
The Python/C API

リリース 3.8.20

Guido van Rossum
and the Python development team

9 月 08, 2024

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	4
1.3	便利なマクロ	5
1.4	オブジェクト、型および参照カウント	6
1.5	例外	11
1.6	Python の埋め込み	13
1.7	デバッグ版ビルド (Debugging Builds)	14
第 2 章	安定 ABI (Stable Application Binary Interface)	17
第 3 章	超高水準レイヤ	19
第 4 章	参照カウント	25
第 5 章	例外処理	27
5.1	出力とクリア	28
5.2	例外の送出	28
5.3	警告	31
5.4	エラーインジケータの問い合わせ	32
5.5	シグナルハンドリング	35
5.6	例外クラス	35
5.7	例外オブジェクト	36
5.8	Unicode 例外オブジェクト	36
5.9	再帰の管理	38
5.10	標準例外	39
5.11	標準警告カテゴリ	41
第 6 章	ユーティリティ	43
6.1	オペレーティングシステム関連のユーティリティ	43
6.2	システム関数	46
6.3	プロセス制御	49
6.4	モジュールのインポート	49
6.5	データ整列化 (data marshallng) のサポート	54

6.6	引数の解釈と値の構築	55
6.7	文字列の変換と書式化	66
6.8	リフレクション	68
6.9	codec レジストリとサポート関数	68
第 7 章	抽象オブジェクトレイヤ (abstract objects layer)	71
7.1	オブジェクトプロトコル (object protocol)	71
7.2	数値型プロトコル (number protocol)	78
7.3	シーケンス型プロトコル (sequence protocol)	82
7.4	マップ型プロトコル (mapping protocol)	85
7.5	イテレータプロトコル (iterator protocol)	86
7.6	バッファプロトコル (buffer Protocol)	87
7.7	古いバッファプロトコル	95
第 8 章	具象オブジェクト (concrete object) レイヤ	97
8.1	基本オブジェクト (fundamental object)	97
8.2	数値型オブジェクト (numeric object)	101
8.3	シーケンスオブジェクト (sequence object)	108
8.4	Container オブジェクト	143
8.5	Function オブジェクト	149
8.6	その他のオブジェクト	154
第 9 章	初期化 (initialization)、終了処理 (finalization)、スレッド	179
9.1	Python 初期化以前	179
9.2	グローバルな設定変数	180
9.3	インタプリタの初期化と終了処理	183
9.4	プロセスワイドのパラメータ	184
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)	188
9.6	サブインタプリタサポート	196
9.7	非同期通知	198
9.8	プロファイルとトレース (profiling and tracing)	199
9.9	高度なデバッガサポート (advanced debugger support)	200
9.10	スレッドローカルストレージのサポート	201
第 10 章	Python 初期化設定	205
10.1	PyWideStringList	206
10.2	PyStatus	207
10.3	PyPreConfig	208
10.4	Preinitialization with PyPreConfig	210
10.5	PyConfig	211
10.6	Initialization with PyConfig	216
10.7	Isolated Configuration	218
10.8	Python Configuration	218
10.9	Path Configuration	219
10.10	Py_RunMain()	221

10.11 Multi-Phase Initialization Private Provisional API	221
第 11 章 メモリ管理	225
11.1 概要	225
11.2 生メモリインタフェース	226
11.3 メモリインタフェース	227
11.4 オブジェクトアロケータ	229
11.5 Default Memory Allocators	230
11.6 メモリアロケータをカスタマイズする	230
11.7 pymalloc アロケータ	232
11.8 tracemalloc C API	233
11.9 使用例	234
第 12 章 オブジェクト実装サポート (object implementation support)	235
12.1 オブジェクトをヒープ上にメモリ確保する	235
12.2 共通のオブジェクト構造体 (common object structure)	236
12.3 型オブジェクト	241
12.4 数値オブジェクト構造体	273
12.5 マップオブジェクト構造体	276
12.6 シーケンスオブジェクト構造体	277
12.7 バッファオブジェクト構造体 (buffer object structure)	278
12.8 async オブジェクト構造体	279
12.9 Slot Type typedefs	280
12.10 使用例	282
12.11 循環参照ガベージコレクションをサポートする	285
第 13 章 API と ABI のバージョンニング	289
付録 A 章 用語集	291
付録 B 章 このドキュメントについて	309
B.1 Python ドキュメント 貢献者	309
付録