags()* の *flags* を NULL にして単純化したインターフェースです。

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

対話的デバイスに関連付けられたファイルから EOF に達するまで文を読み込み実行します。sys.ps1 と sys.ps2 を使って、ユーザにプロンプトを表示します。*filename* は **ファイルシステムのエンコーディングとエラーハンドラ** でデコードされます。EOF に達すると 0 を返すか、失敗したら負の数を返します。

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

次に属します: Stable ABI. `int func(void)` というプロトタイプの関数へのポインタが設定できます。この関数は、Python のインタプリタのプロンプトがアイドル状態になりターミナルからのユーザの入力を待つようになったときに呼び出されます。返り値は無視されます。このフックを上書きすることで、Python のソースコードの中で Modules/_tkinter.c がやっているように、インタプリタのプロンプトと他のイベントループを統合できます。

バージョン 3.12 で変更: This function is only called from the *main interpreter*.

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

`char *func(FILE *stdin, FILE *stdout, char *prompt)` というプロトタイプの関数へのポインタが設定でき、デフォルトの関数を上書きすることでインタプリタのプロンプトへの入力を 1 行だけ読めます。この関数は、文字列 *prompt* が NULL でない場合は *prompt* を出力し、与えられた標準入力ファイルから入力を 1 行読み、結果の文字列を返すという動作が期待されています。例えば、readline モジュールはこのフックを設定して、行編集機能やタブ補完機能を提供しています。

返り値は *PyMem_RawMalloc()* または *PyMem_RawRealloc()* でメモリ確保した文字列、あるいはエラーが起きた場合には NULL でなければなりません。

バージョン 3.4 で変更: 返り値は、*PyMem_Malloc()* や *PyMem_Realloc()* ではなく、*PyMem_RawMalloc()* または *PyMem_RawRealloc()* でメモリ確保したものでなければなりません。

バージョン 3.12 で変更: This function is only called from the *main interpreter*.

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

戻り値: **新しい参照。** 下記の *PyRun_StringFlags()* の *flags* を `NULL` にして単純化したインターフェースです。

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

戻り値: **新しい参照。** オブジェクトの *globals* と *locals* で指定されるコンテキストで、コンパイラフラグに *flags* を設定した状態で、*str* にある Python ソースコードを実行します。*globals* は辞書でなければなりません; *locals* はマッピングプロトコルを実装したオブジェクトなら何でも構いません。引数 *start* はソースコードをパースするために使われるべき開始トークンを指定します。

コードを実行した結果を Python オブジェクトとして返します。または、例外が発生したならば `NULL` を返します。

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

戻り値: **新しい参照。** 下記の *PyRun_FileExFlags()* の *closeit* を `0` にし、*flags* を `NULL` にして単純化したインターフェースです。

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

戻り値: **新しい参照。** 下記の *PyRun_FileExFlags()* の *flags* を `NULL` にして単純化したインターフェースです。

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

戻り値: **新しい参照。** 下記の *PyRun_FileExFlags()* の *closeit* を `0` にして単純化したインターフェースです。

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

戻り値: **新しい参照。** *PyRun_StringFlags()* と似ていますが、Python ソースコードをメモリ内の文字列ではなく *fp* から読み込みます。*filename* はそのファイルの名前でなければならず、**ファイルシステムのエンコーディングとエラーハンドラ** でデコードされます。*closeit* に真を指定した場合は、*PyRun_FileExFlags()* が処理を戻す前にファイルを閉じます。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 下記の *Py_CompileStringFlags()* の *flags* を `NULL` にして単純化したインターフェースです。

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

戻り値: **新しい参照。** 下記の *Py_CompileStringExFlags()* の *optimize* を `-1` にして単純化したインターフェースです。

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

戻り値: 新しい参照。 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.

整数 *optimize* は、コンパイラの最適化レベルを指定します; `-1` は、インタプリタの `-O` オプションで与えられるのと同じ最適化レベルを選びます。明示的なレベルは、`0` (最適化なし、`__debug__` は真)、`1` (assert は取り除かれ、`__debug__` は偽)、`2` (docstring も取り除かれる) です。

Added in version 3.4.

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

戻り値: 新しい参照。 *Py_CompileStringObject()* と似ていますが、*filename* は **ファイルシステムのエンコーディングとエラーハンドラ** でデコードされたバイト文字列です。

Added in version 3.2.

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

戻り値: 新しい参照。次に属します: Stable ABI. *PyEval_EvalCodeEx()* のシンプルなインターフェースで、コードオブジェクトと、グローバル変数とローカル変数だけを受け取ります。他の引数には `NULL` が渡されます。

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)

戻り値: 新しい参照。次に属します: Stable ABI. 与えられた特定の環境で、コンパイル済みのコードオブジェクトを評価します。この環境はグローバル変数の辞書と、ローカル変数のマッピングオブジェクト、引数の配列、キーワードとデフォルト値、**キーワード専用** 引数のデフォルト値の辞書と、セルのクロージャタプルで構成されます。

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

戻り値: 新しい参照。次に属します: Stable ABI. 実行フレームを評価します。これは *PyEval_EvalFrameEx()* に対するシンプルなインターフェースで、後方互換性のためのものです。

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

戻り値: 新しい参照。次に属します: Stable ABI. Python のインタープリタの主要な、直接的な関数です。実行フレーム *f* に関連付けられたコードオブジェクトを実行します。バイトコードを解釈して、必要に応じて呼び出しを実行します。追加の *throwflag* 引数はほとんど無視できます。- もし true なら、すぐに例外を発生させます。これはジェネレータオブジェクトの `throw()` メソッドで利用されます。

バージョン 3.4 で変更: アクティブな例外を黙って捨てないことを保証するのに便利なように、この関数はデバッグアサーションを含むようになりました。

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

現在の評価フレームのフラグを変更します。成功したら true を、失敗したら false を返します。

int **Py_eval_input**

単独の式に対する Python 文法の開始記号で、*Py_CompileString()* と一緒に使います。

int **Py_file_input**

ファイルあるいは他のソースから読み込まれた文の並びに対する Python 文法の開始記号で、*Py_CompileString()* と一緒に使います。これは任意の長さの Python ソースコードをコンパイルするときに使う記号です。

int **Py_single_input**

単一の文に対する Python 文法の開始記号で、*Py_CompileString()* と一緒に使います。これは対話式のインタプリタループのための記号です。

struct **PyCompilerFlags**

コンパイラフラグを収めておくための構造体です。コードをコンパイルするだけの場合、この構造体が `int flags` として渡されます。コードを実行する場合には `PyCompilerFlags *flags` として渡されます。この場合、`from __future__ import` は *flags* の内容を変更できます。

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**

コンパイラフラグ。

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*.

バージョン 3.8 で変更: Added *cf_feature_version* field.

int **CO_FUTURE_DIVISION**

このビットを *flags* にセットすると、除算演算子 `/` は **PEP 238** による「真の除算 (true division)」として扱われます。

第

FOUR

参照カウント

このセクションにある関数とマクロは、Python オブジェクトの参照カウントの管理に使用されます。

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

Python オブジェクト *o* の参照カウントを取得します。

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.

オブジェクトの参照カウントを設定するためには、*Py_SET_REFCNT()* 関数を使用してください。

バージョン 3.10 で変更: *Py_REFCNT()* は static なインライン関数に変更されました。

バージョン 3.11 で変更: 引数の型は `const` *PyObject*`*` ではなくなりました。

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

オブジェクト *o* の参照カウントを *refcnt* に設定します。

Note that this function has no effect on immortal objects.

Added in version 3.9.

バージョン 3.12 で変更: 永続オブジェクトは変更されません。

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

オブジェクト *o* への新しい **強参照** を取得し、それが使用中で破棄されてはならないことを示します。

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

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

オブジェクトが `NULL` であってはいけません。それが `NULL` ではないと確信が持てないならば、*Py_XINCREF()* を使ってください。

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

バージョン 3.12 で変更: 永続オブジェクトは変更されません。

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

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

See also *Py_XNewRef()*.

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

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. Create a new *strong reference* to an object: call *Py_INCREF()* on *o* and return the object *o*.

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

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

例えば:

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

can be written as:

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

See also *Py_INCREF()*.

Added in version 3.10.

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

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. Similar to *Py_NewRef()*, but the object *o* can be NULL.

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

Added in version 3.10.

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

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

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

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

オブジェクトが NULL であってはいけません。それが NULL ではないと確信が持てないならば、*Py_XDECREF()* を使ってください。

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

> **警告**
>
> 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.

バージョン 3.12 で変更: 永続オブジェクトは変更されません。

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.

バージョン 3.12 で変更: The macro argument is now only evaluated once. If the argument has side effects, these are no longer duplicated.

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

次に属します: 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)

次に属します: 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.

Added in version 3.6.

バージョン 3.12 で変更: 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()*.

Added in version 3.6.

バージョン 3.12 で変更: The macro arguments are now only evaluated once. If an argument has side effects, these are no longer duplicated.

第

FIVE

例外処理

この章で説明する関数を使うと、Python の例外の処理や例外の送出ができるようになります。Python の例外処理の基本をいくらか理解することが大切です。例外は POSIX `errno` 変数にやや似た機能を果たします: 発生した中で最も新しいエラーの (スレッド毎の) グローバルなインジケータがあります。実行に成功した場合にはほとんどの C API 関数がこれをクリアしませんが、失敗したときにはエラーの原因を示すために設定します。ほとんどの C API 関数はエラーインジケータも返し、通常は関数がポインタを返すことになっている場合は `NULL` であり、関数が整数を返す場合は `-1` です。(例外: `PyArg_*` 関数は実行に成功したときに `1` を返し、失敗したときに `0` を返します).

具体的には、エラーインジケータは、例外の型、例外の値、トレースバックオブジェクトの 3 つのオブジェクトポインタで構成されます。これらのポインタはどれでも、設定されない場合は `NULL` になりえます（ただし、いくつかの組み合わせは禁止されており、例えば、例外の型が `NULL` の場合は、トレースバックは非 `NULL` の値になりません）

ある関数が呼び出した関数がいくつか失敗したために、その関数が失敗しなければならないとき、一般的にエラーインジケータを設定しません。呼び出した関数がすでに設定しています。エラーを処理して例外をクリアするか、あるいは (オブジェクト参照またはメモリ割り当てのような) それが持つどんなリソースも取り除いた後に戻るかのどちらか一方を行う責任があります。エラーを処理する準備をしていなければ、普通に続けるべきでは **ありません**。エラーのために戻る場合は、エラーが設定されていると呼び出し元に知らせることが大切です。エラーが処理されていない場合または丁寧に伝えられている場合には、Python/C API のさらなる呼び出しは意図した通りには動かない可能性があり、不可解な形で失敗するかもしれません。

> **注釈**
>
> 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 出力とクリア

void `PyErr_Clear()`

> **次に属します**: Stable ABI. エラーインジケータをクリアします。エラーインジケータが設定されていないならば、効果はありません。

void **PyErr_PrintEx**(int set_sys_last_vars)

次に属します: Stable ABI. 標準のトレースバックを `sys.stderr` に出力し、エラーインジケータをクリアします。ただし、エラーが `SystemExit` **である場合を除いて** です。その場合、トレースバックは出力されず、Python プロセスは `SystemExit` インスタンスで指定されたエラーコードで終了します。

エラーインジケータが設定されているときに **だけ**、この関数を呼び出してください。それ以外の場合、致命的なエラーを引き起こすでしょう!

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.

バージョン 3.12 で変更: The setting of `sys.last_exc` was added.

void **PyErr_Print**()

次に属します: Stable ABI. `PyErr_PrintEx(1)` のエイリアスです。

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

次に属します: Stable ABI. 現在の例外と *obj* 引数で `sys.unraisablehook()` を呼び出します。

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.

この関数を呼び出すときには、例外がセットされていなければなりません。

バージョン 3.4 で変更: Print a traceback. Print only traceback if *obj* is `NULL`.

バージョン 3.8 で変更: Use `sys.unraisablehook()`.

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

次に属します: Stable ABI *(**バージョン** 3.12 **より**)*. Print the standard traceback display of `exc` to `sys.stderr`, including chained exceptions and notes.

Added in version 3.12.

5.2 例外の送出

以下の関数は、現在のスレッドのエラーインジケータの設定を補助します。利便性のため、これらの関数のいくつかは、`return` 文で利用できるように常に `NULL` ポインタを返します。

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

次に属します: 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)

次に属します: Stable ABI. この関数は *PyErr_SetString()* に似ていますが、例外の " 値 (value)" として任意の Python オブジェクトを指定することができます。

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

戻り値: 常に *NULL* **。次に属します:** Stable ABI. この関数はエラーインジケータを設定し NULL を返します。*exception* は Python 例外クラスであるべきです。*format* と以降の引数はエラーメッセージを作るためのもので, *PyUnicode_FromFormat()* の引数と同じ意味を持っています。*format* は ASCII エンコードされた文字列です。

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

戻り値: 常に *NULL* **。次に属します:** Stable ABI *(**バージョン** 3.5 **より**)*. *PyErr_Format()* と同じですが、可変長引数の代わりに va_list 引数を受け取ります。

Added in version 3.5.

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

次に属します: Stable ABI. これは PyErr_SetObject(type, Py_None) を省略したものです。

int **PyErr_BadArgument**()

次に属します: Stable ABI. これは PyErr_SetString(PyExc_TypeError, message) を省略したもので、ここで *message* は組み込み操作が不正な引数で呼び出されたということを表しています。主に内部で使用するためのものです。

PyObject ***PyErr_NoMemory**()

戻り値: 常に *NULL* **。次に属します:** Stable ABI. これは PyErr_SetNone(PyExc_MemoryError) を省略したもので、NULL を返します。したがって、メモリ不足になったとき、オブジェクト割り当て関数は return PyErr_NoMemory(); と書くことができます。

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

戻り値: 常に *NULL* **。次に属します:** 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)

戻り値: 常に *NULL* **。次に属します:** 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)

戻り値: 常に *NULL*。**次に属します**: Stable ABI (バージョン *3.7* より). *PyErr_SetFromErrnoWithFilenameObject()* に似てますが、ファイル名を 2 つ取る関数が失敗したときに例外を送出するために、2 つ目のファイル名オブジェクトを受け取ります。

Added in version 3.4.

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

戻り値: 常に *NULL*。**次に属します**: Stable ABI. *PyErr_SetFromErrnoWithFilenameObject()* に似ていますが、ファイル名は C 文字列として与えられます。*filename* は **ファイルシステムのエンコーディングとエラーハンドラ** でデコードされます。

PyObject ***PyErr_SetFromWindowsErr**(int ierr)

戻り値: 常に *NULL*。**次に属します**: Stable ABI *on Windows* (バージョン *3.7* より). 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)

戻り値: 常に *NULL*。**次に属します**: Stable ABI *on Windows* (バージョン *3.7* より). *PyErr_SetFromWindowsErr()* に似ていますが、送出する例外の型を指定する引数が追加されています。

Availability: Windows.

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

戻り値: 常に *NULL*。**次に属します**: Stable ABI *on Windows* (バージョン *3.7* より). 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)

戻り値: 常に *NULL*。**次に属します**: Stable ABI *on Windows* (バージョン *3.7* より). 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)

戻り値: **常に** *NULL* **。次に属します**: Stable ABI *on Windows* (**バージョン** *3.7* **より**). *PyErr_SetExcFromWindowsErrWithFilenameObject()* に似てますが、2 つ目のファイル名オブジェクトを受け取ります。

Availability: Windows.

Added in version 3.4.

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

戻り値: **常に** *NULL* **。次に属します**: Stable ABI *on Windows* (**バージョン** *3.7* **より**). *PyErr_SetFromWindowsErrWithFilename()* に似ていますが、送出する例外の型を指定する引数が追加されています。

Availability: Windows.

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

戻り値: **常に** *NULL* **。次に属します**: Stable ABI (**バージョン** *3.7* **より**). `ImportError` を簡単に送出するための関数です。*msg* は例外のメッセージ文字列としてセットされます。*name* と *path* はどちらも `NULL` にしてよく、それぞれ `ImportError` の `name` 属性と `path` 属性としてセットされます。

Added in version 3.3.

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

戻り値: **常に** *NULL* **。次に属します**: Stable ABI (**バージョン** *3.6* **より**). *PyErr_SetImportError()* とよく似ていますが、この関数は送出する例外として、`ImportError` のサブクラスを指定できます。

Added in version 3.6.

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

現在の例外のファイル、行、オフセットの情報をセットします。現在の例外が `SyntaxError` でない場合は、例外を表示するサブシステムが、例外が `SyntaxError` であると思えるように属性を追加します。

Added in version 3.4.

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

次に属します: Stable ABI (**バージョン** *3.7* **より**). *PyErr_SyntaxLocationObject()* と似ていますが、*filename* は **ファイルシステムのエンコーディングとエラーハンドラ** でデコードされたバイト文字列です。

Added in version 3.2.

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

次に属します: Stable ABI. *PyErr_SyntaxLocationEx()* と似ていますが、*col_offset* 引数が除去されています。

void **PyErr_BadInternalCall**()

次に属します: Stable ABI. `PyErr_SetString(PyExc_SystemError, message)` を省略したものです。ここで *message* は内部操作 (例えば、Python/C API 関数) が不正な引数とともに呼び出されたということを示しています。主に内部で使用するためのものです。

5.3 警告

以下の関数を使い、C コードで起きた警告を報告します。Python の `warnings` モジュールで公開されている同様の関数とよく似てます。これらの関数は通常警告メッセージを *sys.stderr* へ出力しますが、ユーザが警告をエラーへ変更するように指定することもでき、その場合は、関数は例外を送出します。警告機構がもつ問題のためにその関数が例外を送出するということも有り得ます。例外が送出されない場合は戻り値は `0` で、例外が送出された場合は `-1` です。(警告メッセージが実際に出力されるか、およびその例外の原因が何かについては判断できません; これは意図的なものです。) 例外が送出された場合、呼び出し元は通常の例外処理を行います (例えば、保持していた参照に対し *Py_DECREF()* を行い、エラー値を返します)。

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

次に属します: Stable ABI. 警告メッセージを発行します。*category* 引数は警告カテゴリ (以下を参照) かまたは `NULL` で、*message* 引数は UTF-8 エンコードされた文字列です。*stacklevel* はスタックフレームの数を示す正の整数です; 警告はそのスタックフレームの中の実行している行から発行されます。*stacklevel* が 1 だと *PyErr_WarnEx()* を呼び出している関数が、2 だとその上の関数が Warning の発行元になります。

警告カテゴリは `PyExc_Warning` のサブクラスでなければなりません。`PyExc_Warning` は `PyExc_Exception` のサブクラスです。デフォルトの警告カテゴリは `PyExc_RuntimeWarning` です。標準の Python 警告カテゴリは、**標準警告カテゴリ** で名前が列挙されているグローバル変数として利用可能です。

警告をコントロールするための情報については、`warnings` モジュールのドキュメンテーションとコマンドライン・ドキュメンテーションの `-W` オプションを参照してください。警告コントロールのための C API はありません。

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

すべての警告の属性を明示的に制御した警告メッセージを出します。これは Python 関数 `warnings.warn_explicit()` の直接的なラッパーで、さらに情報を得るにはそちらを参照してください。そこに説明されているデフォルトの効果を得るために、*module* と *registry* 引数は `NULL` に設定することができます。

Added in version 3.4.

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

次に属します: Stable ABI. *PyErr_WarnExplicitObject()* に似ていますが、*message* と *module* が UTF-8 エンコードされた文字列であるところが異なり、*filename* は **ファイルシステムのエンコーディングとエラーハンドラ** でデコードされます。

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

次に属します: Stable ABI. *PyErr_WarnEx()* に似たような関数ですが、警告メッセージをフォーマットするのに *PyUnicode_FromFormat()* を使用します。*format* は ASCII にエンコードされた文字列です。

Added in version 3.2.

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

次に属します: Stable ABI (**バージョン** *3.6* **より**). Function similar to *PyErr_WarnFormat()*, but *category* is `ResourceWarning` and it passes *source* to `warnings.WarningMessage`.

Added in version 3.6.

5.4 エラーインジケータの問い合わせ

PyObject ***PyErr_Occurred**()

戻り値: **借用参照。次に属します**: Stable ABI. エラーインジケータが設定されているかテストします。設定されている場合は、例外の **型** (`PyErr_Set*` 関数の一つあるいは *PyErr_Restore()* への最も新しい呼び出しに対する第一引数) を返します。設定されていない場合は `NULL` を返します。あなたは戻り値への参照を持っていませんので、それに *Py_DECREF()* する必要はありません。

The caller must hold the GIL.

> **注釈**
>
> 戻り値を特定の例外と比較しないでください。その代わりに、下に示す *PyErr_ExceptionMatches()* を使ってください。(比較は簡単に失敗するでしょう。なぜなら、例外はクラスではなくインスタンスかもしれないし、あるいは、クラス例外の場合は期待される例外のサブクラスかもしれないからです。)

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

次に属します: Stable ABI. `PyErr_GivenExceptionMatches(PyErr_Occurred(), exc)` と同じ。例外が実際に設定されたときにだけ、これを呼び出だすべきです。例外が発生していないならば、メモリアクセス違反が起きるでしょう。

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

次に属します: Stable ABI. 例外 *given* が *exc* の例外型と適合する場合に真を返します。*exc* がクラスオブジェクトである場合も、*given* がサブクラスのインスタンスであるときに真を返します。*exc* がタプルの場合は、タプルにある (およびそのサブタプルに再帰的にある) すべての例外型が適合するか調べられます。

PyObject ***PyErr_GetRaisedException**(void)

戻り値: **新しい参照。次に属します**: Stable ABI *(**バージョン** 3.12 **より**)*. 現在発生している例外を返し、同時にエラーインジケータを消去します。エラーインジケータが設定されていない場合は `NULL` を返します。

この関数は、例外をキャッチする必要があるコード、または一時的にエラーインジケータを保存・復元する必要があるコードに使用されます。

例えば:

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

   /* ... 他のエラーを発生させる可能性があるコード ... */

   PyErr_SetRaisedException(exc);
}
```

参考

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

Added in version 3.12.

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

次に属します: Stable ABI *(**バージョン** 3.12 **より**)*. Set *exc* as the exception currently being raised, clearing the existing exception if one is set.

警告

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

Added in version 3.12.

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

次に属します: Stable ABI. バージョン 3.12 で非推奨: Use *PyErr_GetRaisedException()* instead.

エラーインジケータをアドレスを渡す三つの変数の中へ取り出します。エラーインジケータが設定されていない場合は、三つすべての変数を `NULL` に設定します。エラーインジケータが設定されている場合はクリアされ、あなたは取り出されたそれぞれのオブジェクトへの参照を持つことになります。型オブジェクトが `NULL` でないときでさえ、その値とトレースバックオブジェクトは `NULL` かもしれません。

注釈

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

例えば:

```
{
   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)

次に属します: Stable ABI. バージョン 3.12 で非推奨: 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.)

注釈

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)

次に属します: Stable ABI. バージョン 3.12 で非推奨: Use *PyErr_GetRaisedException()* instead, to avoid any possible de-normalization.

ある状況では、以下の *PyErr_Fetch()* が返す値は " 正規化されていない" 可能性があります。つまり、`*exc` はクラスオブジェクトだが `*val` は同じクラスのインスタンスではないという意味です。この関数はそのような場合にそのクラスをインスタンス化するために使われます。その値がすでに正規化されている場合は何も起きません。遅延正規化はパフォーマンスを改善するために実装されています。

注釈

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)

次に属します: Stable ABI *(**バージョン** 3.11 **より**)*. 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.

> **注釈**
>
> この関数は、通常は例外を扱うコードでは使用されません。正確に言うと、これは例外の状態を一時的に保存し、元に戻す必要があるコードで使用することができます。例外の状態を元に戻す、もしくはクリアするには *PyErr_SetHandledException()* を使ってください。

Added in version 3.11.

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

次に属します: Stable ABI *(**バージョン** 3.11 **より**)*. 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`.

> **注釈**
>
> この関数は、通常は例外を扱うコードでは使用されません。正確に言うと、これは例外の状態を一時的に保存し、元に戻す必要があるコードで使用することができます。例外の状態を取得するには *PyErr_GetHandledException()* を使ってください。

Added in version 3.11.

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

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. 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()*.

> **注釈**
>
> この関数は、通常は例外を扱うコードでは使用されません。正確に言うと、これは例外の状態を一

時的に保存し、元に戻す必要があるコードで使用することができます。例外の状態を元に戻す、もしくはクリアするには *PyErr_SetExcInfo()* を使ってください。

Added in version 3.3.

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

次に属します: Stable ABI *(バージョン 3.7 より)*. 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()*.

注釈

この関数は、通常は例外を扱うコードでは使用されません。正確に言うと、これは例外の状態を一時的に保存し、元に戻す必要があるコードで使用することができます。例外の状態を取得するには *PyErr_GetExcInfo()* を使ってください。

Added in version 3.3.

バージョン 3.11 で変更: 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 シグナルハンドリング

int **PyErr_CheckSignals**()

次に属します: 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).

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

void **PyErr_SetInterrupt**()

次に属します: Stable ABI. Simulate the effect of a `SIGINT` signal arriving. This is equivalent to `PyErr_SetInterruptEx(SIGINT)`.

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

int **PyErr_SetInterruptEx**(int signum)

次に属します: Stable ABI *(バージョン 3.10 より)*. シグナルが到達した効果をシミュレートします。次に *PyErr_CheckSignals()* が呼ばれたとき、与えられたシグナル番号用の Python のシグナルハンドラが呼び出されます。

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.

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

Added in version 3.10.

int **PySignal_SetWakeupFd**(int fd)

このユーティリティ関数は、シグナルを受け取ったときにシグナル番号をバイトとして書き込むファイル記述子を指定します。*fd* はノンブロッキングでなければなりません。この関数は、1 つ前のファイル記述子を返します。

値 `-1` を渡すと、この機能を無効にします; これが初期状態です。この関数は Python の `signal.set_wakeup_fd()` と同等ですが、どんなエラーチェックも行いません。*fd* は有効なファイル記述子であるべきです。この関数はメインスレッドからのみ呼び出されるべきです。

バージョン 3.5 で変更: Windows で、この関数はソケットハンドルをサポートするようになりました。

5.6 例外クラス

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

戻り値: **新しい参照。次に属します**: Stable ABI. このユーティリティ関数は新しい例外クラスを作成して返します。*name* 引数は新しい例外の名前、`module.classname` 形式の C 文字列でなければならない。*base* と *dict* 引数は通常 `NULL` です。これはすべての例外のためのルート、組み込み名 `Exception` (C では `PyExc_Exception` としてアクセス可能) をルートとして派生したクラスオブジェクトを作成します。

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)

戻り値: **新しい参照。次に属します**: Stable ABI. *PyErr_NewException()* とほぼ同じですが、新しい例外クラスに簡単に docstring を設定できます。*doc* が `NULL` で無い場合、それが例外クラスの docstring になります。

Added in version 3.2.

5.7 例外オブジェクト

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

戻り値: **新しい参照。次に属します**: Stable ABI. Python で `__traceback__` 属性からアクセスできるものと同じ、例外に関する traceback の新しい参照を返します。関係する traceback が無い場合は、`NULL` を返します。

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

次に属します: Stable ABI. その例外に関する traceback に *tb* をセットします。クリアするには `Py_None` を使用してください。

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

戻り値: **新しい参照。次に属します**: 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)

次に属します: Stable ABI. 例外に関するコンテキストに *ctx* をセットします。クリアするには `NULL` を使用してください。*ctx* が例外インスタンスかどうかを確かめる型チェックは行われません。これは *ctx* への参照を盗みます。

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

戻り値: **新しい参照。次に属します**: 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)

次に属します: 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)

戻り値: 新しい参照。次に属します: Stable ABI *(バージョン 3.12 より)*. Return `args` of exception *ex*.

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

次に属します: Stable ABI *(バージョン 3.12 より)*. Set `args` of exception *ex* to *args*.

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

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

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.

Added in version 3.12.

5.8 Unicode 例外オブジェクト

以下の関数は C 言語から Unicode 例外を作ったり修正したりするために利用します。

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

戻り値: 新しい参照。次に属します: Stable ABI. *encoding*, *object*, *length*, *start*, *end*, *reason* 属性をもった `UnicodeDecodeError` オブジェクトを作成します。*encoding* および *reason* は UTF-8 エンコードされた文字列です。

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

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

戻り値: **新しい参照。次に属します**: Stable ABI. 与えられた例外オブジェクトの *encoding* 属性を返します。

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

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

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

戻り値: **新しい参照。次に属します**: Stable ABI. 与えられた例外オブジェクトの *object* 属性を返します。

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)

次に属します: Stable ABI. 渡された例外オブジェクトから *start* 属性を取得して **start* に格納します。*start* は `NULL` であってはなりません。成功したら `0` を、失敗したら `-1` を返します。

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)

次に属します: Stable ABI. Set the *start* attribute of the given exception object to *start*. Return `0` on success, `-1` on failure.

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)

次に属します: Stable ABI. 渡された例外オブジェクトから *end* 属性を取得して **end* に格納します。*end* は `NULL` であってはなりません。成功したら `0` を、失敗したら `-1` を返します。

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)

次に属します: Stable ABI. 渡された例外オブジェクトの *end* 属性を *end* に設定します。成功したら `0` を、失敗したら `-1` を返します。

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

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

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

戻り値: **新しい参照。次に属します**: Stable ABI. 渡された例外オブジェクトの *reason* 属性を返します。

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

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

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

次に属します: Stable ABI. 渡された例外オブジェクトの *reason* 属性を *reason* に設定します。成功したら 0 を、失敗したら -1 を返します。

5.9 再帰の管理

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)

次に属します: Stable ABI *(**バージョン** 3.9 **より**)*. C レベルの再帰呼び出しをしようとしているところに印を付けます。

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

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

where は `" in instance check"` のような UTF-8 エンコードされた文字列にして、再帰の深さの限界に達したことで送出される `RecursionError` のメッセージに連結できるようにすべきです。

バージョン 3.9 で変更: This function is now also available in the *limited API*.

void **Py_LeaveRecursiveCall**(void)

次に属します: Stable ABI *(**バージョン** 3.9 **より**)*. *Py_EnterRecursiveCall()* を終了させます。*Py_EnterRecursiveCall()* の **成功した** 呼び出しに対し 1 回呼ばなければなりません。

バージョン 3.9 で変更: This function is now also available in the *limited API*.

コンテナ型に対し *tp_repr* を適切に実装するには、特殊な再帰の処理が求められます。スタックの防護に加え、*tp_repr* は循環処理を避けるためにオブジェクトを辿っていく必要があります。次の 2 つの関数はその機能を容易にします。実質的には、これらは `reprlib.recursive_repr()` と同等な C の実装です。

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

次に属します: Stable ABI. 循環処理を検知するために、*tp_repr* の実装の先頭で呼び出します。

そのオブジェクトが既に処理されたものだった場合、この関数は正の整数を返します。その場合、*tp_repr* の実装は、循環を示す文字列オブジェクトを返すべきです。例えば、`dict` オブジェクトは `{...}` を返しますし、`list` オブジェクトは `[...]` を返します。

再帰回数の上限に達した場合は、この関数は負の整数を返します。この場合、*tp_repr* の実装は一般的には `NULL` を返すべきです。

それ以外の場合は、関数はゼロを返し、*tp_repr* の実装は通常どおり処理を続けてかまいません。

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

次に属します: Stable ABI. *Py_ReprEnter()* を終了させます。0 を返した *Py_ReprEnter()* の呼び出しに対し 1 回呼ばなければなりません。

5.10 標準例外

PyExc_ の後ろに Python の例外名が続く名前をもつグローバル変数として、すべての標準 Python 例外が利用可能です。これらは型 *PyObject** を持ち、すべてクラスオブジェクトです。完璧を期するために、すべての変数を以下に列挙します:

C 名	Python 名	注釈
PyExc_BaseException	BaseException	*1
PyExc_Exception	Exception	p. 80, *1
PyExc_ArithmeticError	ArithmeticError	p. 80, *1
PyExc_AssertionError	AssertionError	
PyExc_AttributeError	AttributeError	
PyExc_BlockingIOError	BlockingIOError	
PyExc_BrokenPipeError	BrokenPipeError	
PyExc_BufferError	BufferError	
PyExc_ChildProcessError	ChildProcessError	
PyExc_ConnectionAbortedError	ConnectionAbortedError	
PyExc_ConnectionError	ConnectionError	
PyExc_ConnectionRefusedError	ConnectionRefusedError	
PyExc_ConnectionResetError	ConnectionResetError	
PyExc_EOFError	EOFError	
PyExc_FileExistsError	FileExistsError	
PyExc_FileNotFoundError	FileNotFoundError	
PyExc_FloatingPointError	FloatingPointError	
PyExc_GeneratorExit	GeneratorExit	
PyExc_ImportError	ImportError	
PyExc_IndentationError	IndentationError	
PyExc_IndexError	IndexError	
PyExc_InterruptedError	InterruptedError	
PyExc_IsADirectoryError	IsADirectoryError	
PyExc_KeyError	KeyError	
PyExc_KeyboardInterrupt	KeyboardInterrupt	
PyExc_LookupError	LookupError	p. 80, *1
PyExc_MemoryError	MemoryError	
PyExc_ModuleNotFoundError	ModuleNotFoundError	
PyExc_NameError	NameError	
PyExc_NotADirectoryError	NotADirectoryError	
PyExc_NotImplementedError	NotImplementedError	

次のページに続く

表 1 – 前のページからの続き

C 名	Python 名	注釈
PyExc_OSError	OSError	*1
PyExc_OverflowError	OverflowError	
PyExc_PermissionError	PermissionError	
PyExc_ProcessLookupError	ProcessLookupError	
PyExc_RecursionError	RecursionError	
PyExc_ReferenceError	ReferenceError	
PyExc_RuntimeError	RuntimeError	
PyExc_StopAsyncIteration	StopAsyncIteration	
PyExc_StopIteration	StopIteration	
PyExc_SyntaxError	SyntaxError	
PyExc_SystemError	SystemError	
PyExc_SystemExit	SystemExit	
PyExc_TabError	TabError	
PyExc_TimeoutError	TimeoutError	
PyExc_TypeError	TypeError	
PyExc_UnboundLocalError	UnboundLocalError	
PyExc_UnicodeDecodeError	UnicodeDecodeError	
PyExc_UnicodeEncodeError	UnicodeEncodeError	
PyExc_UnicodeError	UnicodeError	
PyExc_UnicodeTranslateError	UnicodeTranslateError	
PyExc_ValueError	ValueError	
PyExc_ZeroDivisionError	ZeroDivisionError	

Added in 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、PyExc_TimeoutError は **PEP 3151** により導入されました。

Added in version 3.5: PyExc_StopAsyncIteration および PyExc_RecursionError。

Added in version 3.6: PyExc_ModuleNotFoundError.

これらは互換性のある PyExc_OSError のエイリアスです:

C 名	注釈
PyExc_EnvironmentError	
PyExc_IOError	
PyExc_WindowsError	*2

*1 これは別の標準例外のためのベースクラスです。

バージョン 3.3 で変更: これらのエイリアスは例外の種類を分けるために使われます。

注釈:

5.11 標準警告カテゴリ

PyExc_ の後ろに Python の例外名が続く名前をもつグローバル変数として、すべての標準 Python 警告カテゴリが利用可能です。これらは型 *PyObject** を持ち、すべてクラスオブジェクトです。完璧を期するために、すべての変数を以下に列挙します:

C 名	Python 名	注釈
PyExc_Warning	Warning	*3
PyExc_BytesWarning	BytesWarning	
PyExc_DeprecationWarning	DeprecationWarning	
PyExc_FutureWarning	FutureWarning	
PyExc_ImportWarning	ImportWarning	
PyExc_PendingDeprecationWarning	PendingDeprecationWarning	
PyExc_ResourceWarning	ResourceWarning	
PyExc_RuntimeWarning	RuntimeWarning	
PyExc_SyntaxWarning	SyntaxWarning	
PyExc_UnicodeWarning	UnicodeWarning	
PyExc_UserWarning	UserWarning	

Added in version 3.2: PyExc_ResourceWarning.

注釈:

*2 Windows でのみ定義されています。プリプロセッサマクロ MS_WINDOWS が定義されているかテストすることで、これを使うコードを保護してください。

*3 これは別の標準警告カテゴリのためのベースクラスです。

第

SIX

ユーティリティ

この章の関数は、C で書かれたコードをプラットフォーム間で可搬性のあるものにする上で役立つものから、C から Python モジュールを使うもの、そして関数の引数を解釈したり、C の値から Python の値を構築するものまで、様々なユーティリティ的タスクを行います。

6.1 オペレーティングシステム関連のユーティリティ

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

戻り値: **新しい参照。次に属します**: Stable ABI *(**バージョン** 3.6 **より**)*. 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.

Added in 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`()

次に属します: Stable ABI *on platforms with fork() (**バージョン** 3.7 **より**)*. プロセスがフォークする前に、いくつかの内部状態を準備するための関数です。`fork()` や現在のプロセスを複製するその他の類似の関数を呼び出す前にこの関数を呼びださなければなりません。`fork()` が定義されているシステムでのみ利用できます。

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

Added in version 3.7.

void PyOS_AfterFork_Parent()

次に属します: Stable ABI *on platforms with fork()* (**バージョン** *3.7* **より**). プロセスがフォークした後に内部状態を更新するための関数です。fork() や、現在のプロセスを複製するその他の類似の関数を呼び出した後に、プロセスの複製が成功したかどうかにかかわらず、親プロセスからこの関数を呼び出さなければなりません。fork() が定義されているシステムでのみ利用できます。

警告

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

Added in version 3.7.

void PyOS_AfterFork_Child()

次に属します: Stable ABI *on platforms with fork()* (**バージョン** *3.7* **より**). 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.

警告

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

Added in version 3.7.

参考

os.register_at_fork() を利用すると *PyOS_BeforeFork()*、*PyOS_AfterFork_Parent()* *PyOS_AfterFork_Child()* によって呼び出されるカスタムの Python 関数を登録できます。

void PyOS_AfterFork()

次に属します: Stable ABI *on platforms with fork()*. プロセスが fork した後の内部状態を更新するための関数です; fork 後 Python インタプリタを使い続ける場合、新たなプロセス内でこの関数を呼び出さねばなりません。新たなプロセスに新たな実行可能物をロードする場合、この関数を呼び出す必要はありません。

バージョン 3.7 で非推奨: この関数は *PyOS_AfterFork_Child()* によって置き換えられました。

int **PyOS_CheckStack**()

次に属します: Stable ABI *on platforms with USE_STACKCHECK (**バージョン** 3.7 **より**)*. 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)

次に属します: Stable ABI.

PyOS_sighandler_t **PyOS_getsig**(int i)

次に属します: 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)

次に属します: 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)

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*.

> **警告**
>
> 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.

ファイルシステムのエンコーディングとエラーハンドラ からバイト文字列をデコードします。エラーハンドラが surrogateescape エラーハンドラ なら、デコードできないバイトは U+DC80 から U+DCFF までの範囲の文字としてデコードされ、バイト列がサロゲート文字としてデコードできる場合は、デコードするのではなく surrogateescape エラーハンドラを使ってバイト列がエスケープされます。

新しくメモリ確保されたワイドキャラクター文字列へのポインタを返します。このメモリを解放するのには *PyMem_RawFree()* を使ってください。引数 size が `NULL` でない場合は、null 文字以外のワイドキャラクターの数を `*size` へ書き込みます。

デコードもしくはメモリ確保でエラーが起きると `NULL` を返します。*size* が `NULL` でない場合は、メモリエラーのときは `(size_t)-1` を、デコードでのエラーのときは `(size_t)-2` を `*size` に設定します。

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

C ライブラリーにバグがない限り、デコードでのエラーは起こりえません。

キャラクター文字列をバイト文字列に戻すには *Py_EncodeLocale()* 関数を使ってください。

参考

PyUnicode_DecodeFSDefaultAndSize() および *PyUnicode_DecodeLocaleAndSize()* 関数。

Added in version 3.5.

バージョン 3.7 で変更: この関数は、Python UTF-8 Mode では UTF-8 エンコーディングを利用するようになりました。

バージョン 3.8 で変更: 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)

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. ワイドキャラクター文字列を **ファイルシステムのエンコーディングとエラーハンドラ** にエンコードします。エラーハンドラが surrogateescape エラーハンドラ なら、U+DC80 から U+DCFF までの範囲のサロゲート文字は 0x80 から 0xFF までのバイトに変換されます。

新しくメモリ確保されたバイト文字列へのポインタを返します。このメモリを解放するのには *PyMem_Free()* を使ってください。エンコードエラーかメモリ確保エラーのときは `NULL` を返します。

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*.

バイト文字列をワイドキャラクター文字列に戻すには *Py_DecodeLocale()* 関数を使ってください。

警告

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.

参考

PyUnicode_EncodeFSDefault() および *PyUnicode_EncodeLocale()* 関数。

Added in version 3.5.

バージョン 3.7 で変更: この関数は、Python UTF-8 Mode では UTF-8 エンコーディングを利用するようになりました。

バージョン 3.8 で変更: The function now uses the UTF-8 encoding on Windows if *PyPreConfig.legacy_windows_fs_encoding* is zero.

6.2 システム関数

sys モジュールが提供している機能に C のコードからアクセスする関数です。すべての関数は現在のインタプリタスレッドの sys モジュールの辞書に対して動作します。この辞書は内部のスレッド状態構造体に格納されています。

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

戻り値: **借用参照。次に属します**: Stable ABI. sys モジュールの *name* オブジェクトを返すか、存在しなければ例外を設定せずに NULL を返します。

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

次に属します: Stable ABI. *v* が NULL で無い場合、sys モジュールの *name* に *v* を設定します。*v* が NULL なら、sys モジュールから *name* を削除します。成功したら 0 を、エラー時は -1 を返します。

void **PySys_ResetWarnOptions**()

次に属します: Stable ABI. Reset sys.warnoptions to an empty list. This function may be called prior to *Py_Initialize()*.

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

次に属します: Stable ABI. This API is kept for backward compatibility: setting *PyConfig.warnoptions* should be used instead, see *Python Initialization Configuration*.

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

バージョン 3.11 で非推奨.

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

次に属します: Stable ABI. This API is kept for backward compatibility: setting *PyConfig.warnoptions* should be used instead, see *Python Initialization Configuration*.

Append *unicode* to sys.warnoptions.

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

バージョン 3.11 で非推奨.

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

次に属します: Stable ABI. This API is kept for backward compatibility: setting *PyConfig.module_search_paths* and *PyConfig.module_search_paths_set* should be used instead, see *Python Initialization Configuration*.

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

バージョン 3.11 で非推奨.

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

次に属します: Stable ABI. *format* で指定された出力文字列を `sys.stdout` に出力します。切り詰めが起こった場合を含め、例外は一切発生しません (後述)。

format は、フォーマット後の出力文字列のトータルの大きさを 1000 バイト以下に抑えるべきです。--1000 バイト以降の出力文字列は切り詰められます。特に、制限のない "%s" フォーマットを使うべきではありません。"%.<N>s" のようにして N に 10 進数の値を指定し、<N> + その他のフォーマット後の最大サイズが 1000 を超えないように設定するべきです。同じように "%f" にも気を付ける必要があります。非常に大きい数値に対して、数百の数字を出力する可能性があります。

問題が発生したり、`sys.stdout` が設定されていなかった場合、フォーマット後のメッセージは本物の (C レベルの) *stdout* に出力されます。

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

次に属します: Stable ABI. *PySys_WriteStdout()* と同じですが、`sys.stderr` もしくは *stderr* に出力します。

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

次に属します: Stable ABI. PySys_WriteStdout() に似た関数ですが、*PyUnicode_FromFormatV()* を使ってメッセージをフォーマットし、メッセージを任意の長さに切り詰めたりはしません。

Added in version 3.2.

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

次に属します: Stable ABI. *PySys_FormatStdout()* と同じですが、`sys.stderr` もしくは *stderr* に出力します。

Added in version 3.2.

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

次に属します: Stable ABI *(バージョン 3.7 より)*. This API is kept for backward compatibility: setting *PyConfig.xoptions* should be used instead, see *Python Initialization Configuration*.

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

Added in version 3.2.

バージョン 3.11 で非推奨.

PyObject ***PySys_GetXOptions**()

戻り値: **借用参照。次に属します**: Stable ABI *(バージョン 3.7 より)*. `sys._xoptions` と同様、`-X` オプションの現在の辞書を返します。エラーが起きると、`NULL` が返され、例外がセットされます。

Added in version 3.2.

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

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

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

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

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

Added in version 3.8.

バージョン 3.8.2 で変更: Require *Py_ssize_t* for `#` format characters. Previously, an unavoidable deprecation warning was raised.

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

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

userData ポインタはフック関数に渡されます。フック関数は別なランタイムから呼び出されるかもしれないので、このポインタは直接 Python の状態を参照すべきではありません。

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

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

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

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

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

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

Added in version 3.8.

6.3 プロセス制御

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

次に属します: 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.

バージョン 3.9 で変更: Log the function name automatically.

void **Py_Exit**(int status)

次に属します: Stable ABI. 現在のプロセスを終了します。*Py_FinalizeEx()* を呼び出した後、標準 C ライブラリ関数の `exit(status)` を呼び出します。*Py_FinalizeEx()* がエラーになった場合、終了ステータスは 120 に設定されます。

バージョン 3.6 で変更: 終了処理のエラーは無視されなくなりました。

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

次に属します: Stable ABI. *Py_FinalizeEx()* から呼び出される後始末処理を行う関数 (cleanup function) を登録します。後始末関数は引数無しで呼び出され、値を返しません。最大で 32 の後始末処理関数を登録できます。登録に成功すると、*Py_AtExit()* は `0` を返します; 失敗すると `-1` を返します。最後に登録した後始末処理関数から先に呼び出されます。各関数は高々一度しか呼び出されません。Python の内部的な終了処理は後始末処理関数より以前に完了しているので、*func* からはいかなる Python API も呼び出してはなりません。

6.4 モジュールのインポート

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

戻り値: **新しい参照。次に属します**: 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)

戻り値: **新しい参照。次に属します**: Stable ABI. この関数は、*PyImport_ImportModule()* の廃止予定のエイリアスです。

バージョン 3.3 で変更: この関数は、従来は別のスレッドによってインポートロックが行われていた場合は即座に失敗していました。しかし Python 3.3 では、大部分の目的でロックスキームがモジュールごとのロックに移行したので、この関数の特別な振る舞いはもはや必要ではありません。

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

戻り値: **新しい参照。**モジュールをインポートします。モジュールのインポートについては組み込みの Python 関数 `__import__()` を読むとよくわかります。

戻り値は、インポートされたモジュールかトップレベルパッケージへの新しい参照か、失敗した場合は例外を設定して NULL を返します。__import__() と同じように、パッケージのサブモジュールが要求されたときは、空でない *fromlist* を渡された時以外は、トップレベルのパッケージを返します。

インポートが失敗した場合は、*PyImport_ImportModule()* と同様に不完全なモジュールのオブジェクトを削除します。

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

戻り値: **新しい参照。次に属します**: Stable ABI *(**バージョン** 3.7 **より**)*. モジュールをインポートします。モジュールのインポートについては組み込みの Python 関数 __import__() を読むとよく分かります。というのも、標準の __import__() はこの関数を直接呼び出しているからです。

戻り値は、インポートされたモジュールかトップレベルパッケージへの新しい参照か、失敗した場合は例外を設定して NULL を返します。__import__() と同じように、パッケージのサブモジュールが要求されたときは、空でない *fromlist* を渡された時以外は、トップレベルのパッケージを返します。

Added in version 3.3.

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

戻り値: **新しい参照。次に属します**: Stable ABI. *PyImport_ImportModuleLevelObject()* と似ていますが、name が Unicode オブジェクトではなく UTF-8 でエンコードされた文字列である点で異なります。

バージョン 3.3 で変更: *level* にはもはや負の値は使用できません。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 現在の ” インポートフック関数” を呼び出すための高水準のインターフェースです (*level* に 0 を明示すると、絶対インポートを意味します)。この関数は現在のグローバル変数辞書内の __builtins__ から __import__() 関数を呼び出します。すなわち、現在の環境にインストールされているインポートフック使ってインポートを行います。

この関数は常に絶対インポートを使用します。

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

戻り値: **新しい参照。次に属します**: Stable ABI. モジュールを再ロード (reload) します。戻り値は再ロードしたモジュールかトップレベルパッケージへの新たな参照になります。失敗した場合には例外をセットし、NULL を返します (その場合でも、モジュールは生成されている場合があります)。

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

戻り値: **借用参照。次に属します**: Stable ABI *(**バージョン** 3.7 **より**)*. Return the module object corresponding to a module name. The *name* argument may be of the form package.module. First check the modules dictionary if there's one there, and if not, create a new one and insert it in the modules dictionary. Return NULL with an exception set on failure.

> **注釈**
>
> 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.

Added in version 3.3.

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

戻り値: 借用参照。次に属します: Stable ABI. Similar to *PyImport_AddModuleObject()*, but the name is a UTF-8 encoded string instead of a Unicode object.

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

戻り値: 新しい参照。次に属します: 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.

この関数は、すでにインポートされているモジュールの場合には再ロードを行います。意図的にモジュールの再ロードを行う方法は *PyImport_ReloadModule()* を参照してください。

name が `package.module` 形式のドット名表記であった場合、まだ作成されていないパッケージ構造はその作成されないままになります。

PyImport_ExecCodeModuleEx() と *PyImport_ExecCodeModuleWithPathnames()* も参照してください。

バージョン 3.12 で変更: The setting of `__cached__` and `__loader__` is deprecated. See `ModuleSpec` for alternatives.

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

戻り値: 新しい参照。次に属します: Stable ABI. Like *PyImport_ExecCodeModule()*, but the `__file__` attribute of the module object is set to *pathname* if it is non-`NULL`.

PyImport_ExecCodeModuleWithPathnames() も参照してください。

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

戻り値: **新しい参照。次に属します**: Stable ABI *(バージョン 3.7 より)*. 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.

Added in version 3.3.

バージョン 3.12 で変更: Setting `__cached__` is deprecated. See `ModuleSpec` for alternatives.

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

戻り値: **新しい参照。次に属します**: Stable ABI. *PyImport_ExecCodeModuleObject()* と似ていますが、*name* と *pathname*、*cpathname* が UTF-8 でエンコードされた文字列である点が異なります。もし *pathname* が `NULL` の場合、*cpathname* から、*pathname* どのような値になるべきかを知る試みもなされます。

Added in version 3.2.

バージョン 3.3 で変更: Uses `imp.source_from_cache()` in calculating the source path if only the bytecode path is provided.

バージョン 3.12 で変更: No longer uses the removed `imp` module.

long **PyImport_GetMagicNumber**()

次に属します: Stable ABI. Python バイトコードファイル (別名 `.pyc` ファイル) のマジックナンバーを返します。マジックナンバーはバイトコードファイルの最初の 4 バイトに、リトルエンディアンバイトオーダーで現れるべきです。エラーの場合は `-1` を返します。

バージョン 3.3 で変更: 失敗した場合は `-1` の値を返します。

const char ***PyImport_GetMagicTag**()

次に属します: Stable ABI. マジックタグ文字列を Python バイトコードファイル名の **PEP 3147** フォーマットで返します。`sys.implementation.cache_tag` の値が信頼でき、かつこの関数の代わりに使用すべきであることを肝に命じましょう。

Added in version 3.2.

PyObject ***PyImport_GetModuleDict**()

戻り値: **借用参照。次に属します**: Stable ABI. モジュール管理のための辞書 (いわゆる `sys.modules`) を返します。この辞書はインタプリタごとに一つだけある変数なので注意してください。

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

戻り値: **新しい参照。次に属します**: Stable ABI *(バージョン 3.8 より)*. 与えられた名前の既にインポート済みのモジュールを返します。モジュールがインポートされていなかった場合は、`NULL` を返しますが、エラーはセットしません。モジュールの検索に失敗した場合は、`NULL` を返し、エラーをセットします。

Added in version 3.7.

PyObject *PyImport_GetImporter(*PyObject* *path)

戻り値: **新しい参照。次に属します**: 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)

次に属します: Stable ABI (**バージョン** *3.7* **より**). *name* という名前のフリーズ (freeze) されたモジュールをロードします。成功すると 1 を、モジュールが見つからなかった場合には 0 を、初期化が失敗した場合には例外をセットして -1 を返します。ロードに成功したモジュールにアクセスするには *PyImport_ImportModule()* を使ってください。(Note この関数はいささか誤解を招く名前です --- この関数はモジュールがすでにインポートされていたらリロードしてしまいます。)

Added in version 3.3.

バージョン 3.4 で変更: __file__ 属性はもうモジュールにセットされません。

int PyImport_ImportFrozenModule(const char *name)

次に属します: Stable ABI. *PyImport_ImportFrozenModuleObject()* と似ていますが、name は UTF-8 でエンコードされた文字列の代わりに、Unicode オブジェクトを使用する点が異なります。

struct _frozen

freeze ユーティリティが生成するようなフリーズ化モジュールデスクリプタの構造体型定義です。(Python ソース配布物の Tools/freeze/ を参照してください) この構造体の定義は Include/import.h にあり、以下のようになっています:

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

バージョン 3.11 で変更: 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

このポインタは *_frozen* のレコードからなり、終端の要素のメンバが NULL かゼロになっているような配列を指すよう初期化されます。フリーズされたモジュールをインポートするとき、このテーブルを検索します。サードパーティ製のコードからこのポインタに仕掛けを講じて、動的に生成されたフリーズ化モジュールの集合を提供するようにできます。

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

次に属します: Stable ABI. 既存の組み込みモジュールテーブルに単一のモジュールを追加します。こ

の関数は利便性を目的とした *PyImport_ExtendInittab()* のラッパー関数で、テーブルが拡張できないときには `-1` を返します。新たなモジュールは *name* でインポートでき、最初にインポートを試みた際に呼び出される関数として *initfunc* を使います。*Py_Initialize()* よりも前に呼び出さなければなりません。

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()*.

Python が複数回初期化される場合、*PyImport_AppendInittab()* または *PyImport_ExtendInittab()* は、それぞれの初期化の前に呼び出される必要があります。

6.5 データ整列化 (data marshalling) のサポート

以下のルーチン群は、`marshal` モジュールと同じ形式を使った整列化オブジェクトを C コードから使えるようにします。整列化形式でデータを書き出す関数に加えて、データを読み戻す関数もあります。整列化されたデータを記録するファイルはバイナリモードで開かれていなければなりません。

数値は最小桁が先にくるように記録されます。

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

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

`long` 型の整数値 *value* を *file* へ整列化します。この関数は *value* の下桁 32 ビットを書き込むだけです; ネイティブの `long` 型サイズには関知しません。*version* はファイルフォーマットを示します。

この関数は失敗することがあり、その場合はエラー指示子を設定します。それを確認するために *PyErr_Occurred()* を使います。

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

Python オブジェクト *value* を *file* へ整列化します。*version* はファイルフォーマットを示します。

この関数は失敗することがあり、その場合はエラー指示子を設定します。それを確認するために *PyErr_Occurred()* を使います。

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

戻り値: 新しい参照。 *value* の整列化表現が入ったバイト列オブジェクトを返します。*version* はファイルフォーマットを示します。

以下の関数を使うと、整列化された値を読み戻せます。

long **PyMarshal_ReadLongFromFile**(FILE *file)

読み出し用に開かれた `FILE*` 内のデータストリームから、C の `long` 型データを読み出して返します。この関数は、ネイティブの `long` のサイズに関係なく、32 ビットの値だけを読み出せます。

エラーの場合、適切な例外 (`EOFError`) を設定し `-1` を返します。

int **PyMarshal_ReadShortFromFile**(FILE *file)

読み出し用に開かれた `FILE*` 内のデータストリームから、C の `short` 型データを読み出して返します。この関数は、ネイティブの `short` のサイズに関係なく、16 ビットの値だけを読み出せます。

エラーの場合、適切な例外 (`EOFError`) を設定し `-1` を返します。

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

戻り値: 新しい参照。 読み出し用に開かれた `FILE*` 内のデータストリームから Python オブジェクトを返します。

エラーの場合、適切な例外 (`EOFError`, `ValueError`, `TypeError`) を設定し `NULL` を返します。

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

戻り値: 新しい参照。 読み出し用に開かれた `FILE*` 内のデータストリームから、Python オブジェクトを読み出して返します。*PyMarshal_ReadObjectFromFile()* と違い、この関数はファイル中に後続のオブジェクトが存在しないと仮定し、ファイルからメモリ上にファイルデータを一気にメモリにロードして、逆整列化機構がファイルから一バイトづつ読み出す代わりにメモリ上のデータを操作できるようにします。対象のファイルから他に何も読み出さないと分かっている場合にのみ、この関数を使ってください。

エラーの場合、適切な例外 (`EOFError`, `ValueError`, `TypeError`) を設定し `NULL` を返します。

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

戻り値: 新しい参照。 *data* が指す *len* バイトのバイト列バッファ内のデータストリームから Python オブジェクトを返します。

エラーの場合、適切な例外 (`EOFError`, `ValueError`, `TypeError`) を設定し `NULL` を返します。

6.6 引数の解釈と値の構築

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

最初に説明する 3 つの関数、*PyArg_ParseTuple()*, *PyArg_ParseTupleAndKeywords()*, および *PyArg_Parse()* はいずれも **書式文字列** *(format string)* を使います。書式文字列は、関数が受け取るはずの引数に関する情報を伝えるのに用いられます。いずれの関数における書式文字列も、同じ書式を使っています。

6.6.1 引数を解析する

書式文字列は、ゼロ個またはそれ以上の ” 書式単位 (format unit)” から成り立ちます。1 つの書式単位は 1 つの Python オブジェクトを表します; 通常は単一の文字か、書式単位からなる文字列を括弧で囲ったものになります。例外として、括弧で囲われていない書式単位文字列が単一のアドレス引数に対応する場合がいくつかあります。以下の説明では、引用符のついた形式は書式単位です; (丸) 括弧で囲った部分は書式単位に対応する Python のオブジェクト型です; [角] 括弧は値をアドレス渡しする際に使う C の変数型です。

文字列とバッファ

以下のフォーマットはオブジェクトに連続したメモリチャンクとしてアクセスするためのものです。返される unicode や bytes のために生のストレージを用意する必要はありません。

特に言及されていない場合、バッファーは NUL 終端されていません。

There are three ways strings and buffers can be converted to 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).
- The `es`, `es#`, `et` and `et#` formats allocate the result buffer. **You have to call** *PyMem_Free()* after you have finished processing the data (or in any early abort case).
- 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).

> **注釈**
>
> For all `#` variants of formats (`s#`, `y#`, etc.), the macro `PY_SSIZE_T_CLEAN` must be defined before including `Python.h`. On Python 3.9 and older, the type of the length argument is *Py_ssize_t* if the `PY_SSIZE_T_CLEAN` macro is defined, or int otherwise.

`s` (`str`) [const char *]
: Unicode オブジェクトを、キャラクタ文字列を指す C のポインタに変換します。キャラクタ型ポインタ変数のアドレスを渡すと、すでに存在している文字列へのポインタをその変数に記録します。C 文字列は NUL で終端されています。Python の文字列型は、null コードポイントが途中に埋め込まれていてはなりません; もし埋め込まれていれば `ValueError` 例外を送出します。Unicode オブジェクトは `'utf-8'` を使って C 文字列に変換されます。変換に失敗すると `UnicodeError` を送出します。

> **注釈**
>
> このフォーマットは *bytes-like objects* をサポートしません。ファイルシステムパスを受け取って C 言語の文字列に変換したい場合は、`O&` フォーマットを、*converter* に *`PyUnicode_FSConverter()`* を指定して利用すると良いです。

: バージョン 3.5 で変更: 以前は Python 文字列に null コードポイントが埋め込まれていたときに `TypeError` を送出していました。

`s*` (`str` または *bytes-like object*) [Py_buffer]
: このフォーマットは Unicode オブジェクトと bytes-like object を受け付けて、呼び出し元から渡された *`Py_buffer`* 構造体に値を格納します。結果の C 文字列は NUL バイトを含むかもしれません。Unicode オブジェクトは `'utf-8'` エンコーディングで C 文字列に変換されます。

`s#` (`str`, 読み取り専用の *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` または `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* または `None`) [Py_buffer]
: Like `s*`, but the Python object may also be `None`, in which case the `buf` member of the *`Py_buffer`* structure is set to `NULL`.

`z#` (`str`, 読み出し専用の *bytes-like object* または `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` (読み出し専用の *bytes-like object*) [const 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.

: バージョン 3.5 で変更: 以前は bytes バッファにヌルバイトが埋め込まれていたときに `TypeError` を送出していました。

`y*` (*bytes-like object*) [Py_buffer]
: `s*` の変形で、Unicode オブジェクトを受け付けず、bytes-like object のみを受け付けます。**バイナリデータを受け付ける目的には、このフォーマットを使うことを推奨します。**

y# (**読み出し専用の** *bytes-like object*) [const char *, *Py_ssize_t*]
: s# の変形で、Unicode オブジェクトを受け付けず、bytes-like object だけを受け付けます。

S (bytes) [PyBytesObject *]
: Python オブジェクトとして、bytes オブジェクトを要求し、いかなる変換も行いません。オブジェクトが bytes オブジェクトでなければ、TypeError を送出します。C 変数は *PyObject** と宣言しても構いません。

Y (bytearray) [PyByteArrayObject *]
: Python オブジェクトとして bytearray オブジェクトを要求し、いかなる変換もおこないません。もしオブジェクトが bytearray でなければ、TypeError を送出します。C 変数は *PyObject** として宣言しても構いません。

U (str) [PyObject *]
: Python オブジェクトとして Unicode オブジェクトを要求し、いかなる変換も行いません。オブジェクトが Unicode オブジェクトではない場合、TypeError が送出されます。C 変数は *PyObject** として宣言しても構いません。

w* (**読み書き可能な** *bytes-like object*) [Py_buffer]
: このフォーマットは、読み書き可能な buffer interface を実装したオブジェクトを受け付けます。呼び出し元から渡された *Py_buffer* 構造体に値を格納します。バッファは null バイトを含むかもしれず、呼び出し元はバッファを使い終わったら *PyBuffer_Release()* を呼び出さなければなりません。

es (str) [const char *encoding, char **buffer]
: これは s の変化形で、Unicode をキャラクタ型バッファにエンコードするために用いられます。NUL バイトが埋め込まれていないデータでのみ動作します。

  この書式には二つの引数が必要です。一つ目は入力にのみ用いられ、NUL で終端されたエンコード名文字列を指す const char* 型または、'utf-8' が使われることを表す NULL でなければなりません。指定したエンコード名を Python が理解できない場合には例外を送出します。第二の引数は char** でなければなりません; この引数が参照しているポインタの値は、引数に指定したテキストの内容が入ったバッファへのポインタになります。テキストは最初の引数に指定したエンコード方式でエンコードされます。

  PyArg_ParseTuple() を使うと、必要なサイズのバッファを確保し、そのバッファにエンコード後のデータをコピーして、**buffer* がこの新たに確保された記憶領域を指すように変更します。呼び出し側には、確保されたバッファを使い終わった後に *PyMem_Free()* で解放する責任があります。

et (str, bytes **または** bytearray) [const char *encoding, char **buffer]
: es と同じです。ただし、バイト文字列オブジェクトをエンコードし直さずに渡します。その代わり、実装ではバイト文字列オブジェクトがパラメタに渡したエンコードを使っているものと仮定します。

es# (str) [const char *encoding, char **buffer, *Py_ssize_t* *buffer_length]
: s# の変化形で、Unicode をキャラクタ型バッファにエンコードするために用いられます。es 書式違って、この変化形はバイトが埋め込まれていてもかまいません。

  この書式には三つの引数が必要です。一つ目は入力にのみ用いられ、NUL で終端されたエンコード名文字列を指す const char* 型か NULL でなければなりません。NULL の場合には 'utf-8' を使います。

指定したエンコード名を Python が理解できない場合には例外を送出します。第二の引数は `char**` でなければなりません; この引数が参照しているポインタの値は、引数に指定したテキストの内容が入ったバッファへのポインタになります。テキストは最初の引数に指定したエンコード方式でエンコードされます。第三の引数は整数へのポインタでなければなりません; ポインタが参照している整数の値は出力バッファ内のバイト数にセットされます。

この書式の処理には二つのモードがあります:

buffer* が `NULL` ポインタを指している場合、関数は必要なサイズのバッファを確保し、そのバッファにエンコード後のデータをコピーして、buffer* がこの新たに確保された記憶領域を指すように変更します。呼び出し側には、確保されたバッファを使い終わった後に *PyMem_Free()* で解放する責任があります。

buffer* が非 `NULL` のポインタ (すでにメモリ確保済みのバッファ) を指している場合、*PyArg_ParseTuple()* はこのメモリ位置をバッファとして用い、buffer_length* の初期値をバッファサイズとして用います。PyArg_ParseTuple は次にエンコード済みのデータをバッファにコピーして、NUL で終端します。バッファの大きさが足りなければ `ValueError` がセットされます。

どちらの場合も、**buffer_length* は終端の NUL バイトを含まないエンコード済みデータの長さにセットされます。

`et#` (`str`, `bytes` または `bytearray`) [const char *encoding, char **buffer, *Py_ssize_t* *buffer_length]
: `es#` と同じです。ただし、バイト文字列オブジェクトをエンコードし直さずに渡します。その代わり、実装ではバイト文字列オブジェクトがパラメタに渡したエンコードを使っているものと仮定します。

バージョン 3.12 で変更: `u`, `u#`, `Z`, and `Z#` are removed because they used a legacy `Py_UNICODE*` representation.

数

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]
: Python の整数を、C の `short int` 型に変換します。

`H` (`int`) [unsigned short int]
: Python の整数を、オーバフローチェックを行わずに、C の `unsigned short int` 型に変換します。

`i` (`int`) [int]
: Python の整数を、C の `int` 型に変換します。

`I` (`int`) [unsigned int]
: Python の整数を、オーバフローチェックを行わずに、C の `unsigned int` 型に変換します。

`l` (`int`) [long int]
: Python の整数を、C の `long int` 型に変換します。

`k` (`int`) [unsigned long]
: Python の整数を、オーバフローチェックを行わずに、C の `unsigned long` 型に変換します。

`L` (`int`) [long long]
: Python の整数を、C の `long long` 型に変換します。

`K` (`int`) [unsigned long long]
: Python の int を C `unsigned long long` へオーバーフローの確認をせず変換する

`n` (`int`) [*Py_ssize_t*]
: Python の整数を C の *Py_ssize_t* 型に変換します。

`c` (**長さ 1 の、**`bytes` **または** `bytearray`) [char]
: 長さ 1 の `bytes` または `bytearray` オブジェクトとして表現されている Python バイトを C の `char` 型に変換します。

  バージョン 3.3 で変更: `bytearray` を受け付けるようになりました。

`C` (**長さ 1 の** `str`) [int]
: 長さ 1 の `str` オブジェクトとして表現されている Python キャラクタを C の `int` 型に変換します。

`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]
: Python の複素数型を、C の *Py_complex* 構造体に変換します。

その他のオブジェクト

`O` (object) [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!` (object) [*typeobject*, PyObject *]
: Python オブジェクトを C の Python オブジェクト型ポインタに保存します。`O` に似ていますが、二つの C の引数をとります: 一つ目の引数は Python の型オブジェクトへのアドレスで、二つ目の引数はオブジェクトへのポインタが保存されている (*PyObject** の) C の変数へのアドレスです。Python オブジェクトが指定した型ではない場合、`TypeError` を送出します。

`O&` (object) [*converter*, *address*]

Python オブジェクトを *converter* 関数を介して C の変数に変換します。二つの引数をとります: 一つ目は関数で、二つ目は (任意の型の) C 変数へのアドレスを `void*` 型に変換したものです。*converter* は以下のようにして呼び出されます:

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

ここで *object* は変換対象の Python オブジェクトで、*address* は `PyArg_Parse*` に渡した `void*` 型の引数です。戻り値 *status* は変換に成功した際に `1`, 失敗した場合には `0` になります。変換に失敗した場合、*converter* 関数は *address* の内容を変更せずに例外を送出しなくてはなりません。 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()*.

バージョン 3.1 で変更: `Py_CLEANUP_SUPPORTED` was added.

`p` (`bool`) [int] 真

偽値が求められる箇所 (a boolean **p**redicate) に渡された値を判定し、その結果を等価な C の true/false 整数値に変換します。もし式が真なら int には `1` が、偽なら `0` が設定されます。この関数は任意の有効な Python 値を受け付けます。Python が値の真偽をどのように判定するかを知りたければ、truth を参照してください。

Added in version 3.3.

`(items)` (`tuple`) [*matching-items*]

The object must be a Python sequence 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.

その他、書式文字列において意味を持つ文字がいくつかあります。それらの文字は括弧による入れ子内には使えません。以下に文字を示します:

`|`

Python 引数リスト中で、この文字以降の引数がオプションであることを示します。オプションの引数に対応する C の変数はデフォルトの値で初期化しておかなければなりません --- オプションの引数が省略された場合、*PyArg_ParseTuple()* は対応する C 変数の内容に手を加えません。

`$`

PyArg_ParseTupleAndKeywords() でのみ使用可能: 後続の Python 引数がキーワード専用であることを示します。現在、すべてのキーワード専用引数は任意の引数でなければならず、そのため フォーマット文字列中の `|` は常に `$` より前に指定されなければなりません。

Added in version 3.3.

`:` こ

の文字があると、書式単位の記述はそこで終わります; コロン以降の文字列は、エラーメッセージに

おける関数名 (*PyArg_ParseTuple()* が送出する例外の ” 付属値 (associated value)”) として使われます。

; この文字があると、書式単位の記述はそこで終わります; セミコロン以降の文字列は、デフォルトエラーメッセージを **置き換える** エラーメッセージとして使われます。: と ; は相互に排他の文字です。

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)!

以下の関数に渡す補助引数 (additional argument) は、書式文字列から決定される型へのアドレスでなければなりません; 補助引数に指定したアドレスは、タプルから入力された値を保存するために使います。上の書式単位のリストで説明したように、補助引数を入力値として使う場合がいくつかあります; その場合、対応する書式単位の指定する形式に従うようにしなければなりません。

変換を正しく行うためには、*arg* オブジェクトは書式文字に一致しなければならず、かつ書式文字列内の書式単位に全て値が入るようにしなければなりません。成功すると、`PyArg_Parse*` 関数は真を返します。それ以外の場合には偽を返し、適切な例外を送出します。書式単位のどれかの変換失敗により `PyArg_Parse*` が失敗した場合、失敗した書式単位に対応するアドレスとそれ以降のアドレスの内容は変更されません。

API 関数

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

次に属します: Stable ABI. 位置引数のみを引数にとる関数のパラメタを解釈して、ローカルな変数に変換します。成功すると真を返します; 失敗すると偽を返し、適切な例外を送出します。

int **PyArg_VaParse**(*PyObject* *args, const char *format, va_list vargs)

次に属します: Stable ABI. *PyArg_ParseTuple()* と同じですが、可変長の引数ではなく *va_list* を引数にとります。

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

次に属します: 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. Empty names denote *positional-only parameters*. Returns true on success; on failure, it returns false and raises the appropriate exception.

バージョン 3.6 で変更: **位置専用引数** を追加した。

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

次に属します: Stable ABI. *PyArg_ParseTupleAndKeywords()* と同じですが、可変長の引数ではなく *va_list* を引数にとります。

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

次に属します: Stable ABI. キーワード引数を格納した辞書のキーが文字列であることを確認します。この関数は *PyArg_ParseTupleAndKeywords()* を使用しないときにのみ必要で、その理由は後者の関数は同様のチェックを実施するためです。

Added in version 3.2.

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

次に属します: Stable ABI. Function used to deconstruct the argument lists of "old-style" functions --- these are functions which use the `METH_OLDARGS` parameter parsing method, which has been removed in Python 3. This is not recommended for use in parameter parsing in new code, and most code in the standard interpreter has been modified to no longer use this for that purpose. It does remain a convenient way to decompose other tuples, however, and may continue to be used for that purpose.

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

次に属します: 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;
}
```

この例における *PyArg_UnpackTuple()* 呼び出しは、*PyArg_ParseTuple()* を使った以下の呼び出しと全く等価です:

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

6.6.2 値の構築

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

戻り値: 新しい参照。次に属します: Stable ABI. PyArg_Parse* ファミリの関数が受け取るのと似た形式の書式文字列および値列に基づいて、新たな値を生成します。生成した値を返します。エラーの場合には NULL を返します; NULL を返す場合、例外を送出するでしょう。

Py_BuildValue() は常にタプルを生成するとは限りません。この関数がタプルを生成するのは、書式文字列に二つ以上の書式単位が入っているときだけです。書式文字列が空の場合 None を返します; 書式単位が厳密に一つだけ入っている場合、書式単位で指定されている何らかのオブジェクト単体を返します。サイズがゼロや 1 のタプルを返すように強制するには、丸括弧で囲われた書式文字列を使います。

書式単位 s や s# の場合のように、オブジェクトを構築する際にデータを供給するためにメモリバッファをパラメタとして渡す場合には、指定したデータはコピーされます。*Py_BuildValue()* が生成したオブジェクトは、呼び出し側が提供したバッファを決して参照しません。別の言い方をすれば、malloc() を呼び出してメモリを確保し、それを *Py_BuildValue()* に渡した場合、コード内で *Py_BuildValue()* が返った後で free() を呼び出す責任があるということです。

以下の説明では、引用符のついた形式は書式単位です; (丸) 括弧で囲った部分は書式単位が返す Python のオブジェクト型です; [角] 括弧は関数に渡す値の C 変数型です。

書式文字列内では、(s# のような書式単位を除いて) スペース、タブ、コロンおよびコンマは無視されます。これらの文字を使うと、長い書式文字列をちょっとだけ読みやすくできます。

s (str または None) [const char *]
: null 終端された C 文字列を、'utf-8' エンコーディングを用いて、Python str オブジェクトに変換します。もし C 文字列ポインタが NULL の場合、None になります。

s# (str or None) [const char *, *Py_ssize_t*]
: 文字列とその長さを 'utf-8' エンコーディングを使って Python str オブジェクトに変換します。C 文字列ポインタが NULL の場合、長さは無視され、None になります。

y (bytes) [const char *]
: 文字列を Python bytes オブジェクトに変換します。もし C 文字列ポインタが NULL だった場合、None を返します。

y# (bytes) [const char *, *Py_ssize_t*]
: これは C 文字列とその長さから Python オブジェクトに変換します。C 文字列ポインタが NULL の場合、長さは無視され None になります。

z (str または None) [const char *]
: と同じです。

z# (str または None) [const char *, *Py_ssize_t*]
: s# と同じです。

u (str) [const wchar_t *]
: null 終端された Unicode (UTF-16 または UCS-4) データの wchar_t バッファから Python

Unicode オブジェクトに変換します。Unicode バッファポインタが NULL の場合、None になります。

u# (str) [const wchar_t *, *Py_ssize_t*]
Unicode (UTF-16 または UCS-4) データのバッファとその長さから Python Unicode オブジェクトに変換します。Unicode バッファポインタが NULL の場合、長さは無視され None になります。

U (str または None) [const char *] s
と同じです。

U# (str または None) [const char *, *Py_ssize_t*]
s# と同じです。

i (int) [int] 通
常の C の int を Python の整数オブジェクトに変換します。

b (int) [char] 通
常の C の char を Python の整数オブジェクトに変換します。

h (int) [short int] 通
常の C の short int を Python の整数オブジェクトに変換します。

l (int) [long int] C
の long int を Python の整数オブジェクトに変換します。

B (int) [unsigned char] C
の unsigned char を Python の整数オブジェクトに変換します。

H (int) [unsigned short int] C
の unsigned short int を Python の整数オブジェクトに変換します。

I (int) [unsigned int] C
の unsigned int を Python の整数オブジェクトに変換します。

k (int) [unsigned long] C
の unsigned long を Python の整数オブジェクトに変換します。

L (int) [long long] C
の long long を Python の整数オブジェクトに変換します。

K (int) [unsigned long long] C
unsigned long long を Python の int オブジェクトへ変換する。

n (int) [*Py_ssize_t*] C
の *Py_ssize_t* を Python の整数オブジェクトに変換します。

c (長さが 1 の bytes) [char] バ
イトを表す通常の C の int を、長さ 1 の Python の bytes オブジェクトに変換します。

C (長さ 1 の str) [int] 文
字を表す通常の C の int を、長さ 1 の Python の str オブジェクトに変換します。

`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 *] C
: の *`Py_complex`* 構造体を Python の複素数型に変換します。

`O` (object) [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` (object) [PyObject *] `O`
: と同じです。

`N` (object) [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&` (object) [*converter*, *anything*]
: *anything* を *converter* 関数を介して Python オブジェクトに変換します。この関数は *anything* (`void*` と互換の型でなければなりません) を引数にして呼び出され、”新たな”オブジェクトを返すか、失敗した場合には `NULL` を返すようにしなければなりません。

`(items)` (`tuple`) [*matching-items*] C
: の値からなる配列を、同じ要素数を持つ Python のタプルに変換します。

`[items]` (`list`) [*matching-items*] C
: の値からなる配列を、同じ要素数を持つ Python のリストに変換します。

`{items}` (`dict`) [*matching-items*] C
: の値からなる配列を Python の辞書に変換します。一連のペアからなる C の値が、それぞれキーおよび値となって辞書に追加されます。

書式文字列に関するエラーが生じると、`SystemError` 例外をセットして `NULL` を返します。

PyObject *`Py_VaBuildValue`(const char *format, va_list vargs)
: **戻り値: 新しい参照。次に属します:** Stable ABI. *`Py_BuildValue()`* と同じですが、可変長引数の代わりに va_list を受け取ります。

6.7 文字列の変換と書式化

数値変換と、書式化文字列出力のための関数群。

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

> **次に属します**: Stable ABI. 書式文字列 *format* と追加の引数から、*size* バイトを超えない文字列を *str* に出力します。Unix man page の *snprintf(3)* を参照してください。

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

> **次に属します**: Stable ABI. 書式文字列 *format* と可変長引数リスト *va* から、*size* バイトを超えない文字列を *str* に出力します。Unix man page の *vsnprintf(3)* を参照してください。

PyOS_snprintf() と *PyOS_vsnprintf()* は標準 C ライブラリの `snprintf()` と `vsnprintf()` 関数をラップします。これらの関数の目的は、C 標準ライブラリが保証していないコーナーケースでの動作を保証することです。

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.

これらの関数の戻り値 (以下では *rv* とします) は以下の意味を持ちます:

- `0 <= rv < size` のとき、変換出力は成功して、(最後の `str[rv]` にある `'\0'` を除いて) *rv* 文字が *str* に出力された。
- `rv >= size` のとき、変換出力は切り詰められており、成功するためには `rv + 1` バイトが必要だったことを示します。`str[size-1]` は `'\0'` です。
- `rv < 0` のときは、何か悪いことが起こった時です。この場合でも `str[size-1]` は `'\0'` ですが、*str* のそれ以外の部分は未定義です。エラーの正確な原因はプラットフォーム依存です。

以下の関数は locale 非依存な文字列から数値への変換を行ないます。

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

> **次に属します**: 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)*.
>
> Added in version 3.2.

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

次に属します: 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)*.

Added in version 3.2.

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

次に属します: Stable ABI. 文字列 `s` を `double` に変換します。失敗したときは Python の例外を発生させます。受け入れられる文字列は、Python の `float()` コンストラクタが受け付ける文字列に準拠しますが、`s` の先頭と末尾に空白文字があってはならないという部分が異なります。この変換は現在のロケールに依存しません。

`endptr` が `NULL` の場合、変換は文字列全体に対して行われます。文字列が正しい浮動小数点数の表現になっていない場合は `-1.0` を返して `ValueError` を発生させます。

endptr が `NULL` で無い場合、文字列を可能な範囲で変換して、`*endptr` に最初の変換されなかった文字へのポインタを格納します。文字列の先頭に正しい浮動小数点数の表現が無かった場合、`*endptr` を文字列の先頭に設定して、ValueError を発生させ、`-1.0` を返します。

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

それ以外のエラーが変換中に発生した場合 (例えば out-of-memory エラー)、適切な Python の例外を設定して `-1.0` を返します。

Added in version 3.1.

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

次に属します: Stable ABI. `double` *val* を指定された *format_code*, *precision*, *flags* に基づいて文字列に変換します。

format_code は `'e'`, `'E'`, `'f'`, `'F'`, `'g'`, `'G'`, `'r'` のどれかでなければなりません。`'r'` の場合、*precision* は 0 でなければならず、無視されます。`'r'` フォーマットコードは標準の `repr()` フォーマットを指定しています。

flags は 0 か、`Py_DTSF_SIGN`, `Py_DTSF_ADD_DOT_0`, `Py_DTSF_ALT` か、これらの or を取ったものです:

- `Py_DTSF_SIGN` は、*val* が負で無いときも常に符号文字を先頭につけることを意味します。
- `Py_DTSF_ADD_DOT_0` は文字列が整数のように見えないことを保証します。

- Py_DTSF_ALT は "alternate" フォーマットルールを適用することを意味します。詳細は *PyOS_snprintf()* の '#' 指定を参照してください。

ptype が NULL で無い場合、*val* が有限数、無限数、NaN のどれかに合わせて、Py_DTST_FINITE, Py_DTST_INFINITE, Py_DTST_NAN のいずれかに設定されます。

戻り値は変換後の文字列が格納された *buffer* へのポインタか、変換が失敗した場合は NULL です。呼び出し側は、返された文字列を *PyMem_Free()* を使って解放する責任があります。

Added in 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.

6.8 PyHash API

See also the *PyTypeObject.tp_hash* member.

type **Py_hash_t**

Hash value type: signed integer.

Added in version 3.2.

type **Py_uhash_t**

Hash value type: unsigned integer.

Added in version 3.2.

type **PyHash_FuncDef**

Hash function definition used by *PyHash_GetFuncDef()*.

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.

Added in version 3.4.

PyHash_FuncDef ***PyHash_GetFuncDef**(void)

Get the hash function definition.

> **参考**
>
> **PEP 456** ”Secure and interchangeable hash algorithm”.

Added in version 3.4.

6.9 リフレクション

PyObject ***PyEval_GetBuiltins**(void)

戻り値: **借用参照。次に属します**: Stable ABI. 現在の実行フレーム内のビルトインの辞書か、もし実行中のフレームがなければスレッド状態のインタプリタのビルトイン辞書を返します。

PyObject ***PyEval_GetLocals**(void)

戻り値: **借用参照。次に属します**: Stable ABI. Return a dictionary of the local variables in the current execution frame, or `NULL` if no frame is currently executing.

PyObject ***PyEval_GetGlobals**(void)

戻り値: **借用参照。次に属します**: Stable ABI. 現在の実行フレーム内のグローバル変数の辞書か、実行中のフレームがなければ `NULL` を返します。

PyFrameObject ***PyEval_GetFrame**(void)

戻り値: **借用参照。次に属します**: Stable ABI. Return the current thread state's frame, which is `NULL` if no frame is currently executing.

See also *PyThreadState_GetFrame()*.

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

次に属します: Stable ABI. *func* が関数、クラス、インスタンスオブジェクトであればその名前を、そうでなければ *func* の型を返します。

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

次に属します: Stable ABI. *func* の型に依存する、解説文字列 (description string) を返します。戻り値は、関数とメソッドに対しては ”()”, ” constructor”, ” instance”, ” object” です。*PyEval_GetFuncName()* と連結された結果、*func* の解説になります。

6.10 codec レジストリとサポート関数

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

次に属します: Stable ABI. 新しい codec 検索関数を登録します。

副作用として、この関数は `encodings` パッケージが常に検索関数の先頭に来るように、まだロードされていない場合はロードします。

int **PyCodec_Unregister**(*PyObject* *search_function)

次に属します: Stable ABI *(バージョン 3.10 より)*. 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.

Added in version 3.10.

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

次に属します: Stable ABI. *encoding* のための登録された codec が存在するかどうかに応じて `1` か `0` を返します。この関数は常に成功します。

PyObject ***PyCodec_Encode**(*PyObject* *object, const char *encoding, const char *errors)

戻り値: **新しい参照。次に属します**: Stable ABI. 汎用の codec ベースの encode API.

encoding に応じて見つかったエンコーダ関数に対して *object* を渡します。エラーハンドリングメソッドは *errors* で指定します。*errors* は `NULL` でもよく、その場合はその codec のデフォルトのメソッドが利用されます。エンコーダが見つからなかった場合は `LookupError` を発生させます。

PyObject ***PyCodec_Decode**(*PyObject* *object, const char *encoding, const char *errors)

戻り値: **新しい参照。次に属します**: Stable ABI. 汎用の codec ベースのデコード API.

encoding に応じて見つかったデコーダ関数に対して *object* を渡します。エラーハンドリングメソッドは *errors* で指定します。*errors* は `NULL` でもよく、その場合はその codec のデフォルトのメソッドが利用されます。デコーダが見つからなかった場合は `LookupError` を発生させます。

6.10.1 コーデック検索 API

次の関数では、文字列 *encoding* は全て小文字に変換することで、効率的に、大文字小文字を無視した検索をします。コーデックが見つからない場合、`KeyError` を設定して `NULL` を返します。

PyObject ***PyCodec_Encoder**(const char *encoding)

戻り値: **新しい参照。次に属します**: Stable ABI. 与えられた *encoding* のエンコーダ関数を返します。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 与えられた *encoding* のデコーダ関数を返します。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 与えられた *encoding* の `IncrementalEncoder` オブジェクトを返します。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 与えられた *encoding* の `IncrementalDecoder` オブジェクトを返します。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 与えられた *encoding* の `StreamReader` ファクトリ関数を返します。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 与えられた *encoding* の StreamWriter ファクトリ関数を返します。

6.10.2 Unicode エラーハンドラ用レジストリ API

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

次に属します: Stable ABI. エラーハンドルのためのコールバック関数 *error* を *name* で登録します。このコールバック関数は、コーデックがエンコードできない文字/デコードできないバイトに遭遇した時に、そのエンコード/デコード関数の呼び出しで *name* が指定されていたら呼び出されます。

コールバックは 1 つの引数として、UnicodeEncodeError, UnicodeDecodeError, UnicodeTranslateError のどれかのインスタンスを受け取ります。このインスタンスは問題のある文字列やバイト列に関する情報と、その元の文字列中のオフセットを持っています。(その情報を取得するための関数については *Unicode* **例外オブジェクト** を参照してください。) コールバックは渡された例外を発生させるか、2 要素のタプルに問題のシーケンスの代替と、encode/decode を再開する元の文字列中のオフセットとなる整数を格納して返します。

成功したら 0 を、エラー時は -1 を返します。

PyObject *PyCodec_LookupError(const char *name)

戻り値: **新しい参照。次に属します**: Stable ABI. *name* で登録されたエラーハンドリングコールバック関数を検索します。特別な場合として、NULL が渡された場合、”strict” のエラーハンドリングコールバック関数を返します。

PyObject *PyCodec_StrictErrors(*PyObject* *exc)

戻り値: **常に** *NULL* **。次に属します**: Stable ABI. *exc* を例外として発生させます。

PyObject *PyCodec_IgnoreErrors(*PyObject* *exc)

戻り値: **新しい参照。次に属します**: Stable ABI. unicode エラーを無視し、問題の入力をスキップします。

PyObject *PyCodec_ReplaceErrors(*PyObject* *exc)

戻り値: **新しい参照。次に属します**: Stable ABI. unicode エラーを ? か U+FFFD で置き換えます。

PyObject *PyCodec_XMLCharRefReplaceErrors(*PyObject* *exc)

戻り値: **新しい参照。次に属します**: Stable ABI. unicode encode エラーを XML 文字参照で置き換えます。

PyObject *PyCodec_BackslashReplaceErrors(*PyObject* *exc)

戻り値: **新しい参照。次に属します**: Stable ABI. unicode encode エラーをバックスラッシュエスケープ (\x, \u, \U) で置き換えます。

PyObject *PyCodec_NameReplaceErrors(*PyObject* *exc)

戻り値: **新しい参照。次に属します**: Stable ABI *(**バージョン** 3.7 **より**)*. unicode encode エラーを \N{...} で置き換えます。

Added in version 3.5.

6.11 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 the Global Interpreter Lock (GIL) is not required for these APIs.

int **PyUnstable_PerfMapState_Init**(void)

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

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)

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

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)

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

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.

第

SEVEN

抽象オブジェクトレイヤ (ABSTRACT OBJECTS LAYER)

この章で説明する関数は、オブジェクトの型に依存しないような Python オブジェクトの操作や、(数値型全て、シーケンス型全てといった) 大まかな型のオブジェクトに対する操作を行ないます。関数を適用対象でないオブジェクトに対して使った場合、Python の例外が送出されることになります。

これらの関数は、*PyList_New()* で作成された後に NULL 以外の値を設定されていないリストのような、適切に初期化されていないオブジェクトに対して使うことはできません。

7.1 オブジェクトプロトコル (object protocol)

PyObject ***Py_NotImplemented**

与えられたオブジェクトとメソッドの引数の型の組み合わせの処理が未実装である印として使われる、**未実装** (NotImplemented) シングルトン。

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_HasAttr**(*PyObject* *o, *PyObject* *attr_name)

次に属します: Stable ABI. Returns 1 if *o* has the attribute *attr_name*, and 0 otherwise. This is equivalent to the Python expression hasattr(o, attr_name). This function always succeeds.

注釈

Exceptions that occur when this calls `__getattr__()` and `__getattribute__()` methods are silently ignored. For proper error handling, use *PyObject_GetAttr()* instead.

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

次に属します: 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**.

注釈

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_GetAttrString()* instead.

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

戻り値: 新しい参照。次に属します: Stable ABI. オブジェクト *o* から、名前 *attr_name* の属性を取得します。成功すると属性値を返し失敗すると `NULL` を返します。この関数は Python の式 `o.attr_name` と同じです。

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

戻り値: 新しい参照。次に属します: 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**.

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

戻り値: 新しい参照。次に属します: Stable ABI. 型オブジェクトの `tp_getattro` スロットに置かれる、属性を取得する総称的な関数です。この関数は、(もし存在すれば) オブジェクトの属性 `__dict__` に加え、オブジェクトの MRO にあるクラスの辞書にあるデスクリプタを探します。descriptors で概要が述べられている通り、データのデスクリプタはインスタンスの属性より優先され、非データデスクリプタは後回しにされます。見付からなかった場合は `AttributeError` を送出します。

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

次に属します: Stable ABI. オブジェクト *o* の *attr_name* という名の属性に、値 *v* を設定します。失敗すると例外を送出し `-1` を返します; 成功すると `0` を返します。この関数は Python の式 `o.attr_name = v` と同じです。

v が `NULL` のとき、アトリビュートは削除されます。この動作は *PyObject_DelAttr()* のため、非推奨となっていますが、削除される予定はありません。

int **PyObject_SetAttrString**(*PyObject* *o, const char *attr_name, *PyObject* *v)

次に属します: 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**.

v が `NULL` の場合は属性が削除されますが、この機能は非推奨であり *PyObject_DelAttrString()* を使うのが望ましいです。

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)

次に属します: Stable ABI. 属性の設定と削除を行う汎用的な関数で、型オブジェクトの *tp_setattro* スロットに置かれます。オブジェクトの MRO にあるクラスの辞書からデータディスクリプタを探し、見付かった場合はインスタンスの辞書にある属性の設定や削除よりも優先されます。そうでない場合は、(もし存在すれば) オブジェクトの `__dict__` に属性を設定もしくは削除します。成功すると `0` が返され、そうでない場合は `AttributeError` が送出され `-1` が返されます。

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

オブジェクト *o* の *attr_name* という名の属性を削除します。失敗すると `-1` を返します。この関数は Python の文 `del o.attr_name` と同じです。

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

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)

戻り値: **新しい参照。次に属します**: Stable ABI *(**バージョン** 3.10 **より**)*. `__dict__` デスクリプタの getter の総称的な実装です。必要な場合は、辞書を作成します。

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.

Added in version 3.3.

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

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. `__dict__` デスクリプタの setter の総称的な実装です。この実装では辞書を削除することは許されていません。

Added in 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)

戻り値: **新しい参照。次に属します**: 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)

次に属します: 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.

注釈

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)

次に属します: 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)

戻り値: **新しい参照。次に属します**: Stable ABI. オブジェクト *o* の文字列表現を計算します。成功すると文字列表現を返し、失敗すると `NULL` を返します。Python 式 `repr(o)` と同じです。この関数は組み込み関数 `repr()` の処理で呼び出されます。

バージョン 3.4 で変更: アクティブな例外を黙って捨てないことを保証するのに便利なように、この関数はデバッグアサーションを含むようになりました。

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

戻り値: **新しい参照。次に属します**: Stable ABI. *PyObject_Repr()* と同様、オブジェクト *o* の文字列表現を計算しますが、*PyObject_Repr()* によって返された文字列に含まれる非 ASCII 文字を、エスケープ文字 `\x`、`\u`、`\U` でエスケープします。この関数は Pythoh 2 の *PyObject_Repr()* が返す文字列と同じ文字列を生成します。`ascii()` によって呼び出されます。

PyObject ***PyObject_Str**(*PyObject* *o)

戻り値: **新しい参照。次に属します**: Stable ABI. オブジェクト *o* の文字列表現を計算します。成功すると文字列表現を返し、失敗すると `NULL` を返します。Python 式 `str(o)` と同じです。この関数は組み込み関数 `str()` や、`print()` 関数の処理で呼び出されます。

バージョン 3.4 で変更: アクティブな例外を黙って捨てないことを保証するのに便利なように、この関数はデバッグアサーションを含むようになりました。

PyObject *PyObject_Bytes(*PyObject* *o)

戻り値: **新しい参照。次に属します**: Stable ABI. オブジェクト *o* のバイト列表現を計算します。失敗すると NULL を返し、成功すると bytes オブジェクトを返します。*o* が整数でないときの、Python 式 bytes(o) と同じです。bytes(o) と違って、*o* が整数のときには、ゼロで初期化された bytes オブジェクトを返すのではなく TypeError が送出されます。

int PyObject_IsSubclass(*PyObject* *derived, *PyObject* *cls)

次に属します: Stable ABI. クラス *derived* がクラス *cls* と同一であるか、そこから派生したクラスである場合は 1 を返し、そうでない場合は 0 を返します。エラーが起きた場合は -1 を返します。

cls がタプルの場合、*cls* の全ての要素に対してチェックします。少なくとも 1 つのチェックで 1 が返ったとき、結果は 1 となり、それ以外のとき 0 になります。

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)

次に属します: Stable ABI. *inst* がクラス *cls* もしくは *cls* の子クラスのインスタンスである場合に 1 を返し、そうでない場合に 0 を返します。エラーが起きると -1 を返し例外を設定します。

cls がタプルの場合、*cls* の全ての要素に対してチェックします。少なくとも 1 つのチェックで 1 が返ったとき、結果は 1 となり、それ以外のとき 0 になります。

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)

次に属します: Stable ABI. オブジェクト *o* のハッシュ値を計算して返します。失敗すると -1 を返します。Python の式 hash(o) と同じです。

バージョン 3.2 で変更: 返り値の型が Py_hash_t になりました。この型は、*Py_ssize_t* と同じサイズをもつ符号付き整数です。

Py_hash_t PyObject_HashNotImplemented(*PyObject* *o)

次に属します: 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)

次に属します: Stable ABI. *o* が真を表すとみなせる場合には 1 を、そうでないときには 0 を返します。Python の式 not not o と同じです。失敗すると -1 を返します。

int PyObject_Not(*PyObject* *o)

次に属します: Stable ABI. *o* が真を表すとみなせる場合には 0 を、そうでないときには 1 を返します。Python の式 not o と同じです。失敗すると -1 を返します。

PyObject *PyObject_Type(*PyObject* *o)

戻り値: 新しい参照。次に属します: 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)

次に属します: Stable ABI. *o* の長さを返します。ただしオブジェクト *o* がシーケンス型プロトコルとマップ型プロトコルの両方を提供している場合、シーケンスとしての長さを返します。エラーが生じると -1 を返します。Python の式 len(o) と同じです。

Py_ssize_t PyObject_LengthHint(*PyObject* *o, *Py_ssize_t* defaultvalue)

オブジェクト *o* の概算の長さを返します。最初に実際の長さを、次に __length_hint__() を使って概算の長さを、そして最後にデフォルトの値を返そうとします。この関数は Python の式 operator.length_hint(o, defaultvalue) と同じです。

Added in version 3.4.

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

戻り値: 新しい参照。次に属します: Stable ABI. オブジェクト *key* に対応する *o* の要素を返します。失敗すると NULL を返します。Python の式 o[key] と同じです。

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

次に属します: Stable ABI. オブジェクト *key* を値 *v* に対応付けます。失敗すると、例外を送出し -1 を返します。成功すると 0 を返します。これは Python の文 o[key] = v と同等です。この関数は *v* への参照を **盗み取りません** 。

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

次に属します: Stable ABI. オブジェクト *o* から *key* に関する対応付けを削除します。失敗すると -1 を返します。Python の文 del o[key] と同じです。

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

次に属します: 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)

戻り値: 新しい参照。次に属します: Stable ABI. この関数は Python の式 `dir(o)` と同じで、オブジェクトの変数名に割り当てている文字列からなるリスト (空の場合もあります) を返します。エラーの場合には `NULL` を返します。引数を `NULL` にすると、Python における `dir()` と同様に、現在のローカルな名前を返します; この場合、アクティブな実行フレームがなければ `NULL` を返しますが、*PyErr_Occurred()* は偽を返します。

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

戻り値: 新しい参照。次に属します: Stable ABI. Python の式 `iter(o)` と同じです。引数にとったオブジェクトに対する新たなイテレータか、オブジェクトがすでにイテレータの場合にはオブジェクト自身を返します。オブジェクトが反復処理不可能であった場合には `TypeError` を送出して `NULL` を返します。

PyObject ***PyObject_SelfIter**(*PyObject* *obj)

戻り値: 新しい参照。次に属します: 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)

戻り値: 新しい参照。次に属します: Stable ABI *(バージョン 3.10 より)*. 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.

Added in version 3.10.

void ***PyObject_GetTypeData**(*PyObject* *o, *PyTypeObject* *cls)

次に属します: Stable ABI *(バージョン 3.12 より)*. 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`.

Added in version 3.12.

Py_ssize_t **PyType_GetTypeDataSize**(*PyTypeObject* *cls)

次に属します: Stable ABI *(バージョン 3.12 より)*. 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.

Added in 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.

Added in version 3.12.

7.2 Call プロトコル

CPython supports two different calling protocols: *tp_call* and vectorcall.

7.2.1 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*.

7.2.2 The Vectorcall Protocol

Added in 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:

> **警告**
>
> A class supporting vectorcall **must** also implement *tp_call* with the same semantics.

バージョン 3.12 で変更: 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)

次に属します: Stable ABI *(**バージョン** 3.12 **より**)*.

- *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`

次に属します: Stable ABI *(**バージョン** 3.12 **より**)*. 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.

Added in 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.

注釈

In CPython 3.8, the vectorcall API and related functions were available provisionally under names with a leading underscore: `_PyObject_Vectorcall`, `_Py_TPFLAGS_HAVE_VECTORCALL`, `_PyObject_VectorcallMethod`, `_PyVectorcall_Function`, `_PyObject_CallOneArg`, `_PyObject_CallMethodNoArgs`, `_PyObject_CallMethodOneArg`. Additionally, `PyObject_VectorcallDict` was available as `_PyObject_FastCallDict`. The old names are still defined as aliases of the new, non-underscored names.

再帰の管理

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)

次に属します: Stable ABI *(バージョン 3.12 より)*. 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.

Added in 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`.

Added in version 3.9.

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

次に属します: Stable ABI *(バージョン 3.12 より)*. 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`.

Added in version 3.8.

7.2.3 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.

関数	callable	args	kwargs
PyObject_Call()	`PyObject *`	tuple	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)

戻り値: 新しい参照。次に属します: Stable ABI. 呼び出し可能な Python のオブジェクト *callable* を、タプル *args* として与えられる引数と辞書 *kwargs* として与えられる名前付き引数とともに呼び出します。

args は *NULL* であってはならず、引数を必要としない場合は空のタプルを使ってください。*kwargs* は *NULL* でも構いません。

成功したら呼び出しの結果を返し、失敗したら例外を送出し *NULL* を返します。

これは次の Python の式と同等です: `callable(*args, **kwargs)` 。

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

戻り値: 新しい参照。次に属します: Stable ABI *(バージョン 3.10 より)*. Call a callable Python object *callable* without any arguments. It is the most efficient way to call a callable Python object without any argument.

成功したら呼び出しの結果を返し、失敗したら例外を送出し *NULL* を返します。

Added in version 3.9.

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

戻り値: 新しい参照。 Call a callable Python object *callable* with exactly 1 positional argument *arg*

and no keyword arguments.

成功したら呼び出しの結果を返し、失敗したら例外を送出し *NULL* を返します。

Added in version 3.9.

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

戻り値: **新しい参照。次に属します**: Stable ABI. 呼び出し可能な Python のオブジェクト *callable* を、タプル *args* として与えられる引数とともに呼び出します。引数が必要な場合は、*args* は *NULL* で構いません。

成功したら呼び出しの結果を返し、失敗したら例外を送出し *NULL* を返します。

これは次の Python の式と同等です: `callable(*args)` 。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 呼び出し可能な Python オブジェクト *callable* を可変数個の C 引数とともに呼び出します。C 引数は *Py_BuildValue()* 形式のフォーマット文字列を使って記述します。*format* は *NULL* かもしれず、与える引数がないことを表します。

成功したら呼び出しの結果を返し、失敗したら例外を送出し *NULL* を返します。

これは次の Python の式と同等です: `callable(*args)` 。

*PyObject** args だけを引数に渡す場合は、*PyObject_CallFunctionObjArgs()* がより速い方法であることを覚えておいてください。

バージョン 3.4 で変更: *format* の型が `char *` から変更されました。

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

戻り値: **新しい参照。次に属します**: Stable ABI. オブジェクト *obj* の *name* という名前のメソッドを、いくつかの C 引数とともに呼び出します。C 引数はタプルを生成する *Py_BuildValue()* 形式のフォーマット文字列で記述されています。

format は *NULL* でもよく、引数が与えられないことを表します。

成功したら呼び出しの結果を返し、失敗したら例外を送出し *NULL* を返します。

これは次の Python の式と同等です: `obj.name(arg1, arg2, ...)` 。

*PyObject** args だけを引数に渡す場合は、*PyObject_CallMethodObjArgs()* がより速い方法であることを覚えておいてください。

バージョン 3.4 で変更: *name* と *format* の型が `char *` から変更されました。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 呼び出し可能な Python オブジェクト *callable* を可変数個の *PyObject** 引数とともに呼び出します。引数列は末尾に *NULL* がついた可変数個のパラメタとして与えます。

成功したら呼び出しの結果を返し、失敗したら例外を送出し *NULL* を返します。

これは次の Python の式と同等です: `callable(arg1, arg2, ...)` 。

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

戻り値: **新しい参照。次に属します**: Stable ABI. Python オブジェクト *obj* のメソッドを呼び出します、メソッド名は Python 文字列オブジェクト *name* で与えます。可変数個の *PyObject** 引数と共に呼び出されます. 引数列は末尾に *NULL* がついた可変数個のパラメタとして与えます。

成功したら呼び出しの結果を返し、失敗したら例外を送出し *NULL* を返します。

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*.

成功したら呼び出しの結果を返し、失敗したら例外を送出し *NULL* を返します。

Added in 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*.

成功したら呼び出しの結果を返し、失敗したら例外を送出し *NULL* を返します。

Added in version 3.9.

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

次に属します: Stable ABI *(**バージョン** 3.12 **より**)*. 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*.

成功したら呼び出しの結果を返し、失敗したら例外を送出し *NULL* を返します。

Added in 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.

Added in version 3.9.

PyObject ***PyObject_VectorcallMethod**(*PyObject* *name, *PyObject* *const *args, size_t nargsf, *PyObject* *kwnames)

次に属します: Stable ABI *(**バージョン** 3.12 **より**)*. 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.

成功したら呼び出しの結果を返し、失敗したら例外を送出し *NULL* を返します。

Added in version 3.9.

7.2.4 Call Support API

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

次に属します: Stable ABI. オブジェクト *o* が呼び出し可能オブジェクトかどうか調べます。オブジェクトが呼び出し可能であるときに `1` を返し、そうでないときには `0` を返します。この関数呼び出しは常に成功します。

7.3 数値型プロトコル (number protocol)

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

次に属します: Stable ABI. オブジェクト *o* が数値型プロトコルを提供している場合に `1` を返し、そうでないときには偽を返します。この関数呼び出しは常に成功します。

バージョン 3.8 で変更: *o* がインデックス整数だった場合、`1` を返します。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* と *o2* を加算した結果を返し、失敗すると `NULL` を返します。Python の式 `o1 + o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* から *o2* を減算した結果を返し、失敗すると `NULL` を返します。Python の式 `o1 - o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* と *o2* を乗算した結果を返し、失敗すると `NULL` を返します。Python の式 `o1 * o2` と同じです。

PyObject ***PyNumber_MatrixMultiply**(*PyObject* *o1, *PyObject* *o2)

戻り値: **新しい参照。次に属します**: Stable ABI *(**バージョン** 3.7 **より**)*. 成功すると *o1* と *o2* を行列乗算した結果を返し、失敗すると `NULL` を返します。Python の式 `o1 @ o2` と同じです。

Added in version 3.5.

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

戻り値: **新しい参照。次に属します**: 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)

戻り値: **新しい参照。次に属します**: 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)

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* を *o2* で除算した剰余を返し、失敗すると NULL を返します。Python の式 `o1 % o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 組み込み関数 `divmod()` を参照してください。失敗すると NULL を返します。Python の式 `divmod(o1, o2)` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 組み込み関数 `pow()` を参照してください。失敗すると NULL を返します。Python の式 `pow(o1, o2, o3)` と同じです。*o3* はオプションです。*o3* を無視させたいなら、*Py_None* を入れてください (*o3* に NULL を渡すと、不正なメモリアクセスを引き起こすことがあります)。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o* の符号反転を返し、失敗すると NULL を返します。Python の式 `-o` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o* を返し、失敗すると NULL を返します。Python の式 `+o` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o* の絶対値を返し、失敗すると NULL を返します。Python の式 `abs(o)` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o* のビット単位反転 (bitwise negation) を返し、失敗すると NULL を返します。Python の式 `~o` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* を *o2* だけ左シフトした結果を返し、失敗すると NULL を返します。Python の式 `o1 << o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* を *o2* だけ右シフトした結果を返し、失敗すると NULL を返します。Python の式 `o1 >> o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* と *o2* の "ビット単位論理積 (bitwise and)" を返し、失敗すると NULL を返します。Python の式 `o1 & o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* と *o2* の "ビット単位排他的論理和 (bitwise exclusive or)" を返し、失敗すると NULL を返します。Python の式 `o1 ^ o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* と *o2* の "ビット単位論理和 (bitwise or)" を返し失敗すると NULL を返します。Python の式 `o1 | o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* と *o2* を加算した結果を返し、失敗すると NULL を返します。*o1* が *in-place* 演算をサポートする場合、in-place 演算を行います。Python の文 `o1 += o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* から *o2* を減算した結果を返し、失敗すると NULL を返します。*o1* が *in-place* 演算をサポートする場合、in-place 演算を行います。Python の文 `o1 -= o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* と *o2* を乗算した結果を返し、失敗すると NULL を返します。*o1* が *in-place* 演算をサポートする場合、in-place 演算を行います。Python の文 `o1 *= o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI *(**バージョン** 3.7 **より**)*. 成功すると *o1* と *o2* を行列乗算した結果を返し、失敗すると NULL を返します。*o1* が *in-place* 演算をサポートする場合、in-place 演算を行います。Python の文 `o1 @= o2` と同じです。

Added in version 3.5.

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* を *o2* で除算した切捨て値を返し、失敗すると NULL を返します。*o1* が *in-place* 演算をサポートする場合、in-place 演算を行います。Python の文 `o1 //= o2` と同じです。

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

戻り値: **新しい参照。次に属します**: 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)

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* を *o2* で除算した剰余を返し、, 失敗すると `NULL` を返します。*o1* が *in-place* 演算をサポートする場合、in-place 演算を行います。Python の文 `o1 %= o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 組み込み関数 `pow()` を参照してください。失敗すると `NULL` を返します。*o1* が *in-place* 演算をサポートする場合、in-place 演算を行います。この関数は *o3* が *Py_None* の場合は Python 文 `o1 **= o2` と同じで、それ以外の場合は `pow(o1, o2, o3)` の in-place 版です。*o3* を無視させたいなら、*Py_None* を入れてください (*o3* に `NULL` を渡すと、不正なメモリアクセスを引き起こすことがあります)。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* を *o2* だけ左シフトした結果を返し、失敗すると `NULL` を返します。*o1* が *in-place* 演算をサポートする場合、in-place 演算を行います。Python の文 `o1 <<= o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* を *o2* だけ右シフトした結果を返し、失敗すると `NULL` を返します。*o1* が *in-place* 演算をサポートする場合、in-place 演算を行います。Python の文 `o1 >>= o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* と *o2* の ” ビット単位論理積 (bitwise and)” を返し、失敗すると `NULL` を返します。*o1* が *in-place* 演算をサポートする場合、in-place 演算を行います。Python の文 `o1 &= o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* と *o2* の ” ビット単位排他的論理和 (bitwise exclusive or)” を返し、失敗すると `NULL` を返します。*o1* が *in-place* 演算をサポートする場合、in-place 演算を行います。Python の文 `o1 ^= o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* と *o2* の ” ビット単位論理和 (bitwise or)” を返し失敗すると `NULL` を返します。*o1* が *in-place* 演算をサポートする場合、in-place 演算を行います。Python の文 `o1 |= o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o* を整数に変換したものを返し、失敗すると `NULL` を返します。Python の式 `int(o)` と同じです。

PyObject *PyNumber_Float(*PyObject* *o)

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o* を浮動小数点数に変換したものを返し、失敗すると NULL を返します。Python の式 float(o) と同じです。

PyObject *PyNumber_Index(*PyObject* *o)

戻り値: **新しい参照。次に属します**: Stable ABI. *o* を Python の int 型に変換し、成功したらその値を返します。失敗したら NULL が返され、TypeError 例外が送出されます。

バージョン 3.10 で変更: 結果は常に厳密な int 型です。以前は、結果は int のサブクラスのインスタンスのこともありました。

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

戻り値: **新しい参照。次に属します**: Stable ABI. *base* 進数に変換された整数 *n* を文字列として返します。*base* 引数は 2, 8, 10 または 16 のいずれかでなければなりません。基数 2、8、16 について、返される文字列の先頭には基数マーカー '0b' 、'0o' または '0x' が、それぞれ付与されます。もし *n* が Python の int 型でなければ、まず *PyNumber_Index()* で変換されます。

Py_ssize_t PyNumber_AsSsize_t(*PyObject* *o, *PyObject* *exc)

次に属します: Stable ABI. *o* を整数として解釈可能だった場合、*Py_ssize_t* 型の値に変換して返します。呼び出しが失敗したら、例外が送出され、-1 が返されます。

もし *o* が Python の int に変換できたのに、*Py_ssize_t* への変換が OverflowError になる場合は、*exc* 引数で渡された型 (普通は IndexError か OverflowError) の例外を送出します。もし、*exc* が NULL なら、例外はクリアされて、値が負の場合は PY_SSIZE_T_MIN へ、正の場合は PY_SSIZE_T_MAX へと制限されます。

int PyIndex_Check(*PyObject* *o)

次に属します: Stable ABI *(**バージョン** 3.8 **より**)*. *o* がインデックス整数である場合 (tp_as_number 構造体の nb_index スロットが埋まっている場合) に 1 を返し、そうでない場合に 0 を返します。この関数は常に成功します。

7.4 シーケンス型プロトコル (sequence protocol)

int PySequence_Check(*PyObject* *o)

次に属します: 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)

次に属します: Stable ABI. 成功するとシーケンス *o* 中のオブジェクトの数を返し、失敗すると -1 を返します。これは、Python の式 len(o) と同じになります。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* と *o2* の連結 (concatenation) を返し、失敗すると NULL を返します。Python の式 `o1 + o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功するとオブジェクト *o* の *count* 回繰り返しを返し、失敗すると NULL を返します。Python の式 `o * count` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o1* と *o2* の連結 (concatenation) を返し、失敗すると NULL を返します。*o1* が *in-place* 演算をサポートする場合、in-place 演算を行います。Python の式 `o1 += o2` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功するとオブジェクト *o* の *count* 回繰り返しを返し、失敗すると NULL を返します。*o* が *in-place* 演算をサポートする場合、in-place 演算を行います。Python の式 `o *= count` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o* の *i* 番目の要素を返し、失敗すると NULL を返します。Python の式 `o[i]` と同じです。

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

戻り値: **新しい参照。次に属します**: Stable ABI. 成功すると *o* の *i1* から *i2* までの間のスライスを返し、失敗すると NULL を返します。Python の式 `o[i1:i2]` と同じです。

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

次に属します: Stable ABI. *o* の *i* 番目の要素に *v* を代入します。失敗すると、例外を送出し `-1` を返します; 成功すると `0` を返します。これは Python の文 `o[i] = v` と同じです。この関数は *v* への参照を **盗み取りません** 。

v が NULL の場合はその要素が削除されますが、この機能は非推奨であり、*PyObject_DelAttr()* を使うのが望ましいです。

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

次に属します: Stable ABI. *o* の *i* 番目の要素を削除します。失敗すると `-1` を返します。Python の文 `del o[i]` と同じです。

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

次に属します: Stable ABI. *o* の *i1* から *i2* までの間のスライスに *v* を代入します。Python の文 `o[i1:i2] = v` と同じです。

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

次に属します: Stable ABI. シーケンスオブジェクト *o* の *i1* から *i2* までの間のスライスを削除します。失敗すると `-1` を返します。Python の文 `del o[i1:i2]` と同じです。

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

次に属します: Stable ABI. *o* における *value* の出現回数、すなわち `o[key] == value` となる *key* の個数を返します。失敗すると `-1` を返します。Python の式 `o.count(value)` と同じです。

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

次に属します: Stable ABI. *o* に *value* が入っているか判定します。*o* のある要素が *value* と等価 (equal) ならば `1` を返し、それ以外の場合には `0` を返します。エラーが発生すると `-1` を返します。Python の式 `value in o` と同じです。

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

次に属します: Stable ABI. `o[i] == value` となる最初に見つかったインデクス *i* を返します。エラーが発生すると `-1` を返します。Python の式 `o.index(value)` と同じです。

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

戻り値: 新しい参照。次に属します: Stable ABI. シーケンスもしくはイテラブル *o* と同じ内容を持つリストオブジェクトを返します。失敗したら `NULL` を返します。返されるリストは新しく作られたことが保証されています。これは Python の式 `list(o)` と同等です。

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

戻り値: 新しい参照。次に属します: Stable ABI. シーケンスあるいはイテラブルである *o* と同じ内容を持つタプルオブジェクトを返します。失敗したら `NULL` を返します。*o* がタプルの場合、新たな参照を返します。それ以外の場合、適切な内容が入ったタプルを構築して返します。Python の式 `tuple(o)` と同等です。

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

戻り値: 新しい参照。次に属します: Stable ABI. シーケンスまたはイテラブルの *o* を `PySequence_Fast*` ファミリの関数で利用できるオブジェクトとして返します。オブジェクトがシーケンスでもイテラブルでもない場合は、メッセージ *m* を持つ、`TypeError` を送出します。失敗したら `NULL` を返します。

`PySequence_Fast*` ファミリの関数は、*o* が *PyTupleObject* または *PyListObject* と仮定し、*o* のデータフィールドに直接アクセスするため、そのように名付けられています。

CPython の実装では、もし *o* が 既にシーケンスかタプルであれば、*o* そのものを返します。

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

o が `NULL` でなく、*PySequence_Fast()* が返したオブジェクトであると仮定して、*o* の長さを返します。*o* のサイズは *PySequence_Size()* を呼び出しても得られますが、*PySequence_Fast_GET_SIZE()* の方が *o* をリストかタプルであると仮定して処理するため、より高速です。

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

戻り値: 借用参照。 *o* が `NULL` でなく、*PySequence_Fast()* が返したオブジェクトであり、かつ *i* がインデクスの範囲内にあると仮定して、*o* の *i* 番目の要素を返します。

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

PyObject ポインタの背後にあるアレイを返します。この関数では、*o* は *PySequence_Fast()* の返したオブジェクトであり、`NULL` でないものと仮定しています。

リストのサイズが変更されるとき、メモリ再確保が要素の配列を再配置するかもしれないことに注意してください。そのため、シーケンスの変更が発生しないコンテキストでのみ背後にあるポインタを使ってください。

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

戻り値: 新しい参照。 *o* の *i* 番目の要素を返し、失敗すると `NULL` を返します。*PySequence_GetItem()* の高速版であり、*PySequence_Check()* で *o* が真を返すかどうかの検証や、負の添え字の調整を行いません。

7.5 マップ型プロトコル (mapping protocol)

PyObject_GetItem(), *PyObject_SetItem()*, *PyObject_DelItem()* も参照してください。

int **PyMapping_Check**(*PyObject* *o)

次に属します: 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)

次に属します: Stable ABI. 成功するとオブジェクト *o* 中のキーの数を返し、失敗すると `-1` を返します。これは、Python の式 `len(o)` と同じになります。

PyObject ***PyMapping_GetItemString**(*PyObject* *o, const char *key)

戻り値: 新しい参照。次に属します: 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_SetItemString**(*PyObject* *o, const char *key, *PyObject* *v)

次に属します: 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_HasKey**(*PyObject* *o, *PyObject* *key)

次に属します: Stable ABI. マップ型オブジェクトがキー *key* を持つ場合に `1` を返し、そうでないときには `0` を返します。これは、Python の式 `key in o` と等価です。この関数呼び出しは常に成功します。

> **注釈**
>
> Exceptions which occur when this calls `__getitem__()` method are silently ignored. For proper error handling, use *PyObject_GetItem()* instead.

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

次に属します: 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**.

> **注釈**
>
> Exceptions that occur when this calls `__getitem__()` method or while creating the temporary `str` object are silently ignored. For proper error handling, use *PyMapping_GetItemString()* instead.

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

戻り値: 新しい参照。次に属します: Stable ABI. 成功するとオブジェクト *o* のキーからなるリストを返します。失敗すると `NULL` を返します。

バージョン 3.7 で変更: 以前は、関数はリストもしくはタプルを返していました。

PyObject ***PyMapping_Values**(*PyObject* *o)

戻り値: 新しい参照。次に属します: Stable ABI. 成功するとオブジェクト *o* の値からなるリストを返します。失敗すると `NULL` を返します。

バージョン 3.7 で変更: 以前は、関数はリストもしくはタプルを返していました。

PyObject ***PyMapping_Items**(*PyObject* *o)

戻り値: 新しい参照。次に属します: Stable ABI. 成功するとオブジェクト *o* の要素からなるリストを返し、各要素はキーと値のペアが入ったタプルになっています。失敗すると `NULL` を返します。

バージョン 3.7 で変更: 以前は、関数はリストもしくはタプルを返していました。

7.6 イテレータプロトコル (iterator protocol)

イテレータを扱うための固有の関数は二つあります。

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

次に属します: Stable ABI *(バージョン 3.8 より)*. Return non-zero if the object *o* can be safely passed to *PyIter_Next()*, and `0` otherwise. This function always succeeds.

int **PyAIter_Check**(*PyObject* *o)

次に属します: Stable ABI *(バージョン 3.10 より)*. Return non-zero if the object *o* provides the `AsyncIterator` protocol, and `0` otherwise. This function always succeeds.

Added in version 3.10.

PyObject *PyIter_Next(*PyObject* *o)

戻り値: **新しい参照。次に属します**: Stable ABI. 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.

To write a loop which iterates over an iterator, the C code should look something like this:

```
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;

if (iterator == NULL) {
    /* propagate error */
}

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

Py_DECREF(iterator);

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

type PySendResult

The enum value used to represent different results of *PyIter_Send()*.

Added in version 3.10.

PySendResult PyIter_Send(*PyObject* *iter, *PyObject* *arg, *PyObject* **presult)

次に属します: Stable ABI *(バージョン 3.10 より)*. 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.

Added in version 3.10.

7.7 バッファプロトコル (buffer Protocol)

Python で利用可能ないくつかのオブジェクトは、下層にあるメモリ配列または *buffer* へのアクセスを提供します。このようなオブジェクトとして、組み込みの `bytes` や `bytearray` 、`array.array` のようないくつかの拡張型が挙げられます。サードパーティのライブラリは画像処理や数値解析のような特別な目的のために、それら自身の型を定義することができます。

それぞれの型はそれ自身のセマンティクスを持ちますが、おそらく大きなメモリバッファからなるという共通の特徴を共有します。いくつかの状況では仲介するコピーを行うことなく直接バッファにアクセスすることが望まれます。

Python provides such a facility at the C 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 **バッファオブジェクト構造体** *(buffer object structure)*;
- 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).

`bytes` や `bytearray` などのシンプルなオブジェクトは、内部のバッファーをバイト列の形式で公開します。バイト列以外の形式も利用可能です。例えば、`array.array` が公開する要素はマルチバイト値になることがあります。

buffer インターフェースの利用者の一例は、ファイルオブジェクトの `write()` メソッドです: buffer インターフェースを通して一連のバイト列を提供できるどんなオブジェクトでもファイルに書き込むことができます。`write()` は、その引数として渡されたオブジェクトの内部要素に対する読み出し専用アクセスのみを必要としますが、`readinto()` のような他のメソッドでは、その引数の内容に対する書き込みアクセスが必要です。buffer インターフェースにより、オブジェクトは読み書き両方、読み出し専用バッファへのアクセスを許可するかそれとも拒否するか選択することができます。

buffer インターフェースの利用者には、対象となるオブジェクトのバッファを得る二つの方法があります:

- 正しい引数で *PyObject_GetBuffer()* を呼び出す;
- *PyArg_ParseTuple()* (またはその同族のひとつ) を `y*` 、`w*` または `s*` *format codes* のいずれかとともに呼び出す。

どちらのケースでも、buffer が必要なくなった時に *PyBuffer_Release()* を呼び出さなければなりません。これを怠ると、リソースリークのような様々な問題につながる恐れがあります。

7.7.1 buffer 構造体

バッファ構造体（または単純に "buffers"）は別のオブジェクトのバイナリデータを Python プログラマに提供するのに便利です。これはまた、ゼロコピースライシング機構としても使用できます。このメモリブロックを参照する機能を使うことで、どんなデータでもとても簡単に Python プログラマに提供することができます。メモリは、C 拡張の大きな配列定数かもしれませんし、オペレーティングシステムライブラリに渡す前のメモリブロックかもしれませんし、構造化データをネイティブのインメモリ形式受け渡すのに使用されるかも

しれません。

Python インタプリタによって提供される多くのデータ型とは異なり、バッファは *PyObject* ポインタではなく、シンプルな C 構造体です。そのため、作成とコピーが非常に簡単に行えます。バッファの一般的なラッパーが必要なときは、*memoryview* オブジェクトが作成されます。

エクスポートされるオブジェクトを書く方法の短い説明には、*Buffer Object Structures* を参照してください。バッファを取得するには、*PyObject_GetBuffer()* を参照してください。

type **Py_buffer**

次に属します: Stable ABI *(**すべてのメンバーを含む**)* *(**バージョン** 3.11 **より**)*.

void ***buf**

バッファフィールドが表している論理構造の先頭を指すポインタ。バッファを提供するオブジェクトの下層物理メモリブロック中のどの位置にもなりえます。例えば *strides* が負だと、この値はメモリブロックの末尾かもしれません。

連続 配列の場合この値はメモリブロックの先頭を指します。

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.

PyMemoryView_FromBuffer() または *PyBuffer_FillInfo()* によってラップされた **一時的な** バッファである特別なケースでは、このフィールドは `NULL` です。一般的に、エクスポートオブジェクトはこの方式を使用してはなりません。

Py_ssize_t **len**

`product(shape) * itemsize`。contiguous 配列では、下層のメモリブロックの長さになります。非 contiguous 配列では、contiguous 表現にコピーされた場合に論理構造がもつ長さです。

`((char *)buf)[0]` から `((char *)buf)[len-1]` の範囲へのアクセスは、連続性 (contiguity) を保証するリクエストによって取得されたバッファに対してのみ許されます。多くの場合に、そのようなリクエストは *PyBUF_SIMPLE* または *PyBUF_WRITABLE* です。

int **readonly**

バッファが読み出し専用であるか示します。このフィールドは *PyBUF_WRITABLE* フラグで制御できます。

Py_ssize_t **itemsize**

要素一つ分の byte 単位のサイズ。`struct.calcsize()` を非 `NULL` の *format* 値に対して呼び出した結果と同じです。

重要な例外: 消費者が *PyBUF_FORMAT* フラグを設定することなくバッファを要求した場合、`format` は `NULL` に設定されます。しかし *itemsize* は元のフォーマットに従った値を保持します。

shape が存在する場合、`product(shape) * itemsize == len` の等式が守られ、利用者は *itemsize* を buffer を読むために利用できます。

PyBUF_SIMPLE または *PyBUF_WRITABLE* で要求した結果、*shape* が NULL であれば、消費者は *itemsize* を無視して itemsize == 1 と見なさなければなりません。

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.

このフィールドは *PyBUF_FORMAT* フラグによって制御されます。

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**

メモリ上の N 次元配列の形を示す、長さが *ndim* である *Py_ssize_t* の配列です。shape[0] * ... * shape[ndim-1] * itemsize は *len* と等しくなければなりません。

shape の値は shape[n] >= 0 に制限されます。shape[n] == 0 の場合に特に注意が必要です。詳細は *complex arrays* を参照してください。

shepe (形状) 配列は利用者からは読み出し専用です。

Py_ssize_t ***strides**

各次元において新しい値を得るためにスキップするバイト数を示す、長さ *ndim* の *Py_ssize_t* の配列。

ストライド値は、任意の整数を指定できます。規定の配列では、ストライドは通常でいけば有効です。しかし利用者は、strides[n] <= 0 のケースを処理することができる必要があります。詳細については *complex arrays* を参照してください。

消費者にとって、この strides 配列は読み出し専用です。

Py_ssize_t ***suboffsets**

Py_ssize_t 型の要素を持つ長さ *ndim* の配列。suboffsets[n] >= 0 の場合は、n 番目の次元に沿って保存されている値はポインタで、suboffset 値は各ポインタの参照を解決した後に何バイト加えればいいかを示しています。suboffset の値が負の数の場合は、ポインタの参照解決は不要 (連続したメモリブロック内に直接配置されいる) ということになります。

全ての suboffset が負数の場合 (つまり参照解決が不要) な場合、このフィールドは NULL (デフォルト値) でなければなりません。

この種の配列表現は Python Imaging Library (PIL) で使われています。このような配列で要素にアクセスする方法についてさらに詳しことは *complex arrays* を参照してください。

消費者にとって、suboffsets 配列は読み出し専用です。

void ***internal**

バッファを提供する側のオブジェクトが内部的に利用するための変数です。例えば、提供側はこの

変数に整数型をキャストして、shape, strides, suboffsets といった配列をバッファを開放するときに同時に解放するべきかどうかを管理するフラグに使うことができるでしょう。バッファを受け取る側は、この値を決して変更してはなりません。

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.

7.7.2 バッファリクエストのタイプ

バッファは通常、*PyObject_GetBuffer()* を使うことで、エクスポートするオブジェクトにバッファリクエストを送ることで得られます。メモリの論理的な構造の複雑性は多岐にわたるため、消費者は *flags* 引数を使って、自身が扱えるバッファの種類を指定します。

Py_buffer の全フィールドは、リクエストの種類によって曖昧さを残さずに定義されます。

リクエストに依存しないフィールド

下記のフィールドは *flags* の影響を受けずに、常に正しい値で設定されます。: *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`

format フィールドを制御します。もしフラグが設定されていれば、このフィールドを正しく埋めなければなりません。フラグが設定されていなければ、このフィールドを `NULL` に設定しなければなりません。

PyBUF_WRITABLE は、次の節に出てくるどのフラグとも | を取ってかまいません。*PyBUF_SIMPLE* は 0 と定義されているので、*PyBUF_WRITABLE* は単純な書き込み可能なバッファを要求する単独のフラグとして使えます。

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

このフラグは、以下で複雑性が大きい順に並べたメモリの論理的な構造を制御します。個々のフラグは、それより下に記載されたフラグのすべてのビットを含むことに注意してください。

リクエスト	shape	strides	suboffsets
PyBUF_INDIRECT	yes	yes	必要な場合
PyBUF_STRIDES	yes	yes	NULL
PyBUF_ND	yes	NULL	NULL
PyBUF_SIMPLE	NULL	NULL	NULL

隣接性のリクエスト

ストライドの情報があってもなくても、C または Fortran の **連続性** が明確に要求される可能性があります。ストライド情報なしに、バッファーは C と隣接している必要があります。

リクエスト	shape	strides	suboffsets	contig
PyBUF_C_CONTIGUOUS	yes	yes	NULL	C
PyBUF_F_CONTIGUOUS	yes	yes	NULL	F
PyBUF_ANY_CONTIGUOUS	yes	yes	NULL	C か F
PyBUF_ND	yes	NULL	NULL	C

複合リクエスト

有り得る全てのリクエストの値は、前の節でのフラグの組み合わせで網羅的に定義されています。便利なように、バッファープロトコルでは頻繁に使用される組み合わせを単一のフラグとして提供してます。

次のテーブルの *U* は連続性が未定義であることを表します。利用者は *PyBuffer_IsContiguous()* を呼び出して連続性を判定する必要があるでしょう。

リクエスト	shape	strides	suboffsets	contig	readonly	format
PyBUF_FULL	yes	yes	必要な場合	U	0	yes
PyBUF_FULL_RO	yes	yes	必要な場合	U	1 か 0	yes
PyBUF_RECORDS	yes	yes	NULL	U	0	yes
PyBUF_RECORDS_RO	yes	yes	NULL	U	1 か 0	yes
PyBUF_STRIDED	yes	yes	NULL	U	0	NULL
PyBUF_STRIDED_RO	yes	yes	NULL	U	1 か 0	NULL
PyBUF_CONTIG	yes	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	yes	NULL	NULL	C	1 か 0	NULL

7.7.3 複雑な配列

NumPy スタイル: shape, strides

NumPy スタイルの配列の論理的構造は *itemsize*, *ndim*, *shape*, *strides* で定義されます。

`ndim == 0` の場合は、*buf* が指すメモリの場所は、サイズが *itemsize* のスカラ値として解釈されます。この場合、*shape* と *strides* の両方とも `NULL` です。

strides が `NULL` の場合は、配列は標準の n 次元 C 配列として解釈されます。そうでない場合は、利用者は次のように n 次元配列にアクセスしなければなりません:

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1] * strides[n-1];
item = *((typeof(item) *)ptr);
```

上記のように、*buf* はメモリブロック内のどの場所でも指すことが可能です。エクスポーターはこの関数を使用することによってバッファの妥当性を確認出来ます。

```
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

    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 スタイル: shape, strides, suboffsets

PIL スタイルの配列では通常の要素の他に、ある次元の上で次の要素を取得するために辿るポインタを持てます。例えば、通常の 3 次元 C 配列 `char v[2][2][3]` は、2 次元配列への 2 つのポインタからなる配列 `char (*v[2])[2][3]` と見ることもできます。suboffset 表現では、これらの 2 つのポインタは *buf* の先頭に埋め込め、メモリのどこにでも配置できる 2 つの `char x[2][3]` 配列を指します。

次の例は、strides も suboffsets も `NULL` でない場合の、N 次元インデックスによって指されている N 次元配列内の要素へのポインタを返す関数です:

```
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];
```

(次のページに続く)

(前のページからの続き)

```
        }
    }
    return (void*)pointer;
}
```

7.7.4 バッファ関連の関数

int **PyObject_CheckBuffer**(*PyObject* *obj)

次に属します: Stable ABI (**バージョン** *3.11* **より**). *obj* が buffer インターフェースをサポートしている場合は `1` を返し、そうでない場合は `0` を返します。`1` を返したとしても、*PyObject_GetBuffer()* が成功することは保証されません。この関数は常に成功します。

int **PyObject_GetBuffer**(*PyObject* *exporter, *Py_buffer* *view, int flags)

次に属します: Stable ABI (**バージョン** *3.11* **より**). *exporter* に *flags* で指定された方法で *view* を埋めるように要求します。もし exporter が指定されたとおりにバッファを提供できない場合、`BufferError` を送出し、`view->obj` を `NULL` に設定した上で、`-1` を返さなければなりません。

成功したときは、*view* を埋め、`view->obj` に *exporter* への新しい参照を設定し、0 を返します。チェイン状のバッファプロバイダがリクエストを単一のオブジェクトにリダイレクトするケースでは、`view->obj` は *exporter* の代わりにこのオブジェクトを参照します (**バッファオブジェクト構造体** を参照してください)。

`malloc()` と `free()` のように、呼び出しに成功した *PyObject_GetBuffer()* と対になる *PyBuffer_Release()* の呼び出しがなければなりません。従って、バッファの利用が済んだら *PyBuffer_Release()* が厳密に 1 回だけ呼び出されなければなりません。

void **PyBuffer_Release**(*Py_buffer* *view)

次に属します: Stable ABI (**バージョン** *3.11* **より**). 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.

PyObject_GetBuffer() を通して取得していないバッファに対してこの関数を呼び出すのは間違いです。

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

次に属します: Stable ABI (**バージョン** *3.11* **より**). Return the implied *itemsize* from *format*. On error, raise an exception and return -1.

Added in version 3.9.

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

次に属します: Stable ABI (**バージョン** *3.11* **より**). *view* で定義されているメモリが、C スタイル (*order* == `'C'`) のときか、Fortran スタイル (*order* == `'F'`) **連続** のときか、そのいずれか (*order* == `'A'`) であれば `1` を返します。それ以外の場合は `0` を返します。この関数は常に成功します。

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

次に属します: Stable ABI *(**バージョン** 3.11 **より**)*. 与えられた *view* 内にある *indices* が指すメモリ領域を取得します。*indices* は `view->ndim` 個のインデックスからなる配列を指していなければなりません。

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

次に属します: Stable ABI *(**バージョン** 3.11 **より**)*. 連続する *len* バイトを *buf* から *view* にコピーします。*fort* には `'C'` か `'F'` を指定できます（それぞれ C 言語スタイルと Fortran スタイルの順序を表します)。成功時には `0`、エラー時には `-1` を返します。

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

次に属します: Stable ABI *(**バージョン** 3.11 **より**)*. *src* から *len* バイトを連続表現で *buf* 上にコピーします。*order* は `'C'` または `'F'` または `'A'` (C スタイル順序または Fortran スタイル順序またはそれ以外) が指定できます。成功したら `0` が返り、エラーなら `-1` が返ります。

len != *src->len* の場合、この関数は失敗します。

int PyObject_CopyData(*PyObject* *dest, *PyObject* *src)

次に属します: Stable ABI *(**バージョン** 3.11 **より**)*. Copy data from *src* to *dest* buffer. Can convert between C-style and or Fortran-style buffers.

成功したら `0` が、エラー時には `-1` が返されます。

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

次に属します: Stable ABI *(**バージョン** 3.11 **より**)*. *strides* 配列を、*itemsize* の大きさの要素がバイト単位の、*shape* の形をした **連続な** (*order* が `'C'` なら C-style 、`'F'` なら Fortran-style の) 多次元配列として埋める。

int PyBuffer_FillInfo(*Py_buffer* *view, *PyObject* *exporter, void *buf, *Py_ssize_t* len, int readonly, int flags)

次に属します: Stable ABI *(**バージョン** 3.11 **より**)*. サイズが *len* の *buf* を *readonly* に従った書き込み可/不可の設定で公開するバッファリクエストを処理します。*buf* は符号無しバイトの列として解釈されます。

flags 引数はリクエストのタイプを示します。この関数は、*buf* が読み出し専用と指定されていて、*flags* に *PyBUF_WRITABLE* が設定されていない限り、常にフラグに指定された通りに *view* を埋めます。

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`;

この関数を *getbufferproc* の一部として使う場合には、*exporter* はエクスポートするオブジェクトに設定しなければならず、さらに *flags* は変更せずに渡さなければなりません。そうでない場合は、*exporter* は `NULL` でなければなりません。

7.8 古いバッファプロトコル

バージョン 3.0 で非推奨.

これらの関数は、Python 2 の「古いバッファプロトコル」API の一部です。Python 3 では、もうこのプロトコルは存在しませんが、2.x のコードを移植しやすいように関数は公開されています。**新しいバッファプロトコル** と互換性のあるラッパー関数のように振る舞いますが、バッファがエクスポートされるときに取得されるリソースの生存期間を管理することはできません。

従って、あるオブジェクトのバッファビューを取得するために、*PyObject_GetBuffer()* (もしくは `y*` および `w*` **フォーマットコード** で *PyArg_ParseTuple()* やその仲間) を呼び出し、バッファビューを解放するときには *PyBuffer_Release()* を呼び出します。

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

次に属します: Stable ABI. 文字ベースの入力として使える読み出し専用メモリ上の位置へのポインタを返します。*obj* 引数は単一セグメントからなる文字バッファインターフェースをサポートしていなければなりません。成功すると `0` を返し、*buffer* をメモリの位置に、*buffer_len* をバッファの長さに設定します。エラーの際には `-1` を返し、`TypeError` をセットします。

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

次に属します: Stable ABI. 任意のデータを収めた読み出し専用のメモリ上の位置へのポインタを返します。*obj* 引数は単一セグメントからなる読み出し可能バッファインターフェースをサポートしていなければなりません。成功すると `0` を返し、*buffer* をメモリの位置に、*buffer_len* をバッファの長さに設定します。エラーの際には `-1` を返し、`TypeError` をセットします。

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

次に属します: Stable ABI. *o* が単一セグメントからなる読み出し可能バッファインターフェースをサポートしている場合に `1` を返します。それ以外の場合には `0` を返します。この関数は常に成功します。

この関数は試しにバッファの取得と解放を行い、それぞれ対応する関数の呼び出し中に起こる例外は抑制されることに注意してください。エラーを報告させるには、*PyObject_GetBuffer()* を代わりに使ってください。

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

次に属します: Stable ABI. 書き込み可能なメモリ上の位置へのポインタを返します。*obj* 引数は単一セグメントからなる文字バッファインターフェースをサポートしていなければなりません。成功すると `0` を返し、*buffer* をメモリの位置に、*buffer_len* をバッファの長さに設定します。エラーの際には `-1` を返し、`TypeError` をセットします。

第

EIGHT

具象オブジェクト (CONCRETE OBJECT) レイヤ

この章では、特定の Python オブジェクト型固有の関数について述べています。これらの関数に間違った型のオブジェクトを渡すのは良い考えではありません; Python プログラムから何らかのオブジェクトを受け取ったとき、そのオブジェクトが正しい型になっているか確信をもてないのなら、まず型チェックを行わなければなりません; 例えば、あるオブジェクトが辞書型か調べるには、*PyDict_Check()* を使います。この章は Python のオブジェクト型における ” 家計図” に従って構成されています。

警告

この章で述べている関数は、渡されたオブジェクトの型を注意深くチェックしはするものの、多くの関数は渡されたオブジェクトが有効な `NULL` なのか有効なオブジェクトなのかをチェックしません。これらの関数に `NULL` を渡させてしまうと、関数はメモリアクセス違反を起こして、インタプリタを即座に終了させてしまうはずです。

8.1 基本オブジェクト (fundamental object)

この節では、Python の型オブジェクトとシングルトン (singleton) オブジェクト `None` について述べます。

8.1.1 型オブジェクト

type **PyTypeObject**

次に属します: Limited API (**不透明な構造体として**). 組み込み型を記述する際に用いられる、オブジェクトを表す C 構造体です。

PyTypeObject **PyType_Type**

次に属します: Stable ABI. 型オブジェクト自身の型オブジェクトです。Python レイヤにおける `type` と同じオブジェクトです。

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**()

次に属します: Stable ABI. 内部の検索キャッシュをクリアします。現在のバージョンタグを返します。

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

次に属します: 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*.

Added in version 3.2.

バージョン 3.4 で変更: 返り値の型が `long` ではなく `unsigned long` になりました。

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.

Added in version 3.12.

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

次に属します: Stable ABI. 内部の検索キャッシュを、その type とすべてのサブタイプに対して無効にします。この関数は type の属性や基底クラス列を変更したあとに手動で呼び出さなければなりません。

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.

Added in 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()*.

Added in 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()*.

Added in 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.

Added in 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)

次に属します: Stable ABI. *a* が *b* のサブタイプの場合に真を返します。

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)

戻り値: **新しい参照。次に属します**: Stable ABI. Generic handler for the *tp_alloc* slot of a type object. Use Python's default memory allocation mechanism to allocate a new instance and initialize all its contents to `NULL`.

PyObject ***PyType_GenericNew**(*PyTypeObject* *type, *PyObject* *args, *PyObject* *kwds)

戻り値: **新しい参照。次に属します**: Stable ABI. Generic handler for the *tp_new* slot of a type object. Create a new instance using the type's *tp_alloc* slot.

int **PyType_Ready**(*PyTypeObject* *type)

次に属します: Stable ABI. 型オブジェクトのファイナライズを行います。この関数は全てのオブジェクトで初期化を完了するために呼び出されなくてはなりません。この関数は、基底クラス型から継承したスロットを型オブジェクトに追加する役割があります。成功した場合には `0` を返し、エラーの場合には `-1` を返して例外情報を設定します。

> **注釈**
>
> 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)

戻り値: **新しい参照。次に属します**: Stable ABI *(**バージョン** 3.11 **より**)*. Return the type's name. Equivalent to getting the type's `__name__` attribute.

Added in version 3.11.

PyObject ***PyType_GetQualName**(*PyTypeObject* *type)

戻り値: **新しい参照。次に属します**: Stable ABI *(**バージョン** 3.11 **より**)*. Return the type's qualified name. Equivalent to getting the type's `__qualname__` attribute.

Added in version 3.11.

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

次に属します: Stable ABI *(**バージョン** 3.4 **より**)*. 与えられたスロットに格納されている関数ポインタを返します。返り値が `NULL` の場合は、スロットが `NULL` か、関数が不正な引数で呼ばれたことを示します。通常、呼び出し側は返り値のポインタを適切な関数型にキャストします。

See *PyType_Slot.slot* for possible values of the *slot* argument.

Added in version 3.4.

バージョン 3.10 で変更: *PyType_GetSlot()* can now accept all types. Previously, it was limited to *heap types*.

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

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. 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.

Added in version 3.9.

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

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. Return the state of the module object asso-

ciated 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.

Added in version 3.9.

PyObject ***PyType_GetModuleByDef**(*PyTypeObject* *type, struct *PyModuleDef* *def)

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.

Added in version 3.11.

int **PyUnstable_Type_AssignVersionTag**(*PyTypeObject* *type)

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

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.

Added in 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)

次に属します: Stable ABI *(バージョン 3.12 より)*. Create and return a *heap type* from the *spec* (see *Py_TPFLAGS_HEAPTYPE*).

The metaclass *metaclass* is used to construct the resulting type object. When *metaclass* is `NULL`, the metaclass is derived from *bases* (or *Py_tp_base[s]* slots if *bases* is `NULL`, see below).

Metaclasses that override *tp_new* are not supported, except if `tp_new` is `NULL`. (For backwards compatibility, other `PyType_From*` functions allow such metaclasses. They ignore `tp_new`, which may result in incomplete initialization. This is deprecated and in Python 3.14+ such metaclasses will not be supported.)

The *bases* argument can be used to specify base classes; it can either be only one class or a tuple of classes. If *bases* is `NULL`, the *Py_tp_bases* slot is used instead. If that also is `NULL`, the *Py_tp_base* slot is used instead. If that also is `NULL`, the new type derives from `object`.

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

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

Note that this function does *not* fully match the behavior of calling `type()` or using the `class` statement. With user-provided base types or metaclasses, prefer *calling* `type` (or the metaclass) over `PyType_From*` functions. Specifically:

- `__new__()` is not called on the new class (and it must be set to `type.__new__`).
- `__init__()` is not called on the new class.
- `__init_subclass__()` is not called on any bases.
- `__set_name__()` is not called on new descriptors.

Added in version 3.12.

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

戻り値: **新しい参照。次に属します**: Stable ABI *(**バージョン** 3.10 **より**)*. Equivalent to `PyType_FromMetaclass(NULL, module, spec, bases)`.

Added in version 3.9.

バージョン 3.10 で変更: The function now accepts a single class as the *bases* argument and `NULL` as the `tp_doc` slot.

バージョン 3.12 で変更: The function now finds and uses a metaclass corresponding to the provided base classes. Previously, only `type` instances were returned.

The *tp_new* of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides *tp_new* is deprecated and in Python 3.14+ it will be no longer allowed.

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

戻り値: **新しい参照。次に属します**: Stable ABI *(**バージョン** 3.3 **より**)*. Equivalent to `PyType_FromMetaclass(NULL, NULL, spec, bases)`.

Added in version 3.3.

バージョン 3.12 で変更: The function now finds and uses a metaclass corresponding to the provided base classes. Previously, only `type` instances were returned.

The *tp_new* of the metaclass is *ignored*. which may result in incomplete initialization. Creating classes whose metaclass overrides *tp_new* is deprecated and in Python 3.14+ it will be no longer

allowed.

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

戻り値: **新しい参照。次に属します**: Stable ABI. Equivalent to `PyType_FromMetaclass(NULL, NULL, spec, NULL)`.

バージョン 3.12 で変更: The function now finds and uses a metaclass corresponding to the base classes provided in *Py_tp_base[s]* slots. Previously, only `type` instances were returned.

The *tp_new* of the metaclass is *ignored.* which may result in incomplete initialization. Creating classes whose metaclass overrides *tp_new* is deprecated and in Python 3.14+ it will be no longer allowed.

type **PyType_Spec**

次に属します: Stable ABI *(**すべてのメンバーを含む**)*. 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.

バージョン 3.12 で変更: 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**

次に属します: Stable ABI *(すべてのメンバーを含む)*. Structure defining optional functionality of a type, containing a slot ID and a value pointer.

int **slot**

A slot ID.

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

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

The following "offset" fields cannot be set using *PyType_Slot*:

- *tp_weaklistoffset* (use *Py_TPFLAGS_MANAGED_WEAKREF* instead if possible)
- *tp_dictoffset* (use *Py_TPFLAGS_MANAGED_DICT* instead if possible)
- *tp_vectorcall_offset* (use `"__vectorcalloffset__"` in *PyMemberDef*)

If it is not possible to switch to a `MANAGED` flag (for example, for vectorcall or to support Python older than 3.12), specify the offset in *Py_tp_members*. See *PyMemberDef documentation* for details.

The following fields cannot be set at all when creating a heap type:

- *tp_vectorcall* (use *tp_new* and/or *tp_init*)
- Internal fields: *tp_dict*, *tp_mro*, *tp_cache*, *tp_subclasses*, and *tp_weaklist*.

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

バージョン 3.9 で変更: Slots in *PyBufferProcs* may be set in the unlimited API.

バージョン 3.11 で変更: *bf_getbuffer* and *bf_releasebuffer* are now available under the *limited API*.

void *pfunc

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

Slots other than Py_tp_doc may not be NULL.

8.1.2 None オブジェクト

None に対する *PyTypeObject* は、Python/C API では直接公開されていないので注意してください。None は単量子 (singleton) なので、オブジェクトの同一性テスト (C では ==) を使うだけで十分だからです。同じ理由から、PyNone_Check() 関数はありません。

PyObject *Py_None

The Python None object, denoting lack of value. This object has no methods and is immortal.

バージョン 3.12 で変更: *Py_None* is immortal.

Py_RETURN_NONE

関数から *Py_None* を返します。

8.2 数値型オブジェクト (numeric object)

8.2.1 整数型オブジェクト (integer object)

すべての整数は任意の長さをもつ "long" 整数として実装されます。

エラーが起きると、ほとんどの PyLong_As* API は (return type)-1 を返しますが、これは数値と見分けが付きません。見分けを付けるためには *PyErr_Occurred()* を使ってください。

type PyLongObject

次に属します: Limited API (**不透明な構造体として**). この *PyObject* のサブタイプは整数型を表現します。

PyTypeObject PyLong_Type

次に属します: Stable ABI. この *PyTypeObject* のインスタンスは Python 整数型を表現します。これは Python レイヤにおける int と同じオブジェクトです。

int PyLong_Check(*PyObject* *p)

引数が *PyLongObject* か *PyLongObject* のサブタイプであるときに真を返します。この関数は常に成功します。

int PyLong_CheckExact(*PyObject* *p)

引数が *PyLongObject* であるが *PyLongObject* のサブタイプでないときに真を返します。この関数は常に成功します。

PyObject *PyLong_FromLong(long v)

戻り値: **新しい参照。次に属します**: Stable ABI. *v* から新たな *PyLongObject* オブジェクトを生成して返します。失敗のときには NULL を返します。

現在の実装では、-5 から 256 までの全ての整数に対する整数オブジェクトの配列を保持します。この範囲の数を生成すると、実際には既存のオブジェクトに対する参照が返るようになっています。

PyObject *PyLong_FromUnsignedLong(unsigned long v)

戻り値: 新しい参照。次に属します: Stable ABI. C の unsigned long から新たな *PyLongObject* オブジェクトを生成して返します。失敗した際には NULL を返します。

PyObject *PyLong_FromSsize_t(*Py_ssize_t* v)

戻り値: 新しい参照。次に属します: Stable ABI. C の *Py_ssize_t* 型から新たな *PyLongObject* オブジェクトを生成して返します。失敗のときには NULL を返します。

PyObject *PyLong_FromSize_t(size_t v)

戻り値: 新しい参照。次に属します: Stable ABI. C の size_t 型から新たな *PyLongObject* オブジェクトを生成して返します。失敗のときには NULL を返します。

PyObject *PyLong_FromLongLong(long long v)

戻り値: 新しい参照。次に属します: Stable ABI. C の long long 型から新たな *PyLongObject* オブジェクトを生成して返します。失敗のときには NULL を返します。

PyObject *PyLong_FromUnsignedLongLong(unsigned long long v)

戻り値: 新しい参照。次に属します: Stable ABI. C の unsigned long long 型から新たな *PyLongObject* オブジェクトを生成して返します。失敗のときには NULL を返します。

PyObject *PyLong_FromDouble(double v)

戻り値: 新しい参照。次に属します: Stable ABI. *v* の整数部から新たな *PyLongObject* オブジェクトを生成して返します。失敗のときには NULL を返します。

PyObject *PyLong_FromString(const char *str, char **pend, int base)

戻り値: 新しい参照。次に属します: 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.

> **参考**
>
> Python methods int.to_bytes() and int.from_bytes() to convert a *PyLongObject* to/from an array of bytes in base 256. You can call those from C using *PyObject_CallMethod()*.

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

戻り値: 新しい参照。 Convert a sequence of Unicode digits in the string *u* to a Python integer value.

Added in version 3.3.

PyObject *PyLong_FromVoidPtr(void *p)

戻り値: **新しい参照。次に属します**: Stable ABI. ポインタ *p* から Python 整数値を生成します。ポインタの値は *PyLong_AsVoidPtr()* を適用した結果から取得されます。

long PyLong_AsLong(*PyObject* *obj)

次に属します: 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*.

もし *obj* の値が `long` の範囲外であれば、`OverflowError` を送出します。

エラーが起きたときに `-1` を返します。見分けを付けるためには *PyErr_Occurred()* を使ってください。

バージョン 3.8 で変更: 可能であれば `__index__()` を使うようになりました。

バージョン 3.10 で変更: 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.

バージョン 3.14 で非推奨: The function is soft deprecated.

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

次に属します: 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.

エラーが起きたときに `-1` を返します。見分けを付けるためには *PyErr_Occurred()* を使ってください。

バージョン 3.8 で変更: 可能であれば `__index__()` を使うようになりました。

バージョン 3.10 で変更: This function will no longer use `__int__()`.

long long PyLong_AsLongLong(*PyObject* *obj)

次に属します: 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*.

もし *obj* の値が `long long` の範囲外であれば、`OverflowError` を送出します。

エラーが起きたときに `-1` を返します。見分けを付けるためには *PyErr_Occurred()* を使ってください。

バージョン 3.8 で変更: 可能であれば `__index__()` を使うようになりました。

バージョン 3.10 で変更: This function will no longer use `__int__()`.

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

次に属します: 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.

エラーが起きたときに `-1` を返します。見分けを付けるためには *PyErr_Occurred()* を使ってください。

Added in version 3.2.

バージョン 3.8 で変更: 可能であれば `__index__()` を使うようになりました。

バージョン 3.10 で変更: This function will no longer use `__int__()`.

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

次に属します: Stable ABI. *pylong* を表す C の *Py_ssize_t* を返します。*pylong* は *PyLongObject* のインスタンスでなければなりません。

もし *pylong* の値が *Py_ssize_t* の範囲外であれば、`OverflowError` を送出します。

エラーが起きたときに `-1` を返します。見分けを付けるためには *PyErr_Occurred()* を使ってください。

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

次に属します: Stable ABI. *pylong* を表す C の `unsigned long` を返します。*pylong* は *PyLongObject* のインスタンスでなければなりません。

もし *pylong* の値が `unsigned long` の範囲外であれば、`OverflowError` を送出します。

エラーが起きたときに `(unsigned long)-1` を返します。見分けを付けるためには *PyErr_Occurred()* を使ってください。

size_t **PyLong_AsSize_t**(*PyObject* *pylong)

次に属します: Stable ABI. *pylong* を表す C の `size_t` を返します。*pylong* は *PyLongObject* のインスタンスでなければなりません。

もし *pylong* の値が `size_t` の範囲外であれば、`OverflowError` を送出します。

エラーが起きたときに `(size_t)-1` を返します。見分けを付けるためには *PyErr_Occurred()* を使ってください。

unsigned long long **PyLong_AsUnsignedLongLong**(*PyObject* *pylong)

次に属します: Stable ABI. *pylong* を表す C の `unsigned long long` を返します。*pylong* は *PyLongObject* のインスタンスでなければなりません。

もし *pylong* の値が `unsigned long long` の範囲外であれば、`OverflowError` を送出します。

エラーが起きたときに (unsigned long long)-1 を返します。見分けを付けるためには *PyErr_Occurred()* を使ってください。

バージョン 3.1 で変更: 負 *pylong* を指定した際に TypeError ではなく、OverflowError を送出するようになりました。

unsigned long **PyLong_AsUnsignedLongMask**(*PyObject* *obj)

次に属します: 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*.

obj の値が unsigned long の範囲から外れていた場合は、ULONG_MAX + 1 を法とした剰余を返します。

エラーが起きたときに (unsigned long)-1 を返します。見分けを付けるためには *PyErr_Occurred()* を使ってください。

バージョン 3.8 で変更: 可能であれば __index__() を使うようになりました。

バージョン 3.10 で変更: This function will no longer use __int__().

unsigned long long **PyLong_AsUnsignedLongLongMask**(*PyObject* *obj)

次に属します: 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*.

obj の値が unsigned long long の範囲から外れていた場合は、ULLONG_MAX + 1 を法とした剰余を返します。

エラーが起きたときに (unsigned long long)-1 を返します。見分けを付けるためには *PyErr_Occurred()* を使ってください。

バージョン 3.8 で変更: 可能であれば __index__() を使うようになりました。

バージョン 3.10 で変更: This function will no longer use __int__().

double **PyLong_AsDouble**(*PyObject* *pylong)

次に属します: Stable ABI. *pylong* を表す C の double を返します。*pylong* は *PyLongObject* のインスタンスでなければなりません。

もし *pylong* の値が double の範囲外であれば、OverflowError を送出します。

エラーが起きたときに -1.0 を返します。見分けを付けるためには *PyErr_Occurred()* を使ってください。

void ***PyLong_AsVoidPtr**(*PyObject* *pylong)

次に属します: Stable ABI. Python の整数型を指す *pylong* を、C の void ポインタに変換します。*pylong* を変換できなければ、OverflowError を送出します。この関数は *PyLong_FromVoidPtr()* で値を生成するときに使うような void ポインタ型を生成できるだけです。

エラーが起きたときに NULL を返します。見分けを付けるためには *PyErr_Occurred()* を使ってください。

PyObject ***PyLong_GetInfo**(void)

次に属します: 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.

Added in version 3.1.

int **PyUnstable_Long_IsCompact**(const *PyLongObject* *op)

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

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 *calling* `int.to_bytes()`.

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.

Added in version 3.12.

Py_ssize_t **PyUnstable_Long_CompactValue**(const *PyLongObject* *op)

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

If *op* is compact, as determined by *PyUnstable_Long_IsCompact()*, return its value.

Otherwise, the return value is undefined.

Added in version 3.12.

8.2.2 Boolean オブジェクト

Python の Bool 型は整数のサブクラスとして実装されています。ブール型の値は、*Py_False* と *Py_True* の 2 つしかありません。従って、通常の生成／削除関数はブール型にはあてはまりません。とはいえ、以下のマクロが利用できます。

PyTypeObject **PyBool_Type**

次に属します: Stable ABI. この *PyTypeObject* のインスタンスは Python の boolean 型を表現します; Python レイヤにおける `bool` と同じオブジェクトです。

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

o が *PyBool_Type* 型の場合に真を返します。この関数は常に成功します。

PyObject ***Py_False**

The Python `False` object. This object has no methods and is immortal.

バージョン 3.12 で変更: *Py_False* is immortal.

PyObject ***Py_True**

The Python `True` object. This object has no methods and is immortal.

バージョン 3.12 で変更: *Py_True* is immortal.

Py_RETURN_FALSE

関数から *Py_False* を返します。

Py_RETURN_TRUE

関数から *Py_True* を返します。

PyObject ***PyBool_FromLong**(long v)

戻り値: **新しい参照。次に属します**: Stable ABI. *v* の真理値に応じて *Py_True* または *Py_False* を返します。

8.2.3 Floating-Point Objects

type **PyFloatObject**

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

PyTypeObject **PyFloat_Type**

次に属します: 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)

引数が *PyFloatObject* か *PyFloatObject* のサブタイプであるときに真を返します。この関数は常に成功します。

int **PyFloat_CheckExact**(*PyObject* *p)

引数が *PyFloatObject* であるが *PyFloatObject* のサブタイプでないときに真を返します。この関数は常に成功します。

PyObject ***PyFloat_FromString**(*PyObject* *str)

戻り値: **新しい参照。次に属します**: Stable ABI. *str* の文字列値をもとに *PyFloatObject* オブジェクトを生成します。失敗すると `NULL` を返します。

PyObject *PyFloat_FromDouble(double v)

戻り値: 新しい参照。次に属します: Stable ABI. *v* から *PyFloatObject* オブジェクトを生成して返します。失敗すると NULL を返します。

double PyFloat_AsDouble(*PyObject* *pyfloat)

次に属します: 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.

バージョン 3.8 で変更: 可能であれば __index__() を使うようになりました。

double PyFloat_AS_DOUBLE(*PyObject* *pyfloat)

pyfloat の指す値を、C の double 型表現で返しますが、エラーチェックを行いません。

PyObject *PyFloat_GetInfo(void)

戻り値: 新しい参照。次に属します: Stable ABI. float の精度、最小値、最大値に関する情報を含む structseq インスタンスを返します。これは、float.h ファイルの薄いラッパーです。

double PyFloat_GetMax()

次に属します: Stable ABI. float の表現できる最大限解値 *DBL_MAX* を C の double 型で返します。

double PyFloat_GetMin()

次に属します: Stable ABI. float の正規化された最小の正の値 *DBL_MIN* を C の double 型で返します。

Pack and Unpack functions

The pack and unpack functions provide an efficient platform-independent way to store floating-point values as byte strings. The Pack routines produce a bytes string from a C double, and the Unpack routines produce a C double from such a bytes string. The suffix (2, 4 or 8) specifies the number of bytes in the bytes string.

On platforms that appear to use IEEE 754 formats these functions work by copying bits. On other platforms, the 2-byte format is identical to the IEEE 754 binary16 half-precision format, the 4-byte format (32-bit) is identical to the IEEE 754 binary32 single precision format, and the 8-byte format to the IEEE 754 binary64 double precision format, although the packing of INFs and NaNs (if such things exist on the platform) isn't handled correctly, and attempting to unpack a bytes string containing an IEEE INF or NaN will raise an exception.

On non-IEEE platforms with more precision, or larger dynamic range, than IEEE 754 supports, not all values can be packed; on non-IEEE platforms with less precision, or smaller dynamic range, not all values can be unpacked. What happens in such cases is partly accidental (alas).

Added in 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, unsigned char *p, int le)

Pack a C double as the IEEE 754 binary16 half-precision format.

int **PyFloat_Pack4**(double x, unsigned char *p, int le)

Pack a C double as the IEEE 754 binary32 single precision format.

int **PyFloat_Pack8**(double x, unsigned char *p, int le)

Pack a C double as the IEEE 754 binary64 double precision format.

Unpack functions

The unpack routines read 2, 4 or 8 bytes, starting at *p*. *le* is an `int` argument, non-zero if the bytes string is in little-endian format (exponent last, at `p+1`, `p+3` or `p+6` and `p+7`), zero if big-endian (exponent first, at *p*). The `PY_BIG_ENDIAN` constant can be used to use the native endian: it is equal to `1` on big endian processor, or `0` on little endian processor.

Return value: The unpacked double. On error, this is `-1.0` and *PyErr_Occurred()* is true (and an exception is set, most likely `OverflowError`).

Note that on a non-IEEE platform this will refuse to unpack a bytes string that represents a NaN or infinity.

double **PyFloat_Unpack2**(const unsigned char *p, int le)

Unpack the IEEE 754 binary16 half-precision format as a C double.

double **PyFloat_Unpack4**(const unsigned char *p, int le)

Unpack the IEEE 754 binary32 single precision format as a C double.

double **PyFloat_Unpack8**(const unsigned char *p, int le)

Unpack the IEEE 754 binary64 double precision format as a C double.

8.2.4 複素数オブジェクト

Python の複素数オブジェクトは、C API 側から見ると二つの別個の型として実装されています: 一方は Python プログラムに対して公開されている Python のオブジェクトで、他方は実際の複素数値を表現する C の構造体です。API では、これら双方を扱う関数を提供しています。

C 構造体としての複素数

複素数の C 構造体を引数として受理したり、戻り値として返したりする関数は、ポインタ渡しを行うのではなく **値渡し** を行うので注意してください。これは API 全体を通して一貫しています。

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)

二つの複素数の和を C の *Py_complex* 型で返します。

Py_complex **_Py_c_diff**(*Py_complex* left, *Py_complex* right)

二つの複素数の差を C の *Py_complex* 型で返します。

Py_complex **_Py_c_neg**(*Py_complex* num)

複素数 *num* の符号反転 C の *Py_complex* 型で返します。

Py_complex **_Py_c_prod**(*Py_complex* left, *Py_complex* right)

二つの複素数の積を C の *Py_complex* 型で返します。

Py_complex **_Py_c_quot**(*Py_complex* dividend, *Py_complex* divisor)

二つの複素数の商を C の *Py_complex* 型で返します。

divisor が null の場合は、このメソッドはゼロを返し、`errno` に `EDOM` をセットします。

Py_complex **_Py_c_pow**(*Py_complex* num, *Py_complex* exp)

指数 *exp* の *num* 乗を C の *Py_complex* 型で返します。

num が null で *exp* が正の実数でない場合は、このメソッドはゼロを返し、`errno` に `EDOM` をセットします。

Python オブジェクトとしての複素数型

type **PyComplexObject**

この *PyObject* のサブタイプは Python の複素数型を表現します。

PyTypeObject **PyComplex_Type**

次に属します: Stable ABI. この *PyTypeObject* のインスタンスは Python の複素数型を表現します。Python レイヤの `complex` と同じオブジェクトです。

int **PyComplex_Check**(*PyObject* *p)

引数が *PyComplexObject* か *PyComplexObject* のサブタイプであるときに真を返します。この関数は常に成功します。

int **PyComplex_CheckExact**(*PyObject* *p)

引数が *PyComplexObject* であるが *PyComplexObject* のサブタイプでないときに真を返します。この関数は常に成功します。

PyObject ***PyComplex_FromCComplex**(*Py_complex* v)

戻り値: **新しい参照。** 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)

戻り値: **新しい参照。次に属します**: Stable ABI. Return a new *PyComplexObject* object from *real* and *imag*. Return `NULL` with an exception set on error.

double **PyComplex_RealAsDouble**(*PyObject* *op)

次に属します: Stable ABI. *op* の実数部分を C の `double` 型で返します。

Upon failure, this method returns `-1.0` with an exception set, so one should call *PyErr_Occurred()* to check for errors.

double **PyComplex_ImagAsDouble**(*PyObject* *op)

次に属します: Stable ABI. *op* の虚数部分を C の `double` 型で返します。

Py_complex **PyComplex_AsCComplex**(*PyObject* *op)

複素数値 *op* から *Py_complex* 型を生成します。

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.

バージョン 3.8 で変更: 可能であれば `__index__()` を使うようになりました。

8.3 シーケンスオブジェクト (sequence object)

シーケンスオブジェクトに対する一般的な操作については前の章ですでに述べました; この節では、Python 言語にもともと備わっている特定のシーケンスオブジェクトについて扱います。

8.3.1 バイトオブジェクト

下記の関数は、バイトオブジェクトを期待している引数にバイトオブジェクトでないパラメタを指定して呼び出されると、`TypeError` を送出します。

type **PyBytesObject**

この *PyObject* のサブタイプは、Python バイトオブジェクトを表します。

PyTypeObject **PyBytes_Type**

次に属します: Stable ABI. この *PyTypeObject* のインスタンスは、Python バイト型を表します; Python レイヤの `bytes` と同じオブジェクトです。

int **PyBytes_Check**(*PyObject* *o)

オブジェクト *o* が bytes オブジェクトか bytes 型のサブタイプのインスタンスである場合に真を返します。この関数は常に成功します。

int **PyBytes_CheckExact**(*PyObject* *o)

オブジェクト *o* が bytes オブジェクトだが bytes 型のサブタイプのインスタンスでない場合に真を返します。この関数は常に成功します。

PyObject ***PyBytes_FromString**(const char *v)

戻り値: 新しい参照。次に属します: Stable ABI. 成功時に、文字列 *v* のコピーを値とする新しいバイトオブジェクトを返し、失敗時に `NULL` を返します。引数 *v* は `NULL` であってはなりません; そのチェックは行われません。

PyObject ***PyBytes_FromStringAndSize**(const char *v, *Py_ssize_t* len)

戻り値: 新しい参照。次に属します: Stable ABI. 成功時に、文字列 *v* のコピーを値とする長さ *len* の新しいバイトオブジェクトを返し、失敗時に `NULL` を返します。引数 *v* が `NULL` の場合、バイトオブジェクトの中身は初期化されていません。

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

戻り値: 新しい参照。次に属します: Stable ABI. C 関数の `printf()` スタイルの *format* 文字列と可変長の引数を取り、結果の Python バイトオブジェクトのサイズを計算し、値を指定した書式にしたがって変換したバイトオブジェクトを返します。可変長の引数は C のデータ型でなければならず、*format* 文字列中のフォーマット文字と厳密に関連付けられていなければなりません。下記のフォーマット文字が使用できます:

書式指定文字	型	備考
%%	*n/a*	リテラルの % 文字
%c	int	C の整数型で表現される単一のバイト。
%d	int	printf("%d") と同等。[*1]
%u	unsigned int	printf("%u") と同等。[*1]
%ld	long	printf("%ld") と同等。[*1]
%lu	unsigned long	printf("%lu") と同等。[*1]
%zd	*Py_ssize_t*	printf("%zd") と同等。[*1]
%zu	size_t	printf("%zu") と同等。[*1]
%i	int	printf("%i") と同等。[*1]
%x	int	printf("%x") と同等。[*1]
%s	const char*	null で終端された C の文字列。
%p	const void*	C ポインタの 16 進表記。printf("%p") とほとんど同じですが、プラットフォームにおける printf の定義に関わりなく先頭にリテラル 0x が付きます。

識別できない書式指定文字があった場合、残りの書式文字列はそのまま結果のオブジェクトにコピーされ、残りの引数は無視されます。

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

戻り値: **新しい参照。次に属します**: Stable ABI. ちょうど 2 つの引数を取ることを除いて、*PyBytes_FromFormat()* と同じです。

PyObject *PyBytes_FromObject(*PyObject* *o)

戻り値: **新しい参照。次に属します**: Stable ABI. バッファプロトコルを実装するオブジェクト *o* のバイト表現を返します。

Py_ssize_t PyBytes_Size(*PyObject* *o)

次に属します: Stable ABI. バイトオブジェクト *o* のバイト単位の長さを返します。

Py_ssize_t PyBytes_GET_SIZE(*PyObject* *o)

PyBytes_Size() に似ていますが、エラーチェックを行いません。

char *PyBytes_AsString(*PyObject* *o)

次に属します: Stable ABI. *o* の中身へのポインタを返します。ポインタは、len(o) + 1 バイトからなる *o* の内部バッファを参照します。他に null のバイトがあるかどうかにかかわらず、バッファの最後のバイトは必ず null になります。PyBytes_FromStringAndSize(NULL, size) で生成された場合を除いて、データを修正してはなりません。またポインタを解放（deallocated）してはなりません。もし、*o* が bytes オブジェクトでなければ、*PyBytes_AsString()* は NULL を返し TypeError を送出します。

[*1] 整数指定子 (d, u, ld, lu, zd, zu, i, x): 精度が与えられていても、0 指定子は有効です。

char *PyBytes_AS_STRING(*PyObject* *string)

PyBytes_AsString() に似ていますが、エラーチェックを行いません。

int PyBytes_AsStringAndSize(*PyObject* *obj, char **buffer, *Py_ssize_t* *length)

次に属します: Stable ABI. Return the null-terminated contents of the object *obj* through the output variables *buffer* and *length*. Returns `0` on success.

length の値が `NULL` の場合、バイトオブジェクトが null バイトを含まない可能性があります。その場合、関数は `-1` を返し、`ValueError` を送出します。

buffer は *obj* の内部バッファを参照していて、これには末尾の null バイトも含んでいます (これは *length* には数えられません)。オブジェクトが `PyBytes_FromStringAndSize(NULL, size)` で生成された場合を除いて、何があってもデータを改変してはいけません。オブジェクトを解放 (deallocate) してもいけません。*obj* が bytes オブジェクトでなかった場合は、*PyBytes_AsStringAndSize()* は `-1` を返し `TypeError` を送出します。

バージョン 3.5 で変更: 以前は bytes オブジェクトにヌルバイトが埋め込まれていたときに `TypeError` を送出していました。

void PyBytes_Concat(*PyObject* **bytes, *PyObject* *newpart)

次に属します: Stable ABI. *newpart* の内容を *bytes* の後ろに連結した新しいバイトオブジェクトを **bytes* に生成します。呼び出し側は新しい参照を所有します。*bytes* の古い値の参照は盗まれます。もし新しいオブジェクトが生成できない場合、古い *bytes* の参照は放棄され、**bytes* の値は `NULL` に設定されます; 適切な例外が設定されます。

void PyBytes_ConcatAndDel(*PyObject* **bytes, *PyObject* *newpart)

次に属します: 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).

int _PyBytes_Resize(*PyObject* **bytes, *Py_ssize_t* newsize)

A way to resize a bytes object even though it is "immutable". Only use this to build up a brand new bytes object; don't use this if the bytes may already be known in other parts of the code. It is an error to call this function if the refcount on the input bytes object is not one. 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.

8.3.2 bytearray オブジェクト

type PyByteArrayObject

この *PyObject* のサブタイプは Python の bytearray オブジェクトを表します。

PyTypeObject PyByteArray_Type

次に属します: Stable ABI. この *PyTypeObject* のインスタンスは、Python bytearray 型を示します。Python レイヤでの `bytearray` と同じオブジェクトです。

型チェックマクロ

int **PyByteArray_Check**(*PyObject* *o)

オブジェクト *o* が bytearray オブジェクトか bytearray 型のサブタイプのインスタンスである場合に真を返します。この関数は常に成功します。

int **PyByteArray_CheckExact**(*PyObject* *o)

オブジェクト *o* が bytearray オブジェクトだが bytearray 型のサブタイプのインスタンスでない場合に真を返します。この関数は常に成功します。

ダイレクト API 関数

PyObject ***PyByteArray_FromObject**(*PyObject* *o)

戻り値: **新しい参照。次に属します**: Stable ABI. *buffer protocol* を実装した任意のオブジェクト *o* から、新しい bytearray オブジェクトを作成し、返します。

On failure, return `NULL` with an exception set.

PyObject ***PyByteArray_FromStringAndSize**(const char *string, *Py_ssize_t* len)

戻り値: **新しい参照。次に属します**: 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)

戻り値: **新しい参照。次に属します**: Stable ABI. bytearray *a* と *b* を連結した結果を新しい bytearray として返します。

On failure, return `NULL` with an exception set.

Py_ssize_t **PyByteArray_Size**(*PyObject* *bytearray)

次に属します: Stable ABI. `NULL` ポインタチェックの後に *bytearray* のサイズを返します。

char ***PyByteArray_AsString**(*PyObject* *bytearray)

次に属します: Stable ABI. `NULL` ポインタチェックの後に *bytearray* の内容を char 配列として返します。返される配列には、常に余分な null バイトが追加されます。

int **PyByteArray_Resize**(*PyObject* *bytearray, *Py_ssize_t* len)

次に属します: Stable ABI. Resize the internal buffer of *bytearray* to *len*.

マクロ

以下のマクロは、ポインタのチェックをしないことにより安全性を犠牲にしてスピードを優先しています。

char ***PyByteArray_AS_STRING**(*PyObject* *bytearray)

PyByteArray_AsString() に似ていますが、エラーチェックを行いません。

Py_ssize_t **PyByteArray_GET_SIZE**(*PyObject* *bytearray)

PyByteArray_Size() に似ていますが、エラーチェックを行いません。

8.3.3 Unicode オブジェクトと codec

Unicode オブジェクト

Python3.3 の **PEP 393** 実装から、メモリ効率を維持しながら Unicode 文字の完全な範囲を扱えるように、Unicode オブジェクトは内部的に多様な表現形式を用いています。すべてのコードポイントが 128、256 または 65536 以下の文字列に対して特別なケースが存在しますが、それ以外ではコードポイントは 1114112 以下 (これはすべての Unicode 範囲です) でなければなりません。

UTF-8 representation is created on demand and cached in the Unicode object.

注釈

The *Py_UNICODE* representation has been removed since Python 3.12 with deprecated APIs. See **PEP 623** for more information.

Unicode 型

以下は Python の Unicode 実装に用いられている基本 Unicode オブジェクト型です:

type **Py_UCS4**

type **Py_UCS2**

type **Py_UCS1**

次に属します: Stable ABI. これらの型は、それぞれ、32 ビット、16 ビット、そして 8 ビットの文字を保持するのに充分な幅を持つ符号なしの整数型の typedef です。単一の Unicode 文字を扱う場合は、*Py_UCS4* を用いてください。

Added in version 3.3.

type **Py_UNICODE**

This is a typedef of `wchar_t`, which is a 16-bit type or 32-bit type depending on the platform.

バージョン 3.3 で変更: 以前のバージョンでは、Python をビルドした際に "narrow" または "wide" Unicode バージョンのどちらを選択したかによって、16 ビットか 32 ビットのどちらかの型になっていました。

type **PyASCIIObject**

type **PyCompactUnicodeObject**

type **PyUnicodeObject**

これらの *PyObject* のサブタイプは Python Unicode オブジェクトを表現します。Unicode オブジェクトを扱う全ての API 関数は *PyObject* へのポインタを受け取って PyObject へのポインタを返すので、ほとんどの場合、これらの型を直接使うべきではありません。

Added in version 3.3.

PyTypeObject **PyUnicode_Type**

次に属します: Stable ABI. This instance of *PyTypeObject* represents the Python Unicode type. It is exposed to Python code as `str`.

The following APIs are C macros and static inlined functions for fast checks and access to internal read-only data of Unicode objects:

int **PyUnicode_Check**(*PyObject* *obj)

オブジェクト *obj* が Unicode オブジェクトか Unicode 型のサブタイプのインスタンスである場合に真を返します。この関数は常に成功します。

int **PyUnicode_CheckExact**(*PyObject* *obj)

オブジェクト *obj* が Unicode オブジェクトだがサブタイプのインスタンスでない場合に真を返します。この関数は常に成功します。

int **PyUnicode_READY**(*PyObject* *unicode)

Returns `0`. This API is kept only for backward compatibility.

Added in version 3.3.

バージョン 3.10 で非推奨: This API does nothing since Python 3.12.

Py_ssize_t **PyUnicode_GET_LENGTH**(*PyObject* *unicode)

Unicode 文字列のコードポイントでの長さを返します。*unicode* は " 正統な" 表現形式の Unicode オブジェクトでなければなりません (ただしチェックはしません)。

Added in version 3.3.

Py_UCS1 ***PyUnicode_1BYTE_DATA**(*PyObject* *unicode)

Py_UCS2 ***PyUnicode_2BYTE_DATA**(*PyObject* *unicode)

Py_UCS4 ***PyUnicode_4BYTE_DATA**(*PyObject* *unicode)

文字に直接アクセスするために、UCS1, UCS2, UCS4 のいずれかの整数型にキャストされた正統な表現形式へのポインタを返します。正統な表現が適正な文字サイズになっているかどうかのチェックはしません; *PyUnicode_KIND()* を使って正しい関数を選んでください。

Added in version 3.3.

PyUnicode_1BYTE_KIND

PyUnicode_2BYTE_KIND

PyUnicode_4BYTE_KIND

PyUnicode_KIND() マクロの返り値です。

Added in version 3.3.

バージョン 3.12 で変更: `PyUnicode_WCHAR_KIND` は削除されました。

int **PyUnicode_KIND**(*PyObject* *unicode)

この Unicode がデータを保存するのに 1 文字あたり何バイト使っているかを示す PyUnicode 種別の定数 (上を読んでください) のうち 1 つを返します。*unicode* は " 正統な" 表現形式の Unicode オブジェクトでなければなりません (ただしチェックはしません)。

Added in version 3.3.

void *PyUnicode_DATA(*PyObject* *unicode)

生の Unicode バッファへの void ポインタを返します。*unicode* は ” 正統な” 表現形式の Unicode オブジェクトでなければなりません (ただしチェックはしません)。

Added in version 3.3.

void PyUnicode_WRITE(int kind, void *data, *Py_ssize_t* index, *Py_UCS4* value)

Write into a canonical representation *data* (as obtained with *PyUnicode_DATA()*). This function performs no sanity checks, and is intended for usage in loops. The caller should cache the *kind* value and *data* pointer as obtained from other calls. *index* is the index in the string (starts at 0) and *value* is the new code point value which should be written to that location.

Added in version 3.3.

Py_UCS4 PyUnicode_READ(int kind, void *data, *Py_ssize_t* index)

正統な表現形式となっている (*PyUnicode_DATA()* で取得した) *data* からコードポイントを読み取ります。チェックや事前確認のマクロ呼び出しは一切行われません。

Added in version 3.3.

Py_UCS4 PyUnicode_READ_CHAR(*PyObject* *unicode, *Py_ssize_t* index)

Unicode オブジェクト *unicode* から文字を読み取ります。この Unicode オブジェクトは ” 正統な” 表現形式でなければなりません。何度も連続して読み取る場合には、このマクロは *PyUnicode_READ()* よりも非効率的です。

Added in version 3.3.

Py_UCS4 PyUnicode_MAX_CHAR_VALUE(*PyObject* *unicode)

unicode に基づいて他の文字列を作るのに適した最大のコードポイントを返します。この Unicode オブジェクトは ” 正統な” 表現形式でなければなりません。この値は常に概算値ですが、文字列全体を調べるよりも効率的です。

Added in version 3.3.

int PyUnicode_IsIdentifier(*PyObject* *unicode)

次に属します: Stable ABI. 文字列が、identifiers 節の言語定義における有効な識別子であれば 1 を返します。それ以外の場合は 0 を返します。

バージョン 3.9 で変更: The function does not call *Py_FatalError()* anymore if the string is not ready.

Unicode 文字プロパティ

Unicode は数多くの異なる文字プロパティ (character property) を提供しています。よく使われる文字プロパティは、以下のマクロで利用できます。これらのマクロは Python の設定に応じて、各々 C の関数に対応付けられています。

int Py_UNICODE_ISSPACE(*Py_UCS4* ch)

ch が空白文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISLOWER(*Py_UCS4* ch)

ch が小文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISUPPER(*Py_UCS4* ch)

ch が大文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISTITLE(*Py_UCS4* ch)

ch がタイトルケース文字 (titlecase character) かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISLINEBREAK(*Py_UCS4* ch)

ch が改行文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISDECIMAL(*Py_UCS4* ch)

ch が decimal 文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISDIGIT(*Py_UCS4* ch)

ch が digit 文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISNUMERIC(*Py_UCS4* ch)

ch が数字 (numeric) 文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISALPHA(*Py_UCS4* ch)

ch がアルファベット文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISALNUM(*Py_UCS4* ch)

ch が英数文字かどうかに応じて 1 または 0 を返します。

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()`.

以下の API は、高速に直接文字変換を行うために使われます:

Py_UCS4 Py_UNICODE_TOLOWER(*Py_UCS4* ch)

ch を小文字に変換したものを返します。

バージョン 3.3 で非推奨: This function uses simple case mappings.

Py_UCS4 Py_UNICODE_TOUPPER(*Py_UCS4* ch)

ch を大文字に変換したものを返します。

バージョン 3.3 で非推奨: This function uses simple case mappings.

Py_UCS4 Py_UNICODE_TOTITLE(*Py_UCS4* ch)

ch をタイトルケース文字に変換したものを返します。

バージョン 3.3 で非推奨: This function uses simple case mappings.

int **Py_UNICODE_TODECIMAL**(*Py_UCS4* ch)

Return the character *ch* converted to a decimal positive integer. Return `-1` if this is not possible. This 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)

ch を double に変換したものを返します。不可能ならば `-1.0` を返します。この関数は例外を送出しません。

これらの API はサロゲートにも使えます:

int **Py_UNICODE_IS_SURROGATE**(*Py_UCS4* ch)

ch がサロゲートかどうか (`0xD800 <= ch <= 0xDFFF`) をチェックします。

int **Py_UNICODE_IS_HIGH_SURROGATE**(*Py_UCS4* ch)

ch が上位サロゲートかどうか (`0xD800 <= ch <= 0xDBFF`) をチェックします。

int **Py_UNICODE_IS_LOW_SURROGATE**(*Py_UCS4* ch)

ch が下位サロゲートかどうか (`0xDC00 <= ch <= 0xDFFF`) をチェックします。

Py_UCS4 **Py_UNICODE_JOIN_SURROGATES**(*Py_UCS4* high, *Py_UCS4* low)

Join two surrogate characters and return a single *Py_UCS4* value. *high* and *low* are respectively the leading and trailing surrogates in a surrogate pair. *high* must be in the range [0xD800; 0xDBFF] and *low* must be in the range [0xDC00; 0xDFFF].

Unicode 文字列の生成とアクセス

Unicode オブジェクトを生成したり、Unicode のシーケンスとしての基本的なプロパティにアクセスしたりするには、以下の API を使ってください:

PyObject ***PyUnicode_New**(*Py_ssize_t* size, *Py_UCS4* maxchar)

戻り値: **新しい参照。**新しい Unicode オブジェクトを生成します。*maxchar* は文字列に並べるコードポイントの正しい最大値にすべきです。その値は概算値として 127, 255, 65535, 1114111 の一番近い値に切り上げられます。

This is the recommended way to allocate a new Unicode object. Objects created using this function are not resizable.

On error, set an exception and return `NULL`.

Added in version 3.3.

PyObject ***PyUnicode_FromKindAndData**(int kind, const void *buffer, *Py_ssize_t* size)

戻り値: **新しい参照。**与えられた *kind* (取り得る値は *PyUnicode_1BYTE_KIND* などの *PyUnicode_KIND()* が返す値です) の Unicode オブジェクトを生成します。*buffer* は、与えられ

た kind に従って 1 文字あたり 1, 2, 4 バイトのいずれかを単位として、長さ *size* の配列へのポインタでなければなりません。

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*).

Added in version 3.3.

PyObject ***PyUnicode_FromStringAndSize**(const char *str, *Py_ssize_t* size)

戻り値: 新しい参照。次に属します: 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

バージョン 3.12 で変更: *str* == `NULL` with *size* > 0 is not allowed anymore.

PyObject ***PyUnicode_FromString**(const char *str)

戻り値: 新しい参照。次に属します: Stable ABI. UTF-8 エンコードされた null 終端の char 型バッファ *str* から Unicode オブジェクトを生成します。

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

戻り値: 新しい参照。次に属します: 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.

一つの変換指定子は 2 またはそれ以上の文字を含み、その構成要素は以下からなりますが、示した順に出現しなければなりません:

1. 指定子の開始を示す文字 `'%'` 。
2. 変換フラグ (オプション)。一部の変換型の結果に影響します。
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. 精度長変換子 (オプション)。
6. 変換型。

変換フラグ文字を以下に示します:

Flag	意味
0	数値型に対してゼロによるパディングを行います。
-	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):

修飾子	型
l	long または unsigned long
ll	long long または 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	型	備考
`%`	*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*` または `const wchar_t*`	null で終端された C の文字列。
`p`	`const void*`	C ポインタの 16 進表記。`printf("%p")` とほとんど同じですが、プラットフォームにおける `printf` の定義に関わりなく先頭にリテラル `0x` が付きます。
`A`	*PyObject*`*`	`ascii()` の戻り値。
`U`	*PyObject*`*`	Unicode オブジェクト。
`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*`*`	*PyObject_Str()* の戻り値。
`R`	*PyObject*`*`	*PyObject_Repr()* の戻り値。

注釈

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`).

注釈

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).

バージョン 3.2 で変更: "%lld", "%llu" のサポートが追加されました。

バージョン 3.3 で変更: "%li", "%lli", "%zi" のサポートが追加されました。

バージョン 3.4 で変更: "%s", "%A", "%U", "%V", "%S", "%R" での幅フォーマッタおよび精度フォーマッタのサポートが追加されました。

バージョン 3.12 で変更: 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.

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

戻り値: **新しい参照。次に属します**: Stable ABI. ちょうど 2 つの引数を取ることを除いて、*PyUnicode_FromFormat()* と同じです。

PyObject ***PyUnicode_FromObject**(*PyObject* *obj)

戻り値: **新しい参照。次に属します**: 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.

Unicode やそのサブタイプ以外のオブジェクトでは TypeError が引き起こされます。

PyObject ***PyUnicode_FromEncodedObject**(*PyObject* *obj, const char *encoding, const char *errors)

戻り値: **新しい参照。次に属します**: Stable ABI. エンコードされている *obj* を Unicode オブジェクトにデコードします。

bytes や bytearray や他の *bytes-like objects* は、与えられた *encoding* に従ってデコードされ、*errors* で定義されたエラーハンドリングが使われます。これらの引数は両方とも NULL にでき、その場合この API はデフォルト値を使います (詳しことは **組み込み** *codec (built-in codec)* を参照してください)。

その他の Unicode オブジェクトを含むオブジェクトは TypeError 例外を引き起こします。

この API は、エラーが生じたときには NULL を返します。呼び出し側は返されたオブジェクトに対し参照カウンタを 1 つ減らす (decref) する責任があります。

const char ***PyUnicode_GetDefaultEncoding**(void)

次に属します: 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)

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. Unicode オブジェクトの長さをコードポイントで返します。

On error, set an exception and return `-1`.

Added in 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)

ある Unicode オブジェクトから他へ文字をコピーします。この関数は必要なときに文字変換を行い、可能な場合は `memcpy()` へ差し戻します。失敗のときには `-1` を返し、例外を設定します。そうでない場合は、コピーした文字数を返します。

Added in version 3.3.

Py_ssize_t **PyUnicode_Fill**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* length, *Py_UCS4* fill_char)

文字列を文字で埋めます: `unicode[start:start+length]` で *fill_char* を埋めることになります。

fill_char が文字列の最大文字よりも大きい場合や、文字列 2 つ以上の参照を持ってた場合は失敗します。

書き込んだ文字数を返すか、失敗のときには `-1` を返し例外を送出します。

Added in version 3.3.

int **PyUnicode_WriteChar**(*PyObject* *unicode, *Py_ssize_t* index, *Py_UCS4* character)

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. Write a character to a string. The string must have been created through *PyUnicode_New()*. Since Unicode strings are supposed to be immutable, the string must not be shared, or have been hashed yet.

This function checks that *unicode* is a Unicode object, that the index is not out of bounds, and that the object can be modified safely (i.e. that it its reference count is one).

Return `0` on success, `-1` on error with an exception set.

Added in version 3.3.

Py_UCS4 **PyUnicode_ReadChar**(*PyObject* *unicode, *Py_ssize_t* index)

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. 文字列から文字を読み取ります。エラーチェックを行わない *PyUnicode_READ_CHAR()* とは対照的に、この関数は *unicode* が Unicode オブジェクトであること、インデックスが範囲内であることをチェックします。

Return character on success, `-1` on error with an exception set.

Added in version 3.3.

PyObject ***PyUnicode_Substring**(*PyObject* *unicode, *Py_ssize_t* start, *Py_ssize_t* end)

戻り値: **新しい参照。次に属します**: Stable ABI *(**バージョン** 3.7 **より**)*. 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`.

Added in version 3.3.

Py_UCS4 ***PyUnicode_AsUCS4**(*PyObject* *unicode, *Py_UCS4* *buffer, *Py_ssize_t* buflen, int copy_null)

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. 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.

Added in version 3.3.

Py_UCS4 ***PyUnicode_AsUCS4Copy**(*PyObject* *unicode)

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. 文字列 *unicode* を *PyMem_Malloc()* でメモリ確保された新しい UCS4 型のバッファにコピーします。これが失敗した場合は、`NULL` を返し `MemoryError` をセットします。返されたバッファは必ず null コードポイントが追加されています。

Added in version 3.3.

ロケールエンコーディング

現在のロケールエンコーディングはオペレーティングシステムのテキストをデコードするのに使えます。

PyObject ***PyUnicode_DecodeLocaleAndSize**(const char *str, *Py_ssize_t* length, const char *errors)

戻り値: **新しい参照。次に属します**: Stable ABI *(**バージョン** 3.7 **より**)*. 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.

> **参考**
>
> *Py_DecodeLocale()* 関数。

Added in version 3.3.

バージョン 3.7 で変更: この関数は、Android 以外では現在のロケールエンコーディングを `surrogateescape` エラーハンドラで使うようになりました。以前は、*Py_DecodeLocale()* が `surrogateescape` で使われ、現在のロケールエンコーディングは `strict` で使われていました。

PyObject *PyUnicode_DecodeLocale(const char *str, const char *errors)

戻り値: 新しい参照。次に属します: Stable ABI (バージョン 3.7 より). Similar to *PyUnicode_DecodeLocaleAndSize()*, but compute the string length using `strlen()`.

Added in version 3.3.

PyObject *PyUnicode_EncodeLocale(*PyObject* *unicode, const char *errors)

戻り値: 新しい参照。次に属します: Stable ABI (バージョン 3.7 より). 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.

参考

Py_EncodeLocale() 関数。

Added in version 3.3.

バージョン 3.7 で変更: この関数は、Android 以外では現在のロケールエンコーディングを `surrogateescape` エラーハンドラで使うようになりました。以前は、*Py_EncodeLocale()* が `surrogateescape` で使われ、現在のロケールエンコーディングは `strict` で使われていました。

ファイルシステムエンコーディング

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)

次に属します: 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`.

Added in version 3.1.

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

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)

次に属します: 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`.

Added in version 3.2.

バージョン 3.6 で変更: *path-like object* を受け入れるようになりました。

PyObject ***PyUnicode_DecodeFSDefaultAndSize**(const char *str, *Py_ssize_t* size)

戻り値: 新しい参照。次に属します: 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()*.

> **参考**
>
> *Py_DecodeLocale()* 関数。

バージョン 3.6 で変更: The *filesystem error handler* is now used.

PyObject ***PyUnicode_DecodeFSDefault**(const char *str)

戻り値: 新しい参照。次に属します: Stable ABI. Decode a null-terminated string from the *filesystem encoding and error handler*.

If the string length is known, use *PyUnicode_DecodeFSDefaultAndSize()*.

バージョン 3.6 で変更: The *filesystem error handler* is now used.

PyObject ***PyUnicode_EncodeFSDefault**(*PyObject* *unicode)

戻り値: 新しい参照。次に属します: 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()*.

> **参考**
>
> *Py_EncodeLocale()* 関数。

Added in version 3.2.

バージョン 3.6 で変更: The *filesystem error handler* is now used.

wchar_t サポート

`wchar_t` をサポートするプラットフォームでの wchar_t サポート:

PyObject ***PyUnicode_FromWideChar**(const wchar_t *wstr, *Py_ssize_t* size)

戻り値: **新しい参照。次に属します**: 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)

次に属します: 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)

次に属します: Stable ABI *(バージョン 3.7 より)*. 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.

Added in version 3.2.

バージョン 3.7 で変更: Raises a `ValueError` if *size* is `NULL` and the `wchar_t*` string contains null characters.

組み込み codec (built-in codec)

Python には、処理速度を高めるために C で書かれた codec が揃えてあります。これら全ての codec は以下の関数を介して直接利用できます。

以下の API の多くが、*encoding* と *errors* という二つの引数をとります。これらのパラメータは、組み込みの文字列コンストラクタである `str()` における同名のパラメータと同じ意味を持ちます。

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.

errors で指定するエラー処理もまた、`NULL` を指定できます。`NULL` を指定すると、codec で定義されているデフォルト処理の使用を意味します。全ての組み込み codec で、デフォルトのエラー処理は "strict" (`ValueError` を送出する) になっています。

個々の codec は全て同様のインターフェースを使っています。個別の codec の説明では、説明を簡単にするために以下の汎用のインターフェースとの違いだけを説明しています。

汎用 codec

以下は汎用 codec の API です:

PyObject ***PyUnicode_Decode**(const char *str, *Py_ssize_t* size, const char *encoding, const char *errors)

戻り値: 新しい参照。次に属します: 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)

戻り値: 新しい参照。次に属します: Stable ABI. Unicode オブジェクトをエンコードし、その結果を Python の bytes オブジェクトとして返します。*encoding* および *errors* は Unicode 型の `encode()` メソッドに与える同名のパラメータと同じ意味を持ちます。使用する codec の検索は、Python の codec レジストリを使って行います。codec が例外を送出した場合には `NULL` を返します。

UTF-8 Codecs

以下は UTF-8 codec の API です:

PyObject ***PyUnicode_DecodeUTF8**(const char *str, *Py_ssize_t* size, const char *errors)

戻り値: 新しい参照。次に属します: Stable ABI. UTF-8 でエンコードされた *size* バイトの文字列 *str* から Unicode オブジェクトを生成します。codec が例外を送出した場合には `NULL` を返します。

PyObject ***PyUnicode_DecodeUTF8Stateful**(const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

戻り値: 新しい参照。次に属します: Stable ABI. *consumed* が `NULL` の場合、*PyUnicode_DecodeUTF8()* と同じように動作します。*consumed* が `NULL` でない場合、末尾の不完全な UTF-8 バイト列はエラーとみなされません。これらのバイト列はデコードされず、デコードされたバイト数は *consumed* に格納されます。

PyObject *PyUnicode_AsUTF8String(*PyObject* *unicode)

戻り値: **新しい参照。次に属します**: Stable ABI. UTF-8 で Unicode オブジェクトをエンコードし、結果を Python バイト列オブジェクトとして返します。エラー処理は "strict" です。codec が例外を送出した場合には NULL を返します。

const char *PyUnicode_AsUTF8AndSize(*PyObject* *unicode, *Py_ssize_t* *size)

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. Unicode オブジェクトを UTF-8 でエンコードしたものへのポインタを返し、エンコードされた表現形式でのサイズ (バイト単位) を *size* に格納します。*size* 引数は NULL でも構いません; その場合はサイズは格納されません。返されるバッファには、null コードポイントがあるかどうかに関わらず、常に null バイトが終端に付加されています (これは *size* には勘定されません)。

In the case of an error, NULL is returned with an exception set and no *size* is stored.

This caches the UTF-8 representation of the string in the Unicode object, and subsequent calls will return a pointer to the same buffer. The caller is not responsible for deallocating the buffer. The buffer is deallocated and pointers to it become invalid when the Unicode object is garbage collected.

Added in version 3.3.

バージョン 3.7 で変更: 返り値の型が `char *` ではなく `const char *` になりました。

バージョン 3.10 で変更: This function is a part of the *limited API*.

const char *PyUnicode_AsUTF8(*PyObject* *unicode)

PyUnicode_AsUTF8AndSize() とほぼ同じですが、サイズを格納しません。

> **警告**
>
> 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.

Added in version 3.3.

バージョン 3.7 で変更: 返り値の型が `char *` ではなく `const char *` になりました。

UTF-32 Codecs

以下は UTF-32 codec API です:

PyObject *PyUnicode_DecodeUTF32(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

戻り値: **新しい参照。次に属します**: Stable ABI. UTF-32 でエンコードされたバッファ文字列から *size* バイトをデコードし、Unicode オブジェクトとして返します。*errors* は (NULL でないなら) エラーハンドラを指定します。デフォルトは "strict" です。

byteorder が NULL でない時、デコーダは与えられたバイトオーダーでデコードを開始します。

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

*byteorder が 0 で、入力データの最初の 4 バイトが byte order mark (BOM) ならば、デコーダはこのバイトオーダーに切り替え、BOM は結果の Unicode 文字列にコピーされません。*byteorder が -1 または 1 ならば、全ての byte order mark は出力にコピーされます。

デコードが完了した後、入力データの終端に来た時点でのバイトオーダーを **byteorder* にセットします。

byteorder が NULL のとき、codec は native order モードで開始します。

codec が例外を発生させたときは NULL を返します。

PyObject ***PyUnicode_DecodeUTF32Stateful**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

戻り値: **新しい参照。次に属します**: Stable ABI. *consumed* が NULL のとき、*PyUnicode_DecodeUTF32()* と同じように振る舞います。*consumed* が NULL でないとき、*PyUnicode_DecodeUTF32Stateful()* は末尾の不完全な (4 で割り切れない長さのバイト列などの) UTF-32 バイト列をエラーとして扱いません。末尾の不完全なバイト列はデコードされず、デコードされたバイト数が *consumed* に格納されます。

PyObject ***PyUnicode_AsUTF32String**(*PyObject* *unicode)

戻り値: **新しい参照。次に属します**: Stable ABI. ネイティブバイトオーダーで UTF-32 エンコーディングされた Python バイト文字列を返します。文字列は常に BOM マークで始まります。エラーハンドラは "strict" です。codec が例外を発生させたときは NULL を返します。

UTF-16 Codecs

以下は UTF-16 codec の API です:

PyObject ***PyUnicode_DecodeUTF16**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder)

戻り値: **新しい参照。次に属します**: Stable ABI. UTF-16 でエンコードされたバッファ *s* から *size* バイトだけデコードして、結果を Unicode オブジェクトで返します。*errors* は (NULL でない場合) エラー処理方法を定義します。デフォルト値は "strict" です。

byteorder が NULL でない時、デコーダは与えられたバイトオーダーでデコードを開始します。

```
*byteorder == -1: little endian
*byteorder == 0:  native order
*byteorder == 1:  big endian
```

*byteorder が 0 で、入力データの先頭 2 バイトがバイトオーダーマーク (BOM) だった場合、デコーダは BOM が示すバイトオーダーに切り替え、その BOM を結果の Unicode 文字列にコピーしませ

ん。*byteorder が -1 か 1 だった場合、すべての BOM は出力へコピーされます (出力では \ufeff か \ufffe のどちらかになるでしょう)。

デコードが完了した後、入力データの終端に来た時点でのバイトオーダーを *byteorder にセットします。

byteorder が NULL のとき、codec は native order モードで開始します。

codec が例外を発生させたときは NULL を返します。

PyObject ***PyUnicode_DecodeUTF16Stateful**(const char *str, *Py_ssize_t* size, const char *errors, int *byteorder, *Py_ssize_t* *consumed)

戻り値: **新しい参照。次に属します**: Stable ABI. *consumed* が NULL の場合、*PyUnicode_DecodeUTF16()* と同じように動作します。*consumed* が NULL でない場合、*PyUnicode_DecodeUTF16Stateful()* は末尾の不完全な UTF-16 バイト列 (奇数長のバイト列や分割されたサロゲートペア) をエラーとみなしません。これらのバイト列はデコードされず、デコードされたバイト数を *consumed* に返します。

PyObject ***PyUnicode_AsUTF16String**(*PyObject* *unicode)

戻り値: **新しい参照。次に属します**: Stable ABI. ネイティブバイトオーダーで UTF-16 エンコーディングされた Python バイト文字列を返します。文字列は常に BOM マークで始まります。エラーハンドラは "strict" です。codec が例外を発生させたときは NULL を返します。

UTF-7 Codecs

以下は UTF-7 codec の API です:

PyObject ***PyUnicode_DecodeUTF7**(const char *str, *Py_ssize_t* size, const char *errors)

戻り値: **新しい参照。次に属します**: 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)

戻り値: **新しい参照。次に属します**: Stable ABI. *consumed* が NULL のとき、*PyUnicode_DecodeUTF7()* と同じように動作します。*consumed* が NULL でないとき、末尾の不完全な UTF-7 base-64 部分をエラーとしません。不完全な部分のバイト列はデコードせずに、デコードしたバイト数を *consumed* に格納します。

Unicode-Escape Codecs

以下は "Unicode Escape" codec の API です:

PyObject ***PyUnicode_DecodeUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

戻り値: **新しい参照。次に属します**: 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)

戻り値: **新しい参照。次に属します**: Stable ABI. Unicode-Escape を使い Unicode オブジェクトをエ

ンコードし、結果を bytes オブジェクトとして返します。エラー処理は ”strict” です。codec が例外を送出した場合には NULL を返します。

Raw-Unicode-Escape Codecs

以下は ”Raw Unicode Escape” codec の API です:

PyObject ***PyUnicode_DecodeRawUnicodeEscape**(const char *str, *Py_ssize_t* size, const char *errors)

戻り値: 新しい参照。次に属します: 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)

戻り値: 新しい参照。次に属します: Stable ABI. Raw-Unicode-Escape を使い Unicode オブジェクトをエンコードし、結果を bytes オブジェクトとして返します。エラー処理は ”strict” です。codec が例外を送出した場合には NULL を返します。

Latin-1 Codecs

以下は Latin-1 codec の API です: Latin-1 は、Unicode 序数の最初の 256 個に対応し、エンコード時にはこの 256 個だけを受理します。

PyObject ***PyUnicode_DecodeLatin1**(const char *str, *Py_ssize_t* size, const char *errors)

戻り値: 新しい参照。次に属します: Stable ABI. Latin-1 でエンコードされた *size* バイトの文字列 *str* から Unicode オブジェクトを生成します。codec が例外を送出した場合には NULL を返します。

PyObject ***PyUnicode_AsLatin1String**(*PyObject* *unicode)

戻り値: 新しい参照。次に属します: Stable ABI. Latin-1 で Unicode オブジェクトをエンコードし、結果を Python bytes オブジェクトとして返します。エラー処理は ”strict” です。codec が例外を送出した場合には NULL を返します。

ASCII Codecs

以下は ASCII codec の API です。7 ビットの ASCII データだけを受理します。その他のコードはエラーになります。

PyObject ***PyUnicode_DecodeASCII**(const char *str, *Py_ssize_t* size, const char *errors)

戻り値: 新しい参照。次に属します: 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)

戻り値: 新しい参照。次に属します: Stable ABI. ASCII で Unicode オブジェクトをエンコードし、結果を Python bytes オブジェクトとして返します。エラー処理は ”strict” です。codec が例外を送出した場合には NULL を返します。

Character Map Codecs

この codec は、多くの様々な codec を実装する際に使われるという点で特殊な codec です (実際、`encodings` パッケージに入っている標準 codecs のほとんどは、この codec を使っています)。この codec は、文字のエンコードやデコードに対応表を使います。提供される対応表のオブジェクトは `__getitem__()` マッピングインターフェースをサポートしていなければなりません; 辞書やシーケンスがそれに適しています。

以下は mapping codec の API です:

PyObject ***PyUnicode_DecodeCharmap**(const char *str, *Py_ssize_t* length, *PyObject* *mapping, const char *errors)

戻り値: 新しい参照。次に属します: Stable ABI. 与えられた *mapping* オブジェクトを使って、*size* バイトのエンコードされた文字列 *str* をデコードして Unicode オブジェクトを作成します。codec が例外を発生させたときは `NULL` を返します。

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)

戻り値: 新しい参照。次に属します: Stable ABI. Unicode オブジェクトを *mapping* に指定されたオブジェクトを使ってエンコードし、結果を bytes オブジェクトとして返します。エラー処理は "strict" です。codec が例外を送出した場合には `NULL` を返します。

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.

以下の codec API は Unicode から Unicode への対応付けを行う特殊なものです。

PyObject ***PyUnicode_Translate**(*PyObject* *unicode, *PyObject* *table, const char *errors)

戻り値: 新しい参照。次に属します: Stable ABI. 文字列に文字対応表 *table* を適用して変換し、変換結果を Unicode オブジェクトで返します。codec が例外を発行した場合には `NULL` を返します。

対応表は、Unicode 序数を表す整数を Unicode 序数を表す整数または `None` (その文字を削除する) に対応付けなければなりません。

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 は codecs で通常使われるのと同じ意味を持ちます。*errors* は `NULL` にしてもよく、デフォルトエラー処理の使用を意味します。

Windows 用の MBCS codec

以下は MBCS codec の API です。この codec は現在のところ、Windows 上だけで利用でき、変換の実装には Win32 MBCS 変換機構 (Win32 MBCS converter) を使っています。MBCS (または DBCS) はエンコード方式の種類 (class) を表す言葉で、単一のエンコード方式を表すわけでなないので注意してください。利用されるエンコード方式 (target encoding) は、codec を動作させているマシン上のユーザ設定で定義されています。

PyObject *`PyUnicode_DecodeMBCS`(const char *str, *Py_ssize_t* size, const char *errors)

戻り値: 新しい参照。次に属します: Stable ABI *on Windows* (**バージョン** *3.7* **より**). 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)

戻り値: 新しい参照。次に属します: Stable ABI *on Windows* (**バージョン** *3.7* **より**). *consumed* が `NULL` のとき、*PyUnicode_DecodeMBCS()* と同じ動作をします。*consumed* が `NULL` でないとき、*PyUnicode_DecodeMBCSStateful()* は文字列の最後にあるマルチバイト文字の前半バイトをデコードせず、*consumed* にデコードしたバイト数を格納します。

PyObject *`PyUnicode_DecodeCodePageStateful`(int code_page, const char *str, *Py_ssize_t* size, const char *errors, *Py_ssize_t* *consumed)

戻り値: 新しい参照。次に属します: Stable ABI *on Windows* (**バージョン** *3.7* **より**). Similar to *PyUnicode_DecodeMBCSStateful()*, except uses the code page specified by *code_page*.

PyObject *`PyUnicode_AsMBCSString`(*PyObject* *unicode)

戻り値: 新しい参照。次に属します: Stable ABI *on Windows* (**バージョン** *3.7* **より**). MBCS で Unicode オブジェクトをエンコードし、結果を Python バイト列オブジェクトとして返します。エラー処理は "strict" です。codec が例外を送出した場合には `NULL` を返します。

PyObject *`PyUnicode_EncodeCodePage`(int code_page, *PyObject* *unicode, const char *errors)

戻り値: 新しい参照。次に属します: Stable ABI *on Windows* (**バージョン** *3.7* **より**). 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.

Added in version 3.3.

メソッドとスロット

メソッドおよびスロット関数 (slot function)

以下の API は Unicode オブジェクトおよび文字列を入力に取り (説明では、どちらも文字列と表記しています)、場合に応じて Unicode オブジェクトか整数を返す機能を持っています。

これらの関数は全て、例外が発生した場合には `NULL` または `-1` を返します。

PyObject *`PyUnicode_Concat`(*PyObject* *left, *PyObject* *right)

戻り値: 新しい参照。次に属します: Stable ABI. 二つの文字列を結合して、新たな Unicode 文字列を生成します。

PyObject *PyUnicode_Split(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

戻り値: 新しい参照。次に属します: Stable ABI. Unicode 文字列のリストを分割して、Unicode 文字列からなるリストを返します。*sep* が NULL の場合、全ての空白文字を使って分割を行います。それ以外の場合、指定された文字を使って分割を行います。最大で *maxsplit* 個までの分割を行います。*maxsplit* が負ならば分割数に制限を設けません。分割結果のリスト内には分割文字は含みません。

On error, return NULL with an exception set.

Equivalent to str.split().

PyObject *PyUnicode_RSplit(*PyObject* *unicode, *PyObject* *sep, *Py_ssize_t* maxsplit)

戻り値: 新しい参照。次に属します: 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)

戻り値: 新しい参照。次に属します: 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)

戻り値: 新しい参照。次に属します: 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)

戻り値: 新しい参照。次に属します: 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)

戻り値: 新しい参照。次に属します: Stable ABI. 指定した *separator* で文字列からなるシーケンスを連結 (join) し、連結結果を Unicode 文字列で返します。

Py_ssize_t **PyUnicode_Tailmatch**(*PyObject* *unicode, *PyObject* *substr, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

次に属します: 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)

次に属します: Stable ABI. `unicode[start:end]` 中に *substr* が最初に出現する場所を返します。このとき指定された検索方向 *direction* (*direction* == `1` は順方向検索、*direction* == `-1` は逆方向検索) で検索します。戻り値は最初にマッチが見つかった場所のインデックスです; 戻り値 `-1` はマッチが見つからなかったことを表し、`-2` はエラーが発生して例外情報が設定されていることを表します。

Py_ssize_t **PyUnicode_FindChar**(*PyObject* *unicode, *Py_UCS4* ch, *Py_ssize_t* start, *Py_ssize_t* end, int direction)

次に属します: Stable ABI *(バージョン 3.7 より)*. `unicode[start:end]` 中に文字 *ch* が最初に出現する場所を返します。このとき指定された検索方向 *direction* (*direction* == `1` は順方向検索、*direction* == `-1` は逆方向検索) で検索します。戻り値は最初にマッチが見つかった場所のインデックスです; 戻り値 `-1` はマッチが見つからなかったことを表し、`-2` はエラーが発生して例外情報が設定されていることを表します。

Added in version 3.3.

バージョン 3.7 で変更: *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)

次に属します: Stable ABI. `unicode[start:end]` に *substr* が重複することなく出現する回数を返します。エラーが発生した場合には `-1` を返します。

PyObject ***PyUnicode_Replace**(*PyObject* *unicode, *PyObject* *substr, *PyObject* *replstr, *Py_ssize_t* maxcount)

戻り値: **新しい参照。次に属します**: Stable ABI. *unicode* 中に出現する *substr* を最大で *maxcount* 個 *replstr* に置換し、置換結果である Unicode オブジェクトを返します。*maxcount* == `-1` にすると、文字列中に現れる全ての *substr* を置換します。

int **PyUnicode_Compare**(*PyObject* *left, *PyObject* *right)

次に属します: Stable ABI. 二つの文字列を比較して、左引数が右引数より小さい場合、左右引数が等価の場合、左引数が右引数より大きい場合に対して、それぞれ `-1`, `0`, `1` を返します。

この関数は、失敗したときに `-1` を返すので、*PyErr_Occurred()* を呼び出して、エラーをチェックすべきです。

int **PyUnicode_CompareWithASCIIString**(*PyObject* *unicode, const char *string)

次に属します: Stable ABI. Unicode オブジェクト *unicode* と *string* を比較して、左引数が右引数より小さい場合、左右引数が等価の場合、左引数が右引数より大きい場合に対して、それぞれ `-1`, `0`, `1` を

返します。ASCII エンコードされた文字列だけを渡すのが最も良いですが、入力文字列に非 ASCII 文字が含まれている場合は ISO-8859-1 として解釈します。

この関数は例外を送出しません。

PyObject ***PyUnicode_RichCompare**(*PyObject* *left, *PyObject* *right, int op)

戻り値: **新しい参照。次に属します**: Stable ABI. 二つの Unicode 文字列を比較して、下のうちの一つを返します:

- `NULL` を、例外が発生したときに返します。
- *Py_True* もしくは *Py_False* を、正しく比較できた時に返します。
- *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)

戻り値: **新しい参照。次に属します**: Stable ABI. 新たな文字列オブジェクトを *format* および *args* から生成して返します; このメソッドは `format % args` のようなものです。

int **PyUnicode_Contains**(*PyObject* *unicode, *PyObject* *substr)

次に属します: Stable ABI. *substr* が *unicode* 内にあるか調べ、その結果に応じて真または偽を返します。

substr は単要素の Unicode 文字に型強制できなければなりません。エラーが生じた場合には `-1` を返します。

void **PyUnicode_InternInPlace**(*PyObject* **p_unicode)

次に属します: 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)

戻り値: **新しい参照。次に属します**: 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 prevent 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.

8.3.4 タプルオブジェクト (tuple object)

type PyTupleObject

この *PyObject* のサブタイプは Python のタプルオブジェクトを表現します。

PyTypeObject PyTuple_Type

次に属します: Stable ABI. この *PyTypeObject* のインスタンスは Python のタプル型を表現します; Python レイヤにおける `tuple` と同じオブジェクトです。

int PyTuple_Check(*PyObject* *p)

p がタプルオブジェクトかタプル型のサブタイプのインスタンスである場合に真を返します。この関数は常に成功します。

int PyTuple_CheckExact(*PyObject* *p)

p がタプルオブジェクトだがタプル型のサブタイプのインスタンスでない場合に真を返します。この関数は常に成功します。

PyObject *PyTuple_New(*Py_ssize_t* len)

戻り値: **新しい参照。次に属します**: 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, ...)

戻り値: **新しい参照。次に属します**: 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)

次に属します: 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)

戻り値: **借用参照。次に属します**: Stable ABI. *p* の指すタプルオブジェクト内の、位置 *pos* にあるオ

ブジェクトを返します。*pos* が負であるか範囲を超えている場合、NULL を返して IndexError 例外をセットします。

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)

戻り値: **借用参照。***PyTuple_GetItem()* に似ていますが、引数に対するエラーチェックを行いません。

PyObject ***PyTuple_GetSlice**(*PyObject* *p, *Py_ssize_t* low, *Py_ssize_t* high)

戻り値: **新しい参照。次に属します**: 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)

次に属します: Stable ABI. *p* の指すタプルオブジェクト内の、位置 *pos* にあるオブジェクトへの参照を入れます。成功すれば 0 を返します。*pos* が範囲を超えている場合、-1 を返して IndexError 例外をセットします。

> **注釈**
>
> この関数は *o* への参照を ”盗み取り” ます。また、変更先のインデクスにすでに別の要素が入っている場合、その要素に対する参照を放棄します。

void **PyTuple_SET_ITEM**(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

PyTuple_SetItem() に似ていますが、エラーチェックを行わず、新たなタプルに値を入れるとき **以外には使ってはなりません** 。

> **注釈**
>
> この関数は *o* への参照を ”盗み取り” ます。また、*PyTuple_SetItem()* と違って、要素の置き換えが生じても置き換えられるオブジェクトへの参照を放棄 **しません** ; その結果、タプル中の位置 *pos* で参照されていたオブジェクトがメモリリークを引き起こします。

int **_PyTuple_Resize**(*PyObject* **p, *Py_ssize_t* newsize)

タプルをリサイズする際に使えます。*newsize* はタプルの新たな長さです。タプルは変更不能なオブジェクト **ということになっている** ので、この関数はこのオブジェクトに対してただ一つしか参照がない時以外には使ってはなりません。タプルがコード中の他の部分ですでに参照されている場合には、この関数を **使ってはなりません** 。タプルは常に指定サイズの末尾まで伸縮します。成功した場合には 0 を返します。クライアントコードは、*p の値が呼び出し前と同じになると期待してはなりません。*p

が置き換えられた場合、オリジナルの `*p` は破壊されます。失敗すると `-1` を返し、`*p` を `NULL` に設定して、`MemoryError` または `SystemError` を送出します。

8.3.5 Struct Sequence オブジェクト

struct sequence オブジェクトは `namedtuple()` オブジェクトと等価な C オブジェクトです。つまり、その要素に属性を通してアクセスすることができるシーケンスです。struct sequence を生成するには、まず特定の struct sequence 型を生成しなければなりません。

PyTypeObject ***PyStructSequence_NewType**(*PyStructSequence_Desc* *desc)

戻り値: **新しい参照。次に属します**: Stable ABI. 後述の *desc* 中のデータから新しい struct sequence 型を生成します。返される型のインスタンスは *PyStructSequence_New()* で生成できます。

Return `NULL` with an exception set on failure.

void **PyStructSequence_InitType**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

struct sequence 型である *type* を *desc* をもとにその場で初期化します。

int **PyStructSequence_InitType2**(*PyTypeObject* *type, *PyStructSequence_Desc* *desc)

Like *PyStructSequence_InitType()*, but returns `0` on success and `-1` with an exception set on failure.

Added in version 3.4.

type **PyStructSequence_Desc**

次に属します: Stable ABI *(**すべてのメンバーを含む**)*. 生成する struct sequence 型のメタデータを保持します。

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**

次に属します: Stable ABI *(**すべてのメンバーを含む**)*. 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

次に属します: Stable ABI *(バージョン 3.11 より)*. フィールド名を名前がないまますするための特殊な値。

バージョン 3.9 で変更: 型が char * から変更されました。

PyObject *PyStructSequence_New(*PyTypeObject* *type)

戻り値: 新しい参照。次に属します: Stable ABI. *type* のインスタンスを生成します。*type* は *PyStructSequence_NewType()* によって事前に生成していなければなりません。

Return NULL with an exception set on failure.

PyObject *PyStructSequence_GetItem(*PyObject* *p, *Py_ssize_t* pos)

戻り値: 借用参照。次に属します: Stable ABI. Return the object at position *pos* in the struct sequence pointed to by *p*. No bounds checking is performed.

PyObject *PyStructSequence_GET_ITEM(*PyObject* *p, *Py_ssize_t* pos)

戻り値: 借用参照。 Macro equivalent of *PyStructSequence_GetItem()*.

void PyStructSequence_SetItem(*PyObject* *p, *Py_ssize_t* pos, *PyObject* *o)

次に属します: Stable ABI. struct sequence *p* の *pos* の位置にあるフィールドに値 *o* を設定します。*PyTuple_SET_ITEM()* のように、生成したてのインスタンスに対してのみ使用すべきです。

> **注釈**
>
> この関数は *o* への参照を ”盗み取り” ます。

void PyStructSequence_SET_ITEM(*PyObject* *p, *Py_ssize_t* *pos, *PyObject* *o)

Similar to *PyStructSequence_SetItem()*, but implemented as a static inlined function.

> **注釈**
>
> この関数は *o* への参照を ”盗み取り” ます。

8.3.6 リストオブジェクト

type **PyListObject**

この *PyObject* のサブタイプは Python のリストオブジェクトを表現します。

PyTypeObject **PyList_Type**

次に属します: Stable ABI. この *PyTypeObject* のインスタンスは Python のリスト型を表現します。これは Python レイヤにおける `list` と同じオブジェクトです。

int **PyList_Check**(*PyObject* *p)

p がリストオブジェクトかリスト型のサブタイプのインスタンスである場合に真を返します。この関数は常に成功します。

int **PyList_CheckExact**(*PyObject* *p)

p がリストオブジェクトだがリスト型のサブタイプのインスタンスでない場合に真を返します。この関数は常に成功します。

PyObject ***PyList_New**(*Py_ssize_t* len)

戻り値: **新しい参照。次に属します**: Stable ABI. サイズが *len* 新たなリストオブジェクトを返します。失敗すると `NULL` を返します。

> **注釈**
>
> 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()*.

Py_ssize_t **PyList_Size**(*PyObject* *list)

次に属します: Stable ABI. リストオブジェクト *list* の長さを返します; リストオブジェクトにおける `len(list)` と同じです。

Py_ssize_t **PyList_GET_SIZE**(*PyObject* *list)

PyList_Size() に似ていますが、エラーチェックを行いません。

PyObject ***PyList_GetItem**(*PyObject* *list, *Py_ssize_t* index)

戻り値: **借用参照。次に属します**: Stable ABI. 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.

PyObject ***PyList_GET_ITEM**(*PyObject* *list, *Py_ssize_t* i)

戻り値: **借用参照。***PyList_GetItem()* に似ていますが、エラーチェックを行いません。

int **PyList_SetItem**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

次に属します: Stable ABI. リストオブジェクト内の位置 *index* に、オブジェクト *item* を挿入します。成功した場合には `0` を返します。*index* が範囲を越えている場合、`-1` を返して `IndexError` をセットします。

> **注釈**
>
> この関数は *item* への参照を ” 盗み取り” ます。また、変更先のインデクスにすでに別の要素が入っている場合、その要素に対する参照を放棄します。

void **PyList_SET_ITEM**(*PyObject* *list, *Py_ssize_t* i, *PyObject* *o)

PyList_SetItem() をマクロによる実装で、エラーチェックを行いません。このマクロは、新たなリストのまだ要素を入れたことのない位置に要素を入れるときにのみ使います。

> **注釈**
>
> このマクロは *item* への参照を ” 盗み取り” ます。また、*PyList_SetItem()* と違って、要素の置き換えが生じても置き換えられるオブジェクトへの参照を放棄 **しません** ; その結果、*list* 中の位置 *i* で参照されていたオブジェクトがメモリリークを引き起こします。

int **PyList_Insert**(*PyObject* *list, *Py_ssize_t* index, *PyObject* *item)

次に属します: Stable ABI. 要素 *item* をリスト *list* のインデックス *index* の前に挿入します。成功すると `0` を返します。失敗すると `-1` を返し、例外をセットします。`list.insert(index, item)` に類似した機能です。

int **PyList_Append**(*PyObject* *list, *PyObject* *item)

次に属します: Stable ABI. オブジェクト *item* を *list* の末尾に追加します。成功すると `0` を返します; 失敗すると `-1` を返し、例外をセットします。`list.append(item)` に類似した機能です。

PyObject ***PyList_GetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high)

戻り値: **新しい参照。次に属します**: Stable ABI. *list* 内の、*low* から *high* **までの** オブジェクトからなるリストを返します。失敗すると `NULL` を返し、例外をセットします。`list[low:high]` に類似した機能です。ただし、リストの末尾からのインデックスはサポートされていません。

int **PyList_SetSlice**(*PyObject* *list, *Py_ssize_t* low, *Py_ssize_t* high, *PyObject* *itemlist)

次に属します: Stable ABI. *low* から *high* までの *list* のスライスを、*itemlist* の内容にします。`list[low:high] = itemlist` と類似の機能です。*itemlist* は `NULL` でもよく、空リストの代入 (指定スライスの削除) になります。成功した場合には `0` を、失敗した場合には `-1` を返します。ただし、リストの末尾からのインデックスはサポートされていません。

int **PyList_Sort**(*PyObject* *list)

次に属します: Stable ABI. *list* の内容をインプレースでソートします。成功した場合には `0` を、失敗した場合には `-1` を返します。`list.sort()` と同じです。

int **PyList_Reverse**(*PyObject* *list)

次に属します: Stable ABI. *list* の要素をインプレースで反転します。成功した場合には `0` を、失敗した場合には `-1` を返します。`list.reverse()` と同じです。

PyObject *PyList_AsTuple(*PyObject* *list)

戻り値: **新しい参照。次に属します**: Stable ABI. *list* の内容が入った新たなタプルオブジェクトを返します; `tuple(list)` と同じです。

8.4 Container オブジェクト

8.4.1 辞書オブジェクト (dictionary object)

type PyDictObject

この *PyObject* のサブタイプは Python の辞書オブジェクトを表現します。

PyTypeObject PyDict_Type

次に属します: Stable ABI. この *PyTypeObject* のインスタンスは Python の辞書を表現します。このオブジェクトは、Python レイヤにおける `dict` と同じオブジェクトです。

int PyDict_Check(*PyObject* *p)

p が辞書オブジェクトか辞書型のサブタイプのインスタンスである場合に真を返します。この関数は常に成功します。

int PyDict_CheckExact(*PyObject* *p)

p が辞書オブジェクトだが辞書型のサブタイプのインスタンスでない場合に真を返します。この関数は常に成功します。

PyObject *PyDict_New()

戻り値: **新しい参照。次に属します**: Stable ABI. 空の新たな辞書を返します。失敗すると `NULL` を返します。

PyObject *PyDictProxy_New(*PyObject* *mapping)

戻り値: **新しい参照。次に属します**: Stable ABI. あるマップ型オブジェクトに対して、読み出し専用に制限された `types.MappingProxyType` オブジェクトを返します。通常、この関数は動的でないクラス型 (non-dynamic class type) のクラス辞書が変更されないようにビューを作成するために使われます。

void PyDict_Clear(*PyObject* *p)

次に属します: Stable ABI. 現在辞書に入っている全てのキーと値のペアを除去して空にします。

int PyDict_Contains(*PyObject* *p, *PyObject* *key)

次に属します: Stable ABI. 辞書 *p* に *key* が入っているか判定します。*p* の要素が *key* に一致した場合は `1` を返し、それ以外の場合には `0` を返します。エラーの場合 `-1` を返します。この関数は Python の式 `key in p` と等価です。

PyObject *PyDict_Copy(*PyObject* *p)

戻り値: **新しい参照。次に属します**: Stable ABI. *p* と同じキーと値のペアが入った新たな辞書を返します。

int **PyDict_SetItem**(*PyObject* *p, *PyObject* *key, *PyObject* *val)

次に属します: Stable ABI. 辞書 *p* に、*key* をキーとして値 *val* を挿入します。*key* はハッシュ可能 (*hashable*) でなければなりません。ハッシュ可能でない場合、`TypeError` を送出します。成功した場合には `0` を、失敗した場合には `-1` を返します。この関数は *val* への参照を盗み取り **ません**。

int **PyDict_SetItemString**(*PyObject* *p, const char *key, *PyObject* *val)

次に属します: 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)

次に属します: Stable ABI. 辞書 *p* から *key* をキーとするエントリを除去します。*key* は **ハッシュ可能** でなければなりません; ハッシュ可能でない場合、`TypeError` を送出します。*key* が辞書になければ、`KeyError` を送出します。成功した場合には `0` を、失敗した場合には `-1` を返します。

int **PyDict_DelItemString**(*PyObject* *p, const char *key)

次に属します: 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**.

PyObject ***PyDict_GetItem**(*PyObject* *p, *PyObject* *key)

戻り値: 借用参照。次に属します: Stable ABI. Return the object from dictionary *p* which has a key *key*. Return `NULL` if the key *key* is not present, but *without* setting an exception.

注釈

Exceptions that occur while this calls `__hash__()` and `__eq__()` methods are silently ignored. Prefer the *PyDict_GetItemWithError()* function instead.

バージョン 3.10 で変更: Calling this API without *GIL* held had been allowed for historical reason. It is no longer allowed.

PyObject ***PyDict_GetItemWithError**(*PyObject* *p, *PyObject* *key)

戻り値: 借用参照。次に属します: Stable ABI. *PyDict_GetItem()* の変種で例外を隠しません。例外が発生した場合は、例外をセット **した上で** `NULL` を返します。キーが存在しなかった場合は、例外をセット **せずに** `NULL` を返します。

PyObject ***PyDict_GetItemString**(*PyObject* *p, const char *key)

戻り値: 借用参照。次に属します: 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**.

注釈

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.

PyObject *PyDict_SetDefault(*PyObject* *p, *PyObject* *key, *PyObject* *defaultobj)

戻り値: **借用参照。** これは Python レベルの `dict.setdefault()` と同じです。もしあれば、辞書 *p* から *key* に対応する値を返します。キーが辞書になければ、値 *defaultobj* を挿入し *defaultobj* を返します。この関数は、*key* のハッシュ値を検索と挿入ごとに別々に評価するのではなく、一度だけしか評価しません。

Added in version 3.4.

PyObject *PyDict_Items(*PyObject* *p)

戻り値: **新しい参照。次に属します**: Stable ABI. 辞書内の全ての要素対が入った *PyListObject* を返します。

PyObject *PyDict_Keys(*PyObject* *p)

戻り値: **新しい参照。次に属します**: Stable ABI. 辞書内の全てのキーが入った *PyListObject* を返します。

PyObject *PyDict_Values(*PyObject* *p)

戻り値: **新しい参照。次に属します**: Stable ABI. 辞書 *p* 内の全ての値が入った *PyListObject* を返します。

Py_ssize_t PyDict_Size(*PyObject* *p)

次に属します: Stable ABI. 辞書内の要素の数を返します。辞書に対して `len(p)` を実行するのと同じです。

int PyDict_Next(*PyObject* *p, *Py_ssize_t* *ppos, *PyObject* **pkey, *PyObject* **pvalue)

次に属します: Stable ABI. 辞書 *p* 内の全てのキー/値のペアにわたる反復処理を行います。*ppos* が参照している *Py_ssize_t* 型は、この関数で反復処理を開始する際に、最初に関数を呼び出すよりも前に `0` に初期化しておかなければなりません。この関数は辞書内の各ペアを取り上げるごとに真を返し、全てのペアを取り上げたことが分かると偽を返します。パラメーター *pkey* および *pvalue* には、それぞれ辞書の各々のキーと値が埋められた *PyObject** 変数を指すポインタか、または `NULL` が入ります。この関数から返される参照はすべて借用参照になります。反復処理中に *ppos* を変更してはなりません。この値は内部的な辞書構造体のオフセットを表現しており、構造体はスパースなので、オフセットの値に一貫性がないためです。

例えば:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}
```

反復処理中に辞書 *p* を変更してはなりません。辞書を反復処理する際に、キーに対応する値を変更しても大丈夫になりましたが、キーの集合を変更しないことが前提です。以下に例を示します:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    long i = PyLong_AsLong(value);
    if (i == -1 && PyErr_Occurred()) {
        return -1;
    }
    PyObject *o = PyLong_FromLong(i + 1);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
    Py_DECREF(o);
}
```

int **PyDict_Merge**(*PyObject* *a, *PyObject* *b, int override)

次に属します: Stable ABI. マップ型オブジェクト *b* の全ての要素にわたって、反復的にキー/値のペアを辞書 *a* に追加します。*b* は辞書か、*PyMapping_Keys()* または *PyObject_GetItem()* をサポートする何らかのオブジェクトにできます。*override* が真ならば、*a* のキーと一致するキーが *b* にある際に、既存のペアを置き換えます。それ以外の場合は、*b* のキーに一致するキーが *a* にないときのみ追加を行います。成功した場合には `0` を返し、例外が送出された場合には `-1` を返します。

int **PyDict_Update**(*PyObject* *a, *PyObject* *b)

次に属します: Stable ABI. C で表せば `PyDict_Merge(a, b, 1)` と同じで、また Python の `a.update(b)` と似ていますが、*PyDict_Update()* は第二引数が "keys" 属性を持たない場合にキー/値ペアのシーケンスを反復することはありません。成功した場合には `0` を返し、例外が送出された場合には `-1` を返します。

int **PyDict_MergeFromSeq2**(*PyObject* *a, *PyObject* *seq2, int override)

次に属します: Stable ABI. *seq2* 内のキー/値ペアを使って、辞書 *a* の内容を更新したり統合したりします。*seq2* は、キー/値のペアとみなせる長さ 2 の反復可能オブジェクト (iterable object) を生成する反復可能オブジェクトでなければなりません。重複するキーが存在する場合、*override* が真ならば先に出現したキーを使い、そうでない場合は後に出現したキーを使います。成功した場合には `0` を返し、例外が送出された場合には `-1` を返します。(戻り値以外は) 等価な Python コードを書くと、以下のようになります:

```
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.

Added in 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.)

Added in 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.

Added in 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.

Added in 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`.

Added in 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.

Added in version 3.12.

8.4.2 Set オブジェクト

このセクションでは `set` と `frozenset` の公開 API について詳しく述べます。以降で説明していない機能は、抽象オブジェクトプロトコル (*PyObject_CallMethod()*, *PyObject_RichCompareBool()*, *PyObject_Hash()*, *PyObject_Repr()*, *PyObject_IsTrue()*, *PyObject_Print()*, *PyObject_GetIter()* を含む) か抽象数値プロトコル (*PyNumber_And()*, *PyNumber_Subtract()*, *PyNumber_Or()*, *PyNumber_Xor()*, *PyNumber_InPlaceAnd()*, *PyNumber_InPlaceSubtract()*, *PyNumber_InPlaceOr()*, *PyNumber_InPlaceXor()* を含む) を使って利用できます。

type **PySetObject**

この *PyObject* を継承した型は、`set` と `frozenset` 両方の内部データを保存するのに用いられます。*PyDictObject* と同じように、小さい集合 (set) に対しては (タプルのように) 固定サイズであり、そうでない集合に対しては (リストと同じように) 可変長のメモリブロックを用います。この構造体のどのフィールドも、公開されていると考えるべきではなく、変更される可能性があります。すべてのアクセスは、構造体の中の値を直接操作するのではなく、ドキュメントされた API を用いて行うべきです。

PyTypeObject **PySet_Type**

次に属します: Stable ABI. この *PyTypeObject* のインスタンスは、Python の `set` 型を表します。

PyTypeObject **PyFrozenSet_Type**

次に属します: Stable ABI. この *PyTypeObject* のインスタンスは、Python の `frozenset` 型を表します。

以降の型チェックマクロはすべての Python オブジェクトに対するポインタに対して動作します。同様に、コンストラクタはすべてのイテレート可能な Python オブジェクトに対して動作します。

int **PySet_Check**(*PyObject* *p)

p が `set` かそのサブタイプのオブジェクトであるときに true を返します。この関数は常に成功します。

int **PyFrozenSet_Check**(*PyObject* *p)

p が `frozenset` かそのサブタイプのオブジェクトであるときに true を返します。この関数は常に成功します。

int **PyAnySet_Check**(*PyObject* *p)

p が `set` か `frozenset` 、あるいはそのサブタイプのオブジェクトであれば、true を返します。この関数は常に成功します。

int **PySet_CheckExact**(*PyObject* *p)

p が `set` オブジェクトだがサブタイプのインスタンスでない場合に真を返します。この関数は常に成功します。

Added in version 3.10.

int **PyAnySet_CheckExact**(*PyObject* *p)

p が `set` か `frozenset` のどちらかのオブジェクトであるときに true を返します。サブタイプのオブジェクトは含みません。この関数は常に成功します。

int **PyFrozenSet_CheckExact**(*PyObject* *p)

p が `frozenset` オブジェクトだがサブタイプのインスタンスでない場合に真を返します。この関数は常に成功します。

PyObject ***PySet_New**(*PyObject* *iterable)

戻り値: 新しい参照。次に属します: Stable ABI. *iterable* が返すオブジェクトを含む新しい `set` を返します。*iterable* が `NULL` のときは、空の set を返します。成功したら新しい set を、失敗したら `NULL` を返します。*iterable* がイテレート可能でない場合は、`TypeError` を送出します。このコンストラクタは set をコピーするときにも使えます (`c=set(s)`)。

PyObject ***PyFrozenSet_New**(*PyObject* *iterable)

戻り値: 新しい参照。次に属します: Stable ABI. *iterable* が返すオブジェクトを含む新しい `frozenset` を返します。*iterable* が `NULL` のときは、空の frozenset を返します。成功時には新しい set を、失敗時には `NULL` を返します。*iterable* がイテレート可能でない場合は、`TypeError` を送出します。

以降の関数やマクロは、`set` と `frozenset` とそのサブタイプのインスタンスに対して利用できます。

Py_ssize_t **PySet_Size**(*PyObject* *anyset)

次に属します: 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)

エラーチェックを行わない、*PySet_Size()* のマクロ形式。

int **PySet_Contains**(*PyObject* *anyset, *PyObject* *key)

次に属します: 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)

次に属します: Stable ABI. `set` のインスタンスに *key* を追加します。`frozenset` に対しても動作します (*PyTuple_SetItem()* のように、他のコードに見える前の新しい frozenset の値を埋めるために使用できます)。成功したら `0` を、失敗したら `-1` を返します。*key* がハッシュ可能でない場合は、`TypeError` を送出します。*set* を大きくする余裕がない場合は、`MemoryError` を送出します。*set* が `set` かそのサブタイプのインスタンスでない場合は、`SystemError` を送出します。

以降の関数は、`set` とそのサブタイプに対して利用可能です。`frozenset` とそのサブタイプには利用できません。

int **PySet_Discard**(*PyObject* *set, *PyObject* *key)

次に属します: 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)

戻り値: **新しい参照。次に属します**: Stable ABI. *set* の中の要素のどれかに対する新しい参照を返し、そのオブジェクトを *set* から削除します。失敗したら `NULL` を返します。set が空の場合には `KeyError` を送出します。*set* が `set` とそのサブタイプのインスタンスでない場合は、`SystemError` を送出します。

int **PySet_Clear**(*PyObject* *set)

次に属します: 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.

8.5 Function オブジェクト

8.5.1 Function オブジェクト

Function オブジェクト固有の関数はわずかです。

type **PyFunctionObject**

関数に使われる C の構造体。

PyTypeObject **PyFunction_Type**

PyTypeObject 型のインスタンスで、Python の関数型を表します。これは Python プログラムに `types.FunctionType` として公開されています。

int **PyFunction_Check**(*PyObject* *o)

o が関数オブジェクト (*PyFunction_Type* 型である) 場合に真を返します。パラメータは `NULL` にできません。この関数は常に成功します。

PyObject *PyFunction_New(*PyObject* *code, *PyObject* *globals)

戻り値: 新しい参照。 コードオブジェクト *code* に関連付けられた新しい関数オブジェクトを返します。*globals* はこの関数からアクセスできるグローバル変数の辞書でなければなりません。

関数のドキュメント文字列と名前はコードオブジェクトから取得されます。`__module__` は *globals* から取得されます。引数のデフォルト値やアノテーション、クロージャは `NULL` に設定されます。`__qualname__` はコードオブジェクトの `co_qualname` フィールドと同じ値に設定されます。

PyObject *PyFunction_NewWithQualName(*PyObject* *code, *PyObject* *globals, *PyObject* *qualname)

戻り値: 新しい参照。 *PyFunction_New()* に似ていますが、関数オブジェクトの `__qualname__` 属性に値をセットできます。*qualname* はユニコードオブジェクトか `NULL` でなくてはなりません。`NULL` だった場合、`__qualname__` 属性にはコードオブジェクトの `co_qualname` フィールドと同じ値がセットされます。

Added in version 3.3.

PyObject *PyFunction_GetCode(*PyObject* *op)

戻り値: 借用参照。 関数オブジェクト *op* に関連付けられたコードオブジェクトを返します。

PyObject *PyFunction_GetGlobals(*PyObject* *op)

戻り値: 借用参照。 関数オブジェクト *op* に関連付けられた globals 辞書を返します。

PyObject *PyFunction_GetModule(*PyObject* *op)

戻り値: 借用参照。 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)

戻り値: 借用参照。 関数オブジェクト *op* の引数のデフォルト値を返します。引数のタプルか `NULL` になります。

int PyFunction_SetDefaults(*PyObject* *op, *PyObject* *defaults)

関数オブジェクト *op* の引数のデフォルト値を設定します。*defaults* は `Py_None` かタプルでなければいけません。

失敗した時は、`SystemError` を発生させ、`-1` を返します。

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!

Added in version 3.12.

PyObject *PyFunction_GetClosure(*PyObject* *op)

戻り値: **借用参照。**関数オブジェクト *op* に設定されたクロージャを返します。NULL か cell オブジェクトのタプルです。

int PyFunction_SetClosure(*PyObject* *op, *PyObject* *closure)

関数オブジェクト *op* にクロージャを設定します。*closure* は、Py_None もしくは cell オブジェクトのタプルでなければなりません。

失敗した時は、SystemError を発生させ、-1 を返します。

PyObject *PyFunction_GetAnnotations(*PyObject* *op)

戻り値: **借用参照。**関数オブジェクト *op* のアノテーションを返します。返り値は修正可能な辞書か NULL になります。

int PyFunction_SetAnnotations(*PyObject* *op, *PyObject* *annotations)

関数オブジェクト *op* のアノテーションを設定します。*annotations* は辞書か、Py_None でなければなりません。

失敗した時は、SystemError を発生させ、-1 を返します。

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.

Added in 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.)

Added in 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

Added in 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.

Added in version 3.12.

8.5.2 インスタンスメソッドオブジェクト (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**

PyTypeObject のインスタンスは Python のインスタンスメソッドの型を表現します。これは Python のプログラムには公開されません。

int **PyInstanceMethod_Check**(*PyObject* *o)

o がインスタンスメソッドオブジェクト (*PyInstanceMethod_Type* 型である) 場合に真を返します。パラメータは `NULL` にできません。この関数は常に成功します。

PyObject ***PyInstanceMethod_New**(*PyObject* *func)

戻り値: 新しい参照。任意の呼び出し可能オブジェクト *func* を使った新たなインスタンスメソッドオ

ブジェクトを返します。*func* はインスタンスメソッドが呼び出されたときに呼び出される関数です。

PyObject *PyInstanceMethod_Function(*PyObject* *im)

戻り値: **借用参照。**インスタンスメソッド *im* に関連付けられた関数オブジェクトを返します。

PyObject *PyInstanceMethod_GET_FUNCTION(*PyObject* *im)

戻り値: **借用参照。***PyInstanceMethod_Function()* のマクロ版で、エラーチェックを行いません。

8.5.3 メソッドオブジェクト

メソッドは関数オブジェクトに束縛されています。メソッドは常にあるユーザー定義のクラスに束縛されているのです。束縛されていないメソッド (クラスオブジェクトに束縛されたメソッド) は利用することができません。

PyTypeObject PyMethod_Type

この *PyTypeObject* のインスタンスは Python のメソッド型を表現します。このオブジェクトは、types.MethodType として Python プログラムに公開されています。

int PyMethod_Check(*PyObject* *o)

o がメソッドオブジェクト (*PyMethod_Type* 型である) 場合に真を返します。パラメータは NULL にできません。この関数は常に成功します。

PyObject *PyMethod_New(*PyObject* *func, *PyObject* *self)

戻り値: **新しい参照。**任意の呼び出し可能オブジェクト *func* とメソッドが束縛されるべきインスタンス *self* を使った新たなメソッドオブジェクトを返します。関数 *func* は、メソッドが呼び出された時に呼び出されるオブジェクトです。*self* は NULL にできません。

PyObject *PyMethod_Function(*PyObject* *meth)

戻り値: **借用参照。**メソッド *meth* に関連付けられている関数オブジェクトを返します。

PyObject *PyMethod_GET_FUNCTION(*PyObject* *meth)

戻り値: **借用参照。***PyMethod_Function()* のマクロ版で、エラーチェックを行いません。

PyObject *PyMethod_Self(*PyObject* *meth)

戻り値: **借用参照。**メソッド *meth* に関連付けられたインスタンスを返します。

PyObject *PyMethod_GET_SELF(*PyObject* *meth)

戻り値: **借用参照。***PyMethod_Self()* のマクロ版で、エラーチェックを行いません。

8.5.4 セルオブジェクト (cell object)

" セル (cell)" オブジェクトは、複数のスコープから参照される変数群を実装するために使われます。セルは各変数について作成され、各々の値を記憶します; この値を参照する各スタックフレームにおけるローカル変数には、そのスタックフレームの外側で同じ値を参照しているセルに対する参照が入ります。セルで表現された値にアクセスすると、セルオブジェクト自体の代わりにセル内の値が使われます。このセルオブジェクトを使った間接参照 (dereference) は、インタプリタによって生成されたバイトコード内でサポートされている必

要があります; セルオブジェクトにアクセスした際に、自動的に間接参照は起こりません。上記以外の状況では、セルオブジェクトは役に立たないはずです。

type PyCellObject

セルオブジェクトに使われる C 構造体です。

PyTypeObject PyCell_Type

セルオブジェクトに対応する型オブジェクトです。

int PyCell_Check(*PyObject* *ob)

ob がセルオブジェクトの場合に真を返します; *ob* は NULL であってはなりません。この関数は常に成功します。

PyObject *PyCell_New(*PyObject* *ob)

戻り値: 新しい参照。値 *ob* の入った新たなセルオブジェクトを生成して返します。引数を NULL にしてもかまいません。

PyObject *PyCell_Get(*PyObject* *cell)

戻り値: 新しい参照。 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)

戻り値: 借用参照。 *cell* の内容を返しますが、*cell* が非 NULL かつセルオブジェクトであるかどうかはチェックしません。

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)

セルオブジェクト *cell* の値を *value* に設定します。参照カウントに対する変更はなく、安全のためのチェックは何も行いません。*cell* は非 NULL でなければならず、かつセルオブジェクトでなければなりません。

8.5.5 コードオブジェクト

コードオブジェクト (Code objects) は CPython 実装の低レベルな詳細部分です。各オブジェクトは関数に束縛されていない実行可能コードの塊を表現しています。

type PyCodeObject

コードオブジェクトを表現するために利用される C 構造体。この型のフィールドは何時でも変更され得ます。

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 variables in a code object.

int **PyCode_GetFirstFree**(*PyCodeObject* *co)

Return the position of the first free variable in a code object.

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)

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

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.

バージョン 3.11 で変更: Added `qualname` and `exceptiontable` parameters.

バージョン 3.12 で変更: 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)

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

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.

Added in version 3.8: as `PyCode_NewWithPosOnlyArgs`

バージョン 3.11 で変更: Added `qualname` and `exceptiontable` parameters.

バージョン 3.12 で変更: 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)

戻り値: 新しい参照。 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.

Added in 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.

Added in 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.

Added in 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.

Added in 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 variables. On error, NULL is returned and an exception is raised.

Added in 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.

Added in 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.)

Added in version 3.12.

type **PyCodeEvent**

Enumeration of possible code object watcher events: - PY_CODE_EVENT_CREATE - PY_CODE_EVENT_DESTROY

Added in 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.

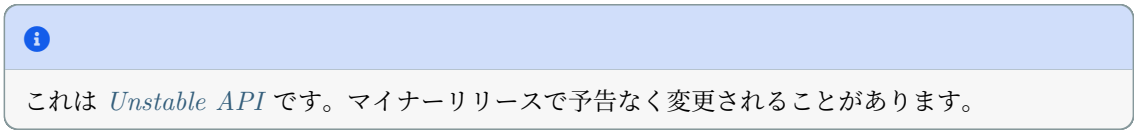
Added in version 3.12.

8.5.6 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)



これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

Return a new an opaque index value used to adding data to code objects.

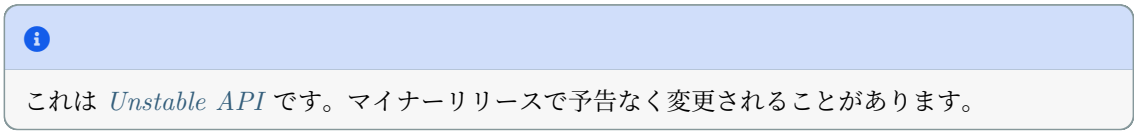
You generally call this function once (per interpreter) and use the result with `PyCode_GetExtra` and `PyCode_SetExtra` to manipulate data on individual code objects.

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*.

Added in version 3.6: as `_PyEval_RequestCodeExtraIndex`

バージョン 3.12 で変更: 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)



これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

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.

Added in version 3.6: as `_PyCode_GetExtra`

バージョン 3.12 で変更: 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)

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

Set the extra data stored under the given index to *extra*. Return 0 on success. Set an exception and return -1 on failure.

Added in version 3.6: as `_PyCode_SetExtra`

バージョン 3.12 で変更: Renamed to `PyUnstable_Code_SetExtra`. The old private name is deprecated, but will be available until the API changes.

8.6 その他のオブジェクト

8.6.1 ファイルオブジェクト

これらの API は、Python 2 の組み込みのファイルオブジェクトの C API を最低限エミュレートするためのものです。それらは、標準 C ライブラリでサポートされているバッファ付き I/O (`FILE*`) に頼るために使われます。Python 3 では、ファイルとストリームは新しい `io` モジュールを使用され、そこに OS の低レベルなバッファ付き I/O の上にいくつかの層が定義されています。下で解説されている関数は、それらの新しい API の便利な C ラッパーであり、インタプリタでの内部的なエラー通知に向いています; サードパーティーのコードは代わりに `io` の API を使うことが推奨されます。

PyObject ***PyFile_FromFd**(int fd, const char *name, const char *mode, int buffering, const char *encoding, const char *errors, const char *newline, int closefd)

戻り値: 新しい参照。次に属します: Stable ABI. 既に開かれているファイル *fd* のファイルディスクリプタから Python のファイルオブジェクトを作成します。引数 *name* 、*encoding* 、*errors* 、*newline* には、デフォルトの値として `NULL` が使えます。*buffering* には *-1* を指定してデフォルトの値を使うことができます。*name* は無視されるのですが、後方互換性のために残されています。失敗すると `NULL` を返します。より包括的な引数の解説は、`io.open()` 関数のドキュメントを参照してください。

警告

Python ストリームは自身のバッファリング層を持つため、ファイル記述子の OS レベルのバッファリングと併用すると、様々な問題 (予期せぬデータ順) などを引き起こします。

バージョン 3.2 で変更: *name* 属性の無視。

int **PyObject_AsFileDescriptor**(*PyObject* *p)

次に属します: Stable ABI. *p* に関連づけられる ファイルディスクリプタを `int` として返します。オブジェクトが整数なら、その値を返します。整数でない場合、オブジェクトに `fileno()` メソッドがあれば呼び出します; このメソッドの返り値は、ファイル記述子の値として返される整数でなければなりません。失敗すると例外を設定して `-1` を返します。

PyObject ***PyFile_GetLine**(*PyObject* *p, int n)

戻り値: 新しい参照。次に属します: Stable ABI. `p.readline([n])` と同じで、この関数はオブジェクト *p* の各行を読み出します。*p* はファイルオブジェクトか、`readline()` メソッドを持つ何らかのオブジェクトでかまいません。*n* が `0` の場合、行の長さに関係なく正確に 1 行だけ読み出します。*n* が `0` より大きければ、*n* バイト以上のデータは読み出しません; 従って、行の一部だけが返される場合があります。どちらの場合でも、読み出し後すぐにファイルの終端に到達した場合には空文字列を 返します。*n* が `0` より小さければ、長さに関わらず 1 行だけを 読み出しますが、すぐにファイルの終端に到達した場合には `EOFError` を送出します。

int **PyFile_SetOpenCodeHook**(*Py_OpenCodeHookFunction* handler)

`io.open_code()` の通常の振る舞いを上書きして、そのパラメーターを提供されたハンドラで渡します。

ハンドラーは次の型の関数です。

typedef *PyObject* *(***Py_OpenCodeHookFunction**)(*PyObject**, void*)

PyObject `*(*)(PyObject *path, void *userData)` と等価で、*path* は *PyUnicodeObject* であることが保証されています。

userData ポインタはフック関数に渡されます。フック関数は別なランタイムから呼び出されるかもしれないので、このポインタは直接 Python の状態を参照すべきではありません。

このフック関数はインポート中に使われることを意図しているため、モジュールが frozen なモジュールであるか `sys.modules` にある利用可能なモジュールであることが分かっている場合を除いては、フック関数の実行中に新しいモジュールをインポートするのは避けてください。

いったんフック関数が設定されたら、削除や置き換えもできず、後からの *PyFile_SetOpenCodeHook()* の呼び出しは失敗します。この関数が失敗したときは、インタープリタが初期化されていた場合、-1 を返して例外をセットします。

この関数は *Py_Initialize()* より前に呼び出しても安全です。

引数無しで 監査イベント `setopencodehook` を送出します。

Added in version 3.8.

int **PyFile_WriteObject**(*PyObject* *obj, *PyObject* *p, int flags)

次に属します: Stable ABI. オブジェクト *obj* をファイルオブジェクト *p* に書き込みます。*flags* がサポートするフラグは *Py_PRINT_RAW* だけです; このフラグを指定すると、オブジェクトに `repr()` ではなく `str()` を適用した結果をファイルに書き出します。成功した場合には `0` を返し、失敗すると `-1` を返して適切な例外をセットします。

int **PyFile_WriteString**(const char *s, *PyObject* *p)

次に属します: Stable ABI. 文字列 *s* をファイルオブジェクト *p* に書き出します。成功した場合には 0 を返し、失敗すると -1 を返して適切な例外をセットします。

8.6.2 モジュールオブジェクト (module object)

PyTypeObject **PyModule_Type**

次に属します: Stable ABI. この *PyTypeObject* のインスタンスは Python のモジュールオブジェクト型を表現します。このオブジェクトは、Python プログラムには types.ModuleType として公開されています。

int **PyModule_Check**(*PyObject* *p)

p がモジュールオブジェクトかモジュールオブジェクトのサブタイプであるときに真を返します。この関数は常に成功します。

int **PyModule_CheckExact**(*PyObject* *p)

p がモジュールオブジェクトで、かつ *PyModule_Type* のサブタイプでないときに真を返します。この関数は常に成功します。

PyObject ***PyModule_NewObject**(*PyObject* *name)

戻り値: **新しい参照。次に属します**: Stable ABI *(バージョン 3.7 より)*. 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.

Added in version 3.3.

バージョン 3.4 で変更: __package__ and __loader__ are now set to None.

PyObject ***PyModule_New**(const char *name)

戻り値: **新しい参照。次に属します**: Stable ABI. *PyModule_NewObject()* に似ていますが、name は Unicode オブジェクトではなく UTF-8 でエンコードされた文字列です。

PyObject ***PyModule_GetDict**(*PyObject* *module)

戻り値: **借用参照。次に属します**: Stable ABI. *module* の名前空間を実装する辞書オブジェクトを返します; このオブジェクトは、モジュールオブジェクトの __dict__ 属性と同じものです。*module* がモジュールオブジェクト (もしくはモジュールオブジェクトのサブタイプ) でない場合は、SystemError が送出され NULL が返されます。

拡張モジュールでは、モジュールの __dict__ を直接操作するよりも、PyModule_* および PyObject_* 関数を使う方が推奨されます。

PyObject ***PyModule_GetNameObject**(*PyObject* *module)

戻り値: **新しい参照。次に属します**: Stable ABI *(バージョン 3.7 より)*. 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.

Added in version 3.3.

const char *`PyModule_GetName`(*PyObject* *module)

次に属します: Stable ABI. *PyModule_GetNameObject()* に似ていますが、`'utf-8'` でエンコードされた name を返します。

void *`PyModule_GetState`(*PyObject* *module)

次に属します: Stable ABI. モジュールの "state"(モジュールを生成したタイミングで確保されるメモリブロックへのポインター) か、なければ `NULL` を返します。*PyModuleDef.m_size* を参照してください。

PyModuleDef *`PyModule_GetDef`(*PyObject* *module)

次に属します: Stable ABI. モジュールが作られる元となった *PyModuleDef* 構造体へのポインタを返します。モジュールが定義によって作られていなかった場合は `NULL` を返します。

PyObject *`PyModule_GetFilenameObject`(*PyObject* *module)

戻り値: 新しい参照。次に属します: 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.

Added in version 3.2.

const char *`PyModule_GetFilename`(*PyObject* *module)

次に属します: Stable ABI. *PyModule_GetFilenameObject()* と似ていますが、'utf-8' でエンコードされたファイル名を返します。

バージョン 3.2 で非推奨: *PyModule_GetFilename()* はエンコードできないファイル名に対しては `UnicodeEncodeError` を送出します。これの代わりに *PyModule_GetFilenameObject()* を使用してください。

C モジュールの初期化

通常、モジュールオブジェクトは拡張モジュール (初期化関数をエクスポートしている共有ライブラリ) または組み込まれたモジュール (*PyImport_AppendInittab()* を使って初期化関数が追加されているモジュール) から作られます。詳細については building または extending-with-embedding を見てください。

初期化関数は、モジュール定義のインスタンスを *PyModule_Create()* に渡して出来上がったモジュールオブジェクトを返してもよいですし、もしくは定義構造体そのものを返し" 多段階初期化" を要求しても構いません。

type `PyModuleDef`

次に属します: Stable ABI *(**すべてのメンバーを含む**)*. モジュール定義構造体はモジュールオブジェクトを生成するのに必要なすべての情報を保持します。通常は、それぞれのモジュールごとに静的に初期化されたこの型の変数が 1 つだけ存在します。

PyModuleDef_Base **m_base**

このメンバーは常に PyModuleDef_HEAD_INIT で初期化してください。

const char ***m_name**

新しいモジュールの名前。

const char ***m_doc**

モジュールの docstring。たいてい docstring は *PyDoc_STRVAR* を利用して生成されます。

Py_ssize_t **m_size**

モジュールの状態は、静的なグローバルな領域ではなく *PyModule_GetState()* で取得できるモジュールごとのメモリ領域に保持されていることがあります。これによってモジュールは複数のサブ・インタプリターで安全に使えます。

このメモリ領域は *m_size* に基づいてモジュール作成時に確保され、モジュールオブジェクトが破棄されるときに、*m_free* 関数があればそれが呼ばれた後で解放されます。

m_size に -1 を設定すると、そのモジュールはグローバルな状態を持つためにサブ・インタープリターをサポートしていないということになります。

m_size を非負の値に設定すると、モジュールは再初期化でき、その状態のために必要となる追加のメモリ量を指定できるということになります。非負の m_size は多段階初期化で必要になります。

詳細は **PEP 3121** を参照。

PyMethodDef ***m_methods**

PyMethodDef で定義される、モジュールレベル関数のテーブルへのポインター。関数が存在しない場合は NULL を設定することが可能。

PyModuleDef_Slot ***m_slots**

多段階初期化のためのスロット定義の配列で、{0, NULL} 要素が終端となります。一段階初期化を使うときは、*m_slots* は NULL でなければなりません。

バージョン 3.5 で変更: バージョン 3.5 より前は、このメンバは常に NULL に設定されていて、次のものとして定義されていました:

inquiry **m_reload**

traverseproc **m_traverse**

GC 走査がモジュールオブジェクトを走査する際に呼び出される走査関数。必要ない場合は NULL.

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.

バージョン 3.9 で変更: No longer called before the module state is allocated.

inquiry **m_clear**

GC がこのモジュールオブジェクトをクリアーする時に呼び出されるクリアー関数。必要ない場合は、NULL.

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.

バージョン 3.9 で変更: No longer called before the module state is allocated.

freefunc **m_free**

GC がこのモジュールオブジェクトを解放するときに呼び出される関数。必要ない場合は NULL.

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.

バージョン 3.9 で変更: No longer called before the module state is allocated.

一段階初期化

モジュールの初期化関数が直接モジュールオブジェクトを生成して返す場合があります。これは” 一段階初期化” と呼ばれ、次の 2 つのモジュール生成関数のどちらか 1 つを使います:

PyObject ***PyModule_Create**(*PyModuleDef* *def)

戻り値: **新しい参照。** Create a new module object, given the definition in *def*. This behaves like *PyModule_Create2()* with *module_api_version* set to PYTHON_API_VERSION.

PyObject ***PyModule_Create2**(*PyModuleDef* *def, int module_api_version)

戻り値: **新しい参照。次に属します**: Stable ABI. API バージョンを *module_api_version* として *def* での定義に従って新しいモジュールオブジェクトを生成します。もし指定されたバージョンが実行しているインタープリターのバージョンと異なる場合は、RuntimeWarning を発生させます。

Return NULL with an exception set on error.

> **注釈**
>
> ほとんどの場合、この関数ではなく *PyModule_Create()* を利用するべきです。この関数は、この関数の必要性を理解しているときにだけ利用してください。

モジュールオブジェクトが初期化関数から返される前に、たいていは *PyModule_AddObjectRef()* などの関数を使ってモジュールオブジェクトにメンバを所属させます。

多段階初期化

An alternate way to specify extensions is to request "multi-phase initialization". Extension modules created this way behave more like Python modules: the initialization is split between the *creation phase*, when the module object is created, and the *execution phase*, when it is populated. The distinction is similar to the `__new__()` and `__init__()` methods of classes.

Unlike modules created using single-phase initialization, these modules are not singletons: if the *sys.modules* entry is removed and the module is re-imported, a new module object is created, and the old module is subject to normal garbage collection -- as with Python modules. By default, multiple modules created from the same definition should be independent: changes to one should not affect the others. This means that all state should be specific to the module object (using e.g. using *PyModule_GetState()*), or its contents (such as the module's `__dict__` or individual classes created with *PyType_FromSpec()*).

多段階初期化を使って生成される全てのモジュールは **サブ・インタプリター** をサポートすることが求められます。複数のモジュールが独立していることを保証するのには、たいていはこのサポートをするだけで十分です。

多段階初期化を要求するために、初期化関数 (PyInit_modulename) は空でない *m_slots* を持つ *PyModuleDef* を返します。これを返す前に、`PyModuleDef` インスタンスは次の関数で初期化されなくてはいけません:

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

戻り値: **借用参照。次に属します**: Stable ABI *(**バージョン** 3.5 **より**)*. モジュール定義が型と参照カウントを正しく報告する、適切に初期化された Python オブジェクトであること保証します。

`PyObject*` にキャストされた *def* を返します。エラーが発生した場合 `NULL` を返します。

Added in version 3.5.

モジュール定義の *m_slots* メンバは `PyModuleDef_Slot` 構造体の配列を指さなければなりません:

type **PyModuleDef_Slot**

int **slot**

スロット ID で、以下で説明されている利用可能な値から選ばれます。

void ***value**

スロットの値で、意味はスロット ID に依存します。

Added in version 3.5.

m_slots 配列は ID 0 のスロットで終端されていなければなりません。

利用可能なスロットの型は以下です:

Py_mod_create

モジュールオブジェクト自身を生成するために呼ばれる関数を指定します。このスロットの *value* ポインタは次のシグネチャを持つ関数を指していなくてはいけません:

PyObject *create_module(*PyObject* *spec, *PyModuleDef* *def)

PEP 451 で定義された ModuleSpec インスタンスと、モジュール定義を受け取る関数です。これは新しいモジュールオブジェクトを返すか、エラーを設定して NULL を返すべきです。

この関数は最小限に留めておくべきです。特に任意の Python コードを呼び出すべきではなく、同じモジュールをインポートしようとすると無限ループに陥るでしょう。

複数の Py_mod_create スロットを 1 つのモジュール定義に設定しない方がよいです。

Py_mod_create が設定されていない場合は、インポート機構は *PyModule_New()* を使って通常のモジュールオブジェクトを生成します。モジュールの名前は定義ではなく *spec* から取得され、これによって拡張モジュールが動的にモジュール階層における位置を調整できたり、シンボリックリンクを通して同一のモジュール定義を共有しつつ別の名前でインポートできたりします。

返されるオブジェクトが *PyModule_Type* のインスタンスである必要はありません。インポートに関連する属性の設定と取得ができる限りは、どんな型でも使えます。しかし、PyModuleDef が NULL でない m_traverse, m_clear, m_free 、もしくはゼロでない m_size 、もしくは Py_mod_create 以外のスロットを持つ場合は、PyModule_Type インスタンスのみが返されるでしょう。

Py_mod_exec

モジュールを **実行する** ときに呼ばれる関数を指定します。これは Python モジュールのコードを実行するのと同等です: この関数はたいていはクラスと定数をモジュールにします。この関数のシグネチャは以下です:

int exec_module(*PyObject* *module)

複数の Py_mod_exec スロットが設定されていた場合は、*m_slots* 配列に現れた順に処理されていきます。

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.

Added in version 3.12.

多段階初期化についてより詳しくは **PEP 489** を見てください。

低水準モジュール作成関数

以下の関数は、多段階初期化を使うときに裏側で呼び出されます。例えばモジュールオブジェクトを動的に生成するときに、これらの関数を直接使えます。`PyModule_FromDefAndSpec` および `PyModule_ExecDef` のどちらも、呼び出した後にはモジュールが完全に初期化されていなければなりません。

PyObject ***PyModule_FromDefAndSpec**(*PyModuleDef* *def, *PyObject* *spec)

戻り値: **新しい参照。** Create a new module object, given the definition in *def* and the ModuleSpec *spec*. This behaves like *PyModule_FromDefAndSpec2()* with *module_api_version* set to `PYTHON_API_VERSION`.

Added in version 3.5.

PyObject ***PyModule_FromDefAndSpec2**(*PyModuleDef* *def, *PyObject* *spec, int module_api_version)

戻り値: **新しい参照。** **次に属します**: Stable ABI *(**バージョン** 3.7 **より**)*. API バージョンを *module_api_version* として、*def* と ModuleSpec オブジェクトの *spec* で定義されたとおりに新しいモジュールオブジェクトを生成します。もし指定されたバージョンが実行しているインタープリターのバージョンと異なる場合は、`RuntimeWarning` を発生させます。

Return `NULL` with an exception set on error.

> **注釈**
>
> ほとんどの場合、この関数ではなく *PyModule_FromDefAndSpec()* を利用するべきです。この関数は、この関数の必要性を理解しているときにだけ利用してください。

Added in version 3.5.

int **PyModule_ExecDef**(*PyObject* *module, *PyModuleDef* *def)

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. *def* で与えられた任意の実行スロット (*Py_mod_exec*) を実行します。

Added in version 3.5.

int **PyModule_SetDocString**(*PyObject* *module, const char *docstring)

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. *module* の docstring を *docstring* に設定します。この関数は、`PyModuleDef` から `PyModule_Create` もしくは `PyModule_FromDefAndSpec` を使ってモジュールを生成するときに自動的に呼び出されます。

Added in version 3.5.

int **PyModule_AddFunctions**(*PyObject* *module, *PyMethodDef* *functions)

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. 終端が `NULL` になっている *functions* 配列にある関数を *module* に追加します。*PyMethodDef* 構造体の個々のエントリについては PyMethodDef の

説明を参照してください (モジュールの名前空間が共有されていないので、C で実装されたモジュールレベル ” 関数” はたいていモジュールを 1 つ目の引数として受け取り、Python クラスのインスタンスメソッドに似た形にします)。この関数は、PyModuleDef から PyModule_Create もしくは PyModule_FromDefAndSpec を使ってモジュールを生成するときに自動的に呼び出されます。

Added in version 3.5.

サポート関数

モジュールの初期化関数 (一段階初期化を使う場合) 、あるいはモジュールの実行スロットから呼び出される関数 (多段階初期化を使う場合) は次の関数を使うと、モジュールの state の初期化を簡単にできます:

int **PyModule_AddObjectRef**(*PyObject* *module, const char *name, *PyObject* *value)

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. *module* にオブジェクトを *name* として追加します。この関数はモジュールの初期化関数から利用される便利関数です。

成功すると 0 を返し、エラーになると例外を送出して -1 を返します。

使用例:

```
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.

この例は、明示的に *obj* が NULL であることを確認せずに書くこともできます:

```
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;
}
```

この場合は、*obj* が NULL でありうるため、Py_DECREF() の代わりに Py_XDECREF() を呼び出す必要

があることに注意してください。

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.

Added in version 3.10.

int **PyModule_AddObject**(*PyObject* *module, const char *name, *PyObject* *value)

次に属します: Stable ABI. Similar to *PyModule_AddObjectRef()*, but steals a reference to *value* on success (if it returns `0`).

The new *PyModule_AddObjectRef()* function is recommended, since it is easy to introduce reference leaks by misusing the *PyModule_AddObject()* function.

注釈

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_DECREF()* *value* manually on error.

使用例:

```
static int
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    if (obj == NULL) {
        return -1;
    }
    if (PyModule_AddObject(module, "spam", obj) < 0) {
        Py_DECREF(obj);
        return -1;
    }
    // PyModule_AddObject() stole a reference to obj:
    // Py_DECREF(obj) is not needed here
    return 0;
}
```

この例は、明示的に *obj* が `NULL` であることを確認せずに書くこともできます:

```
static int
```

(次のページに続く)

(前のページからの続き)

```
add_spam(PyObject *module, int value)
{
    PyObject *obj = PyLong_FromLong(value);
    if (PyModule_AddObject(module, "spam", obj) < 0) {
        Py_XDECREF(obj);
        return -1;
    }
    // PyModule_AddObject() stole a reference to obj:
    // Py_DECREF(obj) is not needed here
    return 0;
}
```

この場合は、*obj* が `NULL` でありうるため、`Py_DECREF()` の代わりに `Py_XDECREF()` を呼び出す必要があることに注意してください。

int **PyModule_AddIntConstant**(*PyObject* *module, const char *name, long value)

次に属します: 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)

次に属します: 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)

文字列定数を *module* に追加します。

int **PyModule_AddType**(*PyObject* *module, *PyTypeObject* *type)

次に属します: Stable ABI *(バージョン 3.10 より)*. 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.

Added in version 3.9.

モジュール検索

一段階初期化は、現在のインタプリタのコンテキストから探せるシングルトンのモジュールを生成します。これによって、後からモジュール定義への参照だけでモジュールオブジェクトが取得できます。

多段階初期化を使うと単一の定義から複数のモジュールが作成できるので、これらの関数は多段階初期化を使って作成されたモジュールには使えません。

PyObject ***PyState_FindModule**(*PyModuleDef* *def)

戻り値: **借用参照。次に属します**: Stable ABI. 現在のインタプリタの *def* から作られたモジュールオブジェクトを返します。このメソッドの前提条件として、前もって *PyState_AddModule()* でインタプリタの state にモジュールオブジェクトを連結しておくことを要求します。対応するモジュールオブジェクトが見付からない、もしくは事前にインタプリタの state に連結されていない場合は、`NULL` を返します。

int **PyState_AddModule**(*PyObject* *module, *PyModuleDef* *def)

次に属します: Stable ABI *(**バージョン** 3.3 **より**)*. 関数に渡されたモジュールオブジェクトを、インタプリタの state に連結します。この関数を使うことで *PyState_FindModule()* からモジュールオブジェクトにアクセスできるようになります。

一段階初期化を使って作成されたモジュールにのみ有効です。

Python calls `PyState_AddModule` automatically after importing a module, so it is unnecessary (but harmless) to call it from module initialization code. An explicit call is needed only if the module's own init code subsequently calls `PyState_FindModule`. The function is mainly intended for implementing alternative import mechanisms (either by calling it directly, or by referring to its implementation for details of the required state updates).

The caller must hold the GIL.

Return `-1` with an exception set on error, `0` on success.

Added in version 3.3.

int **PyState_RemoveModule**(*PyModuleDef* *def)

次に属します: Stable ABI *(**バージョン** 3.3 **より**)*. Removes the module object created from *def* from the interpreter state. Return `-1` with an exception set on error, `0` on success.

The caller must hold the GIL.

Added in version 3.3.

8.6.3 イテレータオブジェクト (iterator object)

Python では二種類のイテレータオブジェクトを提供しています。一つ目はシーケンスイテレータで、`__getitem__()` メソッドをサポートする任意のシーケンスを取り扱います。二つ目は呼び出し可能オブジェクトと番兵 (sentinel value) を扱い、シーケンス内の要素ごとに呼び出し可能オブジェクトを呼び出して、センチネル値が返されたときに反復処理を終了します。

PyTypeObject **PySeqIter_Type**

次に属します: Stable ABI. *PySeqIter_New()* や、組み込みシーケンス型に対して 1 引数形式の組み込み関数 `iter()` を呼び出したときに返される、イテレータオブジェクトの型オブジェクトです。

int **PySeqIter_Check**(*PyObject* *op)

op の型が `PySeqlter_Type` 型の場合に真を返します。この関数は常に成功します。

PyObject ***PySeqIter_New**(*PyObject* *seq)

戻り値: 新しい参照。次に属します: Stable ABI. 一般的なシーケンスオブジェクト *seq* を扱うイテレータを返します。反復処理は、シーケンスが添字指定操作の際に `IndexError` を返したときに終了します。

PyTypeObject **PyCallIter_Type**

次に属します: Stable ABI. *PyCallIter_New()* や、組み込み関数 `iter()` の 2 引数形式が返すイテレータオブジェクトの型オブジェクトです。

int **PyCallIter_Check**(*PyObject* *op)

op の型が *PyCallIter_Type* 型の場合に真を返します。この関数は常に成功します。

PyObject ***PyCallIter_New**(*PyObject* *callable, *PyObject* *sentinel)

戻り値: 新しい参照。次に属します: Stable ABI. 新たなイテレータを返します。最初のパラメタ *callable* は引数なしで呼び出せる Python の呼び出し可能オブジェクトならなんでもかまいません; *callable* は、呼び出されるたびに次の反復処理対象オブジェクトを返さなければなりません。生成されたイテレータは、*callable* が *sentinel* に等しい値を返すと反復処理を終了します。

8.6.4 デスクリプタオブジェクト (descriptor object)

" デスクリプタ (descriptor)" は、あるオブジェクトのいくつかの属性について記述したオブジェクトです。デスクリプタオブジェクトは型オブジェクトの辞書内にあります。

PyTypeObject **PyProperty_Type**

次に属します: Stable ABI. 組み込みデスクリプタ型の型オブジェクトです。

PyObject ***PyDescr_NewGetSet**(*PyTypeObject* *type, struct *PyGetSetDef* *getset)

戻り値: 新しい参照。次に属します: Stable ABI.

PyObject ***PyDescr_NewMember**(*PyTypeObject* *type, struct *PyMemberDef* *meth)

戻り値: 新しい参照。次に属します: Stable ABI.

PyObject ***PyDescr_NewMethod**(*PyTypeObject* *type, struct *PyMethodDef* *meth)

戻り値: 新しい参照。次に属します: Stable ABI.

PyObject ***PyDescr_NewWrapper**(*PyTypeObject* *type, struct wrapperbase *wrapper, void *wrapped)

戻り値: 新しい参照。

PyObject ***PyDescr_NewClassMethod**(*PyTypeObject* *type, *PyMethodDef* *method)

戻り値: 新しい参照。次に属します: Stable ABI.

int **PyDescr_IsData**(*PyObject* *descr)

デスクリプタオブジェクト *descr* がデータ属性のデスクリプタの場合には非ゼロ値を、メソッドデスクリプタの場合には `0` を返します。*descr* はデスクリプタオブジェクトでなければなりません。エラーチェックは行いません。

PyObject ***PyWrapper_New**(*PyObject**, *PyObject**)

戻り値: **新しい参照。次に属します**: Stable ABI.

8.6.5 スライスオブジェクト (slice object)

PyTypeObject **PySlice_Type**

次に属します: Stable ABI. スライスオブジェクトの型オブジェクトです。これは、Python レイヤにおける `slice` と同じオブジェクトです。

int **PySlice_Check**(*PyObject* *ob)

ob がスライスオブジェクトの場合に真を返します; *ob* は `NULL` であってはなりません。この関数は常に成功します。

PyObject ***PySlice_New**(*PyObject* *start, *PyObject* *stop, *PyObject* *step)

戻り値: **新しい参照。次に属します**: 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)

次に属します: Stable ABI. スライスオブジェクト *slice* における *start*, *stop*, および *step* のインデクス値を取得します。このときシーケンスの長さを *length* と仮定します。*length* よりも大きなインデクスになるとエラーとして扱います。

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).

おそらく、あなたはこの関数を使いたくないでしょう。

バージョン 3.2 で変更: 以前は、*slice* 引数の型は `PySliceObject*` でした。

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)

次に属します: Stable ABI. *PySlice_GetIndices()* の便利な代替です。*slice* における、*start*, *stop* および *step* のインデクス値を取得をします。シーケンスの長さを *length* 、スライスの長さを *slicelength* に格納します。境界外のインデクスは通常のスライスと一貫した方法でクリップされます。

Return `0` on success and `-1` on error with an exception set.

注釈

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);
```

バージョン 3.2 で変更: 以前は、*slice* 引数の型は `PySliceObject*` でした。

バージョン 3.6.1 で変更: 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.

バージョン 3.6.1 で非推奨: 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)

次に属します: Stable ABI *(バージョン 3.7 より)*. 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.

Added in 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)

次に属します: Stable ABI *(バージョン 3.7 より)*. 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.

Added in version 3.6.1.

Ellipsis オブジェクト

PyTypeObject **PyEllipsis_Type**

次に属します: 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.

バージョン 3.12 で変更: *Py_Ellipsis* は immortal です。

8.6.6 memoryview オブジェクト

`memoryview` オブジェクトは、他のオブジェクトと同じように扱える Python オブジェクトの形をした C 言語レベルの **バッファのインターフェース** です。

PyObject ***PyMemoryView_FromObject**(*PyObject* *obj)

戻り値: **新しい参照。次に属します**: Stable ABI. バッファインターフェースを提供するオブジェクトから memoryview オブジェクトを生成します。もし *obj* が書き込み可能なバッファのエクスポートをサポートするなら、その memoryview オブジェクトは読み書き可能です。そうでなければ読出しのみになるか、エクスポーターの分別にもとづいて読み書きが可能となります。

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)

戻り値: **新しい参照。次に属します**: Stable ABI **(バージョン** *3.7* **より)**. *mem* を配下のバッファとして memoryview オブジェクトを作成します。*flags* は *PyBUF_READ* か *PyBUF_WRITE* のどちらかになります。

Added in version 3.3.

PyObject ***PyMemoryView_FromBuffer**(const *Py_buffer* *view)

戻り値: **新しい参照。次に属します**: Stable ABI **(バージョン** *3.11* **より)**. *view* として与えられたバッファ構造をラップする memoryview オブジェクトを作成します。単なるバイトバッファ向けには、*PyMemoryView_FromMemory()* のほうが望ましいです。

PyObject ***PyMemoryView_GetContiguous**(*PyObject* *obj, int buffertype, char order)

戻り値: **新しい参照。次に属します**: Stable ABI. buffer インターフェースを定義しているオブジェクトから ('C' か 'F'ortran の *order* で) **連続した** メモリチャンクへの memoryview オブジェクトを作ります。メモリが連続している場合、memoryview オブジェクトは元のメモリを参照します。それ以外の場合、メモリはコピーされて、memoryview オブジェクトは新しい bytes オブジェクトを参照します。

buffertype can be one of *PyBUF_READ* or *PyBUF_WRITE*.

int **PyMemoryView_Check**(*PyObject* *obj)

obj が memoryview オブジェクトの場合に真を返します。現在のところ、`memoryview` のサブクラスの作成は許可されていません。この関数は常に成功します。

Py_buffer ***PyMemoryView_GET_BUFFER**(*PyObject* *mview)

書きだされたバッファーの memoryview のプライベート コピーに、ポインターを返します。*mview* は memoryview インスタンスでなければなりません; このマクロは型をチェックしないので自前で型チェックしなければならず、それを怠るとクラッシュする恐れがあります。

PyObject ***PyMemoryView_GET_BASE**(*PyObject* *mview)

memoryview をエクスポートしているオブジェクトへのポインタを返します。memoryview が *PyMemoryView_FromMemory()* か *PyMemoryView_FromBuffer()* のどちらかで作成されていた場合、`NULL` を返します。

8.6.7 弱参照オブジェクト

Python は **弱参照** を第一級オブジェクト (first-class object) としてサポートします。弱参照を直接実装する二種類の固有のオブジェクト型があります。第一は単純な参照オブジェクトで、第二はオリジナルのオブジェクトに対して可能な限りプロキシとして振舞うオブジェクトです。

int **PyWeakref_Check**(*PyObject* *ob)

Return true if *ob* is either a reference or proxy object. This function always succeeds.

int **PyWeakref_CheckRef**(*PyObject* *ob)

Return true if *ob* is a reference object. This function always succeeds.

int **PyWeakref_CheckProxy**(*PyObject* *ob)

Return true if *ob* is a proxy object. This function always succeeds.

PyObject ***PyWeakref_NewRef**(*PyObject* *ob, *PyObject* *callback)

戻り値: **新しい参照。次に属します**: 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)

戻り値: **新しい参照。次に属します**: 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`.

PyObject ***PyWeakref_GetObject**(*PyObject* *ref)

戻り値: **借用参照。次に属します**: Stable ABI. Return the referenced object from a weak reference, *ref*. If the referent is no longer live, returns `Py_None`.

> **注釈**
>
> 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.

PyObject ***PyWeakref_GET_OBJECT**(*PyObject* *ref)

戻り値: **借用参照。**Similar to *PyWeakref_GetObject()*, but does no error checking.

void **PyObject_ClearWeakRefs**(*PyObject* *object)

次に属します: 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.

8.6.8 カプセル

using-capsules 以下のオブジェクトを使う方法については using-capsules を参照してください。

Added in version 3.1.

type **PyCapsule**

この *PyObject* のサブタイプは、任意の値を表し、C 拡張モジュールから Python コードを経由して他の C 言語のコードに任意の値を (`void*` ポインタの形で) 渡す必要があるときに有用です。あるモジュール内で定義されている C 言語関数のポインタを、他のモジュールに渡してそこから呼び出せるようにするためによく使われます。これにより、動的にロードされるモジュールの中の C API に通常の import 機構を通してアクセスすることができます。

type **PyCapsule_Destructor**

次に属します: Stable ABI. カプセルに対するデストラクタコールバック型. 次のように定義されます:

```
typedef void (*PyCapsule_Destructor)(PyObject *);
```

PyCapsule_Destructor コールバックの動作については *PyCapsule_New()* を参照してください。

int **PyCapsule_CheckExact**(*PyObject* *p)

引数が *PyCapsule* の場合に真を返します。この関数は常に成功します。

PyObject ***PyCapsule_New**(void *pointer, const char *name, *PyCapsule_Destructor* destructor)

戻り値: **新しい参照。次に属します**: Stable ABI. *pointer* を格納する *PyCapsule* を作成します。*pointer* 引数は NULL であってはなりません。

失敗した場合、例外を設定して NULL を返します。

name 文字列は NULL か、有効な C 文字列へのポインタです。NULL で無い場合、この文字列は少なくともカプセルより長く生存する必要があります。(*destructor* の中で解放することは許可されています)

destructor が NULL で無い場合、カプセルが削除されるときにそのカプセルを引数として呼び出されます。

このカプセルがモジュールの属性として保存される場合、*name* は `modulename.attributename` と指定されるべきです。こうすると、他のモジュールがそのカプセルを *PyCapsule_Import()* でインポートすることができます。

void ***PyCapsule_GetPointer**(*PyObject* *capsule, const char *name)

次に属します: Stable ABI. カプセルに保存されている *pointer* を取り出します。失敗した場合は例外を設定して NULL を返します。

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)

次に属します: Stable ABI. カプセルに保存されている現在のデストラクタを返します。失敗した場合、例外を設定して NULL を返します。

カプセルは NULL をデストラクタとして持つことができます。従って、戻り値の NULL がエラーを指してない可能性があります。*PyCapsule_IsValid()* か *PyErr_Occurred()* を利用して確認してください。

void ***PyCapsule_GetContext**(*PyObject* *capsule)

次に属します: Stable ABI. カプセルに保存されている現在のコンテキスト (context) を返します。失敗した場合、例外を設定して NULL を返します。

カプセルは NULL をコンテキストとして持つことができます。従って、戻り値の NULL がエラーを指してない可能性があります。*PyCapsule_IsValid()* か *PyErr_Occurred()* を利用して確認してください。

const char ***PyCapsule_GetName**(*PyObject* *capsule)

次に属します: Stable ABI. カプセルに保存されている現在の name を返します。失敗した場合、例外を設定して NULL を返します。

カプセルは NULL を name として持つことができます。従って、戻り値の NULL がエラーを指してない可能性があります。*PyCapsule_IsValid()* か *PyErr_Occurred()* を利用して確認してください。

void ***PyCapsule_Import**(const char *name, int no_block)

次に属します: 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.

成功した場合、カプセルの内部 **ポインタ** を返します。失敗した場合、例外を設定して `NULL` を返します。

バージョン 3.3 で変更: *no_block* has no effect anymore.

int **PyCapsule_IsValid**(*PyObject* *capsule, const char *name)

次に属します: Stable ABI. *capsule* が有効なカプセルであるかどうかをチェックします。有効な *capsule* は、非 `NULL` で、*PyCapsule_CheckExact()* をパスし、非 `NULL` なポインタを格納していて、内部の name が引数 *name* とマッチします。(name の比較方法については *PyCapsule_GetPointer()* を参照)

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.

オブジェクトが有効で name がマッチした場合に非 `0` を、それ以外の場合に `0` を返します。この関数は絶対に失敗しません。

int **PyCapsule_SetContext**(*PyObject* *capsule, void *context)

次に属します: Stable ABI. *capsule* 内部のコンテキストポインタを *context* に設定します。

成功したら `0` を、失敗したら例外を設定して非 `0` を返します。

int **PyCapsule_SetDestructor**(*PyObject* *capsule, *PyCapsule_Destructor* destructor)

次に属します: Stable ABI. *capsule* 内部のデストラクタを *destructor* に設定します。

成功したら `0` を、失敗したら例外を設定して非 `0` を返します。

int **PyCapsule_SetName**(*PyObject* *capsule, const char *name)

次に属します: Stable ABI. *capsule* 内部の name を *name* に設定します。*name* が非 `NULL` のとき、それは *capsule* よりも長い寿命を持つ必要があります。もしすでに *capsule* に非 `NULL` の *name* が保存されていた場合、それに対する解放は行われません。

成功したら `0` を、失敗したら例外を設定して非 `0` を返します。

int **PyCapsule_SetPointer**(*PyObject* *capsule, void *pointer)

次に属します: Stable ABI. *capsule* 内部のポインタを *pointer* に設定します。*pointer* は `NULL` であってはなりません。

成功したら `0` を、失敗したら例外を設定して非 `0` を返します。

8.6.9 フレーム (frame) オブジェクト

type **PyFrameObject**

次に属します: Limited API (**不透明な構造体として**). フレームオブジェクトを記述する際に用いられる、オブジェクトを表す C 構造体です。

There are no public members in this structure.

バージョン 3.11 で変更: The members of this structure were removed from the public C API. Refer to the What's New entry for details.

The *PyEval_GetFrame()* and *PyThreadState_GetFrame()* functions can be used to get a frame object.

See also *Reflection*.

PyTypeObject **PyFrame_Type**

The type of frame objects. It is the same object as `types.FrameType` in the Python layer.

バージョン 3.11 で変更: Previously, this type was only available after including `<frameobject.h>`.

int **PyFrame_Check**(*PyObject* *obj)

Return non-zero if *obj* is a frame object.

バージョン 3.11 で変更: Previously, this function was only available after including `<frameobject.h>`.

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

Get the *frame* next outer frame.

Return a *strong reference*, or `NULL` if *frame* has no outer frame.

Added in version 3.9.

PyObject ***PyFrame_GetBuiltins**(*PyFrameObject* *frame)

Get the *frame*'s `f_builtins` attribute.

Return a *strong reference*. The result cannot be `NULL`.

Added in version 3.11.

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

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. Get the *frame* code.

Return a *strong reference*.

The result (frame code) cannot be `NULL`.

Added in version 3.9.

PyObject ***PyFrame_GetGenerator**(*PyFrameObject* *frame)

Get the generator, coroutine, or async generator that owns this frame, or `NULL` if this frame is not owned by a generator. Does not raise an exception, even if the return value is `NULL`.

Return a *strong reference*, or `NULL`.

Added in version 3.11.

PyObject ***PyFrame_GetGlobals**(*PyFrameObject* *frame)

Get the *frame*'s `f_globals` attribute.

Return a *strong reference*. The result cannot be `NULL`.

Added in version 3.11.

int **PyFrame_GetLasti**(*PyFrameObject* *frame)

Get the *frame*'s `f_lasti` attribute.

Returns -1 if `frame.f_lasti` is `None`.

Added in version 3.11.

PyObject ***PyFrame_GetVar**(*PyFrameObject* *frame, *PyObject* *name)

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`.

Added in version 3.12.

PyObject ***PyFrame_GetVarString**(*PyFrameObject* *frame, const char *name)

Similar to *PyFrame_GetVar()*, but the variable name is a C string encoded in UTF-8.

Added in version 3.12.

PyObject ***PyFrame_GetLocals**(*PyFrameObject* *frame)

Get the *frame*'s `f_locals` attribute (`dict`).

Return a *strong reference*.

Added in version 3.11.

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

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. *frame* が現在実行している行番号を返します。

Internal Frames

Unless using **PEP 523**, you will not need this.

struct **_PyInterpreterFrame**

The interpreter's internal frame representation.

Added in version 3.11.

PyObject ***PyUnstable_InterpreterFrame_GetCode**(struct *_PyInterpreterFrame* *frame);

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

Return a *strong reference* to the code object for the frame.

Added in version 3.12.

int **PyUnstable_InterpreterFrame_GetLasti**(struct *_PyInterpreterFrame* *frame);

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

Return the byte offset into the last executed instruction.

Added in version 3.12.

int **PyUnstable_InterpreterFrame_GetLine**(struct *_PyInterpreterFrame* *frame);

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

Return the currently executing line number, or -1 if there is no line number.

Added in version 3.12.

8.6.10 ジェネレータオブジェクト

ジェネレータオブジェクトは、Python がジェネレータイテレータを実装するのに使っているオブジェクトです。ジェネレータオブジェクトは通常、*PyGen_New()* や *PyGen_NewWithQualName()* の明示的な呼び出しではなく、値を yield する関数のイテレーションにより生成されます。

type **PyGenObject**

ジェネレータオブジェクトに使われている C 構造体です。

PyTypeObject **PyGen_Type**

ジェネレータオブジェクトに対応する型オブジェクトです。

int **PyGen_Check**(*PyObject* *ob)

ob がジェネレータオブジェクトの場合に真を返します、*ob* は NULL であってはなりません。この関数は常に成功します。

int **PyGen_CheckExact**(*PyObject* *ob)

ob が *PyGen_Type* の場合に真を返します。*o* は NULL であってはなりません。この関数は常に成功します。

PyObject ***PyGen_New**(*PyFrameObject* *frame)

戻り値: **新しい参照。***frame* オブジェクトに基づいて新たなジェネレータオブジェクトを生成して返します。この関数は *frame* への参照を盗みます。引数が NULL であってはなりません。

PyObject ***PyGen_NewWithQualName**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

戻り値: **新しい参照。***frame* オブジェクトから新たなジェネレータオブジェクトを生成し、__name__ と __qualname__ を *name* と *qualname* に設定して返します。この関数は *frame* への参照を盗みます。*frame* 引数は NULL であってはなりません。

8.6.11 コルーチンオブジェクト

Added in version 3.5.

コルーチンオブジェクトは async キーワードを使って定義した関数が返すオブジェクトです。

type **PyCoroObject**

コルーチンオブジェクトのための C 構造体。

PyTypeObject **PyCoro_Type**

コルーチンオブジェクトに対応する型オブジェクト。

int **PyCoro_CheckExact**(*PyObject* *ob)

ob が *PyCoro_Type* の場合に真を返します。*ob* は NULL であってはなりません。この関数は常に成功します。

PyObject ***PyCoro_New**(*PyFrameObject* *frame, *PyObject* *name, *PyObject* *qualname)

戻り値: **新しい参照。***frame* オブジェクトから新しいコルーチンオブジェクトを生成して、__name__ と __qualname__ を *name* と *qualname* に設定して返します。この関数は *frame* への参照を奪います。*frame* 引数は NULL であってはなりません。

8.6.12 コンテキスト変数オブジェクト

Added in version 3.7.

バージョン 3.7.1 で変更:

注釈

Python 3.7.1 で全てのコンテキスト変数の C API のシグネチャは、*PyContext*, *PyContextVar*, *PyContextToken* の代わりに *PyObject* ポインタを使うように **変更** されました。例えば:

```
// in 3.7.0:
PyContext *PyContext_New(void);
```

```
// in 3.7.1+:
PyObject *PyContext_New(void);
```

詳細は bpo-34762 を参照してください。

この節では、contextvars モジュールの公開 C API の詳細について説明します。

type **PyContext**

contextvars.Context オブジェクトを表現するための C 構造体。

type **PyContextVar**

contextvars.ContextVar オブジェクトを表現するための C 構造体。

type **PyContextToken**

contextvars.Token オブジェクトを表現するための C 構造体。

PyTypeObject **PyContext_Type**

コンテキスト 型を表現する型オブジェクト。

PyTypeObject **PyContextVar_Type**

コンテキスト変数 型を表現する型オブジェクト。

PyTypeObject **PyContextToken_Type**

コンテキスト変数トークン 型を表現する型オブジェクト。

型チェックマクロ:

int **PyContext_CheckExact**(*PyObject* *o)

o が *PyContext_Type* の場合に真を返します。*o* は NULL であってはなりません。この関数は常に成功します。

int **PyContextVar_CheckExact**(*PyObject* *o)

o が *PyContextVar_Type* の場合に真を返します。*o* は NULL であってはなりません。この関数は常に成功します。

int **PyContextToken_CheckExact**(*PyObject* *o)

o が *PyContextToken_Type* の場合に真を返します。*o* は NULL であってはなりません。この関数は常に成功します。

コンテキストオブジェクトを取り扱う関数:

PyObject ***PyContext_New**(void)

戻り値: **新しい参照。** 新しい空のコンテキストオブジェクトを作成します。エラーが起きた場合は NULL を返します。

PyObject *PyContext_Copy(*PyObject* *ctx)

戻り値: **新しい参照。** 渡された *ctx* コンテキストオブジェクトの浅いコピーを作成します。エラーが起きた場合は NULL を返します。

PyObject *PyContext_CopyCurrent(void)

戻り値: **新しい参照。** 現在のコンテキストオブジェクトの浅いコピーを作成します。エラーが起きた場合は NULL を返します。

int PyContext_Enter(*PyObject* *ctx)

ctx を現在のスレッドの現在のコンテキストに設定します。成功したら 0 を、失敗したら -1 を返します。

int PyContext_Exit(*PyObject* *ctx)

ctx コンテキストを無効にし、1 つ前のコンテキストを現在のスレッドの現在のコンテキストに復元します。成功したら 0 を、失敗したら -1 を返します。

コンテキスト変数の関数:

PyObject *PyContextVar_New(const char *name, *PyObject* *def)

戻り値: **新しい参照。** 新しい ContextVar オブジェクトを作成します。*name* 引数は内部走査とデバッグの目的で使われます。*def* 引数はコンテキスト変数のデフォルト値を指定するか、デフォルトがない場合は NULL です。エラーが起きた場合は、関数は NULL を返します。

int PyContextVar_Get(*PyObject* *var, *PyObject* *default_value, *PyObject* **value)

コンテキスト変数の値を取得します。取得中にエラーが起きた場合は -1 を、値が見付かっても見付からなくてもエラーが起きなかった場合は 0 を返します。

コンテキスト変数が見付かった場合、*value* はそれを指すポインタになっています。コンテキスト変数が見付から **なかった** 場合は、*value* が指すものは次のようになっています:

- (NULL でなければ) *default_value*
- (NULL でなければ) *var* のデフォルト値
- NULL

NULL を除けば、この関数は新しい参照を返します。

PyObject *PyContextVar_Set(*PyObject* *var, *PyObject* *value)

戻り値: **新しい参照。** 現在のコンテキストにおいて *var* の値を *value* にセットします。この変更による新しいトークンオブジェクトか、エラーが起こった場合は “NULL“を返します。

int PyContextVar_Reset(*PyObject* *var, *PyObject* *token)

var コンテキスト変数の状態をリセットし、*token* を返した *PyContextVar_Set()* が呼ばれる前の状態に戻します。この関数は成功したら 0 、失敗したら -1 を返します。

8.6.13 DateTime オブジェクト

datetime モジュールでは、様々な日付オブジェクトや時刻オブジェクトを提供しています。以下に示す関数を使う場合には、あらかじめヘッダファイル datetime.h をソースに include し (Python.h はこのファイルを include しません)、PyDateTime_IMPORT マクロを、通常はモジュール初期化関数から、起動しておく必要があります。このマクロは以下のマクロで使われる静的変数 PyDateTimeAPI に C 構造体へのポインタを入れます。

type **PyDateTime_Date**

この *PyObject* のサブタイプは、Python の日付オブジェクトを表します。

type **PyDateTime_DateTime**

この *PyObject* のサブタイプは Python の datetime オブジェクトを表します。

type **PyDateTime_Time**

この *PyObject* のサブタイプは Python の time オブジェクトを表現します。

type **PyDateTime_Delta**

この *PyObject* のサブタイプは二つの datetime の値の差分を表します。

PyTypeObject **PyDateTime_DateType**

この *PyTypeObject* のインスタンスは Python の date 型を表します; Python レイヤにおける datetime.date と同じオブジェクトです。

PyTypeObject **PyDateTime_DateTimeType**

この *PyTypeObject* のインスタンスは Python の datetime 型を表現します; Python レイヤにおける datetime.datetime と同じオブジェクトです。

PyTypeObject **PyDateTime_TimeType**

この *PyTypeObject* のインスタンスは Python の time 型を表します; Python レイヤにおける datetime.time と同じオブジェクトです。

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**

この *PyTypeObject* のインスタンスは Python の タイムゾーン情報型を表します; Python レイヤにおける datetime.tzinfo と同じオブジェクトです。

UTC シングルトンにアクセスするためのマクロ:

PyObject ***PyDateTime_TimeZone_UTC**

UTC タイムゾーンに相当するシングルトンを返します。これは datetime.timezone.utc と同じオブジェクトです。

Added in version 3.7.

型チェックマクロ:

int **PyDate_Check**(*PyObject* *ob)

ob が *PyDateTime_DateType* 型か PyDateTime_DateType 型のサブタイプのオブジェクトの場合に真を返します; *ob* は NULL であってはなりません。この関数は常に成功します。

int **PyDate_CheckExact**(*PyObject* *ob)

ob が *PyDateTime_DateType* の場合に真を返します。*ob* は NULL であってはなりません。この関数は常に成功します。

int **PyDateTime_Check**(*PyObject* *ob)

ob が *PyDateTime_DateTimeType* 型か PyDateTime_DateTimeType 型のサブタイプのオブジェクトの場合に真を返します; *ob* は NULL であってはなりません。この関数は常に成功します。

int **PyDateTime_CheckExact**(*PyObject* *ob)

ob が *PyDateTime_DateTimeType* の場合に真を返します。*ob* は NULL であってはなりません。この関数は常に成功します。

int **PyTime_Check**(*PyObject* *ob)

ob が *PyDateTime_TimeType* 型か PyDateTime_TimeType 型のサブタイプのオブジェクトの場合に真を返します; *ob* は NULL であってはなりません。この関数は常に成功します。

int **PyTime_CheckExact**(*PyObject* *ob)

ob が *PyDateTime_TimeType* の場合に真を返します。*ob* は NULL であってはなりません。この関数は常に成功します。

int **PyDelta_Check**(*PyObject* *ob)

ob が *PyDateTime_DeltaType* 型か PyDateTime_DeltaType 型のサブタイプのオブジェクトの場合に真を返します; *ob* は NULL であってはなりません。この関数は常に成功します。

int **PyDelta_CheckExact**(*PyObject* *ob)

ob が *PyDateTime_DeltaType* の場合に真を返します。*ob* は NULL であってはなりません。この関数は常に成功します。

int **PyTZInfo_Check**(*PyObject* *ob)

ob が *PyDateTime_TZInfoType* 型か PyDateTime_TZInfoType 型のサブタイプのオブジェクトの場合に真を返します; *ob* は NULL であってはなりません。この関数は常に成功します。

int **PyTZInfo_CheckExact**(*PyObject* *ob)

ob が *PyDateTime_TZInfoType* の場合に真を返します。*ob* は NULL であってはなりません。この関数は常に成功します。

以下はオブジェクトを作成するためのマクロです:

PyObject ***PyDate_FromDate**(int year, int month, int day)

戻り値: **新しい参照。**指定した年、月、日の datetime.date オブジェクトを返します。

PyObject *PyDateTime_FromDateAndTime(int year, int month, int day, int hour, int minute, int second, int usecond)

戻り値: **新しい参照。** 指定した年、月、日、時、分、秒、マイクロ秒の `datetime.datetime` オブジェクトを返します。

PyObject *PyDateTime_FromDateAndTimeAndFold(int year, int month, int day, int hour, int minute, int second, int usecond, int fold)

戻り値: **新しい参照。** 指定された年、月、日、時、分、秒、マイクロ秒、fold の `datetime.datetime` オブジェクトを返します。

Added in version 3.6.

PyObject *PyTime_FromTime(int hour, int minute, int second, int usecond)

戻り値: **新しい参照。** 指定された時、分、秒、マイクロ秒の `datetime.time` オブジェクトを返します。

PyObject *PyTime_FromTimeAndFold(int hour, int minute, int second, int usecond, int fold)

戻り値: **新しい参照。** 指定された時、分、秒、マイクロ秒、fold の `datetime.time` オブジェクトを返します。

Added in version 3.6.

PyObject *PyDelta_FromDSU(int days, int seconds, int useconds)

戻り値: **新しい参照。** 指定された日、秒、マイクロ秒の `datetime.timedelta` オブジェクトを返します。マイクロ秒と秒が `datetime.timedelta` オブジェクトで定義されている範囲に入るように正規化を行います。

PyObject *PyTimeZone_FromOffset(*PyObject* *offset)

戻り値: **新しい参照。** *offset* 引数で指定した固定オフセットを持つ、名前のない `datetime.timezone` オブジェクトを返します。

Added in version 3.7.

PyObject *PyTimeZone_FromOffsetAndName(*PyObject* *offset, *PyObject* *name)

戻り値: **新しい参照。** *offset*引数で指定した固定のオフセットと、*name* のタイムゾーン名を持つ `datetime.timezone` オブジェクトを返します。

Added in version 3.7.

以下のマクロは date オブジェクトからフィールド値を取り出すためのものです。引数は *PyDateTime_Date* またはそのサブクラス (例えば *PyDateTime_DateTime*) の インスタンスでなければなりません。引数を `NULL` にしてはならず、型チェックは行いません:

int PyDateTime_GET_YEAR(*PyDateTime_Date* *o)

年を正の整数で返します。

int PyDateTime_GET_MONTH(*PyDateTime_Date* *o)

月を 1 から 12 の間の整数で返します。

int **PyDateTime_GET_DAY**(*PyDateTime_Date* *o)

日を 1 から 31 の間の整数で返します。

以下のマクロは datetime オブジェクトからフィールド値を取り出すためのものです。引数は *PyDateTime_DateTime* またはそのサブクラスのインスタンスでなければなりません。引数を NULL にしてはならず、型チェックは行いません:

int **PyDateTime_DATE_GET_HOUR**(*PyDateTime_DateTime* *o)

時を 0 から 23 の間の整数で返します。

int **PyDateTime_DATE_GET_MINUTE**(*PyDateTime_DateTime* *o)

分を 0 から 59 の間の整数で返します。

int **PyDateTime_DATE_GET_SECOND**(*PyDateTime_DateTime* *o)

秒を 0 から 59 の間の整数で返します。

int **PyDateTime_DATE_GET_MICROSECOND**(*PyDateTime_DateTime* *o)

マイクロ秒を 0 から 999999 の間の整数で返します。

int **PyDateTime_DATE_GET_FOLD**(*PyDateTime_DateTime* *o)

フォールド (訳注: サマータイムによる時間のずれのこと) を 0 から 1 までの整数で返します。

Added in version 3.6.

PyObject ***PyDateTime_DATE_GET_TZINFO**(*PyDateTime_DateTime* *o)

Return the tzinfo (which may be None).

Added in version 3.10.

以下のマクロは time オブジェクトからフィールド値を取り出すためのものです。引数は *PyDateTime_Time* またはそのサブクラスのインスタンスでなければなりません。引数を NULL にしてはならず、型チェックは行いません:

int **PyDateTime_TIME_GET_HOUR**(*PyDateTime_Time* *o)

時を 0 から 23 の間の整数で返します。

int **PyDateTime_TIME_GET_MINUTE**(*PyDateTime_Time* *o)

分を 0 から 59 の間の整数で返します。

int **PyDateTime_TIME_GET_SECOND**(*PyDateTime_Time* *o)

秒を 0 から 59 の間の整数で返します。

int **PyDateTime_TIME_GET_MICROSECOND**(*PyDateTime_Time* *o)

マイクロ秒を 0 から 999999 の間の整数で返します。

int **PyDateTime_TIME_GET_FOLD**(*PyDateTime_Time* *o)

フォールド (訳注: サマータイムによる時間のずれのこと) を 0 から 1 までの整数で返します。

Added in version 3.6.

PyObject *PyDateTime_TIME_GET_TZINFO(*PyDateTime_Time* *o)

Return the tzinfo (which may be None).

Added in version 3.10.

以下のマクロは time delta オブジェクトからフィールド値をとりだすためのものです。引数は *PyDateTime_Delta* かそのサブクラスのインスタンスでなければなりません。引数を NULL にしてはならず、型チェックは行いません:

int PyDateTime_DELTA_GET_DAYS(*PyDateTime_Delta* *o)

日数を -999999999 から 999999999 の間の整数で返します。

Added in version 3.3.

int PyDateTime_DELTA_GET_SECONDS(*PyDateTime_Delta* *o)

秒数を 0 から 86399 の間の整数で返します。

Added in version 3.3.

int PyDateTime_DELTA_GET_MICROSECONDS(*PyDateTime_Delta* *o)

マイクロ秒を 0 から 999999 の間の整数で返します。

Added in version 3.3.

以下のマクロは DB API を実装する上での便宜用です:

PyObject *PyDateTime_FromTimestamp(*PyObject* *args)

戻り値: **新しい参照。** Create and return a new datetime.datetime object given an argument tuple suitable for passing to datetime.datetime.fromtimestamp().

PyObject *PyDate_FromTimestamp(*PyObject* *args)

戻り値: **新しい参照。** Create and return a new datetime.date object given an argument tuple suitable for passing to datetime.date.fromtimestamp().

8.6.14 型ヒントのためのオブジェクト

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)

次に属します: Stable ABI (**バージョン** *3.9* **より**). GenericAlias オブジェクトを生成します。Python クラス types.GenericAlias を呼び出すことと同等です。引数 *origin* と *args* は GenericAlias の __origin__ および __args__ 属性をそれぞれ設定します。*origin* は *PyTypeObject** でなければならず、*args* は *PyTupleObject** または任意の PyObject* です。*args* がタプルでない場合には 1 タプルが自動的に生成され、__args__ には (args,) が設定されます。引数チェックは最小限なため、たとえ *origin* が型を示すオブジェクトでなくても関数呼び出しは成功します。GenericAlias の __parameters__ 属性は __args__ から必要に応じて遅延生成されます。失敗した場合、例外が送出されて NULL を返します。

以下は拡張の型をジェネリックにする例です。

```
...
static PyMethodDef my_obj_methods[] = {
    // Other methods.
    ...
    {"__class_getitem__", Py_GenericAlias, METH_O|METH_CLASS, "See PEP 585"}
    ...
}
```

参考

The data model method `__class_getitem__()`.

Added in version 3.9.

PyTypeObject **Py_GenericAliasType**

次に属します: Stable ABI *(**バージョン** 3.9 **より**)*. *Py_GenericAlias()* により返される C の型オブジェクトです。Python の `types.GenericAlias` と同等です。

Added in version 3.9.

第

NINE

初期化 (INITIALIZATION)、終了処理 (FINALIZATION)、スレッド

See also *Python Initialization Configuration*.

9.1 Python 初期化以前

Python が埋め込まれているアプリケーションでは、他の Python/C API 関数を使う前に *Py_Initialize()* 関数を呼ばなければなりません。これには例外として、いくつかの関数と **グローバルな設定変数** があります。

次の関数は Python の初期化の前でも安全に呼び出せます:

- 設定関数:
  - *PyImport_AppendInittab()*
  - *PyImport_ExtendInittab()*
  - PyInitFrozenExtensions()
  - *PyMem_SetAllocator()*
  - *PyMem_SetupDebugHooks()*
  - *PyObject_SetArenaAllocator()*
  - *Py_SetPath()*
  - *Py_SetProgramName()*
  - *Py_SetPythonHome()*
  - *Py_SetStandardStreamEncoding()*
  - *PySys_AddWarnOption()*
  - *PySys_AddXOption()*
  - *PySys_ResetWarnOptions()*
- 情報取得の関数:
  - *Py_IsInitialized()*

  - *PyMem_GetAllocator()*
  - *PyObject_GetArenaAllocator()*
  - *Py_GetBuildInfo()*
  - *Py_GetCompiler()*
  - *Py_GetCopyright()*
  - *Py_GetPlatform()*
  - *Py_GetVersion()*

- ユーティリティ:
  - *Py_DecodeLocale()*
- メモリアロケータ:
  - *PyMem_RawMalloc()*
  - *PyMem_RawRealloc()*
  - *PyMem_RawCalloc()*
  - *PyMem_RawFree()*

注釈

The following functions **should not be called** before *Py_Initialize()*: *Py_EncodeLocale()*, *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()*, *Py_GetProgramFullPath()*, *Py_GetPythonHome()*, *Py_GetProgramName()* and *PyEval_InitThreads()*.

9.2 グローバルな設定変数

Python には、様々な機能やオプションを制御するグローバルな設定のための変数があります。デフォルトでは、これらのフラグは コマンドラインオプション で制御されます。

オプションでフラグがセットされると、フラグの値はそのオプションがセットされた回数になります。例えば、`-b` では *Py_BytesWarningFlag* が 1 に設定され、`-bb` では *Py_BytesWarningFlag* が 2 に設定されます。

int **Py_BytesWarningFlag**

This API is kept for backward compatibility: setting *PyConfig.bytes_warning* should be used instead, see *Python Initialization Configuration*.

`bytes` または `bytearray` を `str` と比較した場合、または、`bytes` を `int` と比較した場合に警告を発生させます。2 以上の値を設定している場合は、エラーを発生させます。

`-b` オプションで設定します。

バージョン 3.12 で非推奨.

int **Py_DebugFlag**

This API is kept for backward compatibility: setting *PyConfig.parser_debug* should be used instead, see *Python Initialization Configuration*.

パーサーのデバッグ出力を有効にします。(専門家専用です。コンパイルオプションに依存します)。

`-d` オプションと `PYTHONDEBUG` 環境変数で設定します。

バージョン 3.12 で非推奨.

int **Py_DontWriteBytecodeFlag**

This API is kept for backward compatibility: setting *PyConfig.write_bytecode* should be used instead, see *Python Initialization Configuration*.

非ゼロに設定した場合、Python はソースモジュールのインポート時に `.pyc` ファイルの作成を試みません。

`-B` オプションと `PYTHONDONTWRITEBYTECODE` 環境変数で設定します。

バージョン 3.12 で非推奨.

int **Py_FrozenFlag**

This API is kept for backward compatibility: setting *PyConfig.pathconfig_warnings* should be used instead, see *Python Initialization Configuration*.

Py_GetPath() の中でモジュール検索パスを割り出しているときのエラーメッセージを抑制します。

`_freeze_module` プログラムと `frozenmain` プログラムが使用する非公開フラグです。

バージョン 3.12 で非推奨.

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*.

`PYTHONHASHSEED` 環境変数が空でない文字列に設定された場合に、`1` が設定されます。

フラグがゼロでない場合、`PYTHONHASHSEED` 環境変数を読みシークレットハッシュシードを初期化します。

バージョン 3.12 で非推奨.

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.

`-E` オプションと `-I` オプションで設定します。

バージョン 3.12 で非推奨.

int **Py_InspectFlag**

This API is kept for backward compatibility: setting *PyConfig.inspect* should be used instead, see *Python Initialization Configuration*.

最初の引数にスクリプトが指定されたときや `-c` オプションが利用された際に、`sys.stdin` がターミナルに出力されないときであっても、スクリプトかコマンドを実行した後にインタラクティブモードに入ります。

`-i` オプションと `PYTHONINSPECT` 環境変数で設定します。

バージョン 3.12 で非推奨.

int **Py_InteractiveFlag**

This API is kept for backward compatibility: setting *PyConfig.interactive* should be used instead, see *Python Initialization Configuration*.

`-i` オプションで設定します。

バージョン 3.12 で非推奨.

int **Py_IsolatedFlag**

This API is kept for backward compatibility: setting *PyConfig.isolated* should be used instead, see *Python Initialization Configuration*.

Python を隔離モードで実行します。隔離モードでは `sys.path` はスクリプトのディレクトリやユーザのサイトパッケージのディレクトリを含みません。

`-I` オプションで設定します。

Added in version 3.4.

バージョン 3.12 で非推奨.

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*.

`PYTHONLEGACYWINDOWSFSENCODING` 環境変数が空でない文字列に設定された場合に、`1` に設定されます。

より詳しくは **PEP 529** を参照してください。

Availability: Windows.

バージョン 3.12 で非推奨.

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.

`PYTHONLEGACYWINDOWSSTDIO` 環境変数が空でない文字列に設定された場合に、`1` に設定されます。

より詳しくは **PEP 528** を参照してください。

Availability: Windows.

バージョン 3.12 で非推奨.

int **Py_NoSiteFlag**

This API is kept for backward compatibility: setting *PyConfig.site_import* should be used instead, see *Python Initialization Configuration*.

`site` モジュールの import と、そのモジュールが行なっていた site ごとの `sys.path` への操作を無効にします。後で `site` を明示的に import しても、これらの操作は実行されません (実行したい場合は、`site.main()` を呼び出してください)。

`-S` オプションで設定します。

バージョン 3.12 で非推奨.

int **Py_NoUserSiteDirectory**

This API is kept for backward compatibility: setting *PyConfig.user_site_directory* should be used instead, see *Python Initialization Configuration*.

ユーザのサイトパッケージのディレクトリ を `sys.path` に追加しません。

`-s` オプション、`-I` 、`PYTHONNOUSERSITE` 環境変数で設定します。

バージョン 3.12 で非推奨.

int **Py_OptimizeFlag**

This API is kept for backward compatibility: setting *PyConfig.optimization_level* should be used instead, see *Python Initialization Configuration*.

`-O` オプションと `PYTHONOPTIMIZE` 環境変数で設定します。

バージョン 3.12 で非推奨.

int **Py_QuietFlag**

This API is kept for backward compatibility: setting *PyConfig.quiet* should be used instead, see *Python Initialization Configuration*.

インタラクティブモードでも copyright とバージョンのメッセージを表示しません。

`-q` オプションで設定します。

Added in version 3.2.

バージョン 3.12 で非推奨.

int **Py_UnbufferedStdioFlag**

This API is kept for backward compatibility: setting *PyConfig.buffered_stdio* should be used instead, see *Python Initialization Configuration*.

標準出力と標準エラーをバッファリングしないように強制します。

-u オプションと PYTHONUNBUFFERED 環境変数で設定します。

バージョン 3.12 で非推奨.

int **Py_VerboseFlag**

This API is kept for backward compatibility: setting *PyConfig.verbose* should be used instead, see *Python Initialization Configuration*.

モジュールが初期化されるたびにメッセージを出力し、それがどこ (ファイル名やビルトインモジュール) からロードされたのかを表示します。値が 2 以上の場合は、モジュールを検索するときにチェックしたファイルごとにメッセージを出力します。また、終了時のモジュールクリーンアップに関する情報も提供します。

-v オプションと PYTHONVERBOSE 環境変数で設定します。

バージョン 3.12 で非推奨.

9.3 インタープリタの初期化と終了処理

void **Py_Initialize**()

次に属します: Stable ABI. Python インタープリタを初期化します。Python の埋め込みを行うアプリケーションでは、他のあらゆる Python/C API を使用するよりも前にこの関数を呼び出さなければなりません。いくつかの例外については *Python* **初期化以前** を参照してください。

This initializes the table of loaded modules (**sys.modules**), and creates the fundamental modules **builtins**, **__main__** and **sys**. It also initializes the module search path (**sys.path**). It does not set **sys.argv**; use *PySys_SetArgvEx()* for that. This is a no-op when called for a second time (without calling *Py_FinalizeEx()* first). There is no return value; it is a fatal error if the initialization fails.

Use the *Py_InitializeFromConfig()* function to customize the *Python Initialization Configuration*.

注釈

Windows では O_TEXT から O_BINARY へコンソールモードが変更されますが、これはその C ランタイムを使っているコンソールでの Python 以外の使い勝手にも影響を及ぼします。

void **Py_InitializeEx**(int initsigs)

次に属します: Stable ABI. This function works like *Py_Initialize()* if *initsigs* is `1`. If *initsigs* is `0`, it skips initialization registration of signal handlers, which might be useful when Python is embedded.

Use the *Py_InitializeFromConfig()* function to customize the *Python Initialization Configuration*.

int **Py_IsInitialized**()

次に属します: Stable ABI. Python インタプリタが初期化済みであれば真 (非ゼロ) を、さもなければ偽 (ゼロ) を返します。*Py_FinalizeEx()* を呼び出した後は、*Py_Initialize()* を再び呼び出すまで、この関数は偽を返します。

int **Py_FinalizeEx**()

次に属します: Stable ABI *(バージョン 3.6 より)*. Undo all initializations made by *Py_Initialize()* and subsequent use of Python/C API functions, and destroy all sub-interpreters (see *Py_NewInterpreter()* below) that were created and not yet destroyed since the last call to *Py_Initialize()*. Ideally, this frees all memory allocated by the Python interpreter. This is a no-op when called for a second time (without calling *Py_Initialize()* again first).

Since this is the reverse of *Py_Initialize()*, it should be called in the same thread with the same interpreter active. That means the main thread and the main interpreter. This should never be called while *Py_RunMain()* is running.

Normally the return value is `0`. If there were errors during finalization (flushing buffered data), `-1` is returned.

この関数が提供されている理由はいくつかあります。Python の埋め込みを行っているアプリケーションでは、アプリケーションを再起動することなく Python を再起動したいことがあります。また、動的ロード可能イブラリ (あるいは DLL) から Python インタプリタをロードするアプリケーションでは、DLL をアンロードする前に Python が確保したメモリを全て解放したいと考えるかもしれません。アプリケーション内で起きているメモリリークを追跡する際に、開発者は Python が確保したメモリをアプリケーションの終了前に解放させたいと思う場合もあります。

Bugs and caveats: The destruction of modules and objects in modules is done in random order; this may cause destructors (`__del__()` methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once; this can happen if an application calls *Py_Initialize()* and *Py_FinalizeEx()* more than once.

引数無しで 監査イベント `cpython._PySys_ClearAuditHooks` を送出します。

Added in version 3.6.

void **Py_Finalize**()

次に属します: Stable ABI. この関数は *Py_FinalizeEx()* の後方互換性バージョンで、戻り値がありません。

9.4 プロセスワイドのパラメータ

int **Py_SetStandardStreamEncoding**(const char *encoding, const char *errors)

This API is kept for backward compatibility: setting *PyConfig.stdio_encoding* and *PyConfig.stdio_errors* should be used instead, see *Python Initialization Configuration*.

This function should be called before *Py_Initialize()*, if it is called at all. It specifies which encoding and error handling to use with standard IO, with the same meanings as in `str.encode()`.

It overrides `PYTHONIOENCODING` values, and allows embedding code to control IO encoding when the environment variable does not work.

encoding and/or *errors* may be `NULL` to use `PYTHONIOENCODING` and/or default values (depending on other settings).

Note that `sys.stderr` always uses the "backslashreplace" error handler, regardless of this (or any other) setting.

If *Py_FinalizeEx()* is called, this function will need to be called again in order to affect subsequent calls to *Py_Initialize()*.

Returns `0` if successful, a nonzero value on error (e.g. calling after the interpreter has already been initialized).

Added in version 3.4.

バージョン 3.11 で非推奨.

void **Py_SetProgramName**(const wchar_t *name)

次に属します: Stable ABI. This API is kept for backward compatibility: setting *PyConfig.program_name* should be used instead, see *Python Initialization Configuration*.

この関数を呼び出すなら、最初に *Py_Initialize()* を呼び出すよりも前に呼び出さなければなりません。この関数はインタプリタにプログラムの `main()` 関数に指定した `argv[0]` 引数の値を教えます (ワイドキャラクタに変換されます)。この引数値は、*Py_GetPath()* や、以下に示すその他の関数が、インタプリタの実行可能形式から Python ランタイムライブラリへの相対パスを取得するために使われます。デフォルトの値は `'python'` です。引数はゼロ終端されたワイドキャラクタ文字列で、静的な記憶領域に入っていなければならず、その内容はプログラムの実行中に変更してはなりません。Python インタプリタ内のコードで、この記憶領域の内容を変更するものは一切ありません。

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_t*` string.

バージョン 3.11 で非推奨.

wchar_t *Py_GetProgramName()

次に属します: Stable ABI. Return the program name set with *Py_SetProgramName()*, or the default. The returned string points into static storage; the caller should not modify its value.

This function should not be called before *Py_Initialize()*, otherwise it returns NULL.

バージョン 3.10 で変更: It now returns NULL if called before *Py_Initialize()*.

wchar_t *Py_GetPrefix()

次に属します: 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 *Py_SetProgramName()* and some environment variables; for example, if the program name is '/usr/local/bin/python', the prefix is '/usr/local'. The returned string points into static storage; the caller should not modify its value. This corresponds to the **prefix** variable in the top-level Makefile and the --prefix argument to the **configure** script at build time. The value is available to Python code as sys.prefix. It is only useful on Unix. See also the next function.

This function should not be called before *Py_Initialize()*, otherwise it returns NULL.

バージョン 3.10 で変更: It now returns NULL if called before *Py_Initialize()*.

wchar_t *Py_GetExecPrefix()

次に属します: 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 *Py_SetProgramName()* and some environment variables; for example, if the program name is '/usr/local/bin/python', the exec-prefix is '/usr/local'. The returned string points into static storage; the caller should not modify its value. This corresponds to the **exec_prefix** variable in the top-level Makefile and the --exec-prefix argument to the **configure** script at build time. The value is available to Python code as sys.exec_prefix. It is only useful on Unix.

背景: プラットフォーム依存のファイル (実行形式や共有ライブラリ) が別のディレクトリツリー内にインストールされている場合、exec-prefix は prefix と異なります。典型的なインストール形態では、プラットフォーム非依存のファイルが /usr/local に収められる一方、プラットフォーム依存のファイルは /usr/local/plat サブツリーに収められます。

一般的に、プラットフォームとは、ハードウェアとソフトウェアファミリの組み合わせを指します。例えば、Solaris 2.x を動作させている Sparc マシンは全て同じプラットフォームであるとみなしますが、Solaris 2.x を動作させている Intel マシンは違うプラットフォームになりますし、同じ Intel マシンでも Linux を動作させているならまた別のプラットフォームです。一般的には、同じオペレーティングシステムでも、メジャーリビジョンの違うものは異なるプラットフォームです。非 Unix のオペレーティングシステムの場合は話はまた別です; 非 Unix のシステムでは、インストール方法はとても異なっていて、prefix や exec-prefix には意味がなく、空文字列が設定されています。コンパイル済みの Python バイトコードはプラットフォームに依存しないので注意してください (ただし、どのバージョンの Python でコンパイルされたかには依存します!)。

システム管理者は、**mount** や **automount** プログラムを使って、各プラットフォーム用の /usr/local/plat を異なったファイルシステムに置き、プラットフォーム間で /usr/local を共有するための設定

方法を知っているでしょう。

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

バージョン 3.10 で変更: It now returns `NULL` if called before *Py_Initialize()*.

wchar_t *`Py_GetProgramFullPath`()

次に属します: 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 *Py_SetProgramName()* above). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.executable`.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

バージョン 3.10 で変更: It now returns `NULL` if called before *Py_Initialize()*.

wchar_t *`Py_GetPath`()

次に属します: Stable ABI. Return the default module search path; this is computed from the program name (set by *Py_SetProgramName()* above) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is `':'` on Unix and macOS, `';'` on Windows. The returned string points into static storage; the caller should not modify its value. The list `sys.path` is initialized with this value on interpreter startup; it can be (and usually is) modified later to change the search path for loading modules.

This function should not be called before *Py_Initialize()*, otherwise it returns `NULL`.

バージョン 3.10 で変更: It now returns `NULL` if called before *Py_Initialize()*.

void `Py_SetPath`(const wchar_t*)

次に属します: Stable ABI *(バージョン 3.7 より)*. This API is kept for backward compatibility: setting *PyConfig.module_search_paths* and *PyConfig.module_search_paths_set* should be used instead, see *Python Initialization Configuration*.

Set the default module search path. If this function is called before *Py_Initialize()*, then *Py_GetPath()* won't attempt to compute a default search path but uses the one provided instead. This is useful if Python is embedded by an application that has full knowledge of the location of all modules. The path components should be separated by the platform dependent delimiter character, which is `':'` on Unix and macOS, `';'` on Windows.

This also causes `sys.executable` to be set to the program full path (see *Py_GetProgramFullPath()*) and for `sys.prefix` and `sys.exec_prefix` to be empty. It is up to the caller to modify these if required after calling *Py_Initialize()*.

Use *Py_DecodeLocale()* to decode a bytes string to get a `wchar_*` string.

The path argument is copied internally, so the caller may free it after the call completes.

バージョン 3.8 で変更: The program full path is now used for `sys.executable`, instead of the program name.

バージョン 3.11 で非推奨.

const char *Py_GetVersion()

次に属します: Stable ABI. Python インタプリタのバージョンを返します。バージョンは、次のような形式の文字列です

```
"3.0a5+ (py3k:63103M, May 12 2008, 00:53:55) \n[GCC 4.2.3]"
```

第一ワード (最初のスペース文字まで) は、現在の Python のバージョンです; 最初の文字は、ピリオドで区切られたメジャーバージョンとマイナーバージョンです。関数が返す文字列ポインタは静的な記憶領域を返します; 関数の呼び出し側はこの値を変更できません。この値は Python コードからは `sys.version` として利用できます。

See also the *Py_Version* constant.

const char *Py_GetPlatform()

次に属します: Stable ABI. 現在のプラットフォームのプラットフォーム識別文字列を返します。Unix では、オペレーティングシステムの ” 公式の” 名前を小文字に変換し、後ろにメジャーリビジョン番号を付けた構成になっています。例えば Solaris 2.x は、SunOS 5.x, としても知られていますが、`'sunos5'` になります。macOS では `'darwin'` です。Windows では `'win'` です。関数が返す文字列ポインタは静的な記憶領域を返します; 関数の呼び出し側はこの値を変更できません。この値は Python コードからは `sys.platform` として利用できます。

const char *Py_GetCopyright()

次に属します: Stable ABI. 現在の Python バージョンに対する公式の著作権表示文字列を返します。例えば

```
'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'
```

関数が返す文字列ポインタは静的な記憶領域を返します; 関数の呼び出し側はこの値を変更できません。この値は Python コードからは `sys.copyright` として利用できます。

const char *Py_GetCompiler()

次に属します: Stable ABI. 現在使っているバージョンの Python をビルドする際に用いたコンパイラを示す文字列を、角括弧で囲った文字列を返します。例えば:

```
"[GCC 2.7.2.2]"
```

関数が返す文字列ポインタは静的な記憶領域を返します; 関数の呼び出し側はこの値を変更できません。この値は Python コードからは `sys.version` の一部として取り出せます。

const char *Py_GetBuildInfo()

次に属します: Stable ABI. 現在使っている Python インタプリタインスタンスの、シーケンス番号とビルド日時に関する情報を返します。例えば

```
"#67, Aug  1 1997, 22:34:28"
```

関数が返す文字列ポインタは静的な記憶領域を返します; 関数の呼び出し側はこの値を変更できません。この値は Python コードからは `sys.version` の一部として取り出せます。

void **PySys_SetArgvEx**(int argc, wchar_t **argv, int updatepath)

次に属します: 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*.

argc および *argv* に基づいて `sys.argv` を設定します。これらの引数はプログラムの `main()` に渡した引数に似ていますが、最初の要素が Python インタプリタの宿主となっている実行形式の名前ではなく、実行されるスクリプト名を参照しなければならない点が違います。実行するスクリプトがない場合、*argv* の最初の要素は空文字列にしてもかまいません。この関数が `sys.argv` の初期化に失敗した場合、致命的エラーを *Py_FatalError()* で知らせます。

updatepath が 0 の場合、ここまでの動作がこの関数がすることの全てです。*updatepath* が 0 でない場合、この関数は `sys.path` を以下のアルゴリズムに基づいて修正します:

- 存在するスクリプトの名前が `argv[0]` に渡された場合、そのスクリプトがある場所の絶対パスを `sys.path` の先頭に追加します。
- それ以外の場合 (*argc* が `0` だったり、`argv[0]` が存在するファイル名を指していない場合)、`sys.path` の先頭に空の文字列を追加します。これは現在の作業ディレクトリ (`"."`) を先頭に追加するのと同じです。

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*.

注釈

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**.

3.1.3 より前のバージョンでは、*PySys_SetArgv()* の呼び出しが完了した後に `sys.path` の先頭の要素を取り出すことで、同じ効果が得られます。例えばこのように使います:

```
PyRun_SimpleString("import sys; sys.path.pop(0)\n");
```

Added in version 3.1.3.

バージョン 3.11 で非推奨.

void **PySys_SetArgv**(int argc, wchar_t **argv)

次に属します: Stable ABI. This API is kept for backward compatibility: setting *PyConfig.argv* and *PyConfig.parse_argv* should be used instead, see *Python Initialization Configuration*.

この関数は、**python** インタプリタが **-I** オプション付きで実行されている場合を除き *PySys_SetArgvEx()* の *updatepath* に 1 を設定したのと同じように動作します。

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*.

バージョン 3.4 で変更: *updatepath* の値は **-I** オプションに依存します。

バージョン 3.11 で非推奨.

void **Py_SetPythonHome**(const wchar_t *home)

次に属します: Stable ABI. This API is kept for backward compatibility: setting *PyConfig.home* should be used instead, see *Python Initialization Configuration*.

Python の標準ライブラリがある、デフォルトの "home" ディレクトリを設定します。引数の文字列の意味については **PYTHONHOME** を参照してください。

引数は静的なストレージに置かれてプログラム実行中に書き換えられないようなゼロ終端の文字列であるべきです。Python インタプリタはこのストレージの内容を変更しません。

Use *Py_DecodeLocale()* to decode a bytes string to get a **wchar_t*** string.

バージョン 3.11 で非推奨.

wchar_t ***Py_GetPythonHome**()

次に属します: Stable ABI. Return the default "home", that is, the value set by a previous call to *Py_SetPythonHome()*, or the value of the **PYTHONHOME** environment variable if it is set.

This function should not be called before *Py_Initialize()*, otherwise it returns **NULL**.

バージョン 3.10 で変更: It now returns **NULL** if called before *Py_Initialize()*.

9.5 スレッド状態 (thread state) とグローバルインタプリタロック (global interpreter lock)

The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there's a global lock, called the *global interpreter lock* or *GIL*, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

このため、*GIL* を獲得したスレッドだけが Python オブジェクトを操作したり、Python/C API 関数を呼び出したりできるというルールがあります。並行処理をエミュレートするために、インタプリタは定期的にロックを解放したり獲得したりします。(**sys.setswitchinterval**() を参照) このロックはブロックが起こりうる I/O 操作の付近でも解放・獲得され、I/O を要求するスレッドが I/O 操作の完了を待つ間、他のスレッドが動作できるようにしています。

The Python interpreter keeps some thread-specific bookkeeping information inside a data structure called *PyThreadState*. There's also one global variable pointing to the current *PyThreadState*: it can be retrieved using *PyThreadState_Get()*.

9.5.1 Releasing the GIL from extension code

Most extension code manipulating the *GIL* has the following simple structure:

```
Save the thread state in a local variable.
Release the global interpreter lock.
... Do some blocking I/O operation ...
Reacquire the global interpreter lock.
Restore the thread state from the local variable.
```

この構造は非常に一般的なので、作業を単純にするために 2 つのマクロが用意されています:

```
Py_BEGIN_ALLOW_THREADS
... Do some blocking I/O operation ...
Py_END_ALLOW_THREADS
```

Py_BEGIN_ALLOW_THREADS マクロは新たなブロックを開始し、隠しローカル変数を宣言します; *Py_END_ALLOW_THREADS* はブロックを閉じます。

上のブロックは次のコードに展開されます:

```
PyThreadState *_save;

_save = PyEval_SaveThread();
... Do some blocking I/O operation ...
PyEval_RestoreThread(_save);
```

Here is how these functions work: the global interpreter lock is used to protect the pointer to the current thread state. When releasing the lock and saving the thread state, the current thread state pointer must be retrieved before the lock is released (since another thread could immediately acquire the lock and store its own thread state in the global variable). Conversely, when acquiring the lock and restoring the thread state, the lock must be acquired before storing the thread state pointer.

注釈

Calling system I/O functions is the most common use case for releasing the GIL, but it can also be useful before calling long-running computations which don't need access to Python objects, such as compression or cryptographic functions operating over memory buffers. For example, the standard `zlib` and `hashlib` modules release the GIL when compressing or hashing data.

9.5.2 Python 以外で作られたスレッド

When threads are created using the dedicated Python APIs (such as the `threading` module), a thread state is automatically associated to them and the code showed above is therefore correct. However, when threads are created from C (for example by a third-party library with its own thread management), they don't hold the GIL, nor is there a thread state structure for them.

If you need to call Python code from these threads (often this will be part of a callback API provided by the aforementioned third-party library), you must first register these threads with the interpreter by creating a thread state data structure, then acquiring the GIL, and finally storing their thread state pointer, before you can start using the Python/C API. When you are done, you should reset the thread state pointer, release the GIL, and finally free the thread state data structure.

PyGILState_Ensure() と *PyGILState_Release()* はこの処理を自動的に行います。C のスレッドから Python を呼び出す典型的な方法は以下のとおりです:

```
PyGILState_STATE gstate;
gstate = PyGILState_Ensure();

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result or handle exception */

/* Release the thread. No Python API allowed beyond this point. */
PyGILState_Release(gstate);
```

Note that the `PyGILState_*` functions assume there is only one global interpreter (created automatically by *Py_Initialize()*). Python supports the creation of additional interpreters (using *Py_NewInterpreter()*), but mixing multiple interpreters and the `PyGILState_*` API is unsupported.

9.5.3 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.

9.5.4 高レベル API

C 拡張を書いたり Python インタプリタを埋め込むときに最も一般的に使われる型や関数は次のとおりです:

type **PyInterpreterState**

次に属します: Limited API *(**不透明な構造体として**)*. このデータ構造体は、協調動作する多数のスレッド間で共有されている状態を表現します。同じインタプリタに属するスレッドはモジュール管理情報やその他いくつかの内部的な情報を共有しています。この構造体には公開 (public) のメンバはありません。

異なるインタプリタに属するスレッド間では、利用可能なメモリ、開かれているファイルデスクリプタなどといったプロセス状態を除いて、初期状態では何も共有されていません。GIL もまた、スレッドがどのインタプリタに属しているかに関わらずすべてのスレッドで共有されています。

type **PyThreadState**

次に属します: Limited API *(**不透明な構造体として**)*. 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**()

次に属します: Stable ABI. Deprecated function which does nothing.

In Python 3.6 and older, this function created the GIL if it didn't exist.

バージョン 3.9 で変更: The function now does nothing.

バージョン 3.7 で変更: この関数は *Py_Initialize()* から呼び出されるようになり、わざわざ呼び出す必要はもう無くなりました。

バージョン 3.2 で変更: この関数は *Py_Initialize()* より前に呼び出すことができなくなりました。

バージョン 3.9 で非推奨.

int **PyEval_ThreadsInitialized**()

次に属します: Stable ABI. Returns a non-zero value if *PyEval_InitThreads()* has been called. This function can be called without holding the GIL, and therefore can be used to avoid calls to the locking API when running single-threaded.

バージョン 3.7 で変更: The *GIL* is now initialized by *Py_Initialize()*.

バージョン 3.9 で非推奨.

PyThreadState ***PyEval_SaveThread**()

次に属します: Stable ABI. Release the global interpreter lock (if it has been created) and reset the thread state to `NULL`, returning the previous thread state (which is not `NULL`). If the lock has been created, the current thread must have acquired it.

void **PyEval_RestoreThread**(*PyThreadState* *tstate)

次に属します: Stable ABI. Acquire the global interpreter lock (if it has been created) and set the thread state to *tstate*, which must not be `NULL`. If the lock has been created, the current thread must not have acquired it, otherwise deadlock ensues.

> **注釈**
>
> Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use `_Py_IsFinalizing()` or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

PyThreadState ***PyThreadState_Get**()

次に属します: Stable ABI. Return the current thread state. The global interpreter lock must be held. When the current thread state is `NULL`, this issues a fatal error (so that the caller needn't check for `NULL`).

PyThreadState ***PyThreadState_Swap**(*PyThreadState* *tstate)

次に属します: Stable ABI. Swap the current thread state with the thread state given by the argument *tstate*, which may be `NULL`. The global interpreter lock must be held and is not released.

以下の関数はスレッドローカルストレージを利用していて、サブインタプリタとの互換性がありません:

PyGILState_STATE **PyGILState_Ensure**()

次に属します: 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 global interpreter lock. This may be called as many times as desired by a thread as long as each call is matched with a call to *PyGILState_Release()*. In general, other thread-related APIs may be used between *PyGILState_Ensure()* and *PyGILState_Release()* calls as long as the thread state is restored to its previous state before the Release(). For example, normal usage of the *Py_BEGIN_ALLOW_THREADS* and *Py_END_ALLOW_THREADS* macros is acceptable.

The return value is an opaque "handle" to the thread state when *PyGILState_Ensure()* was called, and must be passed to *PyGILState_Release()* to ensure Python is left in the same state. Even though recursive calls are allowed, these handles *cannot* be shared - each unique call to *PyGILState_Ensure()* must save the handle for its call to *PyGILState_Release()*.

When the function returns, the current thread will hold the GIL and be able to call arbitrary Python code. Failure is a fatal error.

> **注釈**
>
> Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use `_Py_IsFinalizing()` or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

void **PyGILState_Release**(PyGILState_STATE)

次に属します: Stable ABI. 獲得したすべてのリソースを解放します。この関数を呼び出すと、Python の状態は対応する *PyGILState_Ensure()* を呼び出す前と同じとなります (通常、この状態は呼び出し元でははわかりませんので、GILState API を利用するようにしてください)。

PyGILState_Ensure() を呼び出す場合は、必ず同一スレッド内で対応する *PyGILState_Release()* を呼び出してください。

PyThreadState ***PyGILState_GetThisThreadState**()

次に属します: Stable ABI. Get the current thread state for this thread. May return `NULL` if no GILState API has been used on the current thread. Note that the main thread always has such a thread-state, even if no auto-thread-state call has been made on the main thread. This is mainly a helper/diagnostic function.

int **PyGILState_Check**()

Return `1` if the current thread is holding the GIL and `0` otherwise. This function can be called from any thread at any time. Only if it has had its Python thread state initialized and currently is holding the GIL will it return `1`. This is mainly a helper/diagnostic function. It can be useful for example in callback contexts or memory allocation functions when knowing that the GIL is locked can allow the caller to perform sensitive actions or otherwise behave differently.

Added in version 3.4.

以下のマクロは、通常末尾にセミコロンを付けずに使います; Python ソース配布物内の使用例を見てください。

Py_BEGIN_ALLOW_THREADS

次に属します: Stable ABI. このマクロを展開すると `{ PyThreadState *_save; _save = PyEval_SaveThread();` になります。マクロに開き波括弧が入っていることに注意してください; この波括弧は後で *Py_END_ALLOW_THREADS* マクロと対応させなければなりません。マクロについての詳しい議論は上記を参照してください。

Py_END_ALLOW_THREADS

次に属します: Stable ABI. このマクロを展開すると `PyEval_RestoreThread(_save); }` になります。マクロに開き波括弧が入っていることに注意してください; この波括弧は事前の *Py_BEGIN_ALLOW_THREADS* マクロと対応していなければなりません。マクロについての詳しい議論は上記を参照してください。

Py_BLOCK_THREADS

次に属します: Stable ABI. このマクロを展開すると `PyEval_RestoreThread(_save);` になります: 閉じ波括弧のない *Py_END_ALLOW_THREADS* と同じです。

Py_UNBLOCK_THREADS

次に属します: Stable ABI. このマクロを展開すると `_save = PyEval_SaveThread();` になります: 開き波括弧のない *Py_BEGIN_ALLOW_THREADS* と同じです。

9.5.5 低レベル API

次の全ての関数は *Py_Initialize()* の後に呼び出さなければなりません。

バージョン 3.7 で変更: *Py_Initialize()* now initializes the *GIL*.

PyInterpreterState ***PyInterpreterState_New()**

次に属します: Stable ABI. Create a new interpreter state object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

引数無しで 監査イベント `cpython.PyInterpreterState_New` を送出します。

void **PyInterpreterState_Clear**(*PyInterpreterState* *interp)

次に属します: Stable ABI. Reset all information in an interpreter state object. The global interpreter lock must be held.

引数無しで 監査イベント `cpython.PyInterpreterState_Clear` を送出します。

void **PyInterpreterState_Delete**(*PyInterpreterState* *interp)

次に属します: Stable ABI. Destroy an interpreter state object. The global interpreter lock need not be held. The interpreter state must have been reset with a previous call to *PyInterpreterState_Clear()*.

PyThreadState ***PyThreadState_New**(*PyInterpreterState* *interp)

次に属します: Stable ABI. Create a new thread state object belonging to the given interpreter object. The global interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

void **PyThreadState_Clear**(*PyThreadState* *tstate)

次に属します: Stable ABI. Reset all information in a thread state object. The global interpreter lock must be held.

バージョン 3.9 で変更: This function now calls the `PyThreadState.on_delete` callback. Previously, that happened in *PyThreadState_Delete()*.

void **PyThreadState_Delete**(*PyThreadState* *tstate)

次に属します: Stable ABI. Destroy a thread state object. The global interpreter lock need not be held. The thread state must have been reset with a previous call to *PyThreadState_Clear()*.

void **PyThreadState_DeleteCurrent**(void)

Destroy the current thread state and release the global interpreter lock. Like *PyThreadState_Delete()*, the global interpreter lock must be held. The thread state must have been reset with a previous call to *PyThreadState_Clear()*.

PyFrameObject ***PyThreadState_GetFrame**(*PyThreadState* *tstate)

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. 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`.

Added in version 3.9.

uint64_t **PyThreadState_GetID**(*PyThreadState* *tstate)

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. Get the unique thread state identifier of the Python thread state *tstate*.

tstate must not be `NULL`.

Added in version 3.9.

PyInterpreterState ***PyThreadState_GetInterpreter**(*PyThreadState* *tstate)

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. Get the interpreter of the Python thread state *tstate*.

tstate must not be `NULL`.

Added in 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.

Added in 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.

Added in version 3.11.

PyInterpreterState ***PyInterpreterState_Get**(void)

次に属します: Stable ABI *(**バージョン** 3.9 **より**)*. Get the current interpreter.

Issue a fatal error if there no current Python thread state or no current interpreter. It cannot return NULL.

The caller must hold the GIL.

Added in version 3.9.

int64_t **PyInterpreterState_GetID**(*PyInterpreterState* *interp)

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. インタプリタの一意な ID を返します。処理中に何かエラーが起きたら、`-1` が返され、エラーがセットされます。

The caller must hold the GIL.

Added in version 3.7.

PyObject ***PyInterpreterState_GetDict**(*PyInterpreterState* *interp)

次に属します: Stable ABI *(**バージョン** 3.8 **より**)*. インタプリタ固有のデータを保持している辞書を返します。この関数が `NULL` を返した場合は、ここまでで例外は送出されておらず、呼び出し側はインタプリタ固有の辞書は利用できないと考えなければなりません。

この関数は *PyModule_GetState()* を置き換えるものではなく、拡張モジュールがインタプリタ固有の状態情報を格納するのに使うべきものです。

Added in 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.

バージョン 3.9 で変更: The function now takes a *tstate* parameter.

バージョン 3.11 で変更: 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".

Added in 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".

Added in version 3.9.

PyObject *PyThreadState_GetDict()

戻り値: **借用参照。次に属します**: 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 current thread state is available. If this function returns NULL, no exception has been raised and the caller should assume no current thread state is available.

int PyThreadState_SetAsyncExc(unsigned long id, *PyObject* *exc)

次に属します: 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 the GIL held. Returns the number of thread states modified; this is normally one, but will be zero if the thread id isn't found. If *exc* is NULL, the pending exception (if any) for the thread is cleared. This raises no exceptions.

バージョン 3.7 で変更: *id* 引数の型が long から unsigned long へ変更されました。

void PyEval_AcquireThread(*PyThreadState* *tstate)

次に属します: Stable ABI. Acquire the global interpreter lock and set the current thread state to *tstate*, which must not be NULL. The lock must have been created earlier. If this thread already has the lock, deadlock ensues.

注釈

Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use _Py_IsFinalizing() or sys.is_finalizing() to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

バージョン 3.8 で変更: Updated to be consistent with *PyEval_RestoreThread()*, *Py_END_ALLOW_THREADS()*, and *PyGILState_Ensure()*, and terminate the current thread if called while the interpreter is finalizing.

PyEval_RestoreThread() はいつでも (スレッドが初期化されたいないときでも) 利用可能な高レベル関数です。

void PyEval_ReleaseThread(*PyThreadState* *tstate)

次に属します: Stable ABI. Reset the current thread state to NULL and release the global interpreter lock. The lock must have been created earlier and must be held by the current thread. The *tstate* argument, which must not be NULL, is only used to check that it represents the current thread state --- if it isn't, a fatal error is reported.

PyEval_SaveThread() はより高レベルな関数で常に (スレッドが初期化されていないときでも) 利用できます。

void **PyEval_AcquireLock**()

次に属します: Stable ABI. Acquire the global interpreter lock. The lock must have been created earlier. If this thread already has the lock, a deadlock ensues.

バージョン 3.2 で非推奨: This function does not update the current thread state. Please use *PyEval_RestoreThread()* or *PyEval_AcquireThread()* instead.

> **注釈**
>
> Calling this function from a thread when the runtime is finalizing will terminate the thread, even if the thread was not created by Python. You can use `_Py_IsFinalizing()` or `sys.is_finalizing()` to check if the interpreter is in process of being finalized before calling this function to avoid unwanted termination.

バージョン 3.8 で変更: Updated to be consistent with *PyEval_RestoreThread()*, *Py_END_ALLOW_THREADS()*, and *PyGILState_Ensure()*, and terminate the current thread if called while the interpreter is finalizing.

void **PyEval_ReleaseLock**()

次に属します: Stable ABI. Release the global interpreter lock. The lock must have been created earlier.

バージョン 3.2 で非推奨: This function does not update the current thread state. Please use *PyEval_SaveThread()* or *PyEval_ReleaseThread()* instead.

9.6 サブインタプリタサポート

ほとんどの場合は埋め込む Python インタプリタは 1 つだけですが、いくつかの場合に同一プロセス内、あるいは同一スレッド内で、複数の独立したインタプリタを作成する必要があります。これを可能にするのがサブインタプリタです。

”メイン” インタプリタとは、ランタイムが初期化を行ったときに最初に作成されたインタプリタのことです。サブインタプリタと違い、メインインタプリタにはシグナルハンドリングのような、プロセス全域で唯一な責務があります。メインインタプリタにはランタイムの初期化中の処理実行という責務もあり、通常はランタイムの終了処理中に動いているランタイムでもあります。*PyInterpreterState_Main()* 関数は、メインインタプリタの状態へのポインタを返します。

サブインタプリタを切り替えが *PyThreadState_Swap()* 関数でできます。次の関数を使ってサブインタプリタの作成と削除が行えます:

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.

Added in version 3.12.

構造体フィールド:

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)

新しいサブインタプリタ (sub-interpreter) を生成します。サブインタプリタとは、(ほぼ完全に) 個別に分割された Python コードの実行環境です。特に、新しいサブインタプリタは、import されるモジュール全てについて個別のバージョンを持ち、これには基盤となるモジュール builtins, __main__ および sys も含まれます。ロード済みのモジュールからなるテーブル (sys.modules) およびモジュール検索パス (sys.path) もサブインタプリタ毎に別個のものになります。新たなサブインタプリタ環境には sys.argv 変数がありません。また、サブインタプリタは新たな標準 I/O ストリーム sys.stdin, sys.stdout, sys.stderr を持ちます (とはいえ、これらのストリームは根底にある同じファイル記述子を参照しています)。

The given *config* controls the options with which the interpreter is initialized.

Upon success, *tstate_p* will be set to the first thread state created in the new sub-interpreter. This thread state is made in the current thread state. Note that no actual thread is created; see the discussion of thread states below. If creation of the new interpreter is unsuccessful, *tstate_p* is set to NULL; no exception is set since the exception state is stored in the current thread state and there may not be a current thread state.

Like all other Python/C API functions, the global interpreter lock must be held before calling this function and is still held when it returns. Likewise a current thread state must be set on entry. On success, the returned thread state will be set as current. If the sub-interpreter is created with its own GIL then the GIL of the calling interpreter will be released. When the function returns, the new interpreter's GIL will be held by the current thread and the previously interpreter's GIL will remain released here.

Added in 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,
};
PyThreadState *tstate = NULL;
PyStatus status = Py_NewInterpreterFromConfig(&tstate, &config);
if (PyStatus_Exception(status)) {
```

(次のページに続く)

(前のページからの続き)

```
    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)

次に属します: 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)

次に属します: Stable ABI. Destroy the (sub-)interpreter represented by the given thread state. The given thread state must be the current thread state. See the discussion of thread states below. When the call returns, the current thread state is `NULL`. All thread states associated with this interpreter are destroyed. The global interpreter lock used by the target interpreter must be held before calling this function. No GIL is held when it returns.

Py_FinalizeEx() will destroy all sub-interpreters that haven't been explicitly destroyed at that point.

9.6.1 A Per-Interpreter GIL

Using *Py_NewInterpreterFromConfig()* you can create a sub-interpreter that is completely isolated from other interpreters, including having its own GIL. The most important benefit of this isolation is that such an interpreter can execute Python code without being blocked by other interpreters or blocking any others. Thus a single Python process can truly take advantage of multiple CPU cores when running Python code. The isolation also encourages a different approach to concurrency than that of just using threads. (See **PEP 554**.)

Using an isolated interpreter requires vigilance in preserving that isolation. That especially means not sharing any objects or mutable state without guarantees about thread-safety. Even objects that are otherwise immutable (e.g. `None`, `(1, 5)`) can't normally be shared because of the refcount. One simple but less-efficient approach around this is to use a global lock around all use of some state (or object). Alternately, effectively immutable objects (like integers or strings) can be made safe in spite of their refcounts by making them "immortal". In fact, this has been done for the builtin singletons, small integers, and a number of other builtin objects.

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.

Added in version 3.12.

9.6.2 バグと注意事項

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.

サブインタプリタを `PyGILState_*` API と組み合わせるのが難しいことにも注意してください。これらの API は Python のスレッド状態と OS レベルスレッドが 1 対 1 で対応していることを前提にしていて、サブインタプリタが存在するとその前提が崩れるからです。対応する *PyGILState_Ensure()* と *PyGILState_Release()* の呼び出しのペアの間では、サブインタプリタの切り替えを行わないことを強く推奨します。さらに、(`ctypes` のような) これらの API を使って Python の外で作られたスレッドから Python

コードを実行している拡張モジュールはサブインタプリタを使うと壊れる可能性があります。

9.7 非同期通知

インタプリタのメインスレッドに非同期な通知を行うために提供されている仕組みです。これらの通知は関数ポインタと void ポインタ引数という形態を取ります。

int **Py_AddPendingCall**(int (*func)(void*), void *arg)

次に属します: Stable ABI. インタプリタのメインスレッドから関数が呼び出される予定を組みます。成功すると `0` が返り、*func* はメインスレッドの呼び出しキューに詰められます。失敗すると、例外をセットせずに `-1` が返ります。

無事にキューに詰められると、*func* は **いつかは必ず** インタプリタのメインスレッドから、*arg* を引数として呼び出されます。この関数は、通常の実行中の Python コードに対して非同期に呼び出されますが、次の両方の条件に合致したときに呼び出されます:

- *bytecode* 境界上にいるとき、
- with the main thread holding the *global interpreter lock* (*func* can therefore use the full C API).

func must return `0` on success, or `-1` on failure with an exception set. *func* won't be interrupted to perform another asynchronous notification recursively, but it can still be interrupted to switch threads if the global interpreter lock is released.

This function doesn't need a current thread state to run, and it doesn't need the global interpreter lock.

To call this function in a subinterpreter, the caller must hold the GIL. Otherwise, the function *func* can be scheduled to be called from the wrong interpreter.

> **警告**
>
> これは、非常に特別な場合にのみ役立つ、低レベルな関数です。*func* が可能な限り早く呼び出される保証はありません。メインスレッドがシステムコールを実行するのに忙しい場合は、*func* はシステムコールが返ってくるまで呼び出されないでしょう。この関数は一般的には、任意の C スレッドから Python コードを呼び出すのには **向きません** 。これの代わりに、*PyGILState API* を使用してください。

Added in version 3.1.

バージョン 3.9 で変更: If this function is called in a subinterpreter, the function *func* is now scheduled to be called from the subinterpreter, rather than being called from the main interpreter. Each subinterpreter now has its own list of scheduled calls.

9.8 プロファイルとトレース (profiling and tracing)

Python インタプリタは、プロファイル: 分析 (profile) や実行のトレース: 追跡 (trace) といった機能を組み込むために低水準のサポートを提供しています。このサポートは、プロファイルやデバッグ、適用範囲分析 (coverage analysis) ツールなどに使われます。

この C インターフェースは、プロファイルやトレース作業時に、Python レベルの呼び出し可能オブジェクトが呼び出されることによるオーバヘッドを避け、直接 C 関数呼び出しが行えるようにしています。プロファイルやトレース機能の本質的な特性は変わっていません; インターフェースではトレース関数をスレッドごとにインストールでき、トレース関数に報告される基本イベント (basic event) は以前のバージョンにおいて Python レベルのトレース関数で報告されていたものと同じです。

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*:

what の値	*arg* の意味
PyTrace_CALL	常に *Py_None* 。
PyTrace_EXCEPTION	`sys.exc_info()` の返す例外情報です。
PyTrace_LINE	常に *Py_None* 。
PyTrace_RETURN	呼び出し側に返される予定の値か、例外によって関数を抜ける場合は `NULL` です。
PyTrace_C_CALL	呼び出される関数オブジェクト。
PyTrace_C_EXCEPTION	呼び出される関数オブジェクト。
PyTrace_C_RETURN	呼び出される関数オブジェクト。
PyTrace_OPCODE	常に *Py_None* 。

int **PyTrace_CALL**

関数やメソッドが新たに呼び出されたり、ジェネレータが新たなエントリの処理に入ったことを報告する際の、*Py_tracefunc* の *what* の値です。イテレータやジェネレータ関数の生成は、対応するフレーム内の Python バイトコードに制御の委譲 (control transfer) が起こらないため報告されないので注意してください。

int **PyTrace_EXCEPTION**

例外が送出された際の *Py_tracefunc* の *what* の値です。現在実行されているフレームで例外がセットされ、何らかのバイトコードが処理された後に、*what* にこの値がセットされた状態でコールバック関数が呼び出されます。この結果、例外の伝播によって Python が呼び出しスタックを逆戻りする際に、各フレームから処理が戻るごとにコールバック関数が呼び出されます。トレース関数だけがこれらのイベントを受け取ります; プロファイラはこの種のイベントを必要としません。

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**

呼び出しが返るときに *Py_tracefunc* 関数に *what* 引数として渡す値です。

int **PyTrace_C_CALL**

C 関数を呼び出す直前に *Py_tracefunc* 関数の *what* 引数として渡す値です。

int **PyTrace_C_EXCEPTION**

C 関数が例外を送出したときに *Py_tracefunc* 関数の *what* 引数として渡す値です。

int **PyTrace_C_RETURN**

C 関数から戻るときに *Py_tracefunc* 関数の *what* 引数として渡す値です。

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*.

`sys.setprofile()` 関数も参照してください。

The caller must hold the *GIL*.

void **PyEval_SetProfileAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

Like *PyEval_SetProfile()* but sets the profile function in all running threads belonging to the current interpreter instead of the setting it only on the current thread.

The caller must hold the *GIL*.

As *PyEval_SetProfile()*, this function ignores any exceptions raised while setting the profile functions in all threads.

Added in 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.

`sys.settrace()` 関数も参照してください。

The caller must hold the *GIL*.

void **PyEval_SetTraceAllThreads**(*Py_tracefunc* func, *PyObject* *obj)

Like *PyEval_SetTrace()* but sets the tracing function in all running threads belonging to the current interpreter instead of the setting it only on the current thread.

The caller must hold the *GIL*.

As *PyEval_SetTrace()*, this function ignores any exceptions raised while setting the trace functions in all threads.

Added in version 3.12.

9.9 高度なデバッガサポート (advanced debugger support)

以下の関数は高度なデバッグツールでの使用のためだけのものです。

PyInterpreterState ***PyInterpreterState_Head**()

インタプリタ状態オブジェクトからなるリストのうち、先頭にあるものを返します。

PyInterpreterState ***PyInterpreterState_Main**()

メインインタプリタの状態オブジェクトを返します。

PyInterpreterState ***PyInterpreterState_Next**(*PyInterpreterState* *interp)

インタプリタ状態オブジェクトからなるリストのうち、*interp* の次にあるものを返します。

PyThreadState ***PyInterpreterState_ThreadHead**(*PyInterpreterState* *interp)

インタプリタ *interp* に関連付けられているスレッドからなるリストのうち、先頭にある *PyThreadState* オブジェクトを返します。

PyThreadState ***PyThreadState_Next**(*PyThreadState* *tstate)

tstate と同じ *PyInterpreterState* オブジェクトに属しているスレッド状態オブジェクトのうち、*tstate* の次にあるものを返します。

9.10 スレッドローカルストレージのサポート

Python インタプリタは、スレッドローカルストレージ (thread-local storage, TLS) の低レベルサポートを提供していて、ネイティブの TLS 実装を内部にラップして Python レベルのスレッドローカルストレージ API (`threading.local`) をサポートしています。CPython の C レベル API は pthreads や Windows で与えられる TLS と同様です: スレッドキーとスレッドごとに `void*` 値を関係付ける関数を使います。

The GIL does *not* need to be held when calling these functions; they supply their own locking.

Python.h は TLS API の宣言を include せず、スレッドローカルストレージを使うには pythread.h を include する必要があることに注意してください。

注釈

この API 関数はどれも void* 値の代わりにメモリ管理を行うことはしません。メモリの確保と解放は自前で行う必要があります。void* 値がたまたま *PyObject** だった場合は、API 関数はそれぞれの値の参照カウントの操作は行いません。

9.10.1 スレッド固有ストレージ (Thread Specific Storage, TSS) API

TSS API は、CPython インタプリタに含まれている既存の TLS API を置き換えるために導入されました。この API は、スレッドキーの表現に int の代わりに新しい型 *Py_tss_t* を使います。

Added in version 3.7.

参考

”CPython のスレッドローカルストレージのための新しい C API” (**PEP 539**)

type **Py_tss_t**

このデータ構造体はスレッドキーの状態を表現しています。この構造体の定義は、根底の TLS 実装に依存し、キーの初期化状態を表現する内部フィールドを持ちます。この構造体には公開 (public) のメンバはありません。

Py_LIMITED_API が定義されていないときは、この型の *Py_tss_NEEDS_INIT* による静的メモリ確保ができます。

Py_tss_NEEDS_INIT

このマクロは *Py_tss_t* 変数の初期化子に展開されます。このマクロは *Py_LIMITED_API* があるときは定義されません。

動的メモリ確保

動的な *Py_tss_t* のメモリ確保は *Py_LIMITED_API* でビルドされた拡張モジュールで必要になりますが、その実装がビルド時に不透明なために、この型の静的なメモリ確保は不可能です。

Py_tss_t ***PyThread_tss_alloc**()

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. 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)

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. 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.

> **注釈**
>
> A freed key becomes a dangling pointer. You should reset the key to `NULL`.

メソッド

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)

次に属します: Stable ABI *(バージョン 3.7 より)*. 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)

次に属します: Stable ABI *(バージョン 3.7 より)*. 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)

次に属します: Stable ABI *(バージョン 3.7 より)*. 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)

次に属します: Stable ABI *(バージョン 3.7 より)*. 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)

次に属します: Stable ABI *(バージョン 3.7 より)*. Return the `void*` value associated with a TSS key in the current thread. This returns `NULL` if no value is associated with the key in the current thread.

9.10.2 スレッドローカルストレージ (TLS) API

バージョン 3.7 で非推奨: This API is superseded by *Thread Specific Storage (TSS) API*.

> **注釈**
>
> 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.

前述の互換性の問題により、このバージョンの API は新規のコードで利用すべきではありません。

int **PyThread_create_key**()

次に属します: Stable ABI.

void **PyThread_delete_key**(int key)

次に属します: Stable ABI.

int **PyThread_set_key_value**(int key, void *value)

次に属します: Stable ABI.

void ***PyThread_get_key_value**(int key)

次に属します: Stable ABI.

void **PyThread_delete_key_value**(int key)

次に属します: Stable ABI.

void **PyThread_ReInitTLS**()

次に属します: Stable ABI.

第

TEN

PYTHON 初期化設定

Added in version 3.8.

Python は *Py_InitializeFromConfig()* と *PyConfig* 構造体を使って初期化できます。*Py_PreInitialize()* と *PyPreConfig* 構造体によって事前に初期化できます。

設定には二つの種類があります:

- *Python Configuration* は、通常の Python と同じ振る舞いをするカスタマイズされた Python を構築するために使用されます。例えば、環境変数やコマンドライン引数が Python を設定するために使用されます。
- 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.

Initialization, Finalization, and Threads も参照してください。

参考

PEP 587 "Python 初期化設定"

10.1 使用例

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;
```

(次のページに続く)

(前のページからの続き)

```
    /* 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);
}
```

10.2 PyWideStringList

type **PyWideStringList**

`wchar_t*` 文字列のリスト。

length が非ゼロの場合は、*items* は非 `NULL` かつすべての文字列は非 `NULL` でなければなりません。

メソッド:

PyStatus **PyWideStringList_Append**(*PyWideStringList* *list, const wchar_t *item)

item を *list* に追加します。

Python must be preinitialized to call this function.

PyStatus **PyWideStringList_Insert**(*PyWideStringList* *list, *Py_ssize_t* index, const wchar_t *item)

item を *list* の *index* の位置に挿入します。

index が *list* の長さ以上の場合、*item* を *list* の末尾に追加します。

index must be greater than or equal to `0`.

Python must be preinitialized to call this function.

構造体フィールド:

Py_ssize_t **length**

リストの長さ。

wchar_t ***items**

リストの要素。

10.3 PyStatus

type **PyStatus**

初期化関数のステータス (成功、エラー、終了) を格納する構造体です。

エラー時には、エラーを生成した C 関数の名前を格納できます。

構造体フィールド:

int **exitcode**

終了コード。`exit()` の引数として渡されます。

const char ***err_msg**

エラーメッセージ。

const char ***func**

エラーを生成した関数の名前で、`NULL` になりえます。

ステータスを生成する関数:

PyStatus **PyStatus_Ok**(void)

成功。

PyStatus **PyStatus_Error**(const char *err_msg)

メッセージとともにエラーを初期化します。

err_msg must not be `NULL`.

PyStatus **PyStatus_NoMemory**(void)

メモリ割り当ての失敗 (メモリ不足)。

PyStatus **PyStatus_Exit**(int exitcode)

指定した終了コードで Python を終了します。

ステータスを扱う関数:

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.

注釈

Internally, Python uses macros which set `PyStatus.func`, whereas functions to create a status set `func` to `NULL`.

以下はプログラム例です:

```
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;
}
```

10.4 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*.

構造体フィールド:

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_PYMALLOC` and `PYMEM_ALLOCATOR_PYMALLOC_DEBUG` are not supported if Python is `configured using --without-pymalloc`.

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.

10.5 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();
```

10.6 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.

バージョン 3.10 で変更: 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.

バージョン 3.11 で変更: *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()`.

構造体フィールド:

PyWideStringList **argv**

Command line arguments: `sys.argv`.

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.

Added in version 3.11.

wchar_t ***base_exec_prefix**

`sys.base_exec_prefix`.

Default: `NULL`.

Part of the *Python Path Configuration* output.

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.

wchar_t *base_prefix

sys.base_prefix.

Default: NULL.

Part of the *Python Path Configuration* output.

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.

Added in 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.

Added in 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.

Need a special build of Python with the Py_TRACE_REFS macro defined: see the configure --with-trace-refs option.

Default: 0.

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.

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.

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.

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**

Python home directory.

If *Py_SetPythonHome()* has been called, use its argument if it is not `NULL`.

Set by the `PYTHONHOME` environment variable.

Default: `NULL`.

Part of the *Python Path Configuration* input.

int **import_time**

If non-zero, profile import time.

Set the `1` by the `-X importtime` option and the `PYTHONPROFILEIMPORTTIME` environment variable.

Default: `0`.

int **inspect**

Enter interactive mode after executing a script or a command.

If greater than `0`, enable inspect: when a script is passed as first argument or the -c option is used, enter interactive mode after executing the script or the command, even when `sys.stdin` does not appear to be a terminal.

Incremented by the `-i` command line option. Set to `1` if the `PYTHONINSPECT` environment variable is non-empty.

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.

Added in version 3.12.

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.

PYTHONLEGACYWINDOWSSTDIO 環境変数が空でない文字列に設定された場合に、1 に設定されます。

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.

Added in version 3.9.

バージョン 3.11 で変更: 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.

Added in 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.

バージョン 3.10 で変更: 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.

Need 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.

バージョン 3.11 で変更: 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.

wchar_t ***program_name**

Program name used to initialize *executable* and in early error messages during Python initialization.

- If `Py_SetProgramName()` has been called, use its argument.
- 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.

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`.

int **show_ref_count**

Show total reference count at exit (excluding immortal objects)?

Set to `1` by `-X showrefcount` command line option.

Need 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).

If *Py_SetStandardStreamEncoding()* has been called, use its *error* and *errors* arguments if they are not NULL.

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.

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 compatibility mode with the perf profiler?

If non-zero, initialize the perf trampoline. See perf_profiling for more information.

Set by -X perf command line option and by the PYTHONPERFSUPPORT environment variable.

Default: `-1`.

Added in version 3.12.

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 **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.

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.

バージョン 3.9 で変更: The show_alloc_count field has been removed.

10.7 Initialization with PyConfig

Function to initialize Python:

PyStatus **Py_InitializeFromConfig**(const *PyConfig* *config)

Initialize Python from *config* configuration.

The caller is responsible to handle exceptions (error or exit) using *PyStatus_Exception()* and *Py_ExitStatusException()*.

If *PyImport_FrozenModules()*, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* are used, they must be set or called after Python preinitialization and before the Python initialization. If Python is initialized multiple times, *PyImport_AppendInittab()* or *PyImport_ExtendInittab()* must be called before each Python initialization.

The current configuration (PyConfig type) is stored in PyInterpreterState.config.

Example setting the program name:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name. Implicitly preinitialize Python. */
    status = PyConfig_SetString(&config, &config.program_name,
                                L"/path/to/my_program");
    if (PyStatus_Exception(status)) {
```

(次のページに続く)

(前のページからの続き)

```
        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;
    }
```

(次のページに続く)

(前のページからの続き)

```
    /* Specify sys.path explicitly */
    /* If you want to modify the default set of paths, finish
       initialization first and then use PySys_GetObject("path") */
    config.module_search_paths_set = 1;
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/stdlib");
    if (PyStatus_Exception(status)) {
        goto done;
    }
    status = PyWideStringList_Append(&config.module_search_paths,
                                     L"/path/to/more/modules");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Override executable computed by PyConfig_Read() */
    status = PyConfig_SetString(&config, &config.executable,
                                L"/path/to/my_executable");
    if (PyStatus_Exception(status)) {
        goto done;
    }

    status = Py_InitializeFromConfig(&config);

done:
    PyConfig_Clear(&config);
    return status;
}
```

10.8 Isolated Configuration

PyPreConfig_InitIsolatedConfig() and *PyConfig_InitIsolatedConfig()* functions create a configuration to isolate Python from the system. For example, to embed Python into an application.

This configuration ignores global configuration variables, environment variables, command line arguments (*PyConfig.argv* is not parsed) and user site directory. The C standard streams (ex: `stdout`) and the LC_CTYPE locale are left unchanged. Signal handlers are not installed.

Configuration files are still used with this configuration to determine paths that are unspecified. Ensure *PyConfig.home* is specified to avoid computing the default path configuration.

10.9 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.

10.10 Python Path Configuration

PyConfig contains multiple fields for the path configuration:

- Path configuration inputs:
  - *PyConfig.home*
  - *PyConfig.platlibdir*
  - *PyConfig.pathconfig_warnings*
  - *PyConfig.program_name*
  - *PyConfig.pythonpath_env*
  - current working directory: to get absolute paths
  - `PATH` environment variable to get the program full path (from *PyConfig.program_name*)
  - `__PYVENV_LAUNCHER__` environment variable
  - (Windows only) Application paths in the registry under "SoftwarePythonPythonCoreX.YPythonPath" of HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE (where X.Y is the Python version).
- Path configuration output fields:
  - *PyConfig.base_exec_prefix*
  - *PyConfig.base_executable*
  - *PyConfig.base_prefix*
  - *PyConfig.exec_prefix*
  - *PyConfig.executable*
  - *PyConfig.module_search_paths_set*, *PyConfig.module_search_paths*
  - *PyConfig.prefix*

If at least one "output field" is not set, Python calculates the path configuration to fill unset fields. If *module_search_paths_set* is equal to `0`, *module_search_paths* is overridden and *module_search_paths_set* is set to `1`.

It is possible to completely ignore the function calculating the default path configuration by setting explicitly all path configuration output fields listed above. A string is considered as set even if it is non-empty. `module_search_paths` is considered as set if `module_search_paths_set` is set to `1`. In this case, `module_search_paths` will be used without modification.

Set *pathconfig_warnings* to `0` to suppress warnings when calculating the path configuration (Unix only, Windows does not log any warning).

If *base_prefix* or *base_exec_prefix* fields are not set, they inherit their value from *prefix* and *exec_prefix* respectively.

Py_RunMain() and *Py_Main()* modify `sys.path`:

- If *run_filename* is set and is a directory which contains a `__main__.py` script, prepend *run_filename* to `sys.path`.
- If *isolated* is zero:
  - If *run_module* is set, prepend the current directory to `sys.path`. Do nothing if the current directory cannot be read.
  - If *run_filename* is set, prepend the directory of the filename to `sys.path`.
  - Otherwise, prepend an empty string to `sys.path`.

If *site_import* is non-zero, `sys.path` can be modified by the `site` module. If *user_site_directory* is non-zero and the user's site-package directory exists, the `site` module appends the user's site-package directory to `sys.path`.

The following configuration files are used by the path configuration:

- `pyvenv.cfg`
- `._pth` file (ex: `python._pth`)
- `pybuilddir.txt` (Unix only)

If a `._pth` file is present:

- Set *isolated* to `1`.
- Set *use_environment* to `0`.
- Set *site_import* to `0`.
- Set *safe_path* to `1`.

The `__PYVENV_LAUNCHER__` environment variable is used to set *PyConfig.base_executable*

10.11 Py_RunMain()

int **Py_RunMain**(void)

Execute the command (*PyConfig.run_command*), the script (*PyConfig.run_filename*) or the module (*PyConfig.run_module*) specified on the command line or in the configuration.

By default and when if `-i` option is used, run the REPL.

Finally, finalizes Python and returns an exit status that can be passed to the `exit()` function.

See *Python Configuration* for an example of customized Python always running in isolated mode using *Py_RunMain()*.

10.12 Py_GetArgcArgv()

void **Py_GetArgcArgv**(int *argc, wchar_t ***argv)

Get the original command line arguments, before Python modified them.

See also *PyConfig.orig_argv* member.

10.13 Multi-Phase Initialization Private Provisional API

This section is a private provisional API introducing multi-phase initialization, the core feature of **PEP 432**:

- "Core" initialization phase, "bare minimum Python":
  - Builtin types;
  - Builtin exceptions;
  - Builtin and frozen modules;
  - The `sys` module is only partially initialized (ex: `sys.path` doesn't exist yet).
- "Main" initialization phase, Python is fully initialized:
  - Install and configure `importlib`;
  - Apply the *Path Configuration*;
  - Install signal handlers;
  - Finish `sys` module initialization (ex: create `sys.stdout` and `sys.path`);
  - Enable optional features like `faulthandler` and `tracemalloc`;
  - Import the `site` module;
  - etc.

Private provisional API:

- `PyConfig._init_main`: if set to `0`, *Py_InitializeFromConfig()* stops at the "Core" initialization phase.

PyStatus `_Py_InitializeMain`(void)

Move to the "Main" initialization phase, finish the Python initialization.

No module is imported during the "Core" phase and the `importlib` module is not configured: the *Path Configuration* is only applied during the "Main" phase. It may allow to customize Python in Python to override or tune the *Path Configuration*, maybe install a custom `sys.meta_path` importer or an import hook, etc.

It may become possible to calculate the *Path Configuration* in Python, after the Core phase and before the Main phase, which is one of the **PEP 432** motivation.

The "Core" phase is not properly defined: what should be and what should not be available at this phase is not specified yet. The API is marked as private and provisional: the API can be modified or even be removed anytime until a proper public API is designed.

Example running Python code between "Core" and "Main" initialization phases:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config._init_main = 0;

    /* ... customize 'config' configuration ... */

    status = Py_InitializeFromConfig(&config);
    PyConfig_Clear(&config);
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }

    /* Use sys.stderr because sys.stdout is only created
       by _Py_InitializeMain() */
    int res = PyRun_SimpleString(
        "import sys; "
        "print('Run Python code before _Py_InitializeMain', "
               "file=sys.stderr)");
    if (res < 0) {
        exit(1);
    }
```

(次のページに続く)

(前のページからの続き)

```
    /* ... put more configuration code here ... */

    status = _Py_InitializeMain();
    if (PyStatus_Exception(status)) {
        Py_ExitStatusException(status);
    }
}
```

第

ELEVEN

メモリ管理

11.1 概要

Python におけるメモリ管理には、全ての Python オブジェクトとデータ構造が入ったプライベートヒープ (private heap) が必須です。プライベートヒープの管理は、内部的には *Python* **メモリマネージャ** *(Python memory manager)* が確実に行います。Python メモリマネージャには、共有 (sharing)、セグメント分割 (segmentation)、事前割り当て (preallocation)、キャッシュ化 (caching) といった、様々な動的記憶管理の側面を扱うために、個別のコンポーネントがあります。

最低水準層では、素のメモリ操作関数 (raw memory allocator) がオペレーティングシステムのメモリ管理機構とやりとりして、プライベートヒープ内に Python 関連の全てのデータを記憶するのに十分な空きがあるかどうか確認します。素のメモリ操作関数の上には、いくつかのオブジェクト固有のメモリ操作関数があります。これらは同じヒープを操作し、各オブジェクト型固有の事情に合ったメモリ管理ポリシを実装しています。例えば、整数オブジェクトは文字列やタプル、辞書とは違ったやり方でヒープ内で管理されます。というのも、整数には値を記憶する上で特別な要件があり、速度/容量のトレードオフが存在するからです。このように、Python メモリマネジャは作業のいくつかをオブジェクト固有のメモリ操作関数に委譲しますが、これらの関数がプライベートヒープからはみ出してメモリ管理を行わないようにしています。

重要なのは、たとえユーザがいつもヒープ内のメモリブロックを指すようなオブジェクトポインタを操作しているとしても、Python 用ヒープの管理はインタプリタ自体が行うもので、ユーザがそれを制御する余地はないと理解することです。Python オブジェクトや内部使用されるバッファを入れるためのヒープ空間のメモリ確保は、必要に応じて、Python メモリマネージャがこのドキュメント内で列挙している Python/C API 関数群を介して行います。

メモリ管理の崩壊を避けるため、拡張モジュールの作者は決して Python オブジェクトを C ライブラリが公開している関数: `malloc()` 、`calloc()` 、`realloc()` および `free()` で操作しようとしてはなりません。こうした関数を使うと、C のメモリ操作関数と Python メモリマネージャとの間で関数呼び出しが交錯します。C のメモリ操作関数と Python メモリマネージャは異なるアルゴリズムで実装されていて、異なるヒープを操作するため、呼び出しの交錯は致命的な結果を招きます。とはいえ、個別の目的のためなら、C ライブラリのメモリ操作関数を使って安全にメモリを確保したり解放したりできます。例えば、以下がそのような例です:

```
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
```

(次のページに続く)

(前のページからの続き)

```
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyBytes_FromString(buf);
free(buf); /* malloc'ed */
return res;
```

この例では、I/O バッファに対するメモリ要求は C ライブラリのメモリ操作関数を使っています。Python メモリマネージャーは戻り値として返される bytes オブジェクトを確保する時にだけ必要です。

とはいえ、ほとんどの状況では、メモリの操作は Python ヒープに固定して行うよう勧めます。なぜなら、Python ヒープは Python メモリマネジャの管理下にあるからです。例えば、インタプリタを C で書かれた新たなオブジェクト型で拡張する際には、ヒープでのメモリ管理が必要です。Python ヒープを使った方がよいもう一つの理由として、拡張モジュールが必要としているメモリについて Python メモリマネージャに **情報を提供** してほしいということがあります。たとえ必要なメモリが内部的かつ非常に特化した用途に対して排他的に用いられるものだとしても、全てのメモリ操作要求を Python メモリマネージャに委譲すれば、インタプリタはより正確なメモリフットプリントの全体像を把握できます。その結果、特定の状況では、Python メモリマネージャがガベージコレクションやメモリのコンパクト化、その他何らかの予防措置といった、適切な動作をトリガできることがあります。上の例で示したように C ライブラリのメモリ操作関数を使うと、I/O バッファ用に確保したメモリは Python メモリマネージャの管理から完全に外れることに注意してください。

参考

環境変数 `PYTHONMALLOC` を使用して Python が利用するメモリアロケータを制御することができます。

環境変数 `PYTHONMALLOCSTATS` を使用して、新たなオブジェクトアリーナが生成される時と、シャットダウン時に *pymalloc* **メモリアロケータ** の統計情報を表示できます。

11.2 Allocator Domains

All allocating functions belong to one of three different ”domains” (see also *PyMemAllocatorDomain*). These domains represent different allocation strategies and are optimized for different purposes. The specific details on how every domain allocates memory or what internal functions each domain calls is considered an implementation detail, but for debugging purposes a simplified table can be found at *here*. There is no hard requirement to use the memory returned by the allocation functions belonging to a given domain for only the purposes hinted by that domain (although this is the recommended practice). For example, one could use the memory returned by *PyMem_RawMalloc()* for allocating Python objects or the memory returned by *PyObject_Malloc()* for allocating memory for buffers.

The three allocation domains are:

- Raw domain: intended for allocating memory for general-purpose memory buffers where the allocation *must* go to the system allocator or where the allocator can operate without the *GIL*. The memory is requested directly to the system.
- ”Mem” domain: intended for allocating memory for Python buffers and general-purpose memory

buffers where the allocation must be performed with the *GIL* held. The memory is taken from the Python private heap.

- Object domain: intended for allocating memory belonging to Python objects. The memory is taken from the Python private heap.

When freeing memory previously allocated by the allocating functions belonging to a given domain,the matching specific deallocating functions must be used. For example, *PyMem_Free()* must be used to free memory allocated using *PyMem_Malloc()*.

11.3 生メモリインターフェース

The following function sets are wrappers to the system allocator. These functions are thread-safe, the *GIL* does not need to be held.

The *default raw memory allocator* uses the following functions: `malloc()`, `calloc()`, `realloc()` and `free()`; call `malloc(1)` (or `calloc(1, 1)`) when requesting zero bytes.

Added in version 3.4.

void *__PyMem_RawMalloc__(size_t n)

n バイトを割り当て、そのメモリを指す `void*` 型のポインタを返します。要求が失敗した場合 `NULL` を返します。

0 バイトを要求すると、`PyMem_RawMalloc(1)` が呼ばれたときと同じように、可能なら `NULL` でないユニークなポインタを返します。確保されたメモリーにはいかなる初期化も行われません。

void *__PyMem_RawCalloc__(size_t nelem, size_t elsize)

各要素が *elsize* バイトの要素 *nelem* 個分のメモリーを確保し、そのメモリーを指す `void*` 型のポインタを返します。アロケートに失敗した場合は `NULL` を返します。確保されたメモリー領域はゼロで初期化されます。

要素数か要素のサイズが 0 バイトの要求に対しては、可能なら `PyMem_RawCalloc(1, 1)` が呼ばれたのと同じように、ユニークな `NULL` でないポインタを返します。

Added in version 3.5.

void *__PyMem_RawRealloc__(void *p, size_t n)

p が指すメモリブロックを *n* バイトにリサイズします。古いサイズと新しいサイズの小さい方までの内容は変更されません。

p が `NULL` の場合呼び出しは `PyMem_RawMalloc(n)` と等価です。そうでなく、*n* がゼロに等しい場合、メモリブロックはリサイズされますが解放されません。返されたポインタは非 `NULL` です。

p が `NULL` でない限り、*p* はそれより前の *PyMem_RawMalloc()*, *PyMem_RawRealloc()*, *PyMem_RawCalloc()* の呼び出しにより返されなければなりません。

要求が失敗した場合 *PyMem_RawRealloc()* は `NULL` を返し、*p* は前のメモリエリアをさす有効なポインタのままです。

void **PyMem_RawFree**(void *p)

p が指すメモリブロックを解放します。*p* は以前呼び出した *PyMem_RawMalloc()*, *PyMem_RawRealloc()*, *PyMem_RawCalloc()* の返した値でなければなりません。それ以外の場合や `PyMem_RawFree(p)` を呼び出した後だった場合、未定義の動作になります。

p が `NULL` の場合何もしません。

11.4 メモリインターフェース

以下の関数群が利用して Python ヒープに対してメモリを確保したり解放したり出来ます。これらの関数は ANSI C 標準に従ってモデル化されていますが、0 バイトを要求した際の動作についても定義しています:

The *default memory allocator* uses the *pymalloc memory allocator*.

警告

The *GIL* must be held when using these functions.

バージョン 3.6 で変更: デフォルトのアロケータがシステムの `malloc()` から pymalloc になりました。

void ***PyMem_Malloc**(size_t n)

次に属します: Stable ABI. *n* バイトを割り当て、そのメモリを指す `void*` 型のポインタを返します。要求が失敗した場合 `NULL` を返します。

0 バイトを要求すると、`PyMem_Malloc(1)` が呼ばれたときと同じように、可能なら `NULL` でないユニークなポインタを返します。確保されたメモリーにはいかなる初期化も行われません。

void ***PyMem_Calloc**(size_t nelem, size_t elsize)

次に属します: Stable ABI *(**バージョン** 3.7 **より**)*. 各要素が *elsize* バイトの要素 *nelem* 個分のメモリーを確保し、そのメモリーを指す `void*` 型のポインタを返します。アロケートに失敗した場合は `NULL` を返します。確保されたメモリー領域はゼロで初期化されます。

要素数か要素のサイズが 0 バイトの要求に対しては、可能なら `PyMem_Calloc(1, 1)` が呼ばれたのと同じように、ユニークな `NULL` でないポインタを返します。

Added in version 3.5.

void ***PyMem_Realloc**(void *p, size_t n)

次に属します: Stable ABI. *p* が指すメモリブロックを *n* バイトにリサイズします。古いサイズと新しいサイズの小さい方までの内容は変更されません。

p が `NULL` の場合呼び出しは `PyMem_Malloc(n)` と等価です。そうでなく、*n* がゼロに等しい場合、メモリブロックはリサイズされますが解放されません。返されたポインタは非 `NULL` です。

p が `NULL` でない限り、*p* はそれより前の *PyMem_Malloc()*, *PyMem_Realloc()* または *PyMem_Calloc()* の呼び出しにより返されなければなりません。

要求が失敗した場合 *PyMem_Realloc()* は NULL を返し、*p* は前のメモリエリアをさす有効なポインタのままです。

void **PyMem_Free**(void *p)

次に属します: Stable ABI. *p* が指すメモリブロックを解放します。*p* は以前呼び出した *PyMem_Malloc()*、*PyMem_Realloc()*、または *PyMem_Calloc()* の返した値でなければなりません。それ以外の場合や `PyMem_Free(p)` を呼び出した後だった場合、未定義の動作になります。

p が NULL の場合何もしません。

以下に挙げる型対象のマクロは利便性のために提供されているものです。*TYPE* は任意の C の型を表します。

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.

これは C プリプロセッサマクロです。*p* は常に再代入されます。エラー処理時にメモリを失うのを避けるには *p* の元の値を保存してください。

void **PyMem_Del**(void *p)

PyMem_Free() と同じです。

上記に加えて、C API 関数を介することなく Python メモリ操作関数を直接呼び出すための以下のマクロセットが提供されています。ただし、これらのマクロは Python バージョン間でのバイナリ互換性を保てず、それゆえに拡張モジュールでは撤廃されているので注意してください。

- `PyMem_MALLOC(size)`
- `PyMem_NEW(type, size)`
- `PyMem_REALLOC(ptr, size)`
- `PyMem_RESIZE(ptr, type, size)`
- `PyMem_FREE(ptr)`
- `PyMem_DEL(ptr)`

11.5 オブジェクトアロケータ

以下の関数群が利用して Python ヒープに対してメモリを確保したり解放したり出来ます。これらの関数は ANSI C 標準に従ってモデル化されていますが、0 バイトを要求した際の動作についても定義しています:

注釈

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*.

警告

The *GIL* must be held when using these functions.

void *PyObject_Malloc(size_t n)

次に属します: Stable ABI. *n* バイトを割り当て、そのメモリを指す `void*` 型のポインタを返します。要求が失敗した場合 `NULL` を返します。

0 バイトを要求すると、`PyObject_Malloc(1)` が呼ばれたときと同じように、可能なら `NULL` でないユニークなポインタを返します。確保されたメモリーにはいかなる初期化も行われません。

void *PyObject_Calloc(size_t nelem, size_t elsize)

次に属します: Stable ABI *(バージョン 3.7 より)*. 各要素が *elsize* バイトの要素 *nelem* 個分のメモリーを確保し、そのメモリーを指す `void*` 型のポインタを返します。アロケートに失敗した場合は `NULL` を返します。確保されたメモリー領域はゼロで初期化されます。

要素数か要素のサイズが 0 バイトの要求に対しては、可能なら `PyObject_Calloc(1, 1)` が呼ばれたのと同じように、ユニークな `NULL` でないポインタを返します。

Added in version 3.5.

void *PyObject_Realloc(void *p, size_t n)

次に属します: Stable ABI. *p* が指すメモリブロックを *n* バイトにリサイズします。古いサイズと新しいサイズの小さい方までの内容は変更されません。

p が `NULL` の場合呼び出しは `PyObject_Malloc(n)` と等価です。そうでなく、*n* がゼロに等しい場合、メモリブロックはリサイズされますが解放されません。返されたポインタは非 `NULL` です。

p が `NULL` でない限り、*p* はそれより前の *PyObject_Malloc()*, *PyObject_Realloc()* または *PyObject_Calloc()* の呼び出しにより返されなければなりません。

要求が失敗した場合 *PyObject_Realloc()* は `NULL` を返し、*p* は前のメモリエリアをさす有効なポインタのままです。

void PyObject_Free(void *p)

次に属します: Stable ABI. *p* が指すメモリブロックを解放します。*p* は以前呼び出した *PyObject_Malloc()*、*PyObject_Realloc()*、または *PyObject_Calloc()* の返した値でなければ

なりません。それ以外の場合や `PyObject_Free(p)` を呼び出した後だった場合、未定義の動作になります。

p が `NULL` の場合何もしません。

11.6 Default Memory Allocators

Default memory allocators:

Configuration	名前	PyMem_RawMalloc	PyMem_Malloc	PyObject_Malloc
リリースビルド	`"pymalloc"`	`malloc`	`pymalloc`	`pymalloc`
デバッグビルド	`"pymalloc_debug"`	`malloc` + debug	`pymalloc` + debug	`pymalloc` + debug
pymalloc 無しのリリースビルド	`"malloc"`	`malloc`	`malloc`	`malloc`
pymalloc 無しのデバッグビルド	`"malloc_debug"`	`malloc` + debug	`malloc` + debug	`malloc` + debug

説明:

- Name: value for `PYTHONMALLOC` environment variable.
- `malloc`: system allocators from the standard C library, C functions: `malloc()`, `calloc()`, `realloc()` and `free()`.
- `pymalloc`: *pymalloc memory allocator*.
- "+ debug": with *debug hooks on the Python memory allocators*.
- "Debug build": Python build in debug mode.

11.7 メモリアロケータをカスタマイズする

Added in version 3.4.

type **PyMemAllocatorEx**

Structure used to describe a memory block allocator. The structure has the following fields:

フィールド	意味
`void *ctx`	第一引数として渡されるユーザコンテキスト
`void* malloc(void *ctx, size_t size)`	メモリブロックを割り当てます
`void* calloc(void *ctx, size_t nelem, size_t elsize)`	0 で初期化されたメモリブロックを割り当てます
`void* realloc(void *ctx, void *ptr, size_t new_size)`	メモリブロックを割り当てるかリサイズします
`void free(void *ctx, void *ptr)`	メモリブロックを解放する

バージョン 3.5 で変更: The `PyMemAllocator` structure was renamed to *PyMemAllocatorEx* and a new `calloc` field was added.

type **PyMemAllocatorDomain**

アロケータドメインを同定するための列挙型です。ドメインは:

PYMEM_DOMAIN_RAW

関数:

- *PyMem_RawMalloc()*
- *PyMem_RawRealloc()*
- *PyMem_RawCalloc()*
- *PyMem_RawFree()*

PYMEM_DOMAIN_MEM

関数:

- *PyMem_Malloc()*,
- *PyMem_Realloc()*
- *PyMem_Calloc()*
- *PyMem_Free()*

PYMEM_DOMAIN_OBJ

関数:

- *PyObject_Malloc()*
- *PyObject_Realloc()*
- *PyObject_Calloc()*
- *PyObject_Free()*

void **PyMem_GetAllocator**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

指定されたドメインのメモリブロックアロケータを取得します。

void **PyMem_SetAllocator**(*PyMemAllocatorDomain* domain, *PyMemAllocatorEx* *allocator)

指定されたドメインのメモリブロックアロケータを設定します。

新しいアロケータは、0 バイトを要求されたときユニークな `NULL` でないポインタを返さなければなりません。

For the *PYMEM_DOMAIN_RAW* domain, the allocator must be thread-safe: the *GIL* is not held when the allocator is called.

For the remaining domains, the allocator must also be thread-safe: the allocator may be called in different interpreters that do not share a `GIL`.

新しいアロケータがフックでない (1 つ前のアロケータを呼び出さない) 場合、*PyMem_SetupDebugHooks()* 関数を呼び出して、新しいアロケータの上にデバッグフックを再度設置しなければなりません。

See also *PyPreConfig.allocator* and *Preinitialize Python with PyPreConfig*.

> **警告**
>
> *PyMem_SetAllocator()* does have the following contract:
>
> - It can be called after *Py_PreInitialize()* and before *Py_InitializeFromConfig()* to install a custom memory allocator. There are no restrictions over the installed allocator other than the ones imposed by the domain (for instance, the Raw Domain allows the allocator to be called without the GIL held). See *the section on allocator domains* for more information.
> - If called after Python has finish initializing (after *Py_InitializeFromConfig()* has been called) the allocator **must** wrap the existing allocator. Substituting the current allocator for some other arbitrary one is **not supported**.

バージョン 3.12 で変更: All allocators must be thread-safe.

void **PyMem_SetupDebugHooks**(void)

Setup *debug hooks in the Python memory allocators* to detect memory errors.

11.8 Debug hooks on the Python memory allocators

When Python is built in debug mode, the *PyMem_SetupDebugHooks()* function is called at the *Python preinitialization* to setup debug hooks on Python memory allocators to detect memory errors.

The `PYTHONMALLOC` environment variable can be used to install debug hooks on a Python compiled in release mode (ex: `PYTHONMALLOC=debug`).

The *PyMem_SetupDebugHooks()* function can be used to set debug hooks after calling *PyMem_SetAllocator()*.

These debug hooks fill dynamically allocated memory blocks with special, recognizable bit patterns. Newly allocated memory is filled with the byte `0xCD` (`PYMEM_CLEANBYTE`), freed memory is filled with the byte `0xDD` (`PYMEM_DEADBYTE`). Memory blocks are surrounded by "forbidden bytes" filled with the byte `0xFD` (`PYMEM_FORBIDDENBYTE`). Strings of these bytes are unlikely to be valid addresses, floats, or ASCII strings.

実行時チェック:

- Detect API violations. For example, detect if *PyObject_Free()* is called on a memory block allocated by *PyMem_Malloc()*.
- Detect write before the start of the buffer (buffer underflow).
- Detect write after the end of the buffer (buffer overflow).
- Check that the *GIL* is held when allocator functions of *PYMEM_DOMAIN_OBJ* (ex: *PyObject_Malloc()*) and *PYMEM_DOMAIN_MEM* (ex: *PyMem_Malloc()*) domains are called.

On error, the debug hooks use the `tracemalloc` module to get the traceback where a memory block was allocated. The traceback is only displayed if `tracemalloc` is tracing Python memory allocations and the memory block was traced.

Let S = `sizeof(size_t)`. `2*S` bytes are added at each end of each block of N bytes requested. The memory layout is like so, where p represents the address returned by a malloc-like or realloc-like function (`p[i:j]` means the slice of bytes from `*(p+i)` inclusive up to `*(p+j)` exclusive; note that the treatment of negative indices differs from a Python slice):

`p[-2*S:-S]`
: Number of bytes originally asked for. This is a size_t, big-endian (easier to read in a memory dump).

`p[-S]`
: API identifier (ASCII character):
  - `'r'` for *PYMEM_DOMAIN_RAW*.
  - `'m'` for *PYMEM_DOMAIN_MEM*.
  - `'o'` for *PYMEM_DOMAIN_OBJ*.

`p[-S+1:0]`
: Copies of PYMEM_FORBIDDENBYTE. Used to catch under- writes and reads.

`p[0:N]`
: The requested memory, filled with copies of PYMEM_CLEANBYTE, used to catch reference to uninitialized memory. When a realloc-like function is called requesting a larger memory block, the new excess bytes are also filled with PYMEM_CLEANBYTE. When a free-like function is called, these are overwritten with PYMEM_DEADBYTE, to catch reference to freed memory.

When a realloc- like function is called requesting a smaller memory block, the excess old bytes are also filled with PYMEM_DEADBYTE.

`p[N:N+S]`
: Copies of PYMEM_FORBIDDENBYTE. Used to catch over- writes and reads.

`p[N+S:N+2*S]`
: Only used if the `PYMEM_DEBUG_SERIALNO` macro is defined (not defined by default).

  A serial number, incremented by 1 on each call to a malloc-like or realloc-like function. Big-endian `size_t`. If "bad memory" is detected later, the serial number gives an excellent way to set a breakpoint on the next run, to capture the instant at which this block was passed out. The static function bumpserialno() in obmalloc.c is the only place the serial number is incremented, and exists so you can set such a breakpoint easily.

A realloc-like or free-like function first checks that the PYMEM_FORBIDDENBYTE bytes at each end are intact. If they've been altered, diagnostic output is written to stderr, and the program is aborted via Py_FatalError(). The other main failure mode is provoking a memory error when a program reads up one of the special bit patterns and tries to use it as an address. If you get in a debugger then and look at the object, you're likely to see that it's entirely filled with PYMEM_DEADBYTE (meaning freed memory is getting used) or PYMEM_CLEANBYTE (meaning uninitialized memory is getting used).

バージョン 3.6 で変更: The *PyMem_SetupDebugHooks()* function now also works on Python compiled in release mode. On error, the debug hooks now use `tracemalloc` to get the traceback where a memory block was allocated. The debug hooks now also check if the GIL is held when functions of *PYMEM_DOMAIN_OBJ* and *PYMEM_DOMAIN_MEM* domains are called.

バージョン 3.8 で変更: Byte patterns `0xCB` (`PYMEM_CLEANBYTE`), `0xDB` (`PYMEM_DEADBYTE`) and `0xFB` (`PYMEM_FORBIDDENBYTE`) have been replaced with `0xCD`, `0xDD` and `0xFD` to use the same values than Windows CRT debug `malloc()` and `free()`.

11.9 pymalloc アロケータ

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.

アリーナアロケータは、次の関数を使います:

- `VirtualAlloc()` and `VirtualFree()` on Windows,
- `mmap()` and `munmap()` if available,
- それ以外の場合は `malloc()` と `free()`。

This allocator is disabled if Python is configured with the `--without-pymalloc` option. It can also be disabled at runtime using the `PYTHONMALLOC` environment variable (ex: `PYTHONMALLOC=malloc`).

11.9.1 pymalloc アリーナアロケータのカスタマイズ

Added in version 3.4.

type **PyObjectArenaAllocator**

アリーナアロケータを記述するための構造体です。3 つのフィールドを持ちます:

フィールド	意味
`void *ctx`	第一引数として渡されるユーザコンテキスト
`void* alloc(void *ctx, size_t size)`	size バイトのアリーナを割り当てます
`void free(void *ctx, void *ptr, size_t size)`	アリーナを解放します

void **PyObject_GetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

アリーナアロケータを取得します。

void **PyObject_SetArenaAllocator**(*PyObjectArenaAllocator* *allocator)

アリーナアロケータを設定します。

11.10 tracemalloc C API

Added in 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`.

11.11 使用例

最初に述べた関数セットを使って、**概要** 節の例を Python ヒープに I/O バッファをメモリ確保するように書き換えたものを以下に示します:

```
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;
```

同じコードを型対象の関数セットで書いたものを以下に示します:

```
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyBytes_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

上の二つの例では、バッファを常に同じ関数セットに属する関数で操作していることに注意してください。実際、あるメモリブロックに対する操作は、異なるメモリ操作機構を混用する危険を減らすために、同じメモリ API ファミリを使って行うことが必要です。以下のコードには二つのエラーがあり、そのうちの一つには異なるヒープを操作する別のメモリ操作関数を混用しているので **致命的** *(Fatal)* とラベルづけをしています。

```
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3);  /* Wrong -- should be PyMem_Free() */
free(buf2);       /* Right -- allocated via malloc() */
free(buf1);       /* Fatal -- should be PyMem_Del()  */
```

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with *PyObject_New*, *PyObject_NewVar* and *PyObject_Del()*.

これらの関数については、次章の C による新しいオブジェクト型の定義や実装に関する記述の中で説明します。

第

TWELVE

オブジェクト実装サポート (OBJECT IMPLEMENTATION SUPPORT)

この章では、新しいオブジェクトの型を定義する際に使われる関数、型、およびマクロについて説明します。

12.1 オブジェクトをヒープ上にメモリ確保する

PyObject *`_PyObject_New`(*PyTypeObject* *type)

戻り値: **新しい参照。**

PyVarObject *`_PyObject_NewVar`(*PyTypeObject* *type, *Py_ssize_t* size)

戻り値: **新しい参照。**

PyObject *`PyObject_Init`(*PyObject* *op, *PyTypeObject* *type)

戻り値: **借用参照。次に属します**: 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 affected.

PyVarObject *`PyObject_InitVar`(*PyVarObject* *op, *PyTypeObject* *type, *Py_ssize_t* size)

戻り値: **借用参照。次に属します**: Stable ABI. *`PyObject_Init()`* の全ての処理を行い、可変サイズオブジェクトの場合には長さ情報も初期化します。

`PyObject_New`(TYPE, typeobj)

Allocate a new Python object using the C structure type *TYPE* and the Python type object *typeobj* (`PyTypeObject*`). Fields not defined by the Python object header are not initialized. The caller will own the only reference to the object (i.e. its reference count will be one). The size of the memory allocation is determined from the *`tp_basicsize`* field of the type object.

Note that this function is unsuitable if *typeobj* has *`Py_TPFLAGS_HAVE_GC`* set. For such objects, use *`PyObject_GC_New()`* instead.

`PyObject_NewVar`(TYPE, typeobj, size)

Allocate a new Python object using the C structure type *TYPE* and the Python type object *typeobj* (`PyTypeObject*`). Fields not defined by the Python object header are not initialized. The allocated memory allows for the *TYPE* structure plus *size* (`Py_ssize_t`) fields of the size given by the *`tp_itemsize`* field of *typeobj*. 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.

Note that this function is unsuitable if *typeobj* has *Py_TPFLAGS_HAVE_GC* set. For such objects, use *PyObject_GC_NewVar()* instead.

void **PyObject_Del**(void *op)

Releases memory allocated to an object using *PyObject_New* or *PyObject_NewVar*. This is normally called from the *tp_dealloc* handler specified in the object's type. The fields of the object should not be accessed after this call as the memory is no longer a valid Python object.

PyObject **_Py_NoneStruct**

Python からは `None` に見えるオブジェクトです。この値へのアクセスは、このオブジェクトへのポインタを評価する *Py_None* マクロを使わなければなりません。

参考

PyModule_Create() 拡張モジュールのアロケートと生成。

12.2 共通のオブジェクト構造体 (common object structure)

Python では、オブジェクト型を定義する上で数多くの構造体が使われます。この節では三つの構造体とその利用方法について説明します。

12.2.1 Base object types and macros

All Python objects ultimately share a small number of fields at the beginning of the object's representation in memory. These are represented by the *PyObject* and *PyVarObject* types, which are defined, in turn, by the expansions of some macros also used, whether directly or indirectly, in the definition of all other Python objects. Additional macros can be found under *reference counting*.

type **PyObject**

次に属します: Limited API. *(**いくつかのメンバーのみが安定** ABI **です。**)* 全てのオブジェクト型はこの型を拡張したものです。この型には、あるオブジェクトを指すポインタをオブジェクトとして Python から扱うのに必要な情報が入っています。通常の "リリース" ビルドでは、この構造体にはオブジェクトの参照カウントとオブジェクトに対応する型オブジェクトだけが入っています。実際には *PyObject* であることは宣言されていませんが、全ての Python オブジェクトへのポインタは *PyObject** へキャストできます。メンバにアクセスするには *Py_REFCNT* マクロと *Py_TYPE* マクロを使わなければなりません。

type **PyVarObject**

次に属します: Limited API. *(**いくつかのメンバーのみが安定** ABI **です。**)* This is an extension of *PyObject* that adds the *ob_size* field. This is only used for objects that have some notion of *length*. This type does not often appear in the Python/C API. Access to the members must be

done by using the macros *Py_REFCNT*, *Py_TYPE*, and *Py_SIZE*.

PyObject_HEAD

可変な長さを持たないオブジェクトを表現する新しい型を宣言するときに使うマクロです。PyObject_HEAD マクロは次のように展開されます:

```
PyObject ob_base;
```

上にある *PyObject* のドキュメントを参照してください。

PyObject_VAR_HEAD

インスタンスごとに異なる長さを持つオブジェクトを表現する新しい型を宣言するときに使うマクロです。PyObject_VAR_HEAD マクロは次のように展開されます:

```
PyVarObject ob_base;
```

上にある *PyVarObject* のドキュメントを参照してください。

PyTypeObject **PyBaseObject_Type**

次に属します: Stable ABI. The base class of all other objects, the same as `object` in Python.

int **Py_Is**(*PyObject* *x, *PyObject* *y)

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. Test if the *x* object is the *y* object, the same as `x is y` in Python.

Added in version 3.10.

int **Py_IsNone**(*PyObject* *x)

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. Test if an object is the `None` singleton, the same as `x is None` in Python.

Added in version 3.10.

int **Py_IsTrue**(*PyObject* *x)

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. Test if an object is the `True` singleton, the same as `x is True` in Python.

Added in version 3.10.

int **Py_IsFalse**(*PyObject* *x)

次に属します: Stable ABI *(**バージョン** 3.10 **より**)*. Test if an object is the `False` singleton, the same as `x is False` in Python.

Added in version 3.10.

PyTypeObject ***Py_TYPE**(*PyObject* *o)

Get the type of the Python object *o*.

Return a *borrowed reference*.

Use the *Py_SET_TYPE()* function to set an object type.

バージョン 3.11 で変更: *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`.

Added in version 3.9.

void **Py_SET_TYPE**(*PyObject* *o, *PyTypeObject* *type)

Set the object *o* type to *type*.

Added in version 3.9.

Py_ssize_t **Py_SIZE**(*PyVarObject* *o)

Get the size of the Python object *o*.

Use the *Py_SET_SIZE()* function to set an object size.

バージョン 3.11 で変更: *Py_SIZE()* is changed to an inline static function. The parameter type is no longer `const` *PyVarObject*`*`.

void **Py_SET_SIZE**(*PyVarObject* *o, *Py_ssize_t* size)

Set the object *o* size to *size*.

Added in version 3.9.

PyObject_HEAD_INIT(type)

新しい *PyObject* 型のための初期値に展開するマクロです。このマクロは次のように展開されます。

```
_PyObject_EXTRA_INIT
1, type,
```

PyVarObject_HEAD_INIT(type, size)

This is a macro which expands to initialization values for a new *PyVarObject* type, including the *ob_size* field. This macro expands to:

```
_PyObject_EXTRA_INIT
1, type, size,
```

12.2.2 Implementing functions and methods

type **PyCFunction**

次に属します: 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.

関数のシグネチャは次のとおりです

```
PyObject *PyCFunction(PyObject *self,
                      PyObject *args);
```

type **PyCFunctionWithKeywords**

次に属します: 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**

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**

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)
```

Added in version 3.9.

type **PyMethodDef**

次に属します: Stable ABI *(**すべてのメンバーを含む**)*. 拡張型のメソッドを記述する際に用いる構造体です。この構造体には 4 つのフィールドがあります:

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

PyCFunction 型のメソッドで典型的に使われる呼び出し規約です。関数は *PyObject** 型の引数値を二つ要求します。最初の引数はメソッドの *self* オブジェクトです; モジュール関数の場合、これはモジュールオブジェクトです。第二のパラメタ (よく *args* と呼ばれます) は、全ての引数を表現するタプルオブジェクトです。パラメタは通常、*PyArg_ParseTuple()* や *PyArg_UnpackTuple()* で処理されます。

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).

Added in version 3.7.

バージョン 3.10 で変更: 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.

Added in 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.

Added in 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.

以下の二つの定数は、呼び出し規約を示すものではなく、クラスのメソッドとして使う際の束縛方式を示すものです。モジュールに対して定義された関数で用いてはなりません。メソッドに対しては、最大で一つしかこのフラグをセットできません。

METH_CLASS

メソッドの最初の引数には、型のインスタンスではなく型オブジェクトが渡されます。このフラグは組み込み関数 classmethod() を使って生成するのと同じ **クラスメソッド** *(class method)* を生成するために使われます。

METH_STATIC

メソッドの最初の引数には、型のインスタンスではなく NULL が渡されます。このフラグは、staticmethod() を使って生成するのと同じ **静的メソッド** *(static method)* を生成するために使われます。

もう一つの定数は、あるメソッドを同名の別のメソッド定義と置き換えるかどうかを制御します。

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)

戻り値: **新しい参照。次に属します**: Stable ABI *(**バージョン** 3.9 **より**)*. 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`.

参考

`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`.

Added in version 3.9.

PyObject ***PyCFunction_NewEx**(*PyMethodDef* *ml, *PyObject* *self, *PyObject* *module)

戻り値: **新しい参照。次に属します**: Stable ABI. Equivalent to `PyCMethod_New(ml, self, module, NULL)`.

PyObject ***PyCFunction_New**(*PyMethodDef* *ml, *PyObject* *self)

戻り値: 新しい参照。**次に属します:** Stable ABI *(バージョン 3.4 より)*. Equivalent to `PyCMethod_New(ml, self, NULL, NULL)`.

12.2.3 Accessing attributes of extension types

type **PyMemberDef**

次に属します: Stable ABI *(すべてのメンバーを含む)*. Structure which describes an attribute of a type which corresponds to a C struct member. When defining a class, put a NULL-terminated array of these structures in the *tp_members* slot.

Its fields are, in order:

const char ***name**

Name of the member. A NULL value marks the end of a `PyMemberDef[]` array.

The string should be static, no copy is made of it.

int **type**

The type of the member in the C struct. See *Member types* for the possible values.

Py_ssize_t **offset**

The offset in bytes that the member is located on the type's object struct.

int **flags**

Zero or more of the *Member flags*, combined using bitwise OR.

const char ***doc**

The docstring, or NULL. The string should be static, no copy is made of it. Typically, it is defined using *PyDoc_STR*.

By default (when *flags* is `0`), members allow both read and write access. Use the *Py_READONLY* flag for read-only access. Certain types, like *Py_T_STRING*, imply *Py_READONLY*. Only *Py_T_OBJECT_EX* (and legacy *T_OBJECT*) members can be deleted.

For heap-allocated types (created using *PyType_FromSpec()* or similar), `PyMemberDef` may contain a definition for the special member `"__vectorcalloffset__"`, corresponding to *tp_vectorcall_offset* in type objects. These must be defined with `Py_T_PYSSIZET` and `Py_READONLY`, for example:

```
static PyMemberDef spam_type_members[] = {
    {"__vectorcalloffset__", Py_T_PYSSIZET,
     offsetof(Spam_object, vectorcall), Py_READONLY},
    {NULL}  /* Sentinel */
};
```

(You may need to `#include <stddef.h>` for `offsetof()`.)

The legacy offsets *tp_dictoffset* and *tp_weaklistoffset* can be defined similarly using `"__dictoffset__"` and `"__weaklistoffset__"` members, but extensions are strongly encouraged to use *Py_TPFLAGS_MANAGED_DICT* and *Py_TPFLAGS_MANAGED_WEAKREF* instead.

バージョン 3.12 で変更: `PyMemberDef` is always available. Previously, it required including `"structmember.h"`.

PyObject ***PyMember_GetOne**(const char *obj_addr, struct *PyMemberDef* *m)

次に属します: Stable ABI. Get an attribute belonging to the object at address *obj_addr*. The attribute is described by `PyMemberDef` *m*. Returns `NULL` on error.

バージョン 3.12 で変更: `PyMember_GetOne` is always available. Previously, it required including `"structmember.h"`.

int **PyMember_SetOne**(char *obj_addr, struct *PyMemberDef* *m, *PyObject* *o)

次に属します: 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.

バージョン 3.12 で変更: `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.

バージョン 3.10 で変更: 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.

バージョン 3.12 で変更: 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()`.

マクロ名	C の型	Python の型
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)

(*): 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`.

Added in 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**

次に属します: Stable ABI *(**すべてのメンバーを含む**)*. 型のプロパティのようなアクセスを定義するための構造体です。*PyTypeObject.tp_getset* スロットの説明も参照してください。

const char ***name**

属性名

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**

任意のドキュメンテーション文字列

void ***closure**

Optional user data pointer, providing additional data for getter and setter.

typedef *PyObject* *(***getter**)(*PyObject**, void*)

次に属します: Stable ABI. The `get` function takes one *PyObject** parameter (the instance) and a user data pointer (the associated `closure`):

成功または失敗時に `NULL` と例外の集合にされたときは新しい参照を返します。

typedef int (*`setter`)(*PyObject**, *PyObject**, void*)

次に属します: Stable ABI. `set` functions take two *PyObject*`*` parameters (the instance and the value to be set) and a user data pointer (the associated `closure`):

属性を削除する場合は、2 番目のパラメータに `NULL` を指定します。成功した場合は `0` を、失敗した場合は `-1` を例外として返します。

12.3 Type Object Structures

新スタイルの型を定義する構造体: *PyTypeObject* 構造体は、おそらく Python オブジェクトシステムの中で最も重要な構造体の 1 つでしょう。型オブジェクトは `PyObject_*` 系や `PyType_*` 系の関数で扱えますが、ほとんどの Python アプリケーションにとって、さして面白みのある機能を提供しません。型オブジェクトはオブジェクトがどのように振舞うかを決める基盤ですから、インタプリタ自体や新たな型を定義する拡張モジュールでは非常に重要な存在です。

型オブジェクトは標準の型 (standard type) に比べるとかなり大きな構造体です。各型オブジェクトは多くの値を保持しており、そのほとんどは C 関数へのポインタで、それぞれの関数はその型の機能の小さい部分を実装しています。この節では、型オブジェクトの各フィールドについて詳細を説明します。各フィールドは、構造体内で出現する順番に説明されています。

以下のクイックリファレンスに加えて、**使用例** 節では *PyTypeObject* の意味と使い方を一目で理解できる例を載せています。

12.3.1 クイックリファレンス

tp スロット

PyTypeObject スロット[p. 347, *1]	型	特殊メソッド/特殊属性	Info[p. 348, *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

次のページに続く

表 1 – 前のページからの続き

PyTypeObject スロット[*1]	型	特殊メソッド/特殊属性	Info[p. 348, *2]			
			C	T	D	I
tp_str	*reprfunc*	__str__	X			X
tp_getattro	*getattrofunc*	__getattribute__, __getattr__	X	X		G
tp_setattro	*setattrofunc*	__setattr__, __delattr__	X	X		G
tp_as_buffer	*PyBufferProcs* *					%
tp_flags	unsigned long		X	X		?
tp_doc	const char *	__doc__	X	X		
tp_traverse	*traverseproc*			X		G
tp_clear	*inquiry*			X		G
tp_richcompare	*richcmpfunc*	__lt__, __le__, __eq__, __ne__, __gt__, __ge__	X			G
(*tp_weaklistoffset*)	*Py_ssize_t*			X		?
tp_iter	*getiterfunc*	__iter__				X
tp_iternext	*iternextfunc*	__next__				X
tp_methods	*PyMethodDef* []		X	X		
tp_members	*PyMemberDef* []			X		
tp_getset	*PyGetSetDef* []		X	X		
tp_base	*PyTypeObject* *	__base__			X	
tp_dict	*PyObject* *	__dict__			?	
tp_descr_get	*descrgetfunc*	__get__				X
tp_descr_set	*descrsetfunc*	__set__, __delete__				X
(*tp_dictoffset*)	*Py_ssize_t*			X		?
tp_init	*initproc*	__init__	X	X		X
tp_alloc	*allocfunc*		X		?	?
tp_new	*newfunc*	__new__	X	X	?	?
tp_free	*freefunc*		X	X	?	?
tp_is_gc	*inquiry*			X		X
<*tp_bases*>	*PyObject* *	__bases__			~	
<*tp_mro*>	*PyObject* *	__mro__			~	
[*tp_cache*]	*PyObject* *					
[*tp_subclasses*]	void *	__subclasses__				
[*tp_weaklist*]	*PyObject* *					
(*tp_del*)	*destructor*					
[*tp_version_tag*]	unsigned int					
tp_finalize	*destructor*	__del__				X
tp_vectorcall	*vectorcallfunc*					
[*tp_watched*]	unsigned char					

[*1] (): A slot name in parentheses indicates it is (effectively) deprecated.

sub-slots

Slot	型	特殊メソッド
am_await	*unaryfunc*	__await__
am_aiter	*unaryfunc*	__aiter__
am_anext	*unaryfunc*	__anext__
am_send	*sendfunc*	
nb_add	*binaryfunc*	__add__ __radd__
nb_inplace_add	*binaryfunc*	__iadd__
nb_subtract	*binaryfunc*	__sub__ __rsub__
nb_inplace_subtract	*binaryfunc*	__isub__
nb_multiply	*binaryfunc*	__mul__ __rmul__
nb_inplace_multiply	*binaryfunc*	__imul__
nb_remainder	*binaryfunc*	__mod__ __rmod__
nb_inplace_remainder	*binaryfunc*	__imod__
nb_divmod	*binaryfunc*	__divmod__ __rdivmod__
nb_power	*ternaryfunc*	__pow__ __rpow__
nb_inplace_power	*ternaryfunc*	__ipow__
nb_negative	*unaryfunc*	__neg__
nb_positive	*unaryfunc*	__pos__
nb_absolute	*unaryfunc*	__abs__
nb_bool	*inquiry*	__bool__
nb_invert	*unaryfunc*	__invert__
nb_lshift	*binaryfunc*	__lshift__ __rlshift__
nb_inplace_lshift	*binaryfunc*	__ilshift__
nb_rshift	*binaryfunc*	__rshift__ __rrshift__
nb_xor	*binaryfunc*	__xor__

<>: 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 列:

"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.

表 2 – 前のページからの続き

Slot	型	特殊メソッド
nb_inplace_xor	*binaryfunc*	___ixor___
nb_or	*binaryfunc*	___or___ ___ror___
nb_inplace_or	*binaryfunc*	___ior___
nb_int	*unaryfunc*	___int___
nb_reserved	void *	
nb_float	*unaryfunc*	___float___
nb_floor_divide	*binaryfunc*	___floordiv___
nb_inplace_floor_divide	*binaryfunc*	___ifloordiv___
nb_true_divide	*binaryfunc*	___truediv___
nb_inplace_true_divide	*binaryfunc*	___itruediv___
nb_index	*unaryfunc*	___index___
nb_matrix_multiply	*binaryfunc*	___matmul___ ___rmatmul___
nb_inplace_matrix_multiply	*binaryfunc*	___imatmul___
mp_length	*lenfunc*	___len___
mp_subscript	*binaryfunc*	___getitem___
mp_ass_subscript	*objobjargproc*	___setitem___, ___delitem___
sq_length	*lenfunc*	___len___
sq_concat	*binaryfunc*	___add___
sq_repeat	*ssizeargfunc*	___mul___
sq_item	*ssizeargfunc*	___getitem___
sq_ass_item	*ssizeobjargproc*	___setitem___ ___delitem___
sq_contains	*objobjproc*	___contains___
sq_inplace_concat	*binaryfunc*	___iadd___
sq_inplace_repeat	*ssizeargfunc*	___imul___
bf_getbuffer	*getbufferproc()*	
bf_releasebuffer	*releasebufferproc()*	

スロットの定義型 (typedef)

定義型 (typedef)	引数型	返り値型
allocfunc		*PyObject* *
	PyTypeObject *	
	Py_ssize_t	
destructor	*PyObject* *	void
freefunc	void *	void
traverseproc		int
	PyObject *	
	visitproc	
	void *	
newfunc		*PyObject* *
	PyTypeObject *	
	PyObject *	
	PyObject *	
initproc		int
	PyObject *	
	PyObject *	
	PyObject *	
reprfunc	*PyObject* *	*PyObject* *
getattrfunc		*PyObject* *
	PyObject *	
	const char *	
setattrfunc		int
	PyObject *	
	const char *	
	PyObject *	
getattrofunc		*PyObject* *
	PyObject *	
	PyObject *	
setattrofunc		int
	PyObject *	
	PyObject *	
	PyObject *	

See *Slot Type typedefs* below for more detail.

12.3.2 PyTypeObject 定義

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 */
```

(次のページに続く)

(前のページからの続き)

```
    /* 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;

    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct PyMemberDef *tp_members;
    struct PyGetSetDef *tp_getset;
    // Strong reference on a heap type, borrowed reference on a static type
    struct _typeobject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    Py_ssize_t tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
    freefunc tp_free; /* Low-level free-memory routine */
    inquiry tp_is_gc; /* For PyObject_IS_GC */
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_cache;
    PyObject *tp_subclasses;
    PyObject *tp_weaklist;
    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6 */
    unsigned int tp_version_tag;
```

(次のページに続く)

(前のページからの続き)

```
    destructor tp_finalize;
    vectorcallfunc tp_vectorcall;

    /* bitset of which type-watchers care about this type */
    unsigned char tp_watched;
} PyTypeObject;
```

12.3.3 PyObject スロット

The type object structure extends the *PyVarObject* structure. The *ob_size* field is used for dynamic types (created by `type_new()`, usually called from a class statement). Note that *PyType_Type* (the metatype) initializes *tp_itemsize*, which means that its instances (i.e. type objects) *must* have the *ob_size* field.

Py_ssize_t *PyObject*.ob_refcnt

次に属します: Stable ABI. This is the type object's reference count, initialized to `1` by the `PyObject_HEAD_INIT` macro. Note that for *statically allocated type objects*, the type's instances (objects whose *ob_type* points back to the type) do *not* count as references. But for *dynamically allocated type objects*, the instances *do* count as references.

継承:

サブタイプはこのフィールドを継承しません。

PyTypeObject **PyObject*.ob_type

次に属します: Stable ABI. 型自体の型、別の言い方をするとメタタイプです。`PyObject_HEAD_INIT` マクロで初期化され、通常は `&PyType_Type` になります。しかし、(少なくとも) Windows で利用できる動的ロード可能な拡張モジュールでは、コンパイラは有効な初期化ではないと文句をつけます。そこで、ならわしとして、`PyObject_HEAD_INIT` には `NULL` を渡して初期化しておき、他の操作を行う前にモジュールの初期化関数で明示的にこのフィールドを初期化することになっています。この操作は以下のように行います:

```
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.

継承:

サブタイプはこのフィールドを継承します。

PyObject **PyObject*._ob_next

PyObject **PyObject._ob_prev*

These fields are only present when the macro `Py_TRACE_REFS` is defined (see the `configure --with-trace-refs option`).

Their initialization to `NULL` is taken care of by the `PyObject_HEAD_INIT` macro. For *statically allocated objects*, these fields always remain `NULL`. For *dynamically allocated objects*, these two fields are used to link the object into a doubly linked list of *all* live objects on the heap.

This could be used for various debugging purposes; currently the only uses are the `sys.getobjects()` function and to print the objects that are still alive at the end of a run when the environment variable `PYTHONDUMPREFS` is set.

継承:

These fields are not inherited by subtypes.

12.3.4 PyVarObject スロット

Py_ssize_t *PyVarObject*.ob_size

次に属します: Stable ABI. **静的にメモリ確保されている型オブジェクト** の場合、このフィールドはゼロに初期化されます。**動的にメモリ確保されている型オブジェクト** の場合、このフィールドは内部使用される特殊な意味を持ちます。

This field should be accessed using the *Py_SIZE()* and *Py_SET_SIZE()* macros.

継承:

サブタイプはこのフィールドを継承しません。

12.3.5 PyTypeObject スロット

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"`.

動的にメモリ確保される型オブジェクト の場合、このフィールドは単に型の名前になり、モジュール名は型の辞書内でキー `'__module__'` に対する値として明示的に保存されます。

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*).

継承:

サブタイプはこのフィールドを継承しません。

Py_ssize_t *PyTypeObject*.**tp_basicsize**

Py_ssize_t *PyTypeObject*.**tp_itemsize**

これらのフィールドは、型インスタンスのバイトサイズを計算できるようにします。

There are two kinds of types: types with fixed-length instances have a zero `tp_itemsize` field, types with variable-length instances have a non-zero `tp_itemsize` field. For a type with fixed-length instances, all instances have the same size, given in `tp_basicsize`. (Exceptions to this rule can be made using *PyUnstable_Object_GC_NewWithExtraData()*.)

For a type with variable-length instances, the instances must have an *ob_size* field, and the instance size is `tp_basicsize` plus N times `tp_itemsize`, where N is the "length" of the object.

Functions like *PyObject_NewVar()* will take the value of N as an argument, and store in the instance's *ob_size* field. Note that the *ob_size* field may later be used for other purposes. For example, `int` instances use the bits of *ob_size* in an implementation-defined way; the underlying storage and its size should be acessed using `PyLong_Export()`.

> **注釈**
>
> 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)`.

継承:

These fields are inherited separately by subtypes. (That is, if the field is set to zero, *PyType_Ready()* will copy the value from the base type, indicating that the instances do not need additional storage.)

If the base type has a non-zero *tp_itemsize*, it is generally not safe to set *tp_itemsize* to a different non-zero value in a subtype (though this depends on the implementation of the base type).

destructor *PyTypeObject*.**tp_dealloc**

A pointer to the instance destructor function. This function must be defined unless the type guarantees that its instances will never be deallocated (as is the case for the singletons `None` and `Ellipsis`). The function signature is:

```
void tp_dealloc(PyObject *self);
```

The destructor function is called by the *Py_DECREF()* and *Py_XDECREF()* macros when the new reference count is zero. At this point, the instance is still in existence, but there are no references to it. The destructor function should free all references which the instance owns, free all memory buffers owned by the instance (using the freeing function corresponding to the allocation function used to allocate the buffer), and call the type's *tp_free* function. If the type is not subtypable (doesn't have the *Py_TPFLAGS_BASETYPE* flag bit set), it is permissible to call the object deallocator directly instead of via *tp_free*. The object deallocator should be the one used to allocate the instance; this is normally *PyObject_Del()* if the instance was allocated using *PyObject_New* or *PyObject_NewVar*, or *PyObject_GC_Del()* if the instance was allocated using *PyObject_GC_New* or *PyObject_GC_NewVar*.

If the type supports garbage collection (has the *Py_TPFLAGS_HAVE_GC* flag bit set), the destructor should call *PyObject_GC_UnTrack()* before clearing any member fields.

```
static void foo_dealloc(foo_object *self) {
    PyObject_GC_UnTrack(self);
```

(次のページに続く)

(前のページからの続き)

```
    Py_CLEAR(self->ref);
    Py_TYPE(self)->tp_free((PyObject *)self);
}
```

Finally, if the type is heap allocated (*Py_TPFLAGS_HEAPTYPE*), the deallocator should release the owned reference to its type object (via *Py_DECREF()*) after calling the type deallocator. In order to avoid dangling pointers, the recommended way to achieve this is:

```
static void foo_dealloc(foo_object *self) {
    PyTypeObject *tp = Py_TYPE(self);
    // free references and buffers here
    tp->tp_free(self);
    Py_DECREF(tp);
}
```

警告

In a garbage collected Python, `tp_dealloc` may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a garbage collection on any thread). This is not a problem for Python API calls, since the thread on which `tp_dealloc` is called will own the Global Interpreter Lock (GIL). However, if the object being destroyed in turn destroys objects from some other C or C++ library, care should be taken to ensure that destroying those objects on the thread which called `tp_dealloc` will not violate any assumptions of the library.

継承:

サブタイプはこのフィールドを継承します。

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()*.

バージョン 3.8 で変更: 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.

バージョン 3.12 で変更: 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.

継承:

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**

オプションのポインタで、get-attribute-string を行う関数を指します。

このフィールドは非推奨です。このフィールドを定義するときは、*tp_getattro* 関数と同じように動作し、属性名は Python 文字列 オブジェクトではなく C 文字列で指定するような関数を指すようにしなければなりません。

継承:

Group: *tp_getattr*, *tp_getattro*

このフィールドは *tp_getattro* と共にサブタイプに継承されます: すなわち、サブタイプの *tp_getattr* および *tp_getattro* が共に `NULL` の場合、サブタイプは基底タイプから *tp_getattr* と *tp_getattro* を両方とも継承します。

setattrfunc *PyTypeObject*.**tp_setattr**

オプションのポインタで、属性の設定と削除を行う関数を指します。

このフィールドは非推奨です。このフィールドを定義するときは、*tp_setattro* 関数と同じように動作し、属性名は Python 文字列 オブジェクトではなく C 文字列で指定するような関数を指すようにしなければなりません。

継承:

Group: *tp_setattr*, *tp_setattro*

このフィールドは *tp_setattro* と共にサブタイプに継承されます: すなわち、サブタイプの *tp_setattr* および *tp_setattro* が共に `NULL` の場合、サブタイプは基底タイプから *tp_setattr* と *tp_setattro* を両方とも継承します。

PyAsyncMethods **PyTypeObject*.**tp_as_async**

追加の構造体を指すポインタです。この構造体は、C レベルで *awaitable* プロトコルと *asynchronous iterator* プロトコルを実装するオブジェクトだけに関係するフィールドを持ちます。詳しいことは *async* **オブジェクト構造体** を参照してください。

Added in version 3.5: 以前は `tp_compare` や `tp_reserved` として知られていました。

継承:

tp_as_async フィールドは継承されませんが、これに含まれるフィールドが個別に継承されます。

reprfunc *PyTypeObject*.**tp_repr**

オプションのポインタで、組み込み関数 repr() を実装している関数を指します。

The signature is the same as for *PyObject_Repr()*:

```
PyObject *tp_repr(PyObject *self);
```

この関数は文字列オブジェクトか Unicode オブジェクトを返さなければなりません。理想的には、この関数が返す文字列は、適切な環境で eval() に渡した場合、同じ値を持つオブジェクトになるような文字列でなければなりません。不可能な場合には、オブジェクトの型と値から導出した内容の入った '<' から始まって '>' で終わる文字列を返さなければなりません。

継承:

サブタイプはこのフィールドを継承します。

デフォルト

このフィールドが設定されていない場合、<%s object at %p> の形式をとる文字列が返されます。%s は型の名前に、%p はオブジェクトのメモリアドレスに置き換えられます。

PyNumberMethods **PyTypeObject*.**tp_as_number**

数値プロトコルを実装した追加の構造体を指すポインタです。これらのフィールドについては **数値オブジェクト構造体** で説明されています。

継承:

tp_as_number フィールドは継承されませんが、そこの含まれるフィールドが個別に継承されます。

PySequenceMethods **PyTypeObject*.**tp_as_sequence**

シーケンスプロトコルを実装した追加の構造体を指すポインタです。これらのフィールドについては **シーケンスオブジェクト構造体** で説明されています。

継承:

tp_as_sequence フィールドは継承されませんが、これに含まれるフィールドが個別に継承されます。

PyMappingMethods **PyTypeObject*.**tp_as_mapping**

マッピングプロトコルを実装した追加の構造体を指すポインタです。これらのフィールドについては **マップオブジェクト構造体** で説明されています。

継承:

tp_as_mapping フィールドは継承されませんが、これに含まれるフィールドが個別に継承されます。

hashfunc *PyTypeObject*.**tp_hash**

オプションのポインタで、組み込み関数 hash() を実装している関数を指します。

The signature is the same as for *PyObject_Hash()*:

```
Py_hash_t tp_hash(PyObject *);
```

通常時には -1 を戻り値にしてはなりません; ハッシュ値の計算中にエラーが生じた場合、関数は例外をセットして -1 を返さねばなりません。

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()*.

このフィールドは明示的に *PyObject_HashNotImplemented()* に設定することで、親 type からのハッシュメソッドの継承をブロックすることができます。これは Python レベルでの __hash__ = None と同等に解釈され、isinstance(o, collections.Hashable) が正しく False を返すようになります。逆もまた可能であることに注意してください - Python レベルで __hash__ = None を設定することで tp_hash スロットは *PyObject_HashNotImplemented()* に設定されます。

継承:

Group: *tp_hash*, *tp_richcompare*

このフィールドは *tp_richcompare* と共にサブタイプに継承されます: すなわち、サブタイプの *tp_richcompare* および *tp_hash* が両方とも NULL のとき、サブタイプは基底タイプから *tp_richcompare* と *tp_hash* を両方とも継承します。

ternaryfunc *PyTypeObject*.**tp_call**

オプションのポインタで、オブジェクトの呼び出しを実装している関数を指します。オブジェクトが呼び出し可能でない場合には NULL にしなければなりません。シグネチャは *PyObject_Call()* と同じです。

```
PyObject *tp_call(PyObject *self, PyObject *args, PyObject *kwargs);
```

継承:

サブタイプはこのフィールドを継承します。

reprfunc *PyTypeObject*.**tp_str**

オプションのポインタで、組み込みの演算 str() を実装している関数を指します。(str が型の一つになったため、str() は str のコンストラクタを呼び出すことに注意してください。このコンストラクタは実際の処理を行う上で *PyObject_Str()* を呼び出し、さらに *PyObject_Str()* がこのハンドラを呼び出すことになります。)

The signature is the same as for *PyObject_Str()*:

```
PyObject *tp_str(PyObject *self);
```

この関数は文字列オブジェクトか Unicode オブジェクトを返さなければなりません。それはオブジェクトを " 分かりやすく (friendly)" 表現した文字列でなければなりません。というのは、この文字列はとりわけ print() 関数で使われることになる表記だからです。

継承:

サブタイプはこのフィールドを継承します。

デフォルト

このフィールドが設定されていない場合、文字列表現を返すためには *PyObject_Repr()* が呼び出されます。

getattrofunc *PyTypeObject*.**tp_getattro**

オプションのポインタで、get-attribute を実装している関数を指します。

The signature is the same as for *PyObject_GetAttr()*:

```
PyObject *tp_getattro(PyObject *self, PyObject *attr);
```

通常の属性検索を実装している *PyObject_GenericGetAttr()* をこのフィールドに設定しておくとたいていの場合は便利です。

継承:

Group: *tp_getattr*, *tp_getattro*

このフィールドは *tp_getattr* と共にサブタイプに継承されます: すなわち、サブタイプの *tp_getattr* および *tp_getattro* が共に NULL の場合、サブタイプは基底タイプから *tp_getattr* と *tp_getattro* を両方とも継承します。

デフォルト

PyBaseObject_Type uses *PyObject_GenericGetAttr()*.

setattrofunc *PyTypeObject*.**tp_setattro**

オプションのポインタで、属性の設定と削除を行う関数を指します。

The signature is the same as for *PyObject_SetAttr()*:

```
int tp_setattro(PyObject *self, PyObject *attr, PyObject *value);
```

さらに、*value* に NULL を指定して属性を削除できるようにしなければなりません。通常のオブジェクト属性設定を実装している *PyObject_GenericSetAttr()* をこのフィールドに設定しておくとたいていの場合は便利です。

継承:

Group: *tp_setattr*, *tp_setattro*

このフィールドは *tp_setattr* と共にサブタイプに継承されます: すなわち、サブタイプの *tp_setattr* および *tp_setattro* が共に NULL の場合、サブタイプは基底タイプから *tp_setattr* と *tp_setattro* を両方とも継承します。

デフォルト

PyBaseObject_Type uses *PyObject_GenericSetAttr()*.

PyBufferProcs **PyTypeObject*.tp_as_buffer

バッファインターフェースを実装しているオブジェクトにのみ関連する、一連のフィールド群が入った別の構造体を指すポインタです。構造体内の各フィールドは **バッファオブジェクト構造体** *(buffer object structure)* で説明します。

継承:

tp_as_buffer フィールド自体は継承されませんが、これに含まれるフィールドは個別に継承されます。

unsigned long *PyTypeObject*.tp_flags

このフィールドは様々なフラグからなるビットマスクです。いくつかのフラグは、特定の状況において変則的なセマンティクスが適用されることを示します; その他のフラグは、型オブジェクト (あるいは *tp_as_number* 、*tp_as_sequence* 、*tp_as_mapping* 、および *tp_as_buffer* が参照している拡張機能構造体) の特定のフィールドのうち、過去から現在までずっと存在していたわけではないものが有効になっていることを示すために使われます; フラグビットがクリアされていれば、フラグが保護しているフィールドにはアクセスしない代わりに、その値はゼロか NULL になっているとみなさなければなりません。

継承:

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type's value of the flag bit is copied into the subtype together with a pointer to the extension structure. The *Py_TPFLAGS_HAVE_GC* flag bit is inherited together with the *tp_traverse* and *tp_clear* fields, i.e. if the *Py_TPFLAGS_HAVE_GC* flag bit is clear in the subtype and the *tp_traverse* and *tp_clear* fields in the subtype exist and have NULL values. .. XXX are most flag bits *really* inherited individually?

デフォルト

PyBaseObject_Type uses Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE.

Bit Masks:

以下に挙げるビットマスクは現在定義されているものです; フラグは | 演算子で論理和を取って *tp_flags* フィールドの値を作成できます。*PyType_HasFeature()* マクロは型とフラグ値、*tp* および *f* をとり、tp->tp_flags & f が非ゼロかどうか調べます。

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.

継承:

???

Py_TPFLAGS_BASETYPE

型を別の型の基底タイプとして使える場合にセットされるビットです。このビットがクリアならば、この型のサブタイプは生成できません (Java における "final" クラスに似たクラスになります)。

継承:

???

Py_TPFLAGS_READY

型オブジェクトが *PyType_Ready()* で完全に初期化されるとセットされるビットです。

継承:

???

Py_TPFLAGS_READYING

PyType_Ready() による型オブジェクトの初期化処理中にセットされるビットです。

継承:

???

Py_TPFLAGS_HAVE_GC

This bit is set when the object supports garbage collection. If this bit is set, instances must be created using *PyObject_GC_New* and destroyed using *PyObject_GC_Del()*. More information in section **循環参照ガベージコレクションをサポートする**. This bit also implies that the GC-related fields *tp_traverse* and *tp_clear* are present in the type object.

継承:

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.

継承:

???

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`.

Added in version 3.8.

継承:

This flag is never inherited by types without the *Py_TPFLAGS_IMMUTABLETYPE* flag set. For extension types, it is inherited whenever *tp_descr_get* is inherited.

Py_TPFLAGS_MANAGED_DICT

This bit indicates that instances of the class have a *~object.__dict__* attribute, and that the space for the dictionary is managed by the VM.

If this flag is set, *Py_TPFLAGS_HAVE_GC* should also be set.

Added in version 3.12.

継承:

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.

Added in version 3.12.

継承:

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.

Added in version 3.12.

継承:

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`

これらのフラグは *PyLong_Check()* のような関数が、型がとある組み込み型のサブクラスかどうかを素早く判断するのに使われます; この専用のチェックは *PyObject_IsInstance()* のような汎用的なチェックよりも高速です。組み込み型を継承した独自の型では *tp_flags* を適切に設定すべきで、そうしないとその型が関わるコードでは、どんなチェックの方法が使われるかによって振る舞いが異なってしまうでしょう。

`Py_TPFLAGS_HAVE_FINALIZE`

型構造体に *tp_finalize* スロットが存在しているときにセットされるビットです。

Added in version 3.4.

バージョン 3.8 で非推奨: 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.

継承:

This bit is inherited if *tp_call* is also inherited.

Added in version 3.9.

バージョン 3.12 で変更: 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*.

継承:

This flag is not inherited.

Added in 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.

継承:

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).

> **注釈**
>
> 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.

Added in 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`.

> **注釈**
>
> *Py_TPFLAGS_MAPPING* and *Py_TPFLAGS_SEQUENCE* are mutually exclusive; it is an error to enable both flags simultaneously.

継承:

This flag is inherited by types that do not already set *Py_TPFLAGS_SEQUENCE*.

> **参考**
>
> **PEP 634** -- 構造的パターンマッチ: 仕様

Added in 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`.

> **注釈**
>
> *Py_TPFLAGS_MAPPING* and *Py_TPFLAGS_SEQUENCE* are mutually exclusive; it is an error to enable both flags simultaneously.

継承:

This flag is inherited by types that do not already set *Py_TPFLAGS_MAPPING*.

> **参考**
>
> **PEP 634** -- 構造的パターンマッチ: 仕様

Added in 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()*

> **警告**
>
> This flag is present in header files, but is an internal feature and should 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.

継承:

サブタイプはこのフィールドを継承 **しません** 。

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);
```

Python のガベージコレクションの仕組みについての詳細は、循環参照ガベージコレクションをサポートする にあります。

The *tp_traverse* pointer is used by the garbage collector to detect reference cycles. A typical implementation of a *tp_traverse* function simply calls *Py_VISIT()* on each of the instance's members that are Python objects that the instance owns. For example, this is function `local_traverse()` from the `_thread` extension module:

```
static int
local_traverse(localobject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->args);
    Py_VISIT(self->kw);
    Py_VISIT(self->dict);
    return 0;
}
```

Py_VISIT() が循環参照になる恐れのあるメンバにだけ呼び出されていることに注目してください。`self->key` メンバもありますが、それは `NULL` か Python 文字列なので、循環参照の一部になることはありません。

一方、メンバが循環参照の一部になり得ないと判っていても、デバッグ目的で巡回したい場合があるかもしれないので、`gc` モジュールの `get_referents()` 関数は循環参照になり得ないメンバも返します。

警告

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.

バージョン 3.9 で変更: 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.

継承:

Group: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

This field is inherited by subtypes together with *tp_clear* and the *Py_TPFLAGS_HAVE_GC* flag bit: the flag bit, *tp_traverse*, and *tp_clear* are all inherited from the base type if they are all zero in the subtype.

inquiry *PyTypeObject*.**tp_clear**

An optional pointer to a clear function for the garbage collector. This is only used if the *Py_TPFLAGS_HAVE_GC* flag bit is set. The signature is:

```
int tp_clear(PyObject *);
```

The *tp_clear* member function is used to break reference cycles in cyclic garbage detected by the garbage collector. Taken together, all *tp_clear* functions in the system must combine to break all reference cycles. This is subtle, and if in any doubt supply a *tp_clear* function. For example, the tuple type does not implement a *tp_clear* function, because it's possible to prove that no reference cycle can be composed entirely of tuples. Therefore the *tp_clear* functions of other types must be sufficient to break any cycle containing a tuple. This isn't immediately obvious, and there's rarely a good reason to avoid implementing *tp_clear*.

次の例にあるように、*tp_clear* の実装は、インスタンスから Python オブジェクトだと思われるメンバへの参照を外し、それらのメンバへのポインタに `NULL` をセットすべきです:

```
static int
local_clear(localobject *self)
{
    Py_CLEAR(self->key);
    Py_CLEAR(self->args);
    Py_CLEAR(self->kw);
    Py_CLEAR(self->dict);
    return 0;
}
```

The *Py_CLEAR()* macro should be used, because clearing references is delicate: the reference to the contained object must not be released (via *Py_DECREF()*) until after the pointer to the contained object is set to `NULL`. This is because releasing the reference may cause the contained object to become trash, triggering a chain of reclamation activity that may include invoking arbitrary Python code (due to finalizers, or weakref callbacks, associated with the contained object). If it's possible for such code to reference *self* again, it's important that the pointer to the contained

object be NULL at that time, so that *self* knows the contained object can no longer be used. The *Py_CLEAR()* macro performs the operations in a safe order.

Note that *tp_clear* is not *always* called before an instance is deallocated. For example, when reference counting is enough to determine that an object is no longer used, the cyclic garbage collector is not involved and *tp_dealloc* is called directly.

Because the goal of *tp_clear* functions is to break reference cycles, it's not necessary to clear contained objects like Python strings or Python integers, which can't participate in reference cycles. On the other hand, it may be convenient to clear all contained Python objects, and write the type's *tp_dealloc* function to invoke *tp_clear*.

Python のガベージコレクションの仕組みについての詳細は、**循環参照ガベージコレクションをサポートする** にあります。

継承:

Group: *Py_TPFLAGS_HAVE_GC*, *tp_traverse*, *tp_clear*

This field is inherited by subtypes together with *tp_traverse* and the *Py_TPFLAGS_HAVE_GC* flag bit: the flag bit, *tp_traverse*, and *tp_clear* are all inherited from the base type if they are all zero in the subtype.

richcmpfunc *PyTypeObject*.**tp_richcompare**

オプションのポインタで、拡張比較関数を指します。シグネチャは次の通りです:

```
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*.

この関数は、比較結果を返すべきです。(普通は Py_True か Py_False です。) 比較が未定義の場合は、Py_NotImplemented を、それ以外のエラーが発生した場合には例外状態をセットして NULL を返さなければなりません。

tp_richcompare および *PyObject_RichCompare()* 関数の第三引数に使うための定数としては以下が定義されています:

定数	比較
Py_LT	<
Py_LE	<=
Py_EQ	==
Py_NE	!=
Py_GT	>
Py_GE	>=

拡張比較関数 (rich comparison functions) を簡単に記述するためのマクロが定義されています:

Py_RETURN_RICHCOMPARE(VAL_A, VAL_B, op)

比較した結果に応じて Py_True か Py_False を返します。VAL_A と VAL_B は C の比較演算によって順序付け可能でなければなりません（例えばこれらは C 言語の整数か浮動小数点数になるでしょう）。三番目の引数には *PyObject_RichCompare()* と同様に要求された演算を指定します。

The returned value is a new *strong reference*.

エラー時には例外を設定して、関数から NULL でリターンします。

Added in version 3.7.

継承:

Group: *tp_hash*, *tp_richcompare*

このフィールドは *tp_hash* と共にサブタイプに継承されます: すなわち、サブタイプの *tp_richcompare* および *tp_hash* が両方とも NULL のとき、サブタイプは基底タイプから *tp_richcompare* と *tp_hash* を両方とも継承します。

デフォルト

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.

型のインスタンスが弱参照可能な場合、このフィールドはゼロよりも大きな数になり、インスタンス構造体における弱参照リストの先頭を示すオフセットが入ります (GC ヘッダがある場合には無視します); このオフセット値は *PyObject_ClearWeakRefs()* および PyWeakref_* 関数が利用します。インスタンス構造体には、NULL に初期化された *PyObject** 型のフィールドが入っていなければなりません。

このフィールドを *tp_weaklist* と混同しないようにしてください; これは型オブジェクト自身への弱参照からなるリストの先頭です。

It is an error to set both the *Py_TPFLAGS_MANAGED_WEAKREF* bit and *tp_weaklistoffset*.

継承:

このフィールドはサブタイプに継承されますが、以下の規則を読んでください。サブタイプはこのオフセット値をオーバライドすることがあります; 従って、サブタイプでは弱参照リストの先頭が基底タイプとは異なる場合があります。リストの先頭は常に *tp_weaklistoffset* で分かるはずなので、このことは問題にはならないはずです。

デフォルト

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**

オプションの変数で、そのオブジェクトの **イテレータ** を返す関数へのポインタです。この値が存在することは、通常この型のインスタンスが **イテレート可能** であることを示しています (しかし、シーケンスはこの関数がなくてもイテレート可能です)。

この関数は *PyObject_GetIter()* と同じシグネチャを持っています:

```
PyObject *tp_iter(PyObject *self);
```

継承:

サブタイプはこのフィールドを継承します。

iternextfunc *PyTypeObject*.**tp_iternext**

オプションのポインタで、**イテレーター** の次の要素を返す関数を指します。シグネチャは次の通りです:

```
PyObject *tp_iternext(PyObject *self);
```

イテレータの要素がなくなると、この関数は NULL を返さなければなりません。StopIteration 例外は設定してもしなくても良いです。その他のエラーが発生したときも、NULL を返さなければなりません。このフィールドがあると、この型のインスタンスがイテレータであることを示します。

イテレータ型では、*tp_iter* 関数も定義されていなければならず、その関数は (新たなイテレータインスタンスではなく) イテレータインスタンス自体を返さねばなりません。

この関数のシグネチャは *PyIter_Next()* と同じです。

継承:

サブタイプはこのフィールドを継承します。

struct *PyMethodDef* ***PyTypeObject*.tp_methods**

オプションのポインタで、この型の正規 (regular) のメソッドを宣言している *PyMethodDef* 構造体からなる、NULL で終端された静的な配列を指します。

配列の各要素ごとに、メソッドデスクリプタの入った、要素が型の辞書 (下記の *tp_dict* 参照) に追加されます。

継承:

サブタイプはこのフィールドを継承しません (メソッドは別個のメカニズムで継承されています)。

struct *PyMemberDef* ***PyTypeObject*.tp_members**

オプションのポインタで、型の正規 (regular) のデータメンバ (フィールドおよびスロット) を宣言している *PyMemberDef* 構造体からなる、NULL で終端された静的な配列を指します。

配列の各要素ごとに、メンバデスクリプタの入った要素が型の辞書 (下記の *tp_dict* 参照) に追加されます。

継承:

サブタイプはこのフィールドを継承しません (メンバは別個のメカニズムで継承されています)。

struct *PyGetSetDef* ***PyTypeObject*.tp_getset**

オプションのポインタで、インスタンスの算出属性 (computed attribute) を宣言している *PyGetSetDef* 構造体からなる、NULL で終端された静的な配列を指します。

配列の各要素ごとに、getter/setter デスクリプタの入った、要素が型の辞書 (下記の *tp_dict* 参照) に追加されます。

継承:

サブタイプはこのフィールドを継承しません (算出属性は別個のメカニズムで継承されています)。

PyTypeObject ***PyTypeObject*.tp_base**

オプションのポインタで、型に関するプロパティを継承する基底タイプを指します。このフィールドのレベルでは、単継承 (single inheritance) だけがサポートされています; 多重継承はメタタイプの呼び出しによる動的な型オブジェクトの生成を必要とします。

注釈

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.

継承:

(当たり前ですが) サブタイプはこのフィールドを継承しません。

デフォルト

このフィールドのデフォルト値は (Python プログラマは `object` 型として知っている) `&PyBaseObject_Type` になります。

PyObject ***PyTypeObject*.tp_dict**

型の辞書は *PyType_Ready()* によってこのフィールドに収められます。

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.

バージョン 3.12 で変更: 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.

継承:

サブタイプはこのフィールドを継承しません (が、この辞書内で定義されている属性は異なるメカニズムで継承されます)。

デフォルト

If this field is `NULL`, *PyType_Ready()* will assign a new dictionary to it.

警告

tp_dict に *PyDict_SetItem()* を使ったり、辞書 C-API で編集するのは安全ではありません。

descrgetfunc *PyTypeObject*.**tp_descr_get**

オプションのポインタで、デスクリプタの get 関数を指します。

関数のシグネチャは次のとおりです

```
PyObject * tp_descr_get(PyObject *self, PyObject *obj, PyObject *type);
```

継承:

サブタイプはこのフィールドを継承します。

descrsetfunc *PyTypeObject*.**tp_descr_set**

オプションのポインタで、デスクリプタの値の設定と削除を行う関数を指します。

関数のシグネチャは次のとおりです

```
int tp_descr_set(PyObject *self, PyObject *obj, PyObject *value);
```

値を削除するには、*value* 引数に NULL を設定します。

継承:

サブタイプはこのフィールドを継承します。

Py_ssize_t *PyTypeObject*.**tp_dictoffset**

While this field is still supported, *Py_TPFLAGS_MANAGED_DICT* should be used instead, if at all possible.

型のインスタンスにインスタンス変数の入った辞書がある場合、このフィールドは非ゼロの値になり、型のインスタンスデータ構造体におけるインスタンス変数辞書へのオフセットが入ります; このオフセット値は *PyObject_GenericGetAttr()* が使います。

このフィールドを *tp_dict* と混同しないようにしてください; これは型オブジェクト自身の属性の辞書です。

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*.

継承:

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*.

デフォルト

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_dict* field, then *tp_dictoffset* will be set to `-1`, to indicate that it is unsafe to use this field.

initproc *PyTypeObject*.**tp_init**

オプションのポインタで、インスタンス初期化関数を指します。

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.

関数のシグネチャは次のとおりです

```
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__()`.

tp_init 関数のフィールドが `NULL` でない場合、通常の型を呼び出す方法のインスタンス生成において、型の *tp_new* 関数がインスタンスを返した後に呼び出されます。*tp_new* が元の型のサブタイプでない別の型を返す場合、*tp_init* は全く呼び出されません; *tp_new* が元の型のサブタイプのインスタンスを返す場合、サブタイプの *tp_init* が呼び出されます。

成功のときには `0` を、エラー時には例外をセットして `-1` を返します。

継承:

サブタイプはこのフィールドを継承します。

デフォルト

For *static types* this field does not have a default.

allocfunc *PyTypeObject*.**tp_alloc**

オプションのポインタで、インスタンスのメモリ確保関数を指します。

関数のシグネチャは次のとおりです

```
PyObject *tp_alloc(PyTypeObject *self, Py_ssize_t nitems);
```

継承:

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement).

デフォルト

For dynamic subtypes, this field is always set to *PyType_GenericAlloc()*, to force a standard heap allocation strategy.

For static subtypes, *PyBaseObject_Type* uses *PyType_GenericAlloc()*. That is the recommended value for all statically defined types.

newfunc *PyTypeObject*.**tp_new**

オプションのポインタで、インスタンス生成関数を指します。

関数のシグネチャは次のとおりです

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds);
```

subtype 引数は生成するオブジェクトの型です; *args* および *kwds* 引数は、型を呼び出すときの位置引数およびキーワード引数です。*subtype* は *tp_new* 関数を呼び出すときに使う型と同じである必要はないことに注意してください; その型の (無関係ではない) サブタイプのこともあります。

tp_new 関数は `subtype->tp_alloc(subtype, nitems)` を呼び出してオブジェクトのメモリ領域を確保し、初期化で絶対に必要とされる処理だけを行います。省略したり繰り返したりしても問題のない初期化処理は *tp_init* ハンドラ内に配置しなければなりません。だいたいの目安としては、変更不能な型では初期化は全て *tp_new* で行い、一方、変更可能な型ではほとんどの初期化を *tp_init* に回すべきです。

Set the *Py_TPFLAGS_DISALLOW_INSTANTIATION* flag to disallow creating instances of the type in Python.

継承:

サブタイプはこのフィールドを継承します。例外として、*tp_base* が `NULL` か `&PyBaseObject_Type` になっている **静的な型** では継承しません。

デフォルト

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**

オプションのポインタで、インスタンスのメモリ解放関数を指します。シグネチャは以下の通りです:

```
void tp_free(void *self);
```

An initializer that is compatible with this signature is *PyObject_Free()*.

継承:

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement)

デフォルト

In dynamic subtypes, this field is set to a deallocator suitable to match *PyType_GenericAlloc()* and the value of the *Py_TPFLAGS_HAVE_GC* flag bit.

For static subtypes, *PyBaseObject_Type* uses *PyObject_Del()*.

inquiry *PyTypeObject*.**tp_is_gc**

オプションのポインタで、ガベージコレクタから呼び出される関数を指します。

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);
```

(上記のような型の例は、型オブジェクト自体です。メタタイプ *PyType_Type* は、型のメモリ確保が静的か **動的** かを区別するためにこの関数を定義しています。)

継承:

サブタイプはこのフィールドを継承します。

デフォルト

This slot has no default. If this field is `NULL`, *Py_TPFLAGS_HAVE_GC* is used as the functional equivalent.

PyObject ***PyTypeObject**.**tp_bases**

基底型からなるタプルです。

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.

警告

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.

継承:

このフィールドは継承されません。

PyObject ***PyTypeObject**.**tp_mro**

基底タイプ群を展開した集合が入っているタプルです。集合は該当する型自体からはじまり、`object` で終わります。メソッド解決順序 (Method Resolution Order) に従って並んでいます。

This field should be set to `NULL` and treated as read-only. Python will fill it in when the type is *initialized*.

継承:

このフィールドは継承されません; フィールドの値は *PyType_Ready()* で毎回計算されます。

PyObject ***PyTypeObject*.tp_cache**

未使用のフィールドです。内部でのみ利用されます。

継承:

このフィールドは継承されません。

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__()`.

バージョン 3.12 で変更: For some types, this field does not hold a valid *PyObject**. The type was changed to `void*` to indicate this.

継承:

このフィールドは継承されません。

PyObject ***PyTypeObject*.tp_weaklist**

この型オブジェクトに対する弱参照からなるリストの先頭です。

バージョン 3.12 で変更: 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.

継承:

このフィールドは継承されません。

destructor ***PyTypeObject*.tp_del**

このフィールドは廃止されました。*tp_finalize* を代わりに利用してください。

unsigned int ***PyTypeObject*.tp_version_tag**

メソッドキャッシュへのインデックスとして使われます。内部使用だけのための関数です。

継承:

このフィールドは継承されません。

destructor ***PyTypeObject*.tp_finalize**

An optional pointer to an instance finalization function. Its signature is:

```
void tp_finalize(PyObject *self);
```

If *tp_finalize* is set, the interpreter calls it once when finalizing an instance. It is called either from the garbage collector (if the instance is part of an isolated reference cycle) or just before

the object is deallocated. Either way, it is guaranteed to be called before attempting to break reference cycles, ensuring that it finds the object in a sane state.

tp_finalize should not mutate the current exception status; therefore, a recommended way to write a non-trivial finalizer is:

```
static void
local_finalize(PyObject *self)
{
    PyObject *error_type, *error_value, *error_traceback;

    /* Save the current exception, if any. */
    PyErr_Fetch(&error_type, &error_value, &error_traceback);

    /* ... */

    /* Restore the saved exception. */
    PyErr_Restore(error_type, error_value, error_traceback);
}
```

継承:

サブタイプはこのフィールドを継承します。

Added in version 3.4.

バージョン 3.8 で変更: 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.

参考

”Safe object finalization” (**PEP 442**)

vectorcallfunc *PyTypeObject*.tp_vectorcall

Vectorcall function to use for calls of this type object. In other words, it is used to implement *vectorcall* for `type.__call__`. If `tp_vectorcall` is `NULL`, the default call implementation using `__new__()` and `__init__()` is used.

継承:

このフィールドは決して継承されません。

Added in version 3.9:（このフィールドは 3.8 から存在していますが、3.9 以降でしか利用できません）

unsigned char *PyTypeObject*.tp_watched

Internal. Do not use.

Added in version 3.12.

12.3.6 Static Types

Traditionally, types defined in C code are *static*, that is, a static *PyTypeObject* structure is defined directly in code and initialized using *PyType_Ready()*.

This results in types that are limited relative to types defined in Python:

- Static types are limited to one base, i.e. they cannot use multiple inheritance.
- Static type objects (but not necessarily their instances) are immutable. It is not possible to add or modify the type object's attributes from Python.
- Static type objects are shared across *sub-interpreters*, so they should not include any subinterpreter-specific state.

Also, since *PyTypeObject* is only part of the *Limited API* as an opaque struct, any extension modules using static types must be compiled for a specific Python minor version.

12.3.7 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()*.

12.3.8 数値オブジェクト構造体

type **PyNumberMethods**

この構造体は数値型プロトコルを実装するために使われる関数群へのポインタを保持しています。以下のそれぞれの関数は **数値型プロトコル** *(number protocol)* で解説されている似た名前の関数から利用されます。

以下は構造体の定義です:

```
typedef struct {
     binaryfunc nb_add;
     binaryfunc nb_subtract;
     binaryfunc nb_multiply;
     binaryfunc nb_remainder;
     binaryfunc nb_divmod;
     ternaryfunc nb_power;
     unaryfunc nb_negative;
     unaryfunc nb_positive;
     unaryfunc nb_absolute;
     inquiry nb_bool;
     unaryfunc nb_invert;
     binaryfunc nb_lshift;
```

(次のページに続く)

(前のページからの続き)

```
    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;
```

注釈

二項関数と三項関数は、すべてのオペランドの型をチェックしなければならず、必要な変換を実装しなければなりません (すくなくともオペランドの一つは定義している型のインスタンスです). もし与えられたオペランドに対して操作が定義されなければ、二項関数と三項関数は `Py_NotImplemented` を返さなければならず、他のエラーが起こった場合は、`NULL` を返して例外を設定しなければなりません。

注釈

The *nb_reserved* field should always be `NULL`. It was previously called `nb_long`, and was

renamed in Python 3.0.1.

binaryfunc *PyNumberMethods*.nb_add

binaryfunc *PyNumberMethods*.nb_subtract

binaryfunc *PyNumberMethods*.nb_multiply

binaryfunc *PyNumberMethods*.nb_remainder

binaryfunc *PyNumberMethods*.nb_divmod

ternaryfunc *PyNumberMethods*.nb_power

unaryfunc *PyNumberMethods*.nb_negative

unaryfunc *PyNumberMethods*.nb_positive

unaryfunc *PyNumberMethods*.nb_absolute

inquiry *PyNumberMethods*.nb_bool

unaryfunc *PyNumberMethods*.nb_invert

binaryfunc *PyNumberMethods*.nb_lshift

binaryfunc *PyNumberMethods*.nb_rshift

binaryfunc *PyNumberMethods*.nb_and

binaryfunc *PyNumberMethods*.nb_xor

binaryfunc *PyNumberMethods*.nb_or

unaryfunc *PyNumberMethods*.nb_int

void **PyNumberMethods*.nb_reserved

unaryfunc *PyNumberMethods*.nb_float

binaryfunc *PyNumberMethods*.nb_inplace_add

binaryfunc *PyNumberMethods*.nb_inplace_subtract

binaryfunc *PyNumberMethods*.nb_inplace_multiply

binaryfunc *PyNumberMethods*.**nb_inplace_remainder**

ternaryfunc *PyNumberMethods*.**nb_inplace_power**

binaryfunc *PyNumberMethods*.**nb_inplace_lshift**

binaryfunc *PyNumberMethods*.**nb_inplace_rshift**

binaryfunc *PyNumberMethods*.**nb_inplace_and**

binaryfunc *PyNumberMethods*.**nb_inplace_xor**

binaryfunc *PyNumberMethods*.**nb_inplace_or**

binaryfunc *PyNumberMethods*.**nb_floor_divide**

binaryfunc *PyNumberMethods*.**nb_true_divide**

binaryfunc *PyNumberMethods*.**nb_inplace_floor_divide**

binaryfunc *PyNumberMethods*.**nb_inplace_true_divide**

unaryfunc *PyNumberMethods*.**nb_index**

binaryfunc *PyNumberMethods*.**nb_matrix_multiply**

binaryfunc *PyNumberMethods*.**nb_inplace_matrix_multiply**

12.3.9 マップオブジェクト構造体

type **PyMappingMethods**

この構造体はマップ型プロトコルを実装するために使われる関数群へのポインタを保持しています。以下の 3 つのメンバを持っています:

lenfunc *PyMappingMethods*.**mp_length**

この関数は *PyMapping_Size()* や *PyObject_Size()* から利用され、それらと同じシグネチャを持っています。オブジェクトが定義された長さを持たない場合は、このスロットは NULL に設定されることがあります。

binaryfunc *PyMappingMethods*.**mp_subscript**

この関数は *PyObject_GetItem()* および *PySequence_GetSlice()* から利用され、PyObject_GetItem() と同じシグネチャを持っています。このスロットは *PyMapping_Check()* が 1 を返すためには必要で、そうでなければ NULL の場合があります。

objobjargproc *PyMappingMethods*.**mp_ass_subscript**

This function is used by *PyObject_SetItem()*, *PyObject_DelItem()*, *PySequence_SetSlice()* and *PySequence_DelSlice()*. It has the same signature as PyObject_SetItem(), but *v* can also

be set to NULL to delete an item. If this slot is NULL, the object does not support item assignment and deletion.

12.3.10 シーケンスオブジェクト構造体

type **PySequenceMethods**

この構造体はシーケンス型プロトコルを実装するために使われる関数群へのポインタを保持しています。

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**

この関数は *PySequence_Concat()* で利用され、同じシグネチャを持っています。また、+ 演算子でも、*nb_add* スロットによる数値加算を試した後に利用されます。

ssizeargfunc *PySequenceMethods*.**sq_repeat**

この関数は *PySequence_Repeat()* で利用され、同じシグネチャを持っています。また、* 演算でも、*nb_multiply* スロットによる数値乗算を試したあとに利用されます。

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**

この関数は *PySequence_Contains()* から利用され、同じシグネチャを持っています。このスロットは NULL の場合があり、その時 PySequence_Contains() はシンプルにマッチするオブジェクトを見つけるまでシーケンスを巡回します。

binaryfunc *PySequenceMethods*.**sq_inplace_concat**

This function is used by *PySequence_InPlaceConcat()* and has the same signature. It should modify its first operand, and return it. This slot may be left to NULL, in this case PySequence_InPlaceConcat() will fall back to *PySequence_Concat()*. It is also used by the augmented assignment +=, after trying numeric in-place addition via the *nb_inplace_add* slot.

ssizeargfunc *PySequenceMethods*.sq_inplace_repeat

This function is used by *PySequence_InPlaceRepeat()* and has the same signature. It should modify its first operand, and return it. This slot may be left to NULL, in this case PySequence_InPlaceRepeat() will fall back to *PySequence_Repeat()*. It is also used by the augmented assignment *=, after trying numeric in-place multiplication via the *nb_inplace_multiply* slot.

12.3.11 バッファオブジェクト構造体 (buffer object structure)

type PyBufferProcs

この構造体は *buffer* **プロトコル** が要求する関数群へのポインタを保持しています。そのプロトコルは、エクスポーターオブジェクトが如何にして、その内部データをコンシューマオブジェクトに渡すかを定義します。

getbufferproc *PyBufferProcs*.bf_getbuffer

この関数のシグネチャは以下の通りです:

```
int (PyObject *exporter, Py_buffer *view, int flags);
```

flags で指定された方法で *view* を埋めてほしいという *exporter* に対する要求を処理します。ステップ (3) を除いて、この関数の実装では以下のステップを行わなければなりません:

(1) Check if the request can be met. If not, raise BufferError, set view->obj to NULL and return -1.

(2) 要求されたフィールドを埋めます。

(3) エクスポートした回数を保持する内部カウンタをインクリメントします。

(4) view->obj に *exporter* を設定し、view->obj をインクリメントします。

(5) 0 を返します。

exporter がバッファプロバイダのチェインかツリーの一部であれば、2 つの主要な方式が使用できます:

- 再エクスポート: ツリーの各要素がエクスポートされるオブジェクトとして振る舞い、自身への新しい参照を view->obj へセットします。
- リダイレクト: バッファ要求がツリーのルートオブジェクトにリダイレクトされます。ここでは、view->obj はルートオブジェクトへの新しい参照になります。

view の個別のフィールドは **バッファ構造体** の節で説明されており、エクスポートが特定の要求に対しどう対応しなければならないかの規則は、**バッファ要求のタイプ** の節にあります。

Py_buffer 構造体の中から参照している全てのメモリはエクスポータに属し、コンシューマがいなくなるまで有効でなくてはなりません。*format* 、*shape* 、*strides* 、*suboffsets* 、*internal* はコンシューマからは読み出し専用です。

PyBuffer_FillInfo() は、全てのリクエストタイプを正しく扱う際に、単純なバイトバッファを公開する簡単な方法を提供します。

PyObject_GetBuffer() は、この関数をラップするコンシューマ向けのインターフェースです。

releasebufferproc *PyBufferProcs*.**bf_releasebuffer**

この関数のシグネチャは以下の通りです:

```
void (PyObject *exporter, Py_buffer *view);
```

バッファのリソースを開放する要求を処理します。もし開放する必要のあるリソースがない場合、*PyBufferProcs.bf_releasebuffer* は `NULL` にしても構いません。そうでない場合は、この関数の標準的な実装は、以下の任意の処理手順 (optional step) を行います:

(1) エクスポートした回数を保持する内部カウンタをデクリメントします。

(2) カウンタが `0` の場合は、*view* に関連付けられた全てのメモリを解放します。

エクスポータは、バッファ固有のリソースを監視し続けるために *internal* フィールドを使わなければなりません。このフィールドは、コンシューマが *view* 引数としてオリジナルのバッファのコピーを渡しているであろう間、変わらないことが保証されています。

この関数は、`view->obj` をデクリメントしてはいけません、なぜならそれは *PyBuffer_Release()* で自動的に行われるからです (この方式は参照の循環を防ぐのに有用です)。

PyBuffer_Release() は、この関数をラップするコンシューマ向けのインターフェースです。

12.3.12 async オブジェクト構造体

Added in version 3.5.

type **PyAsyncMethods**

この構造体は *awaitable* オブジェクトと *asynchronous iterator* オブジェクトを実装するのに必要な関数へのポインタを保持しています。

以下は構造体の定義です:

```
typedef struct {
    unaryfunc am_await;
    unaryfunc am_aiter;
    unaryfunc am_anext;
    sendfunc am_send;
} PyAsyncMethods;
```

unaryfunc *PyAsyncMethods*.**am_await**

この関数のシグネチャは以下の通りです:

```
PyObject *am_await(PyObject *self);
```

返されるオブジェクトは **イテレータ** でなければなりません。つまりこのオブジェクトに対して *PyIter_Check()* が 1 を返さなければなりません。

オブジェクトが *awaitable* でない場合、このスロットを NULL に設定します。

unaryfunc *PyAsyncMethods*.**am_aiter**

この関数のシグネチャは以下の通りです:

```
PyObject *am_aiter(PyObject *self);
```

Must return an *asynchronous iterator* object. See `__anext__()` for details.

オブジェクトが非同期反復処理のプロトコルを実装していない場合、このスロットを NULL に設定します。

unaryfunc *PyAsyncMethods*.**am_anext**

この関数のシグネチャは以下の通りです:

```
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**

この関数のシグネチャは以下の通りです:

```
PySendResult am_send(PyObject *self, PyObject *arg, PyObject **result);
```

See *PyIter_Send()* for details. This slot may be set to NULL.

Added in version 3.10.

12.3.13 Slot Type typedefs

typedef *PyObject* *(***allocfunc**)(*PyTypeObject* *cls, *Py_ssize_t* nitems)

次に属します: 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*.

この関数では他のいかなるインスタンス初期化も行ってはなりません。追加のメモリ割り当てすらも行ってはなりません。そのような処理は *tp_new* で行われるべきです。

typedef void (*__destructor__)(*PyObject**)

次に属します: Stable ABI.

typedef void (*__freefunc__)(void*)

tp_free を参照してください。

typedef *PyObject* *(*__newfunc__)(*PyTypeObject**, *PyObject**, *PyObject**)

次に属します: Stable ABI. *tp_new* を参照してください。

typedef int (*__initproc__)(*PyObject**, *PyObject**, *PyObject**)

次に属します: Stable ABI. *tp_init* を参照してください。

typedef *PyObject* *(*__reprfunc__)(*PyObject**)

次に属します: Stable ABI. *tp_repr* を参照してください。

typedef *PyObject* *(*__getattrfunc__)(*PyObject* *self, char *attr)

次に属します: Stable ABI. オブジェクトの属性の値を返します。

typedef int (*__setattrfunc__)(*PyObject* *self, char *attr, *PyObject* *value)

次に属します: Stable ABI. オブジェクトの属性に値を設定します。属性を削除するには、value (実) 引数に `NULL` を設定します。

typedef *PyObject* *(*__getattrofunc__)(*PyObject* *self, *PyObject* *attr)

次に属します: Stable ABI. オブジェクトの属性の値を返します。

tp_getattro を参照してください。

typedef int (*__setattrofunc__)(*PyObject* *self, *PyObject* *attr, *PyObject* *value)

次に属します: Stable ABI. オブジェクトの属性に値を設定します。属性を削除するには、value (実) 引数に `NULL` を設定します。

tp_setattro を参照してください。

typedef *PyObject* *(*__descrgetfunc__)(*PyObject**, *PyObject**, *PyObject**)

次に属します: Stable ABI. *tp_descr_get* を参照してください。

typedef int (*__descrsetfunc__)(*PyObject**, *PyObject**, *PyObject**)

次に属します: Stable ABI. *tp_descr_set* を参照してください。

typedef *Py_hash_t* (*__hashfunc__)(*PyObject**)

次に属します: Stable ABI. *tp_hash* を参照してください。

typedef *PyObject* *(*__richcmpfunc__)(*PyObject**, *PyObject**, int)

次に属します: Stable ABI. *tp_richcompare* を参照してください。

typedef *PyObject* *(*__getiterfunc__)(*PyObject**)

次に属します: Stable ABI. *tp_iter* を参照してください。

typedef *PyObject* *(***iternextfunc**)(*PyObject**)

次に属します: Stable ABI. *tp_iternext* を参照してください。

typedef *Py_ssize_t* (***lenfunc**)(*PyObject**)

次に属します: Stable ABI.

typedef int (***getbufferproc**)(*PyObject**, *Py_buffer**, int)

次に属します: Stable ABI *(バージョン 3.12 より)*.

typedef void (***releasebufferproc**)(*PyObject**, *Py_buffer**)

次に属します: Stable ABI *(バージョン 3.12 より)*.

typedef *PyObject* *(***unaryfunc**)(*PyObject**)

次に属します: Stable ABI.

typedef *PyObject* *(***binaryfunc**)(*PyObject**, *PyObject**)

次に属します: Stable ABI.

typedef *PySendResult* (***sendfunc**)(*PyObject**, *PyObject**, *PyObject***)

am_send を参照してください。

typedef *PyObject* *(***ternaryfunc**)(*PyObject**, *PyObject**, *PyObject**)

次に属します: Stable ABI.

typedef *PyObject* *(***ssizeargfunc**)(*PyObject**, *Py_ssize_t*)

次に属します: Stable ABI.

typedef int (***ssizeobjargproc**)(*PyObject**, *Py_ssize_t*, *PyObject**)

次に属します: Stable ABI.

typedef int (***objobjproc**)(*PyObject**, *PyObject**)

次に属します: Stable ABI.

typedef int (***objobjargproc**)(*PyObject**, *PyObject**, *PyObject**)

次に属します: Stable ABI.

12.3.14 使用例

ここでは Python の型定義の簡単な例をいくつか挙げます。これらの例にはあなたが遭遇する共通的な利用例を含んでいます。いくつかの例ではトリッキーなコーナーケースを実演しています。より多くの例や実践的な情報、チュートリアルが必要なら、defining-new-types や new-types-topics を参照してください。

A basic *static type*:

```
typedef struct {
    PyObject_HEAD
    const char *data;
```

(次のページに続く)

(前のページからの続き)

```
} 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,
};
```

より冗長な初期化子を用いた古いコードを（特に CPython のコードベース中で）見かけることがあるかもしれません:

```
static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    "mymod.MyObject",               /* tp_name */
    sizeof(MyObject),               /* tp_basicsize */
    0,                              /* tp_itemsize */
    (destructor)myobj_dealloc,      /* tp_dealloc */
    0,                              /* tp_vectorcall_offset */
    0,                              /* tp_getattr */
    0,                              /* tp_setattr */
    0,                              /* tp_as_async */
    (reprfunc)myobj_repr,           /* tp_repr */
    0,                              /* tp_as_number */
    0,                              /* tp_as_sequence */
    0,                              /* tp_as_mapping */
    0,                              /* tp_hash */
    0,                              /* tp_call */
    0,                              /* tp_str */
    0,                              /* tp_getattro */
    0,                              /* tp_setattro */
    0,                              /* tp_as_buffer */
    0,                              /* tp_flags */
    PyDoc_STR("My objects"),        /* tp_doc */
    0,                              /* tp_traverse */
    0,                              /* tp_clear */
    0,                              /* tp_richcompare */
    0,                              /* tp_weaklistoffset */
    0,                              /* tp_iter */
```

(次のページに続く)

(前のページからの続き)

```
    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 */
};
```

弱参照やインスタンス辞書、ハッシュをサポートする型:

```
typedef struct {
    PyObject_HEAD
    const char *data;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = PyDoc_STR("My objects"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
        Py_TPFLAGS_HAVE_GC | Py_TPFLAGS_MANAGED_DICT |
        Py_TPFLAGS_MANAGED_WEAKREF,
    .tp_new = myobj_new,
    .tp_traverse = (traverseproc)myobj_traverse,
    .tp_clear = (inquiry)myobj_clear,
    .tp_alloc = PyType_GenericNew,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
    .tp_hash = (hashfunc)myobj_hash,
    .tp_richcompare = PyBaseObject_Type.tp_richcompare,
};
```

A str subclass that cannot be subclassed and cannot be called to create instances (e.g. uses a separate factory func) using *Py_TPFLAGS_DISALLOW_INSTANTIATION* flag:

```
typedef struct {
    PyUnicodeObject raw;
    char *extra;
} MyStr;

static PyTypeObject MyStr_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyStr",
    .tp_basicsize = sizeof(MyStr),
    .tp_base = NULL,  // set to &PyUnicode_Type in module init
    .tp_doc = PyDoc_STR("my custom str"),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_DISALLOW_INSTANTIATION,
    .tp_repr = (reprfunc)myobj_repr,
};
```

The simplest *static type* with fixed-length instances:

```
typedef struct {
    PyObject_HEAD
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
};
```

The simplest *static type* with variable-length instances:

```
typedef struct {
    PyObject_VAR_HEAD
    const char *data[1];
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject) - sizeof(char *),
    .tp_itemsize = sizeof(char *),
};
```

12.4 循環参照ガベージコレクションをサポートする

Python が循環参照を含むガベージの検出とコレクションをサポートするには、他のオブジェクトに対する "コンテナ" (他のオブジェクトには他のコンテナも含みます) となるオブジェクト型によるサポートが必要です。他のオブジェクトに対する参照を記憶しないオブジェクトや、(数値や文字列のような) アトム型 (atomic type) への参照だけを記憶するような型では、ガベージコレクションに際して特別これといったサポートを提供する必要はありません。

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 こ
のフラグをセットした型のオブジェクトは、この節に述べた規則に適合しなければなりません。簡単のため、このフラグをセットした型のオブジェクトをコンテナオブジェクトと呼びます。

コンテナ型のコンストラクタは以下の二つの規則に適合しなければなりません:

1. The memory for the object must be allocated using *PyObject_GC_New* or *PyObject_GC_NewVar*.
2. 他のコンテナへの参照が入るかもしれないフィールドが全て初期化されたら、すぐに *PyObject_GC_Track()* を呼び出さなければなりません。

同様に、オブジェクトのメモリ解放関数も以下の二つの規則に適合しなければなりません:

1. 他のコンテナを参照しているフィールドを無効化する前に、*PyObject_GC_UnTrack()* を呼び出さなければなりません。
2. オブジェクトのメモリは *PyObject_GC_Del()* で解放しなければなりません。

> **警告**
>
> If a type adds the Py_TPFLAGS_HAVE_GC, then it *must* implement at least a *tp_traverse* handler or explicitly use one from its subclass or subclasses.
>
> When calling *PyType_Ready()* or some of the APIs that indirectly call it like *PyType_FromSpecWithBases()* or *PyType_FromSpec()* the interpreter will automatically populate the *tp_flags*, *tp_traverse* and *tp_clear* fields if the type inherits from a class that implements the garbage collector protocol and the child class does *not* include the *Py_TPFLAGS_HAVE_GC* flag.

PyObject_GC_New(TYPE, typeobj)

Analogous to *PyObject_New* but for container objects with the *Py_TPFLAGS_HAVE_GC* flag set.

PyObject_GC_NewVar(TYPE, typeobj, size)

Analogous to *PyObject_NewVar* but for container objects with the *Py_TPFLAGS_HAVE_GC* flag set.

PyObject ***PyUnstable_Object_GC_NewWithExtraData**(*PyTypeObject* *type, size_t extra_size)

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

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.

警告

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.

Added in 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)

次に属します: Stable ABI. オブジェクト *op* を、コレクタによって追跡されるオブジェクトの集合に追加します。コレクタは何回動くのかは予想できないので、追跡されている間はオブジェクトは正しい状態でいなければなりません。*tp_traverse* の対象となる全てのフィールドが正しい状態になってすぐに、たいていはコンストラクタの末尾付近で、呼び出すべきです。

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)

次に属します: Stable ABI *(**バージョン** 3.9 **より**)*. 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()`.

Added in version 3.9.

int `PyObject_GC_IsFinalized`(*PyObject* *op)

次に属します: Stable ABI *(**バージョン** 3.9 **より**)*. 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()`.

Added in version 3.9.

void **PyObject_GC_Del**(void *op)

次に属します: Stable ABI. Releases memory allocated to an object using *PyObject_GC_New* or *PyObject_GC_NewVar*.

void **PyObject_GC_UnTrack**(void *op)

次に属します: Stable ABI. オブジェクト *op* を、コレクタによって追跡されるオブジェクトの集合から除去します。このオブジェクトに対して *PyObject_GC_Track()* を再度呼び出して、追跡されるオブジェクトの集合に戻すことも可能です。*tp_traverse* ハンドラの対象となるフィールドが正しくない状態になる前に、デアロケータ (*tp_dealloc* ハンドラ) はオブジェクトに対して、この関数を呼び出すべきです。

バージョン 3.8 で変更: The `_PyObject_GC_TRACK()` and `_PyObject_GC_UNTRACK()` macros have been removed from the public C API.

tp_traverse ハンドラはこの型の関数パラメータを受け取ります:

typedef int (***visitproc**)(*PyObject* *object, void *arg)

次に属します: Stable ABI. *tp_traverse* ハンドラに渡されるビジター関数 (visitor function) の型です。この関数は、探索するオブジェクトを *object* として、*tp_traverse* ハンドラの第 3 引数を *arg* として呼び出します。Python のコアはいくつかのビジター関数を使って、ゴミとなった循環参照を検出する仕組みを実装します; ユーザが自身のためにビジター関数を書く必要が出てくることはないでしょう。

tp_traverse ハンドラは次の型を持っていなければなりません:

typedef int (***traverseproc**)(*PyObject* *self, *visitproc* visit, void *arg)

次に属します: Stable ABI. コンテナオブジェクトのためのトラバーサル関数 (traversal function) です。実装では、*self* に直接入っている各オブジェクトに対して *visit* 関数を呼び出さなければなりません。このとき、*visit* へのパラメタはコンテナに入っている各オブジェクトと、このハンドラに渡された *arg* の値です。*visit* 関数は `NULL` オブジェクトを引数に渡して呼び出してはなりません。*visit* が非ゼロの値を返す場合、エラーが発生し、戻り値をそのまま返すようにしなければなりません。

tp_traverse ハンドラを簡潔に書くために、*Py_VISIT()* マクロが提供されています。このマクロを使うためには、*tp_traverse* の実装関数の引数は、一文字も違わず *visit* と *arg* でなければなりません:

void **Py_VISIT**(*PyObject* *o)

If *o* is not `NULL`, call the *visit* callback, with arguments *o* and *arg*. If *visit* returns a non-zero value, then return it. Using this macro, *tp_traverse* handlers look like:

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
```

(次のページに続く)

(前のページからの続き)

```
    Py_VISIT(self->bar);
    return 0;
}
```

tp_clear ハンドラは *inquiry* 型であるか、オブジェクトが不変 (immutable) な場合は NULL でなければなりません。

typedef int (***inquiry**)(*PyObject* *self)

次に属します: Stable ABI. 循環参照を形成しているとおぼしき参照群を放棄します。変更不可能なオブジェクトは循環参照を直接形成することが決してないので、この関数を定義する必要はありません。このメソッドを呼び出した後でもオブジェクトは有効なままでなければならないので注意してください (参照に対して *Py_DECREF()* を呼ぶだけにしないでください)。ガベージコレクタは、オブジェクトが循環参照を形成していることを検出した際にこのメソッドを呼び出します。

12.4.1 Controlling the Garbage Collector State

The C-API provides the following functions for controlling garbage collection runs.

Py_ssize_t **PyGC_Collect**(void)

次に属します: 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)

次に属します: Stable ABI *(バージョン 3.10 より)*. Enable the garbage collector: similar to `gc.enable()`. Returns the previous state, 0 for disabled and 1 for enabled.

Added in version 3.10.

int **PyGC_Disable**(void)

次に属します: Stable ABI *(バージョン 3.10 より)*. Disable the garbage collector: similar to `gc.disable()`. Returns the previous state, 0 for disabled and 1 for enabled.

Added in version 3.10.

int **PyGC_IsEnabled**(void)

次に属します: Stable ABI *(バージョン 3.10 より)*. Query the state of the garbage collector: similar to `gc.isenabled()`. Returns the current state, 0 for disabled and 1 for enabled.

Added in version 3.10.

12.4.2 Querying Garbage Collector State

The C-API provides the following interface for querying information about the garbage collector.

void **PyUnstable_GC_VisitObjects**(*gcvisitobjects_t* callback, void *arg)

これは *Unstable API* です。マイナーリリースで予告なく変更されることがあります。

Run supplied *callback* on all live GC-capable objects. *arg* is passed through to all invocations of *callback*.

警告

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.

Added in 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 `0` to continue iteration, return `1` to stop iteration. Other return values are reserved for now so behavior on returning anything else is undefined.

Added in version 3.12.

第

THIRTEEN

API と ABI のバージョニング

CPython exposes its version number in the following macros. Note that these correspond to the version code is **built** with, not necessarily the version used at **run time**.

バージョン間の API と ABI の安定性については *C API* **の安定性** を参照してください。

`PY_MAJOR_VERSION`

`3.4.1a2` の `3` 。

`PY_MINOR_VERSION`

`3.4.1a2` の `4` 。

`PY_MICRO_VERSION`

`3.4.1a2` の `1` 。

`PY_RELEASE_LEVEL`

`3.4.1a2` の `a` 。アルファでは `0xA` 、ベータでは `0xB` 、リリース候補では `0xC` 、最終版では `0xF` となります。

`PY_RELEASE_SERIAL`

`3.4.1a2` の `2` 。最終リリースでは 0 になります。

`PY_VERSION_HEX`

The Python version number encoded in a single integer.

The underlying version information can be found by treating it as a 32 bit number in the following manner:

Bytes	Bits (big endian order)	Meaning	Value for `3.4.1a2`
1	1-8	`PY_MAJOR_VERSION`	`0x03`
2	9-16	`PY_MINOR_VERSION`	`0x04`
3	17-24	`PY_MICRO_VERSION`	`0x01`
4	25-28	`PY_RELEASE_LEVEL`	`0xA`
	29-32	`PY_RELEASE_SERIAL`	`0x2`

Thus `3.4.1a2` is hexversion `0x030401a2` and `3.10.0` is hexversion `0x030a00f0`.

Use this for numeric comparisons, e.g. `#if PY_VERSION_HEX >= ...`.

This version is also available via the symbol *Py_Version*.

const unsigned long **Py_Version**

次に属します: Stable ABI *(**バージョン** 3.11 **より**)*. The Python runtime version number encoded in a single constant integer, with the same format as the *PY_VERSION_HEX* macro. This contains the Python version used at run time.

Added in version 3.11.

All the given macros are defined in Include/patchlevel.h.

付録

A

用語集

>>>

The default Python prompt of the interactive shell. Often seen for code examples which can be executed interactively in the interpreter.

...

次のものが考えられます:

- 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.
- 組み込みの定数 `Ellipsis` 。

2to3

A tool that tries to convert Python 2.x code to Python 3.x code by handling most of the incompatibilities which can be detected by parsing the source and traversing the parse tree.

2to3 is available in the standard library as `lib2to3`; a standalone entry point is provided as `Tools/scripts/2to3`. See 2to3-reference.

abstract base class

(抽象基底クラス) 抽象基底クラスは *duck-typing* を補完するもので、`hasattr()` などの別のテクニックでは不恰好であったり微妙に誤る (例えば magic methods の場合) 場合にインターフェースを定義する方法を提供します。ABC は仮想 (virtual) サブクラスを導入します。これは親クラスから継承しませんが、それでも `isinstance()` や `issubclass()` に認識されます; `abc` モジュールのドキュメントを参照してください。Python には、多くの組み込み ABC が同梱されています。その対象は、(`collections.abc` モジュールで) データ構造、(`numbers` モジュールで) 数、(`io` モジュールで) ストリーム、(`importlib.abc` モジュールで) インポートファインダ及びローダーです。`abc` モジュールを利用して独自の ABC を作成できます。

annotation

(アノテーション) 変数、クラス属性、関数のパラメータや返り値に関係するラベルです。慣例により *type hint* として使われています。

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions are stored in the `__annotations__` special attribute of modules,

classes, and functions, respectively.

See *variable annotation*, *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

引数 (argument)

(実引数) 関数を呼び出す際に、**関数** (または **メソッド**) に渡す値です。実引数には 2 種類あります:

- **キーワード引数**: 関数呼び出しの際に引数の前に識別子がついたもの (例: `name=`) や、`**` に続けた辞書の中の値として渡された引数。例えば、次の `complex()` の呼び出しでは、`3` と `5` がキーワード引数です:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- **位置引数**: キーワード引数以外の引数。位置引数は引数リストの先頭に書くことができ、また `*` に続けた *iterable* の要素として渡すことができます。例えば、次の例では `3` と `5` は両方共位置引数です:

```
complex(3, 5)
complex(*(3, 5))
```

実引数は関数の実体において名前付きのローカル変数に割り当てられます。割り当てを行う規則については calls を参照してください。シンタックスにおいて実引数を表すためにあらゆる式を使うことが出来ます。評価された値はローカル変数に割り当てられます。

仮引数 、FAQ の 実引数と仮引数の違いは何ですか? 、**PEP 362** を参照してください。

asynchronous context manager

(非同期コンテキストマネージャ) `__aenter__()` と `__aexit__()` メソッドを定義することで `async with` 文内の環境を管理するオブジェクトです。**PEP 492** で導入されました。

asynchronous generator

(非同期ジェネレータ) *asynchronous generator iterator* を返す関数です。`async def` で定義されたコルーチン関数に似ていますが、`yield` 式を持つ点で異なります。`yield` 式は `async for` ループで使用できる値の並びを生成するのに使用されます。

通常は非同期ジェネレータ関数を指しますが、文脈によっては **非同期ジェネレータイテレータ** を指す場合があります。意図された意味が明らかでない場合、明瞭化のために完全な単語を使用します。

非同期ジェネレータ関数には、`async for` 文や `async with` 文だけでなく `await` 式もあることがあります。

asynchronous generator iterator

(非同期ジェネレータイテレータ) *asynchronous generator* 関数で生成されるオブジェクトです。

これは *asynchronous iterator* で、`__anext__()` メソッドを使って呼ばれると awaitable オブジェクトを返します。この awaitable オブジェクトは、次の `yield` 式まで非同期ジェネレータ関数の本体を実行します。

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**.

asynchronous iterable
: (非同期イテラブル) `async for` 文の中で使用できるオブジェクトです。自身の `__aiter__()` メソッドから *asynchronous iterator* を返さなければなりません。**PEP 492** で導入されました。

asynchronous iterator
: (非同期イテレータ) `__aiter__()` と `__anext__()` メソッドを実装したオブジェクトです。`__anext__()` は *awaitable* オブジェクトを返さなければなりません。`async for` は `StopAsyncIteration` 例外を送出するまで、非同期イテレータの `__anext__()` メソッドが返す awaitable を解決します。**PEP 492** で導入されました。

属性
: (属性) オブジェクトに関連付けられ、ドット表記式によって名前で通常参照される値です。例えば、オブジェクト *o* が属性 *a* を持っているとき、その属性は *o.a* で参照されます。

  オブジェクトには、identifiers で定義される識別子ではない名前の属性を与えることができます。たとえば `setattr()` を使い、オブジェクトがそれを許可している場合に行えます。このような属性はドット表記式ではアクセスできず、代わりに `getattr()` を使って取る必要があります。

awaitable
: (待機可能) `await` 式で使用することが出来るオブジェクトです。*coroutine* か、`__await__()` メソッドがあるオブジェクトです。**PEP 492** を参照してください。

BDFL
: 慈悲深き終身独裁者 (Benevolent Dictator For Life) の略です。Python の作者、Guido van Rossum のことです。

binary file
: (バイナリファイル) *bytes-like* **オブジェクト** の読み込みおよび書き込みができる **ファイルオブジェクト** です。バイナリファイルの例は、バイナリモード (`'rb'`, `'wb'` or `'rb+'`) で開かれたファイル、`sys.stdin.buffer`、`sys.stdout.buffer`、`io.BytesIO` や `gzip.GzipFile` のインスタンスです。

  `str` オブジェクトの読み書きができるファイルオブジェクトについては、*text file* も参照してください。

borrowed reference
: 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.

  Calling *Py_INCREF()* on the *borrowed reference* is recommended to convert it to a *strong reference* in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The *Py_NewRef()* function can be used to create a new *strong reference*.

bytes-like object
: **バッファプロトコル** *(buffer Protocol)* をサポートしていて、C 言語の意味で **連続した** バッファーを提供

可能なオブジェクト。bytes, bytearray, array.array や、多くの一般的な memoryview オブジェクトがこれに当たります。bytes-like オブジェクトは、データ圧縮、バイナリファイルへの保存、ソケットを経由した送信など、バイナリデータを要求するいろいろな操作に利用することができます。

幾つかの操作ではバイナリデータを変更する必要があります。その操作のドキュメントではよく ” 読み書き可能な bytes-like オブジェクト” に言及しています。変更可能なバッファーオブジェクトには、bytearray と bytearray の memoryview などが含まれます。また、他の幾つかの操作では不変なオブジェクト内のバイナリデータ (” 読み出し専用の bytes-like オブジェクト”) を必要します。それには bytes と bytes の memoryview オブジェクトが含まれます。

bytecode

(バイトコード) Python のソースコードは、Python プログラムの CPython インタプリタの内部表現であるバイトコードへとコンパイルされます。バイトコードは .pyc ファイルにキャッシュされ、同じファイルが二度目に実行されるときはより高速になります (ソースコードからバイトコードへの再度のコンパイルは回避されます)。この ” 中間言語 (intermediate language)” は、各々のバイトコードに対応する機械語を実行する **仮想マシン** で動作するといえます。重要な注意として、バイトコードは異なる Python 仮想マシン間で動作することや、Python リリース間で安定であることは期待されていません。

バイトコードの命令一覧は dis モジュール にあります。

callable

A callable is an object that can be called, possibly with a set of arguments (see *argument*), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A *function*, and by extension a *method*, is a callable. An instance of a class that implements the `__call__()` method is also a callable.

callback

(コールバック) 将来のある時点で実行されるために引数として渡される関数

クラス

(クラス) ユーザー定義オブジェクトを作成するためのテンプレートです。クラス定義は普通、そのクラスのインスタンス上の操作をするメソッドの定義を含みます。

class variable

(クラス変数) クラス上に定義され、クラスレベルで (つまり、クラスのインスタンス上ではなしに) 変更されることを目的としている変数です。

complex number

(複素数) よく知られている実数系を拡張したもので、すべての数は実部と虚部の和として表されます。虚数は虚数単位 (-1 の平方根) に実数を掛けたもので、一般に数学では i と書かれ、工学では j と書かれます。Python は複素数に組み込みで対応し、後者の表記を取っています。虚部は末尾に j をつけて書きます。例えば 3+1j です。math モジュールの複素数版を利用するには、cmath を使います。複素数の使用はかなり高度な数学の機能です。必要性を感じなければ、ほぼ間違いなく無視してしまってよいでしょう。

context manager
: An object which controls the environment seen in a `with` statement by defining `__enter__()` and `__exit__()` methods. See **PEP 343**.

context variable
: A variable which can have different values depending on its context. This is similar to Thread-Local Storage in which each execution thread may have a different value for a variable. However, with context variables, there may be several contexts in one execution thread and the main usage for context variables is to keep track of variables in concurrent asynchronous tasks. See `contextvars`.

contiguous
: (隣接、連続) バッファが厳密に *C-**連続*** または *Fortran **連続*** である場合に、そのバッファは連続しているとみなせます。ゼロ次元バッファは C 連続であり Fortran 連続です。一次元の配列では、その要素は必ずメモリ上で隣接するように配置され、添字がゼロから始まり増えていく順序で並びます。多次元の C-連続な配列では、メモリアドレス順に要素を巡る際には最後の添え字が最初に変わるのに対し、Fortran 連続な配列では最初の添え字が最初に動きます。

コルーチン
: (コルーチン) コルーチンはサブルーチンのより一般的な形式です。サブルーチンには決められた地点から入り、別の決められた地点から出ます。コルーチンには多くの様々な地点から入る、出る、再開することができます。コルーチンは `async def` 文で実装できます。**PEP 492** を参照してください。

coroutine function
: (コルーチン関数) *coroutine* オブジェクトを返す関数です。コルーチン関数は `async def` 文で実装され、`await`、`async for`、および `async with` キーワードを持つことが出来ます。これらは **PEP 492** で導入されました。

CPython
: python.org で配布されている、Python プログラミング言語の標準的な実装です。”CPython” という単語は、この実装を Jython や IronPython といった他の実装と区別する必要が有る場合に利用されます。

decorator
: (デコレータ) 別の関数を返す関数で、通常、`@wrapper` 構文で関数変換として適用されます。デコレータの一般的な利用例は、`classmethod()` と `staticmethod()` です。

  デコレータの文法はシンタックスシュガーです。次の 2 つの関数定義は意味的に同じものです:

```
def f(arg):
    ...
f = staticmethod(f)

@staticmethod
def f(arg):
    ...
```

  同じ概念がクラスにも存在しますが、あまり使われません。デコレータについて詳しくは、関数定義 お

よび クラス定義 のドキュメントを参照してください。

descriptor

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.

デスクリプタのメソッドに関しての詳細は、descriptors や Descriptor How To Guide を参照してください。

dictionary

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.

dictionary comprehension

(辞書内包表記) iterable 内の全てあるいは一部の要素を処理して、その結果からなる辞書を返すコンパクトな書き方です。`results = {n: n ** 2 for n in range(10)}` とすると、キー `n` を値 `n ** 2` に対応付ける辞書を生成します。comprehensions を参照してください。

dictionary view

(辞書ビュー) `dict.keys()`、`dict.values()`、`dict.items()` が返すオブジェクトです。辞書の項目の動的なビューを提供します。すなわち、辞書が変更されるとビューはそれを反映します。辞書ビューを強制的に完全なリストにするには `list(dictview)` を使用してください。dict-views を参照してください。

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.

duck-typing

あるオブジェクトが正しいインターフェースを持っているかを決定するのにオブジェクトの型を見ないプログラミングスタイルです。代わりに、単純にオブジェクトのメソッドや属性が呼ばれたり使われたりします。(「アヒルのように見えて、アヒルのように鳴けば、それはアヒルである。」) インターフェースを型より重視することで、上手くデザインされたコードは、ポリモーフィックな代替を許して柔軟性を向上させます。ダックタイピングは `type()` や `isinstance()` による判定を避けます。(ただし、ダックタイピングを **抽象基底クラス** で補完することもできます。) その代わり、典型的に `hasattr()` 判定や *EAFP* プログラミングを利用します。

EAFP

「認可をとるより許しを請う方が容易 (easier to ask for forgiveness than permission、マーフィーの法則)」の略です。この Python で広く使われているコーディングスタイルでは、通常は有効なキーや属性が存在するものと仮定し、その仮定が誤っていた場合に例外を捕捉します。この簡潔で手早く書ける

コーディングスタイルには、try 文および except 文がたくさんあるのが特徴です。このテクニックは、C のような言語でよく使われている *LBYL* スタイルと対照的なものです。

expression
: (式) 何かの値と評価される、一まとまりの構文 (a piece of syntax) です。言い換えると、式とはリテラル、名前、属性アクセス、演算子や関数呼び出しなど、値を返す式の要素の積み重ねです。他の多くの言語と違い、Python では言語の全ての構成要素が式というわけではありません。while のように、式としては使えない **文** もあります。代入も式ではなく文です。

extension module
: (拡張モジュール) C や C++ で書かれたモジュールで、Python の C API を利用して Python コアやユーザーコードとやりとりします。

f-string
: 'f' や 'F' が先頭に付いた文字列リテラルは "f-string" と呼ばれ、これは フォーマット済み文字列リテラル の短縮形の名称です。**PEP 498** も参照してください。

file object
: 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*.

  ファイルオブジェクトには実際には 3 種類あります: 生の **バイナリーファイル**、バッファされた **バイナリーファイル**、そして **テキストファイル** です。インターフェイスは io モジュールで定義されています。ファイルオブジェクトを作る標準的な方法は open() 関数を使うことです。

file-like object
: *file object* と同義です。

filesystem encoding and error handler
: Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

  ファイルシステムのエンコーディングでは、すべてが 128 バイト以下に正常にデコードされることが保証されなくてはなりません。ファイルシステムのエンコーディングでこれが保証されなかった場合は、API 関数が UnicodeError を送出することがあります。

  The sys.getfilesystemencoding() and sys.getfilesystemencodeerrors() functions can be used to get the filesystem encoding and error handler.

  The *filesystem encoding and error handler* are configured at Python startup by the *PyConfig_Read()* function: see *filesystem_encoding* and *filesystem_errors* members of *PyConfig*.

  See also the *locale encoding*.

finder

(ファインダ) インポートされているモジュールの *loader* の発見を試行するオブジェクトです。

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.

floor division
: (切り捨て除算) 一番近い整数に切り捨てる数学的除算。切り捨て除算演算子は `//` です。例えば、`11 // 4` は `2` になり、それとは対称に浮動小数点数の真の除算では `2.75` が 返ってきます。`(-11) // 4` は `-2.75` を **小さい方に** 丸める (訳注: 負の無限大への丸めを行う) ので `-3` になることに注意してください。**PEP 238** を参照してください。

関数
: (関数) 呼び出し側に値を返す一連の文のことです。関数には 0 以上の **実引数** を渡すことが出来ます。実体の実行時に引数を使用することが出来ます。**仮引数**、**メソッド**、function を参照してください。

function annotation
: (関数アノテーション) 関数のパラメータや返り値の *annotation* です。

関数アノテーションは、通常は **型ヒント** のために使われます: 例えば、この関数は 2 つの `int` 型の引数を取ると期待され、また `int` 型の返り値を持つと期待されています。

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

関数アノテーションの文法は function の節で解説されています。

機能の説明がある *variable annotation*, **PEP 484**, を参照してください。また、アノテーションを利用するベストプラクティスとして annotations-howto も参照してください。

___future___
: `from __future__ import <feature>` という future 文 は、コンパイラーに将来の Python リリースで標準となる構文や意味を使用して現在のモジュールをコンパイルするよう指示します。`__future__` モジュールでは、*feature* のとりうる値をドキュメント化しています。このモジュールをインポートし、その変数を評価することで、新機能が最初に言語に追加されたのはいつかや、いつデフォルトになるか (またはなったか) を見ることができます:

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

garbage collection
: (ガベージコレクション) これ以降使われることのないメモリを解放する処理です。Python は、参照カウントと、循環参照を検出し破壊する循環ガベージコレクタを使ってガベージコレクションを行います。ガベージコレクタは `gc` モジュールを使って操作できます。

ジェネレータ
: (ジェネレータ) *generator iterator* を返す関数です。通常の関数に似ていますが、`yield` 式を持つ点で

異なります。`yield` 式は、for ループで使用できたり、`next()` 関数で値を 1 つずつ取り出したりできる、値の並びを生成するのに使用されます。

通常はジェネレータ関数を指しますが、文脈によっては **ジェネレータイテレータ** を指す場合があります。意図された意味が明らかでない場合、明瞭化のために完全な単語を使用します。

generator iterator
: (ジェネレータイテレータ) *generator* 関数で生成されるオブジェクトです。

  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).

generator expression
: 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
```

generic function
: (ジェネリック関数) 異なる型に対し同じ操作をする関数群から構成される関数です。呼び出し時にどの実装を用いるかはディスパッチアルゴリズムにより決定されます。

  single dispatch、`functools.singledispatch()` デコレータ、**PEP 443** を参照してください。

generic type
: A *type* that can be parameterized; typically a container class such as `list` or `dict`. Used for *type hints* and *annotations*.

  For more details, see generic alias types, **PEP 483**, **PEP 484**, **PEP 585**, and the `typing` module.

GIL
: *global interpreter lock* を参照してください。

global interpreter lock
: (グローバルインタプリタロック) *CPython* インタプリタが利用している、一度に Python の **バイトコード** を実行するスレッドは一つだけであることを保証する仕組みです。これにより (`dict` などの重要な組み込み型を含む) オブジェクトモデルが同時アクセスに対して暗黙的に安全になるので、CPython の実装がシンプルになります。インタプリタ全体をロックすることで、マルチプロセッサマシンが生じる並列化のコストと引き換えに、インタプリタを簡単にマルチスレッド化できるようになります。

  ただし、標準あるいは外部のいくつかの拡張モジュールは、圧縮やハッシュ計算などの計算の重い処理をするときに GIL を解除するように設計されています。また、I/O 処理をする場合 GIL は常に解除されます。

  Past efforts to create a "free-threaded" interpreter (one which locks shared data at a much finer

granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.

hash-based pyc
: (ハッシュベース pyc ファイル) 正当性を判別するために、対応するソースファイルの最終更新時刻ではなくハッシュ値を使用するバイトコードのキャッシュファイルです。pyc-invalidation を参照してください。

hashable
: 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.

  ハッシュ可能なオブジェクトは辞書のキーや集合のメンバーとして使えます。辞書や集合のデータ構造は内部でハッシュ値を使っているからです。

  Python のイミュータブルな組み込みオブジェクトは、ほとんどがハッシュ可能です。(リストや辞書のような) ミュータブルなコンテナはハッシュ不可能です。(タプルや frozenset のような) イミュータブルなコンテナは、要素がハッシュ可能であるときのみハッシュ可能です。ユーザー定義のクラスのインスタンスであるようなオブジェクトはデフォルトでハッシュ可能です。それらは全て (自身を除いて) 比較結果は非等価であり、ハッシュ値は `id()` より得られます。

IDLE
: Python の統合開発環境 (Integrated DeveLopment Environment) 及び学習環境 (Learning Environment) です。idle は Python の標準的な配布に同梱されている基本的な機能のエディタとインタプリタ環境です。

永続オブジェクト (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.

immutable
: (イミュータブル) 固定の値を持ったオブジェクトです。イミュータブルなオブジェクトには、数値、文字列、およびタプルなどがあります。これらのオブジェクトは値を変えられません。別の値を記憶させる際には、新たなオブジェクトを作成しなければなりません。イミュータブルなオブジェクトは、固定のハッシュ値が必要となる状況で重要な役割を果たします。辞書のキーがその例です。

import path
: *path based finder* が import するモジュールを検索する場所 (または *path entry*) のリスト。import 中、このリストは通常 `sys.path` から来ますが、サブパッケージの場合は親パッケージの `__path__` 属性からも来ます。

importing
: あるモジュールの Python コードが別のモジュールの Python コードで使えるようにする処理です。

importer
: モ

ジュールを探してロードするオブジェクト。*finder* と *loader* のどちらでもあるオブジェクト。

interactive

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)`).

interpreted

Python はインタプリタ形式の言語であり、コンパイラ言語の対極に位置します。(バイトコードコンパイラがあるために、この区別は曖昧ですが。) ここでのインタプリタ言語とは、ソースコードのファイルを、まず実行可能形式にしてから実行させるといった操作なしに、直接実行できることを意味します。インタプリタ形式の言語は通常、コンパイラ形式の言語よりも開発／デバッグのサイクルは短いものの、プログラムの実行は一般に遅いです。**対話的** も参照してください。

interpreter shutdown

Python インタープリターはシャットダウンを要請された時に、モジュールやすべてのクリティカルな内部構造をなどの、すべての確保したリソースを段階的に開放する、特別なフェーズに入ります。このフェーズは **ガベージコレクタ** を複数回呼び出します。これによりユーザー定義のデストラクターや weakref コールバックが呼び出されることがあります。シャットダウンフェーズ中に実行されるコードは、それが依存するリソースがすでに機能しない (よくある例はライブラリーモジュールや warning 機構です) ために様々な例外に直面します。

インタープリタがシャットダウンする主な理由は `__main__` モジュールや実行されていたスクリプトの実行が終了したことです。

iterable

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*.

iterator

データの流れを表現するオブジェクトです。イテレータの `__next__()` メソッドを繰り返し呼び出す (または組み込み関数 `next()` に渡す) と、流れの中の要素を一つずつ返します。データがなくなると、代わりに `StopIteration` 例外を送出します。その時点で、イテレータオブジェクトは尽きており、それ以降は `__next__()` を何度呼んでも `StopIteration` を送出します。イテレータは、そのイテレータオブジェクト自体を返す `__iter__()` メソッドを実装しなければならないので、イテレータは他の iterable を受理するほとんどの場所で利用できます。はっきりとした例外は複数の反復を行うようなコードです。(`list` のような) コンテナオブジェクトは、自身を `iter()` 関数にオブジェクトに渡した

り for ループ内で使うたびに、新たな未使用のイテレータを生成します。これをイテレータで行おうとすると、前回のイテレーションで使用済みの同じイテレータオブジェクトを単純に返すため、空のコンテナのようになってしまします。

詳細な情報は typeiter にあります。

CPython 実装の詳細: CPython does not consistently apply the requirement that an iterator define `__iter__()`.

key function

(キー関数) キー関数、あるいは照合関数とは、ソートや順序比較のための値を返す呼び出し可能オブジェクト (callable) です。例えば、`locale.strxfrm()` をキー関数に使えば、ロケール依存のソートの慣習にのっとったソートキーを返します。

Python の多くのツールはキー関数を受け取り要素の並び順やグループ化を管理します。`min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, `itertools.groupby()` 等があります。

キー関数を作る方法はいくつかあります。例えば `str.lower()` メソッドを大文字小文字を区別しないソートを行うキー関数として使うことが出来ます。あるいは、`lambda r: (r[0], r[2])` のような `lambda` 式からキー関数を作ることができます。また、`operator.attrgetter()`, `operator.itemgetter()`, `operator.methodcaller()` の 3 つのキー関数コンストラクタがあります。キー関数の作り方と使い方の例は Sorting HOW TO を参照してください。

keyword argument 実

引数 を参照してください。

lambda

(ラムダ) 無名のインライン関数で、関数が呼び出されたときに評価される 1 つの 式 を含みます。ラムダ関数を作る構文は `lambda [parameters]: expression` です。

LBYL

「ころばぬ先の杖 (look before you leap)」の略です。このコーディングスタイルでは、呼び出しや検索を行う前に、明示的に前提条件 (pre-condition) 判定を行います。*EAFP* アプローチと対照的で、`if` 文がたくさん使われるのが特徴的です。

マルチスレッド化された環境では、LBYL アプローチは ” 見る” 過程と ” 飛ぶ” 過程の競合状態を引き起こすリスクがあります。例えば、`if key in mapping: return mapping[key]` というコードは、判定の後、別のスレッドが探索の前に *mapping* から *key* を取り除くと失敗します。この問題は、ロックするか EAFP アプローチを使うことで解決できます。

lexical analyzer

Formal name for the *tokenizer*; see *token*.

list 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 $O(1)$.

list comprehension

(リスト内包表記) シーケンス中の全てあるいは一部の要素を処理して、その結果からなるリストを返す、コンパクトな方法です。`result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` とすると、0 から 255 までの偶数を 16 進数表記 (0x..) した文字列からなるリストを生成します。`if` 節はオプションです。`if` 節がない場合、`range(256)` の全ての要素が処理されます。

loader

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**

ロケールエンコーディング

On Unix, it is the encoding of the LC_CTYPE locale. It can be set with `locale.setlocale(locale.LC_CTYPE, new_locale)`.

On Windows, it is the ANSI code page (ex: `"cp1252"`).

On Android and VxWorks, Python uses `"utf-8"` as the locale encoding.

`locale.getencoding()` can be used to get the locale encoding.

See also the *filesystem encoding and error handler*.

magic method

special method のくだけた同義語です。

mapping

(マッピング) 任意のキー探索をサポートしていて、`collections.abc.Mapping` か `collections.abc.MutableMapping` の 抽象基底クラス で指定されたメソッドを実装しているコンテナオブジェクトです。例えば、`dict`, `collections.defaultdict`, `collections.OrderedDict`, `collections.Counter` などです。

meta path finder

`sys.meta_path` を検索して得られた *finder*. meta path finder は *path entry finder* と関係はありますが、別物です。

meta path finder が実装するメソッドについては `importlib.abc.MetaPathFinder` を参照してください。

metaclass

(メタクラス) クラスのクラスです。クラス定義は、クラス名、クラスの辞書と、基底クラスのリストを作ります。メタクラスは、それら 3 つを引数として受け取り、クラスを作る責任を負います。ほとんどのオブジェクト指向言語は (訳注: メタクラスの) デフォルトの実装を提供しています。Python が特別なのはカスタムのメタクラスを作成できる点です。ほとんどのユーザーに取って、メタクラスは全く必要のないものです。しかし、一部の場面では、メタクラスは強力でエレガントな方法を提供します。た

とえば属性アクセスのログを取ったり、スレッドセーフ性を追加したり、オブジェクトの生成を追跡したり、シングルトンを実装するなど、多くの場面で利用されます。

詳細は metaclasses を参照してください。

メソッド
: (メソッド) クラス本体の中で定義された関数。そのクラスのインスタンスの属性として呼び出された場合、メソッドはインスタンスオブジェクトを第一 **引数** として受け取ります (この第一引数は通常 `self` と呼ばれます)。**関数** と **ネストされたスコープ** も参照してください。

method resolution order
: 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
: (モジュール) Python コードの組織単位としてはたらくオブジェクトです。モジュールは任意の Python オブジェクトを含む名前空間を持ちます。モジュールは *importing* の処理によって Python に読み込まれます。

パッケージ を参照してください。

module spec
: モジュールをロードするのに使われるインポート関連の情報を含む名前空間です。`importlib.machinery.ModuleSpec` のインスタンスです。

See also module-specs.

MRO
: *method resolution order* を参照してください。

mutable
: (ミュータブル) ミュータブルなオブジェクトは、`id()` を変えることなく値を変更できます。**イミュータブル**) も参照してください。

named tuple
: ”名前付きタプル” という用語は、タプルを継承していて、インデックスが付く要素に対し属性を使ってのアクセスもできる任意の型やクラスに応用されています。その型やクラスは他の機能も持っていることもあります。

`time.localtime()` や `os.stat()` の返り値を含むいくつかの組み込み型は名前付きタプルです。他の例は `sys.float_info` です:

```
>>> 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.

namespace

(名前空間) 変数が格納される場所です。名前空間は辞書として実装されます。名前空間にはオブジェクトの (メソッドの) 入れ子になったものだけでなく、局所的なもの、大域的なもの、そして組み込みのものがあります。名前空間は名前の衝突を防ぐことによってモジュール性をサポートする。例えば関数 `builtins.open` と `os.open()` は名前空間で区別されています。また、どのモジュールが関数を実装しているか明示することによって名前空間は可読性と保守性を支援します。例えば、`random.seed()` や `itertools.islice()` と書くと、それぞれモジュール `random` や `itertools` で実装されていることが明らかです。

namespace package

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.

module を参照してください。

nested scope

(ネストされたスコープ) 外側で定義されている変数を参照する機能です。例えば、ある関数が別の関数の中で定義されている場合、内側の関数は外側の関数中の変数を参照できます。ネストされたスコープはデフォルトでは変数の参照だけができ、変数の代入はできないので注意してください。ローカル変数は、最も内側のスコープで変数を読み書きします。同様に、グローバル変数を使うとグローバル名前空間の値を読み書きします。`nonlocal` で外側の変数に書き込めます。

new-style class

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.

object

(オブジェクト) 状態 (属性や値) と定義された振る舞い (メソッド) をもつ全てのデータ。もしくは、全ての **新スタイルクラス** の究極の基底クラスのこと。

package

(パッケージ) サブモジュールや再帰的にサブパッケージを含むことの出来る *module* のことです。専門的には、パッケージは `__path__` 属性を持つ Python オブジェクトです。

regular package と *namespace package* を参照してください。

parameter

(仮引数) 名前付の実体で 関数 (や メソッド) の定義において関数が受ける 実引数 を指定します。仮引数には 5 種類あります:

- **位置またはキーワード**: 位置 であるいは キーワード引数 として渡すことができる引数を指定します。これはたとえば以下の *foo* や *bar* のように、デフォルトの仮引数の種類です:

```
def func(foo, bar=None): ...
```

- 位置専用: 位置によってのみ与えられる引数を指定します。位置専用の引数は 関数定義の引数のリストの中でそれらの後ろに `/` を含めることで定義できます。例えば下記の *posonly1* と *posonly2* は位置専用引数になります:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- **キーワード専用**: キーワードによってのみ与えられる引数を指定します。キーワード専用の引数を定義できる場所は、例えば以下の *kw_only1* や *kw_only2* のように、関数定義の仮引数リストに含めた可変長位置引数または裸の `*` の後です:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- **可変長位置**: (他の仮引数で既に受けられた任意の位置引数に加えて) 任意の個数の位置引数が与えられることを指定します。このような仮引数は、以下の *args* のように仮引数名の前に `*` をつけることで定義できます:

```
def func(*args, **kwargs): ...
```

- **可変長キーワード**: (他の仮引数で既に受けられた任意のキーワード引数に加えて) 任意の個数のキーワード引数が与えられることを指定します。このような仮引数は、上の例の *kwargs* のように仮引数名の前に `**` をつけることで定義できます。

仮引数はオプションと必須の引数のどちらも指定でき、オプションの引数にはデフォルト値も指定できます。

仮引数 、FAQ の 実引数と仮引数の違いは何ですか? 、`inspect.Parameter` クラス、function セクション、**PEP 362** を参照してください。

path entry

path based finder が import するモジュールを探す *import path* 上の 1 つの場所です。

path entry finder

`sys.path_hooks` にある callable (つまり *path entry hook*) が返した *finder* です。与えられた *path entry* にあるモジュールを見つける方法を知っています。

パスエントリーファインダが実装するメソッドについては `importlib.abc.PathEntryFinder` を参照してください。

path entry hook 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*.

path based finder
: デフォルトの *meta path finder* の 1 つは、モジュールの *import path* を検索します。

path-like object
: (path-like オブジェクト) ファイルシステムパスを表します。path-like オブジェクトは、パスを表す `str` オブジェクトや `bytes` オブジェクト、または `os.PathLike` プロトコルを実装したオブジェクトのどれかです。`os.PathLike` プロトコルをサポートしているオブジェクトは `os.fspath()` を呼び出すことで `str` または `bytes` のファイルシステムパスに変換できます。`os.fsdecode()` と `os.fsencode()` はそれぞれ `str` あるいは `bytes` になるのを保証するのに使えます。**PEP 519** で導入されました。

PEP
: Python Enhancement Proposal。PEP は、Python コミュニティに対して情報を提供する、あるいは Python の新機能やその過程や環境について記述する設計文書です。PEP は、機能についての簡潔な技術的仕様と提案する機能の論拠 (理論) を伝えるべきです。

  PEP は、新機能の提案にかかる、コミュニティによる問題提起の集積と Python になされる設計決断の文書化のための最上位の機構となることを意図しています。PEP の著者にはコミュニティ内の合意形成を行うこと、反対意見を文書化することの責務があります。

  PEP 1 を参照してください。

portion
: **PEP 420** で定義されている、namespace package に属する、複数のファイルが (zip ファイルに格納されている場合もある) 1 つのディレクトリに格納されたもの。

位置引数 (positional argument)
: **実引数** を参照してください。

provisional API
: (暫定 API) 標準ライブラリの後方互換性保証から計画的に除外されたものです。そのようなインターフェースへの大きな変更は、暫定であるとされている間は期待されていませんが、コア開発者によって必要とみなされれば、後方非互換な変更 (インターフェースの削除まで含まれる) が行われえます。このような変更はむやみに行われるものではありません -- これは API を組み込む前には見落とされていた重大な欠陥が露呈したときにのみ行われます。

  暫定 API についても、後方互換性のない変更は「最終手段」とみなされています。問題点が判明した場合でも後方互換な解決策を探すべきです。

  このプロセスにより、標準ライブラリは問題となるデザインエラーに長い間閉じ込められることなく、時代を超えて進化を続けられます。詳細は **PEP 411** を参照してください。

provisional package
: *provisional API* を参照してください。

Python 3000

Python 3.x リリースラインのニックネームです。(Python 3 が遠い将来の話だった頃に作られた言葉です。) "Py3k" と略されることもあります。

Pythonic
他の言語で一般的な考え方で書かれたコードではなく、Python の特に一般的なイディオムに従った考え方やコード片。例えば、Python の一般的なイディオムでは `for` 文を使ってイテラブルのすべての要素に渡ってループします。他の多くの言語にはこの仕組みはないので、Python に慣れていない人は代わりに数値のカウンターを使うかもしれません:

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

これに対し、きれいな Pythonic な方法は:

```
for piece in food:
    print(piece)
```

qualified name
(修飾名) モジュールのグローバルスコープから、そのモジュールで定義されたクラス、関数、メソッドへの、" パス" を表すドット名表記です。**PEP 3155** で定義されています。トップレベルの関数やクラスでは、修飾名はオブジェクトの名前と同じです:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

モジュールへの参照で使われると、**完全修飾名** *(fully qualified name)* はすべての親パッケージを含む全体のドット名表記、例えば `email.mime.text` を意味します:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count
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.

regular package
伝統的な、`__init__.py` ファイルを含むディレクトリとしての *package*。

namespace package を参照してください。

__slots__
クラス内での宣言で、インスタンス属性の領域をあらかじめ定義しておき、インスタンス辞書を排除することで、メモリを節約します。これはよく使われるテクニックですが、正しく扱うには少しトリッキーなので、稀なケース、例えばメモリが死活問題となるアプリケーションでインスタンスが大量に存在する、といったときを除き、使わないのがベストです。

sequence
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.

set comprehension
(集合内包表記) iterable 内の全てあるいは一部の要素を処理して、その結果からなる集合を返すコンパクトな書き方です。`results = {c for c in 'abracadabra' if c not in 'abc'}` とすると、`{'r', 'd'}` という文字列の辞書を生成します。comprehensions を参照してください。

single dispatch
generic function の一種で実装は一つの引数の型により選択されます。

slice
(スライス) 一般に **シーケンス** の一部を含むオブジェクト。スライスは、添字表記 `[]` で与えられた複数の数の間にコロンを書くことで作られます。例えば、`variable_name[1:3:5]` です。角括弧 (添字) 記号は `slice` オブジェクトを内部で利用しています。

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.

special method
: (特殊メソッド) ある型に特定の操作、例えば加算をするために Python から暗黙に呼び出されるメソッド。この種類のメソッドは、メソッド名の最初と最後にアンダースコア 2 つがついています。特殊メソッドについては specialnames で解説されています。

statement
: (文) 文はスイート (コードの” ブロック”) に不可欠な要素です。文は **式** かキーワードから構成されるもののどちらかです。後者には `if`、`while`、`for` があります。

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.

strong reference
: 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.

  The *Py_NewRef()* function can be used to create a strong reference to an object. Usually, the *Py_DECREF()* function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

  See also *borrowed reference*.

text encoding
: A string in Python is a sequence of Unicode code points (in range `U+0000--U+10FFFF`). To store or transfer a string, it needs to be serialized as a sequence of bytes.

  Serializing a string into a sequence of bytes is known as ”encoding”, and recreating the string from the sequence of bytes is known as ”decoding”.

  There are a variety of different text serialization codecs, which are collectively referred to as ”text encodings”.

text file
: (テキストファイル) `str` オブジェクトを読み書きできる *file object* です。しばしば、テキストファイルは実際にバイト指向のデータストリームにアクセスし、**テキストエンコーディング** を自動的に行います。テキストファイルの例は、`sys.stdin`, `sys.stdout`, `io.StringIO` インスタンスなどをテキストモード (`'r'` or `'w'`) で開いたファイルです。

  bytes-like **オブジェクト** を読み書きできるファイルオブジェクトについては、**バイナリファイル** も参照してください。

トークン
: 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.

triple-quoted string
: (三重クォート文字列) 3 つの連続したクォート記号 (") かアポストロフィー (') で囲まれた文字列。通常の (一重) クォート文字列に比べて表現できる文字列に違いはありませんが、幾つかの理由で有用です。1 つか 2 つの連続したクォート記号をエスケープ無しに書くことができますし、行継続文字 (\) を使わなくても複数行にまたがることができるので、ドキュメンテーション文字列を書く時に特に便利です。

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)`.

type alias
: (型エイリアス) 型の別名で、型を識別子に代入して作成します。

  型エイリアスは **型ヒント** を単純化するのに有用です。例えば:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

  これは次のようにより読みやすくできます:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

  機能の説明がある `typing` と **PEP 484** を参照してください。

type hint
: (型ヒント) 変数、クラス属性、関数のパラメータや返り値の期待される型を指定する *annotation* です。

  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.

  グローバル変数、クラス属性、関数で、ローカル変数でないものの型ヒントは `typing.get_type_hints()` で取得できます。

  機能の説明がある `typing` と **PEP 484** を参照してください。

universal newlines
: テキストストリームの解釈法の一つで、以下のすべてを行末と認識します: Unix の行末規定 `'\n'`、Windows の規定 `'\r\n'`、古い Macintosh の規定 `'\r'`。利用法について詳しくは、**PEP 278** と **PEP 3116** 、さらに `bytes.splitlines()` も参照してください。

variable annotation
: (変数アノテーション) 変数あるいはクラス属性の *annotation* 。

  変数あるいはクラス属性に注釈を付けたときは、代入部分は任意です:

```
class C:
    field: 'annotation'
```

変数アノテーションは通常は **型ヒント** のために使われます: 例えば、この変数は `int` の値を取ることを期待されています:

```
count: int = 0
```

変数アノテーションの構文については annassign 節で解説しています。

機能の説明がある *function annotation*, **PEP 484** , **PEP 526** を参照してください。また、アノテーションを利用するベストプラクティスとして annotations-howto も参照してください。

virtual environment
: (仮想環境) 協調的に切り離された実行環境です。これにより Python ユーザとアプリケーションは同じシステム上で動いている他の Python アプリケーションの挙動に干渉することなく Python パッケージのインストールと更新を行うことができます。

  `venv` を参照してください。

virtual machine
: (仮想マシン) 完全にソフトウェアにより定義されたコンピュータ。Python の仮想マシンは、バイトコードコンパイラが出力した **バイトコード** を実行します。

Zen of Python
: (Python の悟り) Python を理解し利用する上での導きとなる、Python の設計原則と哲学をリストにしたものです。対話プロンプトで "`import this`" とするとこのリストを読めます。

付録

B

このドキュメントについて

Python のドキュメントは、Sphinx を使って、reStructuredText のソースから生成されています。Sphinx はもともと Python のために作られ、現在は独立したプロジェクトとして保守されています。

ドキュメントとそのツール群の開発は、Python 自身と同様に完全にボランティアの努力です。もしあなたが貢献したいなら、どのようにすればよいかについて reporting-bugs ページをご覧下さい。新しいボランティアはいつでも歓迎です!（訳注: 日本語訳の問題については、GitHub 上の Issue Tracker で報告をお願いします。）

多大な感謝を:

- Fred L. Drake, Jr., オリジナルの Python ドキュメントツールセットの作成者で、ドキュメントの多くを書きました。
- Docutils プロジェクトは、reStructuredText と Docutils ツールセットを作成しました。
- Fredrik Lundh の Alternative Python Reference プロジェクトから Sphinx は多くのアイデアを得ました。

B.1 Python ドキュメントへの貢献者

多くの方々が Python 言語、Python 標準ライブラリ、そして Python ドキュメンテーションに貢献してくれています。ソース配布物の Misc/ACKS に、それら貢献してくれた人々を部分的にではありますがリストアップしてあります。

Python コミュニティからの情報提供と貢献がなければこの素晴らしいドキュメントは生まれませんでした -- ありがとう!

付録

C

歴史とライセンス

C.1 Python の歴史

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.

リリース	ベース	西暦年	権利	GPL-compatible? (1)
0.9.0 - 1.2	n/a	1991-1995	CWI	yes
1.3 - 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	yes (2)
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 以降	2.1.1	2001-現在	PSF	yes

注釈

(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.

Guido の指示の下、これらのリリースを可能にしてくださった多くのボランティアのみなさんに感謝します。

C.2 Terms and conditions for accessing or otherwise using 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*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Licenses and Acknowledgements for Incorporated Software* for an incomplete list of these licenses.

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 © 2001-2023 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.
```

(次のページに続く)

(前のページからの続き)

```
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
   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 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT 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
```

(次のページに続く)

(前のページからの続き)

```
   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 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
```

(次のページに続く)

(前のページからの続き)

```
   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
```

(次のページに続く)

(前のページからの続き)

```
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 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 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

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
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
```

(次のページに続く)

(前のページからの続き)

```
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 ソケット

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
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 Asynchronous socket services

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 Cookie management

The `http.cookies` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
```

(次のページに続く)

(前のページからの続き)

```
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Execution tracing

The `trace` module contains the following notice:

```
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 UUencode and UUdecode functions

The uu module 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.

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 XML Remote Procedure Calls

The xmlrpc.client module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
```

(次のページに続く)

(前のページからの続き)

```
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
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

select モジュールは kqueue インターフェースについての次の告知を含んでいます:

```
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

The file Python/pyhash.c contains Marek Majkowski' implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<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
```

(次のページに続く)

(前のページからの続き)

```
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod と 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`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. 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
   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
```

(次のページに続く)

(前のページからの続き)

```
   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,
   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
```

(次のページに続く)

(前のページからの続き)

```
   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,
```

(次のページに続く)

(前のページからの続き)

```
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
```

(次のページに続く)

(前のページからの続き)

```
   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
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

The `zlib` extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

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

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

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

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

(次のページに続く)

(前のページからの続き)

```
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

The implementation of the hash table used by the `tracemalloc` で使用しているハッシュテーブルの実装は、cfuhash プロジェクトのものに基づきます:

```
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)
```

(次のページに続く)

(前のページからの続き)

```
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 W3C C14N test suite

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

```
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,
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 Audioop

The audioop module uses the code base in g771.c file of the SoX project. https://sourceforge.net/projects/sox/files/sox/12.17.7/sox-12.17.7.tar.gz

> This source code is a product of Sun Microsystems, Inc. and is provided for unrestricted use. Users may copy or modify this source code without charge.
>
> SUN SOURCE CODE IS PROVIDED AS IS WITH NO WARRANTIES OF ANY KIND INCLUDING THE WARRANTIES OF DESIGN, MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE, OR ARISING FROM A COURSE OF DEALING, USAGE OR TRADE PRACTICE.
>
> Sun source code is provided with no support and without any obligation on the part of Sun Microsystems, Inc. to assist in its use, correction, modification or enhancement.
>
> SUN MICROSYSTEMS, INC. SHALL HAVE NO LIABILITY WITH RESPECT TO THE IN-

FRINGEMENT OF COPYRIGHTS, TRADE SECRETS OR ANY PATENTS BY THIS SOFTWARE OR ANY PART THEREOF.

In no event will Sun Microsystems, Inc. be liable for any lost revenue or profits or other special, indirect and consequential damages, even if Sun has been advised of the possibility of such damages.

Sun Microsystems, Inc. 2550 Garcia Avenue Mountain View, California 94043

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.
```

付録

D

COPYRIGHT

Python and this documentation is:

Copyright © 2001-2023 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

ライセンスおよび許諾に関する完全な情報は、歴史とライセンス を参照してください。

索引

アルファベット以外

... , **401**
2to3, **401**
>>>, **401**
__all__ (*package variable*), 90
__dict__ (**モジュール属性**), 223
__doc__ (**モジュール属性**), 223
__file__ (**モジュール属性**), 223, 224
__future__, **408**
__import__
    組み込み関数, 90
__loader__ (*module attribute*), 223
__main__
    module, 15, 260, 279, 280
__name__ (**モジュール属性**), 223
__package__ (*module attribute*), 223
__PYVENV_LAUNCHER__, 299, 306
__slots__, **419**
_frozen (*C struct*), 94
_inittab (*C struct*), 95
_inittab.initfunc (*C member*), 95
_inittab.name (*C member*), 95
_Py_c_diff (*C function*), 168
_Py_c_neg (*C function*), 168
_Py_c_pow (*C function*), 168
_Py_c_prod (*C function*), 168
_Py_c_quot (*C function*), 168
_Py_c_sum (*C function*), 168
_Py_InitializeMain (*C function*), 316
_Py_NoneStruct (*C var*), 334
_PyBytes_Resize (*C function*), 172
_PyCFunctionFast (*C type*), 337
_PyCFunctionFastWithKeywords (*C type*), 337
_PyCode_GetExtra (*C* **関数**), 220
_PyCode_SetExtra (*C* **関数**), 221
_PyEval_RequestCodeExtraIndex (*C* **関数**), 220
_PyFrameEvalFunction (*C type*), 275
_PyInterpreterFrame (*C struct*), 243
_PyInterpreterState_GetEvalFrameFunc (*C function*), 275
_PyInterpreterState_SetEvalFrameFunc (*C function*), 275
_PyObject_GetDictPtr (*C function*), 119
_PyObject_New (*C function*), 333
_PyObject_NewVar (*C function*), 333
_PyTuple_Resize (*C function*), 199
_thread
    module, 270
クラス, **404**
クラスメソッド
    組み込み関数, 340
コルーチン, **405**
コンパイル
    組み込み関数, 92
コードオブジェクト, 216
ジェネレータ, **408**
トークン, **420**
パス
    module **検索**, 15, 260, 264
ファイル
    object, 221
メソッド, **414**
    magic, 413
    object, 215
    特殊, 420
ロケールエンコーディング, **413**
位置引数 (*positional argument*), **417**
特殊
    メソッド, 420
環境変数
    __PYVENV_LAUNCHER__, 299, 306
    PATH, 15
    PYTHONCOERCECLOCALE, 313
    PYTHONDEBUG, 257, 305
    PYTHONDEVMODE, 300
    PYTHONDONTWRITEBYTECODE, 257, 310
    PYTHONDUMPREFS, 300, 354
    PYTHONEXECUTABLE, 306
    PYTHONFAULTHANDLER, 301
    PYTHONHASHSEED, 257, 302
    PYTHONHOME, 15, 16, 257, 267, 302
    PYTHONINSPECT, 258, 302
    PYTHONINTMAXSTRDIGITS, 303
    PYTHONIOENCODING, 262, 308
    PYTHONLEGACYWINDOWSFSENCODING, 258, 294
    PYTHONLEGACYWINDOWSSTDIO, 259, 303
    PYTHONMALLOC, 320, 325, 327, 330
    PYTHONMALLOCSTATS, 304, 320
    PYTHONNODEBUGRANGES, 299
    PYTHONNOUSERSITE, 259, 309
    PYTHONOPTIMIZE, 259, 305
    PYTHONPATH, 15, 16, 257, 304
    PYTHONPERFSUPPORT, 308
    PYTHONPLATLIBDIR, 304
    PYTHONPROFILEIMPORTTIME, 302
    PYTHONPYCACHEPREFIX, 306
    PYTHONSAFEPATH, 298
    PYTHONTRACEMALLOC, 308
    PYTHONUNBUFFERED, 260, 299
    PYTHONUTF8, 294, 313
    PYTHONVERBOSE, 260, 309
    PYTHONWARNINGS, 309
組み込み関数
    __import__, 90
    abs, 131
    ascii, 120
    bytes, 121
    divmod, 131
    hash, 121, 359
    int, 133
    len, 122, 134, 137, 202, 206, 210
    pow, 131, 133
    repr, 120, 359
    tuple, 136, 204
    type, 122
    クラスメソッド, 340
    コンパイル, 92
    浮動小数点数, 134
    静的メソッド, 340
関数, **408**
    object, 211
集合
    object, 209
静的メソッド
    組み込み関数, 340

A

abort (*C function*), 90
abs
    組み込み関数, 131
abstract base class, **401**
allocfunc (*C type*), 388
annotation, **401**
argv (*in module sys*), 266
ascii
    組み込み関数, 120
asynchronous context manager, **402**
asynchronous generator, **402**
asynchronous generator iterator, **402**
asynchronous iterable, **403**
asynchronous iterator, **403**
awaitable, **403**

B

BDFL, **403**
binary file, **403**
binaryfunc (*C type*), 390
borrowed reference, **403**
buffer interface
    (see buffer protocol), 139
buffer object
    (see buffer protocol), 139

buffer protocol, 139
builtins
    module, 15, 260, 279, 280
bytearray
    object, 172
bytecode, **404**
bytes
    object, 170
    組み込み関数, 121
bytes-like object, **403**

C

callable, **404**
callback, **404**
calloc (*C function*), 319
Capsule
    object, 239
C-contiguous, 144, 405
class variable, **404**
cleanup functions, 90
close (*in module os*), 280
CO_FUTURE_DIVISION (*C var*), 58
Common Vulnerabilities and Exposures
    CVE 2008-5983, 266
complex number, **404**
    object, 168
context manager, **405**
context variable, **405**
contiguous, 144, **405**
copyright (*in module sys*), 265
coroutine function, **405**
CPython, **405**

D

decorator, **405**
descrgetfunc (*C type*), 389
descriptor, **406**
descrsetfunc (*C type*), 389
destructor (*C type*), 388
dictionary, **406**
    object, 204
dictionary comprehension, **406**
dictionary view, **406**
divmod
    組み込み関数, 131
docstring, **406**
duck-typing, **406**

E

EAFP, **406**
EOFError (**組み込み例外**), 222
exc_info (*sys* **モジュール**), 13
executable (*in module sys*), 264
exit (*C function*), 90
expression, **407**
extension module, **407**

F

f-string, **407**
file object, **407**
file-like object, **407**
filesystem encoding and error handler, **407**
finder, **407**
floor division, **408**
Fortran contiguous, 144, 405
free (*C function*), 319
freefunc (*C type*), 389
frozenset
    object, 209
function annotation, **408**

G

garbage collection, **408**
gcvisitobjects_t (*C type*), 398
generator expression, **409**
generator iterator, **409**
generic function, **409**
generic type, **409**
getattrfunc (*C type*), 389
getattrofunc (*C type*), 389
getbufferproc (*C type*), 390
getiterfunc (*C type*), 389
getter (*C type*), 345
GIL, **409**
global interpreter lock, 267, **409**

H

hash
    組み込み関数, 121, 359
hash-based pyc, **410**
hashable, **410**
hashfunc (*C type*), 389

I

IDLE, **410**
immutable, **410**
import path, **410**
importer, **410**
importing, **410**
incr_item(), 14, 15
initproc (*C type*), 389
inquiry (*C type*), 397
instancemethod
    object, 214
int
    組み込み関数, 133
interactive, **411**
interpreted, **411**
interpreter lock, 267
interpreter shutdown, **411**
iterable, **411**
iterator, **411**
iternextfunc (*C type*), 389

K

key function, **412**
KeyboardInterrupt (**組み込み例外**), 73, 74
keyword argument, **412**

L

lambda, **412**
LBYL, **412**
len
    組み込み関数, 122, 134, 137, 202, 206, 210
lenfunc (*C type*), 390
lexical analyzer, **412**
list, **412**
    object, 202
list comprehension, **412**
loader, **413**
lock, interpreter, 267
long integer
    object, 159
LONG_MAX (*C macro*), 161

M

magic
    メソッド, 413
magic method, **413**
main(), 262, 266
malloc (*C function*), 319
mapping, **413**
    object, 204
memoryview
    object, 237
meta path finder, **413**
metaclass, **413**
METH_CLASS (*C macro*), 339
METH_COEXIST (*C macro*), 340
METH_FASTCALL (*C macro*), 338
METH_KEYWORDS (*C macro*), 338
METH_METHOD (*C macro*), 339
METH_NOARGS (*C macro*), 339
METH_O (*C macro*), 339
METH_STATIC (*C macro*), 340
METH_VARARGS (*C macro*), 338
method resolution order, **414**
MethodType (*in module types*), 211, 215
module, **414**
    __main__, 15, 260, 279, 280
    _thread, 270
    builtins, 15, 260, 279, 280
    object, 223
    signal, 73, 74
    sys, 15, 260, 279, 280
    検索 パス, 15, 260, 264
module spec, **414**
modules (*in module sys*), 90, 260
ModuleType (*in module types*), 223
MRO, **414**
mutable, **414**

N

named tuple, **414**
namespace, **415**
namespace package, **415**
nested scope, **415**
new-style class, **415**
newfunc (*C type*), 389
None
    object, 159

O

object, **415**
    bytearray, 172
    bytes, 170
    Capsule, 239
    complex number, 168
    dictionary, 204
    frozenset, 209
    instancemethod, 214
    list, 202
    long integer, 159
    mapping, 204
    memoryview, 237
    module, 223
    None, 159
    sequence, 170
    tuple, 198
    type, 8, 151
    コード, 216
    ファイル, 221
    メソッド, 215
    数値, 159
    整数, 159
    浮動小数点, 165

    関数, 211
    集合, 209
objobjargproc (*C type*), 390
objobjproc (*C type*), 390
OverflowError (組み込み例外), 161, 162

P

package, **415**
package variable
    __all__, 90
parameter, **416**
PATH, 15
path (*sys* モジュール), 15, 260, 264
path based finder, **417**
path entry, **416**
path entry finder, **416**
path entry hook, **416**
path-like object, **417**
PEP, **417**
platform (*in module sys*), 265
portion, **417**
pow
    組み込み関数, 131, 133
provisional API, **417**
provisional package, **417**
Py_ABS (*C macro*), 5
Py_AddPendingCall (*C function*), 282
Py_ALWAYS_INLINE (*C macro*), 5
Py_AtExit (*C function*), 90
Py_AUDIT_READ (*C macro*), 342
Py_AuditHookFunction (*C type*), 89
Py_BEGIN_ALLOW_THREADS (*C macro*), 268, 272
Py_BLOCK_THREADS (*C macro*), 272
Py_buffer (*C type*), 141
Py_buffer.buf (*C member*), 141
Py_buffer.format (*C member*), 142
Py_buffer.internal (*C member*), 142
Py_buffer.itemsize (*C member*), 141
Py_buffer.len (*C member*), 141
Py_buffer.ndim (*C member*), 142
Py_buffer.obj (*C member*), 141
Py_buffer.readonly (*C member*), 141
Py_buffer.shape (*C member*), 142
Py_buffer.strides (*C member*), 142
Py_buffer.suboffsets (*C member*), 142
Py_BuildValue (*C function*), 105
Py_BytesMain (*C function*), 53
Py_BytesWarningFlag (*C var*), 256
Py_CHARMASK (*C macro*), 5
Py_CLEANUP_SUPPORTED (*C macro*), 102
Py_CLEAR (*C function*), 61
Py_CompileString (*C function*), 56, 58
Py_CompileStringExFlags (*C function*), 57
Py_CompileStringFlags (*C function*), 56
Py_CompileStringObject (*C function*), 56
Py_complex (*C type*), 168
Py_complex.imag (*C member*), 168
Py_complex.real (*C member*), 168
Py_DEBUG (*C macro*), 16
Py_DebugFlag (*C var*), 256
Py_DecodeLocale (*C function*), 85
Py_DECREF (*C function*), 8, 60
Py_DecRef (*C function*), 61
Py_DEPRECATED (*C macro*), 5
Py_DontWriteBytecodeFlag (*C var*), 257
Py_Ellipsis (*C var*), 237
Py_EncodeLocale (*C function*), 86
Py_END_ALLOW_THREADS (*C macro*), 268, 272
Py_EndInterpreter (*C function*), 280
Py_EnterRecursiveCall (*C function*), 78
Py_EQ (*C macro*), 371
Py_eval_input (*C var*), 58
Py_Exit (*C function*), 90
Py_ExitStatusException (*C function*), 292
Py_False (*C var*), 165
Py_FatalError (*C function*), 90
Py_FatalError(), 266
Py_FdIsInteractive (*C function*), 83
Py_file_input (*C var*), 58
Py_Finalize (*C function*), 261
Py_FinalizeEx (*C function*), 90, 260, 261, 280
Py_FrozenFlag (*C var*), 257
Py_GE (*C macro*), 371
Py_GenericAlias (*C function*), 252
Py_GenericAliasType (*C var*), 253
Py_GetArgcArgv (*C function*), 315
Py_GetBuildInfo (*C function*), 265
Py_GetCompiler (*C function*), 265
Py_GetCopyright (*C function*), 265
Py_GETENV (*C macro*), 6
Py_GetExecPrefix (*C function*), 16, 263
Py_GetPath (*C function*), 16, 264
Py_GetPath(), 262, 264
Py_GetPlatform (*C function*), 265
Py_GetPrefix (*C function*), 16, 263
Py_GetProgramFullPath (*C function*), 16, 264
Py_GetProgramName (*C function*), 262
Py_GetPythonHome (*C function*), 267
Py_GetVersion (*C function*), 265
Py_GT (*C macro*), 371
Py_hash_t (*C type*), 110
Py_HashRandomizationFlag (*C var*), 257
Py_IgnoreEnvironmentFlag (*C var*), 257
Py_INCREF (*C function*), 8, 59
Py_IncRef (*C function*), 61
Py_Initialize (*C function*), 15, 260, 280
Py_Initialize(), 262
Py_InitializeEx (*C function*), 260
Py_InitializeFromConfig (*C function*), 310
Py_InspectFlag (*C var*), 257
Py_InteractiveFlag (*C var*), 258
Py_Is (*C function*), 335
Py_IS_TYPE (*C function*), 336
Py_IsFalse (*C function*), 335
Py_IsInitialized (*C function*), 16, 261
Py_IsNone (*C function*), 335
Py_IsolatedFlag (*C var*), 258
Py_IsTrue (*C function*), 335
Py_LE (*C macro*), 371
Py_LeaveRecursiveCall (*C function*), 78
Py_LegacyWindowsFSEncodingFlag (*C var*), 258
Py_LegacyWindowsStdioFlag (*C var*), 258
Py_LIMITED_API (*C macro*), 18
Py_LT (*C macro*), 371
Py_Main (*C function*), 53
PY_MAJOR_VERSION (*C macro*), 399
Py_MAX (*C macro*), 6
Py_MEMBER_SIZE (*C macro*), 6
PY_MICRO_VERSION (*C macro*), 399
Py_MIN (*C macro*), 6
PY_MINOR_VERSION (*C macro*), 399
Py_mod_create (*C macro*), 227
Py_mod_exec (*C macro*), 228
Py_mod_multiple_interpreters (*C macro*), 228
Py_MOD_MULTIPLE_INTERPRETERS_NOT_SUPPORTED (*C macro*), 228
Py_MOD_MULTIPLE_INTERPRETERS_SUPPORTED (*C macro*), 228
Py_MOD_PER_INTERPRETER_GIL_SUPPORTED (*C macro*), 228
Py_NE (*C macro*), 371
Py_NewInterpreter (*C function*), 280
Py_NewInterpreterFromConfig (*C function*), 279
Py_NewRef (*C function*), 60
Py_NO_INLINE (*C macro*), 6
Py_None (*C var*), 159
Py_NoSiteFlag (*C var*), 259
Py_NotImplemented (*C var*), 117
Py_NoUserSiteDirectory (*C var*), 259
Py_OpenCodeHookFunction (*C type*), 222
Py_OptimizeFlag (*C var*), 259
Py_PreInitialize (*C function*), 295
Py_PreInitializeFromArgs (*C function*), 295
Py_PreInitializeFromBytesArgs (*C function*), 295
Py_PRINT_RAW (*C macro*), 117
Py_PRINT_RAW (*C* のマクロ), 222
Py_QuietFlag (*C var*), 259
Py_READONLY (*C macro*), 342
Py_REFCNT (*C function*), 59
Py_RELATIVE_OFFSET (*C macro*), 342
PY_RELEASE_LEVEL (*C macro*), 399
PY_RELEASE_SERIAL (*C macro*), 399
Py_ReprEnter (*C function*), 78
Py_ReprLeave (*C function*), 78
Py_RETURN_FALSE (*C macro*), 165
Py_RETURN_NONE (*C macro*), 159
Py_RETURN_NOTIMPLEMENTED (*C macro*), 117
Py_RETURN_RICHCOMPARE (*C macro*), 371
Py_RETURN_TRUE (*C macro*), 165
Py_RunMain (*C function*), 315
Py_SET_REFCNT (*C function*), 59
Py_SET_SIZE (*C function*), 336
Py_SET_TYPE (*C function*), 336
Py_SetPath (*C function*), 264
Py_SetPath(), 264
Py_SetProgramName (*C function*), 16, 262
Py_SetProgramName(), 260, 263, 264
Py_SetPythonHome (*C function*), 267
Py_SETREF (*C macro*), 61
Py_SetStandardStreamEncoding (*C function*), 262
Py_single_input (*C var*), 58
Py_SIZE (*C function*), 336
Py_ssize_t (*C type*), 12
PY_SSIZE_T_MAX (*C macro*), 162
Py_STRINGIFY (*C macro*), 6
Py_T_BOOL (*C macro*), 344
Py_T_BYTE (*C macro*), 344
Py_T_CHAR (*C macro*), 344
Py_T_DOUBLE (*C macro*), 344
Py_T_FLOAT (*C macro*), 344
Py_T_INT (*C macro*), 344
Py_T_LONG (*C macro*), 344
Py_T_LONGLONG (*C macro*), 344
Py_T_OBJECT_EX (*C macro*), 344
Py_T_PYSSIZET (*C macro*), 344
Py_T_SHORT (*C macro*), 344

Py_T_STRING (C macro), 344
Py_T_STRING_INPLACE (C macro), 344
Py_T_UBYTE (C macro), 344
Py_T_UINT (C macro), 344
Py_T_ULONG (C macro), 344
Py_T_ULONGLONG (C macro), 344
Py_T_USHORT (C macro), 344
Py_TPFLAGS_BASE_EXC_SUBCLASS (C macro), 365
Py_TPFLAGS_BASETYPE (C macro), 363
Py_TPFLAGS_BYTES_SUBCLASS (C macro), 365
Py_TPFLAGS_DEFAULT (C macro), 363
Py_TPFLAGS_DICT_SUBCLASS (C macro), 365
Py_TPFLAGS_DISALLOW_INSTANTIATION (C macro), 366
Py_TPFLAGS_HAVE_FINALIZE (C macro), 365
Py_TPFLAGS_HAVE_GC (C macro), 363
Py_TPFLAGS_HAVE_VECTORCALL (C macro), 365
Py_TPFLAGS_HEAPTYPE (C macro), 362
Py_TPFLAGS_IMMUTABLETYPE (C macro), 365
Py_TPFLAGS_ITEMS_AT_END (C macro), 364
Py_TPFLAGS_LIST_SUBCLASS (C macro), 365
Py_TPFLAGS_LONG_SUBCLASS (C macro), 365
Py_TPFLAGS_MANAGED_DICT (C macro), 364
Py_TPFLAGS_MANAGED_WEAKREF (C macro), 364
Py_TPFLAGS_MAPPING (C macro), 366
Py_TPFLAGS_METHOD_DESCRIPTOR (C macro), 363
Py_TPFLAGS_READY (C macro), 363
Py_TPFLAGS_READYING (C macro), 363
Py_TPFLAGS_SEQUENCE (C macro), 367
Py_TPFLAGS_TUPLE_SUBCLASS (C macro), 365
Py_TPFLAGS_TYPE_SUBCLASS (C macro), 365
Py_TPFLAGS_UNICODE_SUBCLASS (C macro), 365
Py_TPFLAGS_VALID_VERSION_TAG (C macro), 367
Py_tracefunc (C type), 283
Py_True (C var), 165
Py_tss_NEEDS_INIT (C macro), 286
Py_tss_t (C type), 286
Py_TYPE (C function), 335
Py_UCS1 (C type), 174
Py_UCS2 (C type), 174
Py_UCS4 (C type), 174
Py_uhash_t (C type), 110
Py_UNBLOCK_THREADS (C macro), 273
Py_UnbufferedStdioFlag (C var), 260
Py_UNICODE (C type), 174
Py_UNICODE_IS_HIGH_SURROGATE (C function), 178
Py_UNICODE_IS_LOW_SURROGATE (C function), 178
Py_UNICODE_IS_SURROGATE (C function), 178
Py_UNICODE_ISALNUM (C function), 177
Py_UNICODE_ISALPHA (C function), 177
Py_UNICODE_ISDECIMAL (C function), 177
Py_UNICODE_ISDIGIT (C function), 177
Py_UNICODE_ISLINEBREAK (C function), 177
Py_UNICODE_ISLOWER (C function), 176
Py_UNICODE_ISNUMERIC (C function), 177
Py_UNICODE_ISPRINTABLE (C function), 177
Py_UNICODE_ISSPACE (C function), 176
Py_UNICODE_ISTITLE (C function), 177
Py_UNICODE_ISUPPER (C function), 177
Py_UNICODE_JOIN_SURROGATES (C function), 178
Py_UNICODE_TODECIMAL (C function), 177
Py_UNICODE_TODIGIT (C function), 178
Py_UNICODE_TOLOWER (C function), 177
Py_UNICODE_TONUMERIC (C function), 178
Py_UNICODE_TOTITLE (C function), 177
Py_UNICODE_TOUPPER (C function), 177
Py_UNREACHABLE (C macro), 6
Py_UNUSED (C macro), 7
Py_VaBuildValue (C function), 107
PY_VECTORCALL_ARGUMENTS_OFFSET (C macro), 125
Py_VerboseFlag (C var), 260
Py_Version (C var), 400
PY_VERSION_HEX (C macro), 399
Py_VISIT (C function), 396
Py_XDECREF (C function), 15, 61
Py_XINCREF (C function), 60
Py_XNewRef (C function), 60
Py_XSETREF (C macro), 62
PyAIter_Check (C function), 138
PyAnySet_Check (C function), 210
PyAnySet_CheckExact (C function), 210
PyArg_Parse (C function), 104
PyArg_ParseTuple (C function), 103
PyArg_ParseTupleAndKeywords (C function), 103
PyArg_UnpackTuple (C function), 104
PyArg_ValidateKeywordArguments (C function), 103
PyArg_VaParse (C function), 103
PyArg_VaParseTupleAndKeywords (C function), 103
PyASCIIObject (C type), 174
PyAsyncMethods (C type), 387
PyAsyncMethods.am_aiter (C member), 388
PyAsyncMethods.am_anext (C member), 388
PyAsyncMethods.am_await (C member), 387
PyAsyncMethods.am_send (C member), 388
PyBaseObject_Type (C var), 335
PyBool_Check (C function), 165
PyBool_FromLong (C function), 165
PyBool_Type (C var), 164
PyBUF_ANY_CONTIGUOUS (C macro), 144
PyBUF_C_CONTIGUOUS (C macro), 144
PyBUF_CONTIG (C macro), 145
PyBUF_CONTIG_RO (C macro), 145
PyBUF_F_CONTIGUOUS (C macro), 144
PyBUF_FORMAT (C macro), 143
PyBUF_FULL (C macro), 145
PyBUF_FULL_RO (C macro), 145
PyBUF_INDIRECT (C macro), 144
PyBUF_MAX_NDIM (C macro), 143
PyBUF_ND (C macro), 144
PyBUF_READ (C macro), 237
PyBUF_RECORDS (C macro), 145
PyBUF_RECORDS_RO (C macro), 145
PyBUF_SIMPLE (C macro), 144
PyBUF_STRIDED (C macro), 145
PyBUF_STRIDED_RO (C macro), 145
PyBUF_STRIDES (C macro), 144
PyBUF_WRITABLE (C macro), 143
PyBUF_WRITE (C macro), 237
PyBuffer_FillContiguousStrides (C function), 148
PyBuffer_FillInfo (C function), 148
PyBuffer_FromContiguous (C function), 148
PyBuffer_GetPointer (C function), 147
PyBuffer_IsContiguous (C function), 147
PyBuffer_Release (C function), 147
PyBuffer_SizeFromFormat (C function), 147
PyBuffer_ToContiguous (C function), 148
PyBufferProcs (C type), 140, 386
PyBufferProcs.bf_getbuffer (C member), 386
PyBufferProcs.bf_releasebuffer (C member), 387
PyByteArray_AS_STRING (C function), 173
PyByteArray_AsString (C function), 173
PyByteArray_Check (C function), 173
PyByteArray_CheckExact (C function), 173
PyByteArray_Concat (C function), 173
PyByteArray_FromObject (C function), 173
PyByteArray_FromStringAndSize (C function), 173
PyByteArray_GET_SIZE (C function), 173
PyByteArray_Resize (C function), 173
PyByteArray_Size (C function), 173
PyByteArray_Type (C var), 172
PyByteArrayObject (C type), 172
PyBytes_AS_STRING (C function), 171
PyBytes_AsString (C function), 171
PyBytes_AsStringAndSize (C function), 172
PyBytes_Check (C function), 170
PyBytes_CheckExact (C function), 170
PyBytes_Concat (C function), 172
PyBytes_ConcatAndDel (C function), 172
PyBytes_FromFormat (C function), 170
PyBytes_FromFormatV (C function), 171
PyBytes_FromObject (C function), 171
PyBytes_FromString (C function), 170
PyBytes_FromStringAndSize (C function), 170
PyBytes_GET_SIZE (C function), 171
PyBytes_Size (C function), 171
PyBytes_Type (C var), 170
PyBytesObject (C type), 170
PyCallable_Check (C function), 130
PyCallIter_Check (C function), 234
PyCallIter_New (C function), 234
PyCallIter_Type (C var), 234
PyCapsule (C type), 239
PyCapsule_CheckExact (C function), 239
PyCapsule_Destructor (C type), 239
PyCapsule_GetContext (C function), 240

PyCapsule_GetDestructor (C function), 240
PyCapsule_GetName (C function), 240
PyCapsule_GetPointer (C function), 240
PyCapsule_Import (C function), 240
PyCapsule_IsValid (C function), 241
PyCapsule_New (C function), 239
PyCapsule_SetContext (C function), 241
PyCapsule_SetDestructor (C function), 241
PyCapsule_SetName (C function), 241
PyCapsule_SetPointer (C function), 241
PyCell_Check (C function), 216
PyCell_GET (C function), 216
PyCell_Get (C function), 216
PyCell_New (C function), 216
PyCell_SET (C function), 216
PyCell_Set (C function), 216
PyCell_Type (C var), 216
PyCellObject (C type), 216
PyCFunction (C type), 336
PyCFunction_New (C function), 340
PyCFunction_NewEx (C function), 340
PyCFunctionWithKeywords (C type), 337
PyCMethod (C type), 337
PyCMethod_New (C function), 340
PyCode_Addr2Line (C function), 218
PyCode_Addr2Location (C function), 218
PyCode_AddWatcher (C function), 219
PyCode_Check (C function), 216
PyCode_ClearWatcher (C function), 219
PyCode_GetCellvars (C function), 218
PyCode_GetCode (C function), 218
PyCode_GetFirstFree (C function), 217
PyCode_GetFreevars (C function), 219
PyCode_GetNumFree (C function), 217
PyCode_GetVarnames (C function), 218
PyCode_New (C 関数), 217
PyCode_NewEmpty (C function), 218
PyCode_NewWithPosOnlyArgs (C 関数), 218
PyCode_Type (C var), 216
PyCode_WatchCallback (C type), 219
PyCodec_BackslashReplaceErrors (C function), 113
PyCodec_Decode (C function), 112
PyCodec_Decoder (C function), 112
PyCodec_Encode (C function), 112
PyCodec_Encoder (C function), 112
PyCodec_IgnoreErrors (C function), 113
PyCodec_IncrementalDecoder (C function), 112
PyCodec_IncrementalEncoder (C function), 112
PyCodec_KnownEncoding (C function), 112
PyCodec_LookupError (C function), 113
PyCodec_NameReplaceErrors (C function), 113
PyCodec_Register (C function), 111
PyCodec_RegisterError (C function), 113
PyCodec_ReplaceErrors (C function), 113
PyCodec_StreamReader (C function), 112
PyCodec_StreamWriter (C function), 112
PyCodec_StrictErrors (C function), 113
PyCodec_Unregister (C function), 111
PyCodec_XMLCharRefReplaceErrors (C function), 113
PyCodeEvent (C type), 219
PyCodeObject (C type), 216
PyCompactUnicodeObject (C type), 174
PyCompilerFlags (C struct), 58
PyCompilerFlags.cf_feature_version (C member), 58
PyCompilerFlags.cf_flags (C member), 58
PyComplex_AsCComplex (C function), 169
PyComplex_Check (C function), 169
PyComplex_CheckExact (C function), 169
PyComplex_FromCComplex (C function), 169
PyComplex_FromDoubles (C function), 169
PyComplex_ImagAsDouble (C function), 169
PyComplex_RealAsDouble (C function), 169
PyComplex_Type (C var), 169
PyComplexObject (C type), 169
PyConfig (C type), 296
PyConfig_Clear (C function), 297
PyConfig_InitIsolatedConfig (C function), 296
PyConfig_InitPythonConfig (C function), 296
PyConfig_Read (C function), 297
PyConfig_SetArgv (C function), 297
PyConfig_SetBytesArgv (C function), 297
PyConfig_SetBytesString (C function), 296
PyConfig_SetString (C function), 296
PyConfig_SetWideStringList (C function), 297
PyConfig.argv (C member), 298
PyConfig.base_exec_prefix (C member), 298
PyConfig.base_executable (C member), 298
PyConfig.base_prefix (C member), 299
PyConfig.buffered_stdio (C member), 299
PyConfig.bytes_warning (C member), 299
PyConfig.check_hash_pycs_mode (C member), 300
PyConfig.code_debug_ranges (C member), 299
PyConfig.configure_c_stdio (C member), 300
PyConfig.dev_mode (C member), 300
PyConfig.dump_refs (C member), 300
PyConfig.exec_prefix (C member), 300
PyConfig.executable (C member), 301
PyConfig.faulthandler (C member), 301
PyConfig.filesystem_encoding (C member), 301
PyConfig.filesystem_errors (C member), 301
PyConfig.hash_seed (C member), 302
PyConfig.home (C member), 302
PyConfig.import_time (C member), 302
PyConfig.inspect (C member), 302
PyConfig.install_signal_handlers (C member), 303
PyConfig.int_max_str_digits (C member), 303
PyConfig.interactive (C member), 303
PyConfig.isolated (C member), 303
PyConfig.legacy_windows_stdio (C member), 303
PyConfig.malloc_stats (C member), 304
PyConfig.module_search_paths (C member), 304
PyConfig.module_search_paths_set (C member), 304
PyConfig.optimization_level (C member), 305
PyConfig.orig_argv (C member), 305
PyConfig.parse_argv (C member), 305
PyConfig.parser_debug (C member), 305
PyConfig.pathconfig_warnings (C member), 306
PyConfig.perf_profiling (C member), 308
PyConfig.platlibdir (C member), 304
PyConfig.prefix (C member), 306
PyConfig.program_name (C member), 306
PyConfig.pycache_prefix (C member), 306
PyConfig.pythonpath_env (C member), 304
PyConfig.quiet (C member), 306
PyConfig.run_command (C member), 307
PyConfig.run_filename (C member), 307
PyConfig.run_module (C member), 307
PyConfig.safe_path (C member), 298
PyConfig.show_ref_count (C member), 307
PyConfig.site_import (C member), 307
PyConfig.skip_source_first_line (C member), 308
PyConfig.stdio_encoding (C member), 308
PyConfig.stdio_errors (C member), 308
PyConfig.tracemalloc (C member), 308
PyConfig.use_environment (C member), 309
PyConfig.use_hash_seed (C member), 302
PyConfig.user_site_directory (C member), 309
PyConfig.verbose (C member), 309
PyConfig.warn_default_encoding (C member), 299
PyConfig.warnoptions (C member), 309
PyConfig.write_bytecode (C member), 309
PyConfig.xoptions (C member), 310
PyContext (C type), 246
PyContext_CheckExact (C function), 246
PyContext_Copy (C function), 246
PyContext_CopyCurrent (C function), 247
PyContext_Enter (C function), 247
PyContext_Exit (C function), 247
PyContext_New (C function), 246
PyContext_Type (C var), 246
PyContextToken (C type), 246

PyContextToken_CheckExact (C function), 246
PyContextToken_Type (C var), 246
PyContextVar (C type), 246
PyContextVar_CheckExact (C function), 246
PyContextVar_Get (C function), 247
PyContextVar_New (C function), 247
PyContextVar_Reset (C function), 247
PyContextVar_Set (C function), 247
PyContextVar_Type (C var), 246
PyCoro_CheckExact (C function), 245
PyCoro_New (C function), 245
PyCoro_Type (C var), 245
PyCoroObject (C type), 245
PyDate_Check (C function), 249
PyDate_CheckExact (C function), 249
PyDate_FromDate (C function), 249
PyDate_FromTimestamp (C function), 252
PyDateTime_Check (C function), 249
PyDateTime_CheckExact (C function), 249
PyDateTime_Date (C type), 248
PyDateTime_DATE_GET_FOLD (C function), 251
PyDateTime_DATE_GET_HOUR (C function), 251
PyDateTime_DATE_GET_MICROSECOND (C function), 251
PyDateTime_DATE_GET_MINUTE (C function), 251
PyDateTime_DATE_GET_SECOND (C function), 251
PyDateTime_DATE_GET_TZINFO (C function), 251
PyDateTime_DateTime (C type), 248
PyDateTime_DateTimeType (C var), 248
PyDateTime_DateType (C var), 248
PyDateTime_Delta (C type), 248
PyDateTime_DELTA_GET_DAYS (C function), 252
PyDateTime_DELTA_GET_MICROSECONDS (C function), 252
PyDateTime_DELTA_GET_SECONDS (C function), 252
PyDateTime_DeltaType (C var), 248
PyDateTime_FromDateAndTime (C function), 249
PyDateTime_FromDateAndTimeAndFold (C function), 250
PyDateTime_FromTimestamp (C function), 252
PyDateTime_GET_DAY (C function), 250
PyDateTime_GET_MONTH (C function), 250
PyDateTime_GET_YEAR (C function), 250
PyDateTime_Time (C type), 248
PyDateTime_TIME_GET_FOLD (C function), 251
PyDateTime_TIME_GET_HOUR (C function), 251
PyDateTime_TIME_GET_MICROSECOND (C function), 251
PyDateTime_TIME_GET_MINUTE (C function), 251
PyDateTime_TIME_GET_SECOND (C function), 251
PyDateTime_TIME_GET_TZINFO (C function), 251
PyDateTime_TimeType (C var), 248
PyDateTime_TimeZone_UTC (C var), 248
PyDateTime_TZInfoType (C var), 248
PyDelta_Check (C function), 249
PyDelta_CheckExact (C function), 249
PyDelta_FromDSU (C function), 250
PyDescr_IsData (C function), 235
PyDescr_NewClassMethod (C function), 234
PyDescr_NewGetSet (C function), 234
PyDescr_NewMember (C function), 234
PyDescr_NewMethod (C function), 234
PyDescr_NewWrapper (C function), 234
PyDict_AddWatcher (C function), 207
PyDict_Check (C function), 204
PyDict_CheckExact (C function), 204
PyDict_Clear (C function), 204
PyDict_ClearWatcher (C function), 208
PyDict_Contains (C function), 204
PyDict_Copy (C function), 204
PyDict_DelItem (C function), 205
PyDict_DelItemString (C function), 205
PyDict_GetItem (C function), 205
PyDict_GetItemString (C function), 205
PyDict_GetItemWithError (C function), 205
PyDict_Items (C function), 206
PyDict_Keys (C function), 206
PyDict_Merge (C function), 207
PyDict_MergeFromSeq2 (C function), 207
PyDict_New (C function), 204
PyDict_Next (C function), 206
PyDict_SetDefault (C function), 205
PyDict_SetItem (C function), 204
PyDict_SetItemString (C function), 205
PyDict_Size (C function), 206
PyDict_Type (C var), 204
PyDict_Unwatch (C function), 208
PyDict_Update (C function), 207
PyDict_Values (C function), 206
PyDict_Watch (C function), 208
PyDict_WatchCallback (C type), 208
PyDict_WatchEvent (C type), 208
PyDictObject (C type), 204
PyDictProxy_New (C function), 204
PyDoc_STR (C macro), 7
PyDoc_STRVAR (C macro), 7
PyEllipsis_Type (C var), 237
PyErr_BadArgument (C function), 65
PyErr_BadInternalCall (C function), 68
PyErr_CheckSignals (C function), 73
PyErr_Clear (C function), 13, 15, 63
PyErr_DisplayException (C function), 64
PyErr_ExceptionMatches (C function), 15, 69
PyErr_Fetch (C function), 70
PyErr_Format (C function), 65
PyErr_FormatV (C function), 65
PyErr_GetExcInfo (C function), 72
PyErr_GetHandledException (C function), 72
PyErr_GetRaisedException (C function), 69
PyErr_GivenExceptionMatches (C function), 69
PyErr_NewException (C function), 75
PyErr_NewExceptionWithDoc (C function), 75
PyErr_NoMemory (C function), 65
PyErr_NormalizeException (C function), 71
PyErr_Occurred (C function), 13, 69
PyErr_Print (C function), 64
PyErr_PrintEx (C function), 63
PyErr_ResourceWarning (C function), 69
PyErr_Restore (C function), 71
PyErr_SetExcFromWindowsErr (C function), 66
PyErr_SetExcFromWindowsErrWithFilename (C function), 67
PyErr_SetExcFromWindowsErrWithFilenameObject (C function), 66
PyErr_SetExcFromWindowsErrWithFilenameObject (C function), 67
PyErr_SetExcInfo (C function), 73
PyErr_SetFromErrno (C function), 65
PyErr_SetFromErrnoWithFilename (C function), 66
PyErr_SetFromErrnoWithFilenameObject (C function), 65
PyErr_SetFromErrnoWithFilenameObjects (C function), 66
PyErr_SetFromWindowsErr (C function), 66
PyErr_SetFromWindowsErrWithFilename (C function), 66
PyErr_SetHandledException (C function), 72
PyErr_SetImportError (C function), 67
PyErr_SetImportErrorSubclass (C function), 67
PyErr_SetInterrupt (C function), 74
PyErr_SetInterruptEx (C function), 74
PyErr_SetNone (C function), 65
PyErr_SetObject (C function), 65
PyErr_SetRaisedException (C function), 70
PyErr_SetString (C function), 13, 64
PyErr_SyntaxLocation (C function), 67
PyErr_SyntaxLocationEx (C function), 67
PyErr_SyntaxLocationObject (C function), 67
PyErr_WarnEx (C function), 68
PyErr_WarnExplicit (C function), 68
PyErr_WarnExplicitObject (C function), 68
PyErr_WarnFormat (C function), 69
PyErr_WriteUnraisable (C function), 64
PyEval_AcquireLock (C function), 276
PyEval_AcquireThread (C function), 276
PyEval_AcquireThread(), 270
PyEval_EvalCode (C function), 57
PyEval_EvalCodeEx (C function), 57
PyEval_EvalFrame (C function), 57
PyEval_EvalFrameEx (C function), 57
PyEval_GetBuiltins (C function), 111
PyEval_GetFrame (C function), 111
PyEval_GetFuncDesc (C function), 111
PyEval_GetFuncName (C function), 111
PyEval_GetGlobals (C function), 111
PyEval_GetLocals (C function), 111
PyEval_InitThreads (C function), 270
PyEval_InitThreads(), 260
PyEval_MergeCompilerFlags (C function), 57
PyEval_ReleaseLock (C function), 277
PyEval_ReleaseThread (C function), 276

PyEval_ReleaseThread(), 270
PyEval_RestoreThread (C function), 268, 271
PyEval_RestoreThread(), 270
PyEval_SaveThread (C function), 268, 270
PyEval_SaveThread(), 270
PyEval_SetProfile (C function), 284
PyEval_SetProfileAllThreads (C function), 284
PyEval_SetTrace (C function), 284
PyEval_SetTraceAllThreads (C function), 285
PyEval_ThreadsInitialized (C function), 270
PyExc_ArithmeticError (C var), 79
PyExc_AssertionError (C var), 79
PyExc_AttributeError (C var), 79
PyExc_BaseException (C var), 79
PyExc_BlockingIOError (C var), 79
PyExc_BrokenPipeError (C var), 79
PyExc_BufferError (C var), 79
PyExc_BytesWarning (C var), 81
PyExc_ChildProcessError (C var), 79
PyExc_ConnectionAbortedError (C var), 79
PyExc_ConnectionError (C var), 79
PyExc_ConnectionRefusedError (C var), 79
PyExc_ConnectionResetError (C var), 79
PyExc_DeprecationWarning (C var), 81
PyExc_EnvironmentError (C var), 80
PyExc_EOFError (C var), 79
PyExc_Exception (C var), 79
PyExc_FileExistsError (C var), 79
PyExc_FileNotFoundError (C var), 79
PyExc_FloatingPointError (C var), 79
PyExc_FutureWarning (C var), 81
PyExc_GeneratorExit (C var), 79
PyExc_ImportError (C var), 79
PyExc_ImportWarning (C var), 81
PyExc_IndentationError (C var), 79
PyExc_IndexError (C var), 79
PyExc_InterruptedError (C var), 79
PyExc_IOError (C var), 80
PyExc_IsADirectoryError (C var), 79
PyExc_KeyboardInterrupt (C var), 79
PyExc_KeyError (C var), 79
PyExc_LookupError (C var), 79
PyExc_MemoryError (C var), 79
PyExc_ModuleNotFoundError (C var), 79
PyExc_NameError (C var), 79
PyExc_NotADirectoryError (C var), 79
PyExc_NotImplementedError (C var), 79
PyExc_OSError (C var), 79
PyExc_OverflowError (C var), 79
PyExc_PendingDeprecationWarning (C var), 81
PyExc_PermissionError (C var), 79
PyExc_ProcessLookupError (C var), 79
PyExc_RecursionError (C var), 79
PyExc_ReferenceError (C var), 79
PyExc_ResourceWarning (C var), 81
PyExc_RuntimeError (C var), 79
PyExc_RuntimeWarning (C var), 81
PyExc_StopAsyncIteration (C var), 79
PyExc_StopIteration (C var), 79
PyExc_SyntaxError (C var), 79
PyExc_SyntaxWarning (C var), 81
PyExc_SystemError (C var), 79
PyExc_SystemExit (C var), 79
PyExc_TabError (C var), 79
PyExc_TimeoutError (C var), 79
PyExc_TypeError (C var), 79
PyExc_UnboundLocalError (C var), 79
PyExc_UnicodeDecodeError (C var), 79
PyExc_UnicodeEncodeError (C var), 79
PyExc_UnicodeError (C var), 79
PyExc_UnicodeTranslateError (C var), 79
PyExc_UnicodeWarning (C var), 81
PyExc_UserWarning (C var), 81
PyExc_ValueError (C var), 79
PyExc_Warning (C var), 81
PyExc_WindowsError (C var), 80
PyExc_ZeroDivisionError (C var), 79
PyException_GetArgs (C function), 76
PyException_GetCause (C function), 75
PyException_GetContext (C function), 75
PyException_GetTraceback (C function), 75
PyException_SetArgs (C function), 76
PyException_SetCause (C function), 76
PyException_SetContext (C function), 75
PyException_SetTraceback (C function), 75
PyFile_FromFd (C function), 221
PyFile_GetLine (C function), 222
PyFile_SetOpenCodeHook (C function), 222
PyFile_WriteObject (C function), 222
PyFile_WriteString (C function), 222
PyFloat_AS_DOUBLE (C function), 166
PyFloat_AsDouble (C function), 166
PyFloat_Check (C function), 165
PyFloat_CheckExact (C function), 165
PyFloat_FromDouble (C function), 165
PyFloat_FromString (C function), 165
PyFloat_GetInfo (C function), 166
PyFloat_GetMax (C function), 166
PyFloat_GetMin (C function), 166
PyFloat_Pack2 (C function), 167
PyFloat_Pack4 (C function), 167
PyFloat_Pack8 (C function), 167
PyFloat_Type (C var), 165
PyFloat_Unpack2 (C function), 167
PyFloat_Unpack4 (C function), 167
PyFloat_Unpack8 (C function), 167
PyFloatObject (C type), 165
PyFrame_Check (C function), 242
PyFrame_GetBack (C function), 242
PyFrame_GetBuiltins (C function), 242
PyFrame_GetCode (C function), 242
PyFrame_GetGenerator (C function), 242
PyFrame_GetGlobals (C function), 242
PyFrame_GetLasti (C function), 243
PyFrame_GetLineNumber (C function), 243
PyFrame_GetLocals (C function), 243
PyFrame_GetVar (C function), 243
PyFrame_GetVarString (C function), 243
PyFrame_Type (C var), 242
PyFrameObject (C type), 241
PyFrozenSet_Check (C function), 209
PyFrozenSet_CheckExact (C function), 210
PyFrozenSet_New (C function), 210
PyFrozenSet_Type (C var), 209
PyFunction_AddWatcher (C function), 213
PyFunction_Check (C function), 211
PyFunction_ClearWatcher (C function), 213
PyFunction_GetAnnotations (C function), 213
PyFunction_GetClosure (C function), 212
PyFunction_GetCode (C function), 212
PyFunction_GetDefaults (C function), 212
PyFunction_GetGlobals (C function), 212
PyFunction_GetModule (C function), 212
PyFunction_New (C function), 211
PyFunction_NewWithQualName (C function), 212
PyFunction_SetAnnotations (C function), 213
PyFunction_SetClosure (C function), 213
PyFunction_SetDefaults (C function), 212
PyFunction_SetVectorcall (C function), 212
PyFunction_Type (C var), 211
PyFunction_WatchCallback (C type), 213
PyFunction_WatchEvent (C type), 213
PyFunctionObject (C type), 211
PyGC_Collect (C function), 397
PyGC_Disable (C function), 397
PyGC_Enable (C function), 397
PyGC_IsEnabled (C function), 397
PyGen_Check (C function), 244
PyGen_CheckExact (C function), 244
PyGen_New (C function), 245
PyGen_NewWithQualName (C function), 245
PyGen_Type (C var), 244
PyGenObject (C type), 244
PyGetSetDef (C type), 345
PyGetSetDef.closure (C member), 345
PyGetSetDef.doc (C member), 345
PyGetSetDef.get (C member), 345
PyGetSetDef.name (C member), 345
PyGetSetDef.set (C member), 345
PyGILState_Check (C function), 272
PyGILState_Ensure (C function), 271
PyGILState_GetThisThreadState (C function), 272
PyGILState_Release (C function), 272
PyHash_FuncDef (C type), 110
PyHash_FuncDef.hash_bits (C member), 110
PyHash_FuncDef.name (C member), 110
PyHash_FuncDef.seed_bits (C member), 110
PyHash_GetFuncDef (C function), 110
PyImport_AddModule (C function), 92
PyImport_AddModuleObject (C function), 91
PyImport_AppendInittab (C function), 94
PyImport_ExecCodeModule (C function), 92
PyImport_ExecCodeModuleEx (C function), 92
PyImport_ExecCodeModuleObject (C function), 92
PyImport_ExecCodeModuleWithPathnames (C function), 93
PyImport_ExtendInittab (C function), 95

PyImport_FrozenModules (C var), 94
PyImport_GetImporter (C function), 93
PyImport_GetMagicNumber (C function), 93
PyImport_GetMagicTag (C function), 93
PyImport_GetModule (C function), 93
PyImport_GetModuleDict (C function), 93
PyImport_Import (C function), 91
PyImport_ImportFrozenModule (C function), 94
PyImport_ImportFrozenModuleObject (C function), 94
PyImport_ImportModule (C function), 90
PyImport_ImportModuleEx (C function), 90
PyImport_ImportModuleLevel (C function), 91
PyImport_ImportModuleLevelObject (C function), 91
PyImport_ImportModuleNoBlock (C function), 90
PyImport_ReloadModule (C function), 91
PyIndex_Check (C function), 134
PyInstanceMethod_Check (C function), 214
PyInstanceMethod_Function (C function), 215
PyInstanceMethod_GET_FUNCTION (C function), 215
PyInstanceMethod_New (C function), 214
PyInstanceMethod_Type (C var), 214
PyInterpreterConfig (C type), 277
PyInterpreterConfig_DEFAULT_GIL (C macro), 278
PyInterpreterConfig_OWN_GIL (C macro), 278
PyInterpreterConfig_SHARED_GIL (C macro), 278
PyInterpreterConfig.allow_daemon_threads (C member), 278
PyInterpreterConfig.allow_exec (C member), 278
PyInterpreterConfig.allow_fork (C member), 278
PyInterpreterConfig.allow_threads (C member), 278
PyInterpreterConfig.check_multi_interp_extensions (C member), 278
PyInterpreterConfig.gil (C member), 278
PyInterpreterConfig.use_main_obmalloc (C member), 277
PyInterpreterState (C type), 270
PyInterpreterState_Clear (C function), 273
PyInterpreterState_Delete (C function), 273
PyInterpreterState_Get (C function), 274
PyInterpreterState_GetDict (C function), 275
PyInterpreterState_GetID (C function), 275
PyInterpreterState_Head (C function), 285
PyInterpreterState_Main (C function), 285
PyInterpreterState_New (C function), 273
PyInterpreterState_Next (C function), 285
PyInterpreterState_ThreadHead (C function), 285
PyIter_Check (C function), 138
PyIter_Next (C function), 138
PyIter_Send (C function), 139
PyList_Append (C function), 203
PyList_AsTuple (C function), 203
PyList_Check (C function), 202
PyList_CheckExact (C function), 202
PyList_GET_ITEM (C function), 202
PyList_GET_SIZE (C function), 202
PyList_GetItem (C function), 11, 202
PyList_GetSlice (C function), 203
PyList_Insert (C function), 203
PyList_New (C function), 202
PyList_Reverse (C function), 203
PyList_SET_ITEM (C function), 203
PyList_SetItem (C function), 9, 202
PyList_SetSlice (C function), 203
PyList_Size (C function), 202
PyList_Sort (C function), 203
PyList_Type (C var), 202
PyListObject (C type), 202
PyLong_AS_LONG (C function), 161
PyLong_AsDouble (C function), 163
PyLong_AsLong (C function), 161
PyLong_AsLongAndOverflow (C function), 161
PyLong_AsLongLong (C function), 161
PyLong_AsLongLongAndOverflow (C function), 162
PyLong_AsSize_t (C function), 162
PyLong_AsSsize_t (C function), 162
PyLong_AsUnsignedLong (C function), 162
PyLong_AsUnsignedLongLong (C function), 162
PyLong_AsUnsignedLongLongMask (C function), 163
PyLong_AsUnsignedLongMask (C function), 163
PyLong_AsVoidPtr (C function), 163
PyLong_Check (C function), 159
PyLong_CheckExact (C function), 159
PyLong_FromDouble (C function), 160
PyLong_FromLong (C function), 159
PyLong_FromLongLong (C function), 160
PyLong_FromSize_t (C function), 160
PyLong_FromSsize_t (C function), 160
PyLong_FromString (C function), 160
PyLong_FromUnicodeObject (C function), 160
PyLong_FromUnsignedLong (C function), 160
PyLong_FromUnsignedLongLong (C function), 160
PyLong_FromVoidPtr (C function), 161
PyLong_GetInfo (C function), 164
PyLong_Type (C var), 159
PyLongObject (C type), 159
PyMapping_Check (C function), 137
PyMapping_DelItem (C function), 137
PyMapping_DelItemString (C function), 137
PyMapping_GetItemString (C function), 137
PyMapping_HasKey (C function), 137
PyMapping_HasKeyString (C function), 138
PyMapping_Items (C function), 138
PyMapping_Keys (C function), 138
PyMapping_Length (C function), 137
PyMapping_SetItemString (C function), 137
PyMapping_Size (C function), 137
PyMapping_Values (C function), 138
PyMappingMethods (C type), 384
PyMappingMethods.mp_ass_subscript (C member), 384
PyMappingMethods.mp_length (C member), 384
PyMappingMethods.mp_subscript (C member), 384
PyMarshal_ReadLastObjectFromFile (C function), 96
PyMarshal_ReadLongFromFile (C function), 96
PyMarshal_ReadObjectFromFile (C function), 96
PyMarshal_ReadObjectFromString (C function), 96
PyMarshal_ReadShortFromFile (C function), 96
PyMarshal_WriteLongToFile (C function), 95
PyMarshal_WriteObjectToFile (C function), 95
PyMarshal_WriteObjectToString (C function), 96
PyMem_Calloc (C function), 322
PyMem_Del (C function), 323
PYMEM_DOMAIN_MEM (C macro), 326
PYMEM_DOMAIN_OBJ (C macro), 326
PYMEM_DOMAIN_RAW (C macro), 326
PyMem_Free (C function), 323
PyMem_GetAllocator (C function), 326
PyMem_Malloc (C function), 322
PyMem_New (C macro), 323
PyMem_RawCalloc (C function), 321
PyMem_RawFree (C function), 321
PyMem_RawMalloc (C function), 321
PyMem_RawRealloc (C function), 321
PyMem_Realloc (C function), 322
PyMem_Resize (C macro), 323
PyMem_SetAllocator (C function), 327
PyMem_SetupDebugHooks (C function), 327
PyMemAllocatorDomain (C type), 326
PyMemAllocatorEx (C type), 325
PyMember_GetOne (C function), 342
PyMember_SetOne (C function), 342
PyMemberDef (C type), 341
PyMemberDef.doc (C member), 341
PyMemberDef.flags (C member), 341
PyMemberDef.name (C member), 341
PyMemberDef.offset (C member), 341
PyMemberDef.type (C member), 341
PyMemoryView_Check (C function), 238
PyMemoryView_FromBuffer (C function), 237
PyMemoryView_FromMemory (C function), 237
PyMemoryView_FromObject (C function), 237
PyMemoryView_GET_BASE (C function), 238
PyMemoryView_GET_BUFFER (C function), 238
PyMemoryView_GetContiguous (C function), 237
PyMethod_Check (C function), 215
PyMethod_Function (C function), 215
PyMethod_GET_FUNCTION (C function), 215

PyMethod_GET_SELF (C function), 215
PyMethod_New (C function), 215
PyMethod_Self (C function), 215
PyMethod_Type (C var), 215
PyMethodDef (C type), 337
PyMethodDef.ml_doc (C member), 338
PyMethodDef.ml_flags (C member), 338
PyMethodDef.ml_meth (C member), 338
PyMethodDef.ml_name (C member), 338
PyMODINIT_FUNC (C macro), 4
PyModule_AddFunctions (C function), 229
PyModule_AddIntConstant (C function), 232
PyModule_AddIntMacro (C macro), 232
PyModule_AddObject (C function), 231
PyModule_AddObjectRef (C function), 230
PyModule_AddStringConstant (C function), 232
PyModule_AddStringMacro (C macro), 232
PyModule_AddType (C function), 232
PyModule_Check (C function), 223
PyModule_CheckExact (C function), 223
PyModule_Create (C function), 226
PyModule_Create2 (C function), 226
PyModule_ExecDef (C function), 229
PyModule_FromDefAndSpec (C function), 229
PyModule_FromDefAndSpec2 (C function), 229
PyModule_GetDef (C function), 224
PyModule_GetDict (C function), 223
PyModule_GetFilename (C function), 224
PyModule_GetFilenameObject (C function), 224
PyModule_GetName (C function), 224
PyModule_GetNameObject (C function), 223
PyModule_GetState (C function), 224
PyModule_New (C function), 223
PyModule_NewObject (C function), 223
PyModule_SetDocString (C function), 229
PyModule_Type (C var), 223
PyModuleDef (C type), 224
PyModuleDef_Init (C function), 227
PyModuleDef_Slot (C type), 227
PyModuleDef_Slot.slot (C member), 227
PyModuleDef_Slot.value (C member), 227
PyModuleDef.m_base (C member), 224
PyModuleDef.m_clear (C member), 225
PyModuleDef.m_doc (C member), 225
PyModuleDef.m_free (C member), 226
PyModuleDef.m_methods (C member), 225
PyModuleDef.m_name (C member), 225
PyModuleDef.m_size (C member), 225
PyModuleDef.m_slots (C member), 225
PyModuleDef.m_slots.m_reload (C member), 225
PyModuleDef.m_traverse (C member), 225
PyNumber_Absolute (C function), 131
PyNumber_Add (C function), 130
PyNumber_And (C function), 132
PyNumber_AsSsize_t (C function), 134
PyNumber_Check (C function), 130
PyNumber_Divmod (C function), 131
PyNumber_Float (C function), 133
PyNumber_FloorDivide (C function), 130
PyNumber_Index (C function), 134
PyNumber_InPlaceAdd (C function), 132
PyNumber_InPlaceAnd (C function), 133
PyNumber_InPlaceFloorDivide (C function), 132
PyNumber_InPlaceLshift (C function), 133
PyNumber_InPlaceMatrixMultiply (C function), 132
PyNumber_InPlaceMultiply (C function), 132
PyNumber_InPlaceOr (C function), 133
PyNumber_InPlacePower (C function), 133
PyNumber_InPlaceRemainder (C function), 133
PyNumber_InPlaceRshift (C function), 133
PyNumber_InPlaceSubtract (C function), 132
PyNumber_InPlaceTrueDivide (C function), 132
PyNumber_InPlaceXor (C function), 133
PyNumber_Invert (C function), 131
PyNumber_Long (C function), 133
PyNumber_Lshift (C function), 131
PyNumber_MatrixMultiply (C function), 130
PyNumber_Multiply (C function), 130
PyNumber_Negative (C function), 131
PyNumber_Or (C function), 132
PyNumber_Positive (C function), 131
PyNumber_Power (C function), 131
PyNumber_Remainder (C function), 131
PyNumber_Rshift (C function), 131
PyNumber_Subtract (C function), 130
PyNumber_ToBase (C function), 134
PyNumber_TrueDivide (C function), 131
PyNumber_Xor (C function), 132
PyNumberMethods (C type), 381
PyNumberMethods.nb_absolute (C member), 383
PyNumberMethods.nb_add (C member), 383
PyNumberMethods.nb_and (C member), 383
PyNumberMethods.nb_bool (C member), 383
PyNumberMethods.nb_divmod (C member), 383
PyNumberMethods.nb_float (C member), 383
PyNumberMethods.nb_floor_divide (C member), 384
PyNumberMethods.nb_index (C member), 384
PyNumberMethods.nb_inplace_add (C member), 383
PyNumberMethods.nb_inplace_and (C member), 384
PyNumberMethods.nb_inplace_floor_divide (C member), 384
PyNumberMethods.nb_inplace_lshift (C member), 384
PyNumberMethods.nb_inplace_matrix_multiply (C member), 384
PyNumberMethods.nb_inplace_multiply (C member), 383
PyNumberMethods.nb_inplace_or (C member), 384
PyNumberMethods.nb_inplace_power (C member), 384
PyNumberMethods.nb_inplace_remainder (C member), 383
PyNumberMethods.nb_inplace_rshift (C member), 384
PyNumberMethods.nb_inplace_subtract (C member), 383
PyNumberMethods.nb_inplace_true_divide (C member), 384
PyNumberMethods.nb_inplace_xor (C member), 384
PyNumberMethods.nb_int (C member), 383
PyNumberMethods.nb_invert (C member), 383
PyNumberMethods.nb_lshift (C member), 383
PyNumberMethods.nb_matrix_multiply (C member), 384
PyNumberMethods.nb_multiply (C member), 383
PyNumberMethods.nb_negative (C member), 383
PyNumberMethods.nb_or (C member), 383
PyNumberMethods.nb_positive (C member), 383
PyNumberMethods.nb_power (C member), 383
PyNumberMethods.nb_remainder (C member), 383
PyNumberMethods.nb_reserved (C member), 383
PyNumberMethods.nb_rshift (C member), 383
PyNumberMethods.nb_subtract (C member), 383
PyNumberMethods.nb_true_divide (C member), 384
PyNumberMethods.nb_xor (C member), 383
PyObject (C type), 334
PyObject_AsCharBuffer (C function), 149
PyObject_ASCII (C function), 120
PyObject_AsFileDescriptor (C function), 221
PyObject_AsReadBuffer (C function), 149
PyObject_AsWriteBuffer (C function), 149
PyObject_Bytes (C function), 121
PyObject_Call (C function), 127
PyObject_CallFunction (C function), 128
PyObject_CallFunctionObjArgs (C function), 128
PyObject_CallMethod (C function), 128
PyObject_CallMethodNoArgs (C function), 129
PyObject_CallMethodObjArgs (C function), 129
PyObject_CallMethodOneArg (C function), 129
PyObject_CallNoArgs (C function), 127
PyObject_CallObject (C function), 128
PyObject_Calloc (C function), 324
PyObject_CallOneArg (C function), 127
PyObject_CheckBuffer (C function), 147
PyObject_CheckReadBuffer (C function), 149

PyObject_ClearWeakRefs (C function), 239
PyObject_CopyData (C function), 148
PyObject_Del (C function), 334
PyObject_DelAttr (C function), 119
PyObject_DelAttrString (C function), 119
PyObject_DelItem (C function), 122
PyObject_DelItemString (C function), 123
PyObject_Dir (C function), 123
PyObject_Format (C function), 120
PyObject_Free (C function), 324
PyObject_GC_Del (C function), 396
PyObject_GC_IsFinalized (C function), 395
PyObject_GC_IsTracked (C function), 395
PyObject_GC_New (C macro), 394
PyObject_GC_NewVar (C macro), 394
PyObject_GC_Resize (C macro), 395
PyObject_GC_Track (C function), 395
PyObject_GC_UnTrack (C function), 396
PyObject_GenericGetAttr (C function), 118
PyObject_GenericGetDict (C function), 119
PyObject_GenericSetAttr (C function), 119
PyObject_GenericSetDict (C function), 119
PyObject_GetAIter (C function), 123
PyObject_GetArenaAllocator (C function), 330
PyObject_GetAttr (C function), 118
PyObject_GetAttrString (C function), 118
PyObject_GetBuffer (C function), 147
PyObject_GetItem (C function), 122
PyObject_GetItemData (C function), 124
PyObject_GetIter (C function), 123
PyObject_GetTypeData (C function), 123
PyObject_HasAttr (C function), 117
PyObject_HasAttrString (C function), 118
PyObject_Hash (C function), 121
PyObject_HashNotImplemented (C function), 121
PyObject_HEAD (C macro), 335
PyObject_HEAD_INIT (C macro), 336
PyObject_Init (C function), 333
PyObject_InitVar (C function), 333
PyObject_IS_GC (C function), 395
PyObject_IsInstance (C function), 121
PyObject_IsSubclass (C function), 121
PyObject_IsTrue (C function), 122
PyObject_Length (C function), 122
PyObject_LengthHint (C function), 122
PyObject_Malloc (C function), 324
PyObject_New (C macro), 333
PyObject_NewVar (C macro), 333
PyObject_Not (C function), 122
PyObject._ob_next (C member), 353
PyObject._ob_prev (C member), 353
PyObject_Print (C function), 117
PyObject_Realloc (C function), 324
PyObject_Repr (C function), 120
PyObject_RichCompare (C function), 120
PyObject_RichCompareBool (C function), 120
PyObject_SelfIter (C function), 123
PyObject_SetArenaAllocator (C function), 330
PyObject_SetAttr (C function), 118
PyObject_SetAttrString (C function), 118
PyObject_SetItem (C function), 122
PyObject_Size (C function), 122
PyObject_Str (C function), 120
PyObject_Type (C function), 122
PyObject_TypeCheck (C function), 122
PyObject_VAR_HEAD (C macro), 335
PyObject_Vectorcall (C function), 129
PyObject_VectorcallDict (C function), 129
PyObject_VectorcallMethod (C function), 129
PyObjectArenaAllocator (C type), 330
PyObject.ob_refcnt (C member), 353
PyObject.ob_type (C member), 353
PyOS_AfterFork (C function), 84
PyOS_AfterFork_Child (C function), 84
PyOS_AfterFork_Parent (C function), 84
PyOS_BeforeFork (C function), 83
PyOS_CheckStack (C function), 84
PyOS_double_to_string (C function), 109
PyOS_FSPath (C function), 83
PyOS_getsig (C function), 85
PyOS_InputHook (C var), 55
PyOS_ReadlineFunctionPointer (C var), 55
PyOS_setsig (C function), 85
PyOS_sighandler_t (C type), 85
PyOS_snprintf (C function), 108
PyOS_stricmp (C function), 110
PyOS_string_to_double (C function), 109
PyOS_strnicmp (C function), 110
PyOS_strtol (C function), 108
PyOS_strtoul (C function), 108
PyOS_vsnprintf (C function), 108
PyPreConfig (C type), 293
PyPreConfig_InitIsolatedConfig (C function), 293
PyPreConfig_InitPythonConfig (C function), 293
PyPreConfig.allocator (C member), 293
PyPreConfig.coerce_c_locale (C member), 293
PyPreConfig.coerce_c_locale_warn (C member), 294
PyPreConfig.configure_locale (C member), 293
PyPreConfig.dev_mode (C member), 294
PyPreConfig.isolated (C member), 294
PyPreConfig.legacy_windows_fs_encoding (C member), 294
PyPreConfig.parse_argv (C member), 294
PyPreConfig.use_environment (C member), 294
PyPreConfig.utf8_mode (C member), 294
PyProperty_Type (C var), 234
PyRun_AnyFile (C function), 53
PyRun_AnyFileEx (C function), 54
PyRun_AnyFileExFlags (C function), 54
PyRun_AnyFileFlags (C function), 53
PyRun_File (C function), 56
PyRun_FileEx (C function), 56
PyRun_FileExFlags (C function), 56
PyRun_FileFlags (C function), 56
PyRun_InteractiveLoop (C function), 55
PyRun_InteractiveLoopFlags (C function), 55
PyRun_InteractiveOne (C function), 54
PyRun_InteractiveOneFlags (C function), 55
PyRun_SimpleFile (C function), 54
PyRun_SimpleFileEx (C function), 54
PyRun_SimpleFileExFlags (C function), 54
PyRun_SimpleString (C function), 54
PyRun_SimpleStringFlags (C function), 54
PyRun_String (C function), 55
PyRun_StringFlags (C function), 56
PySendResult (C type), 139
PySeqIter_Check (C function), 234
PySeqIter_New (C function), 234
PySeqIter_Type (C var), 233
PySequence_Check (C function), 134
PySequence_Concat (C function), 134
PySequence_Contains (C function), 136
PySequence_Count (C function), 135
PySequence_DelItem (C function), 135
PySequence_DelSlice (C function), 135
PySequence_Fast (C function), 136
PySequence_Fast_GET_ITEM (C function), 136
PySequence_Fast_GET_SIZE (C function), 136
PySequence_Fast_ITEMS (C function), 136
PySequence_GetItem (C function), 11, 135
PySequence_GetSlice (C function), 135
PySequence_Index (C function), 136
PySequence_InPlaceConcat (C function), 135
PySequence_InPlaceRepeat (C function), 135
PySequence_ITEM (C function), 137
PySequence_Length (C function), 134
PySequence_List (C function), 136
PySequence_Repeat (C function), 135
PySequence_SetItem (C function), 135
PySequence_SetSlice (C function), 135
PySequence_Size (C function), 134
PySequence_Tuple (C function), 136
PySequenceMethods (C type), 385
PySequenceMethods.sq_ass_item (C member), 385
PySequenceMethods.sq_concat (C member), 385
PySequenceMethods.sq_contains (C member), 385
PySequenceMethods.sq_inplace_concat (C member), 385
PySequenceMethods.sq_inplace_repeat (C member), 385
PySequenceMethods.sq_item (C member), 385
PySequenceMethods.sq_length (C member), 385
PySequenceMethods.sq_repeat (C member), 385
PySet_Add (C function), 211
PySet_Check (C function), 209
PySet_CheckExact (C function), 210
PySet_Clear (C function), 211
PySet_Contains (C function), 210

PySet_Discard (C function), 211
PySet_GET_SIZE (C function), 210
PySet_New (C function), 210
PySet_Pop (C function), 211
PySet_Size (C function), 210
PySet_Type (C var), 209
PySetObject (C type), 209
PySignal_SetWakeupFd (C function), 74
PySlice_AdjustIndices (C function), 236
PySlice_Check (C function), 235
PySlice_GetIndices (C function), 235
PySlice_GetIndicesEx (C function), 235
PySlice_New (C function), 235
PySlice_Type (C var), 235
PySlice_Unpack (C function), 236
PyState_AddModule (C function), 233
PyState_FindModule (C function), 233
PyState_RemoveModule (C function), 233
PyStatus (C type), 291
PyStatus_Error (C function), 291
PyStatus_Exception (C function), 291
PyStatus_Exit (C function), 291
PyStatus_IsError (C function), 292
PyStatus_IsExit (C function), 292
PyStatus_NoMemory (C function), 291
PyStatus_Ok (C function), 291
PyStatus.err_msg (C member), 291
PyStatus.exitcode (C member), 291
PyStatus.func (C member), 291
PyStructSequence_Desc (C type), 200
PyStructSequence_Desc.doc (C member), 200
PyStructSequence_Desc.fields (C member), 200
PyStructSequence_Desc.n_in_sequence (C member), 200
PyStructSequence_Desc.name (C member), 200
PyStructSequence_Field (C type), 200
PyStructSequence_Field.doc (C member), 201
PyStructSequence_Field.name (C member), 200
PyStructSequence_GET_ITEM (C function), 201
PyStructSequence_GetItem (C function), 201
PyStructSequence_InitType (C function), 200
PyStructSequence_InitType2 (C function), 200
PyStructSequence_New (C function), 201
PyStructSequence_NewType (C function), 200
PyStructSequence_SET_ITEM (C function), 201
PyStructSequence_SetItem (C function), 201
PyStructSequence_UnnamedField (C var), 201
PySys_AddAuditHook (C function), 89
PySys_AddWarnOption (C function), 87
PySys_AddWarnOptionUnicode (C function), 87
PySys_AddXOption (C function), 88
PySys_Audit (C function), 88
PySys_FormatStderr (C function), 88
PySys_FormatStdout (C function), 88
PySys_GetObject (C function), 87
PySys_GetXOptions (C function), 88
PySys_ResetWarnOptions (C function), 87
PySys_SetArgv (C function), 260, 266
PySys_SetArgvEx (C function), 260, 266
PySys_SetObject (C function), 87
PySys_SetPath (C function), 87
PySys_WriteStderr (C function), 88
PySys_WriteStdout (C function), 88
Python 3000, 417
Python Enhancement Proposals
PEP 1, 417
PEP 7, 3, 7
PEP 238, 58, 408
PEP 278, 421
PEP 302, 413
PEP 343, 405
PEP 353, 12
PEP 362, 402, 416
PEP 383, 184, 185
PEP 387, 17, 18
PEP 393, 174
PEP 411, 417
PEP 420, 415, 417
PEP 432, 315, 316
PEP 442, 380
PEP 443, 409
PEP 451, 228
PEP 456, 111
PEP 483, 409
PEP 484, 402, 408, 409, 421, 422
PEP 489, 229, 278
PEP 492, 402, 403, 405
PEP 498, 407
PEP 519, 417
PEP 523, 243, 275
PEP 525, 403
PEP 526, 402, 422
PEP 528, 259, 304
PEP 529, 185, 258
PEP 538, 313
PEP 539, 286
PEP 540, 313
PEP 552, 300
PEP 554, 281
PEP 578, 89
PEP 585, 409
PEP 587, 289
PEP 590, 124
PEP 623, 174
PEP 0626#out-of-process-debuggers-and-profilers, 218
PEP 634, 367
PEP 0683, 59, 60, 410
PEP 3116, 421
PEP 3119, 121
PEP 3121, 225
PEP 3147, 93
PEP 3151, 80
PEP 3155, 418
PYTHONCOERCECLOCALE, 313
PYTHONDEBUG, 257, 305
PYTHONDEVMODE, 300
PYTHONDONTWRITEBYTECODE, 257, 310
PYTHONDUMPREFS, 300, 354
PYTHONEXECUTABLE, 306
PYTHONFAULTHANDLER, 301
PYTHONHASHSEED, 257, 302
PYTHONHOME, 15, 16, 257, 267, 302
Pythonic, 418
PYTHONINSPECT, 258, 302
PYTHONINTMAXSTRDIGITS, 303
PYTHONIOENCODING, 262, 308
PYTHONLEGACYWINDOWSFSENCODING, 258, 294
PYTHONLEGACYWINDOWSSTDIO, 259, 303
PYTHONMALLOC, 320, 325, 327, 330
PYTHONMALLOCSTATS, 304, 320
PYTHONNODEBUGRANGES, 299
PYTHONNOUSERSITE, 259, 309
PYTHONOPTIMIZE, 259, 305
PYTHONPATH, 15, 16, 257, 304
PYTHONPERFSUPPORT, 308
PYTHONPLATLIBDIR, 304
PYTHONPROFILEIMPORTTIME, 302
PYTHONPYCACHEPREFIX, 306
PYTHONSAFEPATH, 298
PYTHONTRACEMALLOC, 308
PYTHONUNBUFFERED, 260, 299
PYTHONUTF8, 294, 313
PYTHONVERBOSE, 260, 309
PYTHONWARNINGS, 309
PyThread_create_key (C function), 288
PyThread_delete_key (C function), 288
PyThread_delete_key_value (C function), 288
PyThread_get_key_value (C function), 288
PyThread_ReInitTLS (C function), 288
PyThread_set_key_value (C function), 288
PyThread_tss_alloc (C function), 286
PyThread_tss_create (C function), 287
PyThread_tss_delete (C function), 287
PyThread_tss_free (C function), 286
PyThread_tss_get (C function), 287
PyThread_tss_is_created (C function), 287
PyThread_tss_set (C function), 287
PyThreadState (C type), 267, 270
PyThreadState_Clear (C function), 273
PyThreadState_Delete (C function), 273
PyThreadState_DeleteCurrent (C function), 273
PyThreadState_EnterTracing (C function), 274
PyThreadState_Get (C function), 271
PyThreadState_GetDict (C function), 275
PyThreadState_GetFrame (C function), 274
PyThreadState_GetID (C function), 274
PyThreadState_GetInterpreter (C function), 274
PyThreadState_LeaveTracing (C function), 274
PyThreadState_New (C function), 273
PyThreadState_Next (C function), 285
PyThreadState_SetAsyncExc (C function), 276
PyThreadState_Swap (C function), 271
PyThreadState.interp (C member), 270
PyTime_Check (C function), 249
PyTime_CheckExact (C function), 249
PyTime_FromTime (C function), 250
PyTime_FromTimeAndFold (C function), 250
PyTimeZone_FromOffset (C function), 250
PyTimeZone_FromOffsetAndName (C function), 250
PyTrace_C_CALL (C var), 284
PyTrace_C_EXCEPTION (C var), 284
PyTrace_C_RETURN (C var), 284

PyTrace_CALL (C var), 283
PyTrace_EXCEPTION (C var), 283
PyTrace_LINE (C var), 283
PyTrace_OPCODE (C var), 284
PyTrace_RETURN (C var), 284
PyTraceMalloc_Track (C function), 330
PyTraceMalloc_Untrack (C function), 330
PyTuple_Check (C function), 198
PyTuple_CheckExact (C function), 198
PyTuple_GET_ITEM (C function), 199
PyTuple_GET_SIZE (C function), 198
PyTuple_GetItem (C function), 198
PyTuple_GetSlice (C function), 199
PyTuple_New (C function), 198
PyTuple_Pack (C function), 198
PyTuple_SET_ITEM (C function), 199
PyTuple_SetItem (C function), 9, 199
PyTuple_Size (C function), 198
PyTuple_Type (C var), 198
PyTupleObject (C type), 198
PyType_AddWatcher (C function), 152
PyType_Check (C function), 151
PyType_CheckExact (C function), 151
PyType_ClearCache (C function), 152
PyType_ClearWatcher (C function), 152
PyType_FromMetaclass (C function), 155
PyType_FromModuleAndSpec (C function), 156
PyType_FromSpec (C function), 157
PyType_FromSpecWithBases (C function), 156
PyType_GenericAlloc (C function), 153
PyType_GenericNew (C function), 153
PyType_GetDict (C function), 152
PyType_GetFlags (C function), 152
PyType_GetModule (C function), 154
PyType_GetModuleByDef (C function), 155
PyType_GetModuleState (C function), 154
PyType_GetName (C function), 154
PyType_GetQualName (C function), 154
PyType_GetSlot (C function), 154
PyType_GetTypeDataSize (C function), 123
PyType_HasFeature (C function), 153
PyType_IS_GC (C function), 153
PyType_IsSubtype (C function), 153
PyType_Modified (C function), 152
PyType_Ready (C function), 153
PyType_Slot (C type), 158
PyType_Slot.pfunc (C member), 158
PyType_Slot.slot (C member), 158
PyType_Spec (C type), 157
PyType_Spec.basicsize (C member), 157
PyType_Spec.flags (C member), 157
PyType_Spec.itemsize (C member), 157
PyType_Spec.name (C member), 157
PyType_Spec.slots (C member), 158
PyType_Type (C var), 151
PyType_Watch (C function), 152
PyType_WatchCallback (C type), 153
PyTypeObject (C type), 151
PyTypeObject.tp_alloc (C member), 376
PyTypeObject.tp_as_async (C member), 358
PyTypeObject.tp_as_buffer (C member), 361
PyTypeObject.tp_as_mapping (C member), 359
PyTypeObject.tp_as_number (C member), 359
PyTypeObject.tp_as_sequence (C member), 359
PyTypeObject.tp_base (C member), 373
PyTypeObject.tp_bases (C member), 378
PyTypeObject.tp_basicsize (C member), 355
PyTypeObject.tp_cache (C member), 379
PyTypeObject.tp_call (C member), 360
PyTypeObject.tp_clear (C member), 369
PyTypeObject.tp_dealloc (C member), 356
PyTypeObject.tp_del (C member), 379
PyTypeObject.tp_descr_get (C member), 374
PyTypeObject.tp_descr_set (C member), 375
PyTypeObject.tp_dict (C member), 374
PyTypeObject.tp_dictoffset (C member), 375
PyTypeObject.tp_doc (C member), 367
PyTypeObject.tp_finalize (C member), 379
PyTypeObject.tp_flags (C member), 362
PyTypeObject.tp_free (C member), 377
PyTypeObject.tp_getattr (C member), 358
PyTypeObject.tp_getattro (C member), 361
PyTypeObject.tp_getset (C member), 373
PyTypeObject.tp_hash (C member), 359
PyTypeObject.tp_init (C member), 376
PyTypeObject.tp_is_gc (C member), 378
PyTypeObject.tp_itemsize (C member), 355
PyTypeObject.tp_iter (C member), 372
PyTypeObject.tp_iternext (C member), 372
PyTypeObject.tp_members (C member), 373
PyTypeObject.tp_methods (C member), 373
PyTypeObject.tp_mro (C member), 378
PyTypeObject.tp_name (C member), 354
PyTypeObject.tp_new (C member), 377
PyTypeObject.tp_repr (C member), 358
PyTypeObject.tp_richcompare (C member), 370
PyTypeObject.tp_setattr (C member), 358
PyTypeObject.tp_setattro (C member), 361
PyTypeObject.tp_str (C member), 360
PyTypeObject.tp_subclasses (C member), 379
PyTypeObject.tp_traverse (C member), 367
PyTypeObject.tp_vectorcall (C member), 380
PyTypeObject.tp_vectorcall_offset (C member), 357
PyTypeObject.tp_version_tag (C member), 379
PyTypeObject.tp_watched (C member), 380
PyTypeObject.tp_weaklist (C member), 379
PyTypeObject.tp_weaklistoffset (C member), 372
PyTZInfo_Check (C function), 249
PyTZInfo_CheckExact (C function), 249
PyUnicode_1BYTE_DATA (C function), 175
PyUnicode_1BYTE_KIND (C macro), 175
PyUnicode_2BYTE_DATA (C function), 175
PyUnicode_2BYTE_KIND (C macro), 175
PyUnicode_4BYTE_DATA (C function), 175
PyUnicode_4BYTE_KIND (C macro), 175
PyUnicode_AsASCIIString (C function), 192
PyUnicode_AsCharmapString (C function), 193
PyUnicode_AsEncodedString (C function), 188
PyUnicode_AsLatin1String (C function), 192
PyUnicode_AsMBCSString (C function), 194
PyUnicode_AsRawUnicodeEscapeString (C function), 192
PyUnicode_AsUCS4 (C function), 184
PyUnicode_AsUCS4Copy (C function), 184
PyUnicode_AsUnicodeEscapeString (C function), 191
PyUnicode_AsUTF8 (C function), 189
PyUnicode_AsUTF8AndSize (C function), 189
PyUnicode_AsUTF8String (C function), 189
PyUnicode_AsUTF16String (C function), 191
PyUnicode_AsUTF32String (C function), 190
PyUnicode_AsWideChar (C function), 187
PyUnicode_AsWideCharString (C function), 187
PyUnicode_Check (C function), 175
PyUnicode_CheckExact (C function), 175
PyUnicode_Compare (C function), 196
PyUnicode_CompareWithASCIIString (C function), 196
PyUnicode_Concat (C function), 194
PyUnicode_Contains (C function), 197
PyUnicode_CopyCharacters (C function), 183
PyUnicode_Count (C function), 196
PyUnicode_DATA (C function), 175
PyUnicode_Decode (C function), 188
PyUnicode_DecodeASCII (C function), 192
PyUnicode_DecodeCharmap (C function), 193
PyUnicode_DecodeCodePageStateful (C function), 194
PyUnicode_DecodeFSDefault (C function), 186
PyUnicode_DecodeFSDefaultAndSize (C function), 186
PyUnicode_DecodeLatin1 (C function), 192

PyUnicode_DecodeLocale (C function), 184
PyUnicode_DecodeLocaleAndSize (C function), 184
PyUnicode_DecodeMBCS (C function), 194
PyUnicode_DecodeMBCSStateful (C function), 194
PyUnicode_DecodeRawUnicodeEscape (C function), 192
PyUnicode_DecodeUnicodeEscape (C function), 191
PyUnicode_DecodeUTF7 (C function), 191
PyUnicode_DecodeUTF7Stateful (C function), 191
PyUnicode_DecodeUTF8 (C function), 188
PyUnicode_DecodeUTF8Stateful (C function), 188
PyUnicode_DecodeUTF16 (C function), 190
PyUnicode_DecodeUTF16Stateful (C function), 191
PyUnicode_DecodeUTF32 (C function), 189
PyUnicode_DecodeUTF32Stateful (C function), 190
PyUnicode_EncodeCodePage (C function), 194
PyUnicode_EncodeFSDefault (C function), 186
PyUnicode_EncodeLocale (C function), 185
PyUnicode_Fill (C function), 183
PyUnicode_Find (C function), 196
PyUnicode_FindChar (C function), 196
PyUnicode_Format (C function), 197
PyUnicode_FromEncodedObject (C function), 182
PyUnicode_FromFormat (C function), 179
PyUnicode_FromFormatV (C function), 182
PyUnicode_FromKindAndData (C function), 178
PyUnicode_FromObject (C function), 182
PyUnicode_FromString (C function), 179
PyUnicode_FromStringAndSize (C function), 179
PyUnicode_FromWideChar (C function), 187
PyUnicode_FSConverter (C function), 185
PyUnicode_FSDecoder (C function), 186
PyUnicode_GET_LENGTH (C function), 175
PyUnicode_GetDefaultEncoding (C function), 182
PyUnicode_GetLength (C function), 182
PyUnicode_InternFromString (C function), 197
PyUnicode_InternInPlace (C function), 197
PyUnicode_IsIdentifier (C function), 176
PyUnicode_Join (C function), 195
PyUnicode_KIND (C function), 175
PyUnicode_MAX_CHAR_VALUE (C function), 176
PyUnicode_New (C function), 178
PyUnicode_Partition (C function), 195
PyUnicode_READ (C function), 176
PyUnicode_READ_CHAR (C function), 176
PyUnicode_ReadChar (C function), 183
PyUnicode_READY (C function), 175
PyUnicode_Replace (C function), 196
PyUnicode_RichCompare (C function), 197
PyUnicode_RPartition (C function), 195
PyUnicode_RSplit (C function), 195
PyUnicode_Split (C function), 195
PyUnicode_Splitlines (C function), 195
PyUnicode_Substring (C function), 183
PyUnicode_Tailmatch (C function), 195
PyUnicode_Translate (C function), 193
PyUnicode_Type (C var), 174
PyUnicode_WRITE (C function), 176
PyUnicode_WriteChar (C function), 183
PyUnicodeDecodeError_Create (C function), 76
PyUnicodeDecodeError_GetEncoding (C function), 76
PyUnicodeDecodeError_GetEnd (C function), 77
PyUnicodeDecodeError_GetObject (C function), 77
PyUnicodeDecodeError_GetReason (C function), 77
PyUnicodeDecodeError_GetStart (C function), 77
PyUnicodeDecodeError_SetEnd (C function), 77
PyUnicodeDecodeError_SetReason (C function), 77
PyUnicodeDecodeError_SetStart (C function), 77
PyUnicodeEncodeError_GetEncoding (C function), 76
PyUnicodeEncodeError_GetEnd (C function), 77
PyUnicodeEncodeError_GetObject (C function), 77
PyUnicodeEncodeError_GetReason (C function), 77
PyUnicodeEncodeError_GetStart (C function), 77
PyUnicodeEncodeError_SetEnd (C function), 77
PyUnicodeEncodeError_SetReason (C function), 77
PyUnicodeEncodeError_SetStart (C function), 77
PyUnicodeObject (C type), 174
PyUnicodeTranslateError_GetEnd (C function), 77
PyUnicodeTranslateError_GetObject (C function), 77
PyUnicodeTranslateError_GetReason (C function), 77
PyUnicodeTranslateError_GetStart (C function), 77
PyUnicodeTranslateError_SetEnd (C function), 77
PyUnicodeTranslateError_SetReason (C function), 77
PyUnicodeTranslateError_SetStart (C function), 77
PyUnstable, 17
PyUnstable_Code_GetExtra (C function), 220
PyUnstable_Code_New (C function), 217
PyUnstable_Code_NewWithPosOnlyArgs (C function), 217
PyUnstable_Code_SetExtra (C function), 221
PyUnstable_Eval_RequestCodeExtraIndex (C function), 220
PyUnstable_Exc_PrepReraiseStar (C function), 76
PyUnstable_GC_VisitObjects (C function), 398
PyUnstable_InterpreterFrame_GetCode (C function), 243
PyUnstable_InterpreterFrame_GetLasti (C function), 244
PyUnstable_InterpreterFrame_GetLine (C function), 244
PyUnstable_Long_CompactValue (C function), 164
PyUnstable_Long_IsCompact (C function), 164
PyUnstable_Object_GC_NewWithExtraData (C function), 394
PyUnstable_PerfMapState_Fini (C function), 114
PyUnstable_PerfMapState_Init (C function), 114
PyUnstable_Type_AssignVersionTag (C function), 155
PyUnstable_WritePerfMapEntry (C function), 114
PyVarObject (C type), 334
PyVarObject_HEAD_INIT (C macro), 336
PyVarObject.ob_size (C member), 354
PyVectorcall_Call (C function), 126
PyVectorcall_Function (C function), 126
PyVectorcall_NARGS (C function), 126
PyWeakref_Check (C function), 238
PyWeakref_CheckProxy (C function), 238
PyWeakref_CheckRef (C function), 238
PyWeakref_GET_OBJECT (C function), 239
PyWeakref_GetObject (C function), 239
PyWeakref_NewProxy (C function), 238
PyWeakref_NewRef (C function), 238
PyWideStringList (C type), 290
PyWideStringList_Append (C function), 290
PyWideStringList_Insert (C function), 290
PyWideStringList.items (C member), 291
PyWideStringList.length (C member), 291
PyWrapper_New (C function), 235

Q

qualified name, **418**

R

READ_RESTRICTED (C macro), 342
READONLY (C macro), 342
realloc (C function), 319
reference count, **418**
regular package, **419**
releasebufferproc (C type), 390
repr
    組み込み関数, 120, 359
reprfunc (C type), 389
RESTRICTED (C macro), 342
richcmpfunc (C type), 389

S

sdterr
    stdin stdout, 262
sendfunc (*C type*), 390
sequence, **419**
    object, 170
set comprehension, **419**
set_all(), 11
setattrfunc (*C type*), 389
setattrofunc (*C type*), 389
setswitchinterval (*in module sys*), 267
setter (*C type*), 345
SIGINT (*C macro*), 73, 74
signal
    module, 73, 74
single dispatch, **419**
SIZE_MAX (*C macro*), 162
slice, **419**
soft deprecated, **419**
special method, **420**
ssizeargfunc (*C type*), 390
ssizeobjargproc (*C type*), 390
statement, **420**
static type checker, **420**
stderr (*sys* モジュール), 279, 280
stdin
    stdout sdterr, 262
stdin (*sys* モジュール), 279, 280
stdout
    sdterr, stdin, 262
stdout (*sys* モジュール), 279, 280
strerror (*C function*), 65
string
    PyObject_Str (*C* 関数), 120
strong reference, **420**
structmember.h, 345
sum_list(), 12
sum_sequence(), 12, 13
sys
    module, 15, 260, 279, 280
SystemError (組み込み例外), 223, 224

T

T_BOOL (*C macro*), 345
T_BYTE (*C macro*), 345
T_CHAR (*C macro*), 345
T_DOUBLE (*C macro*), 345
T_FLOAT (*C macro*), 345
T_INT (*C macro*), 345
T_LONG (*C macro*), 345
T_LONGLONG (*C macro*), 345
T_NONE (*C macro*), 345
T_OBJECT (*C macro*), 345
T_OBJECT_EX (*C macro*), 345
T_PYSSIZET (*C macro*), 345
T_SHORT (*C macro*), 345
T_STRING (*C macro*), 345
T_STRING_INPLACE (*C macro*), 345
T_UBYTE (*C macro*), 345
T_UINT (*C macro*), 345
T_ULONG (*C macro*), 345
T_ULONGULONG (*C macro*), 345
T_USHORT (*C macro*), 345
ternaryfunc (*C type*), 390
text encoding, **420**
text file, **420**
traverseproc (*C type*), 396
triple-quoted string, **421**
tuple
    object, 198
    組み込み関数, 136, 204
type, **421**
    object, 8, 151
    組み込み関数, 122
type alias, **421**
type hint, **421**

U

ULONG_MAX (*C macro*), 162
unaryfunc (*C type*), 390
universal newlines, **421**
USE_STACKCHECK (*C macro*), 85

V

variable annotation, **421**
凍結ユーティリティ, 94
vectorcallfunc (*C type*), 125
version (*in module sys*), 265, 266
virtual environment, **422**
virtual machine, **422**
visitproc (*C type*), 396
属性, **403**
引数 (*argument*), **402**

W

数値
    object, 159
整数
    object, 159
検索
    パス, module, 15, 260, 264
永続オブジェクト (*immortal*), **410**
浮動小数点
    object, 165
浮動小数点数
    組み込み関数, 134
WRITE_RESTRICTED (*C macro*), 342

Z

Zen of Python, **422**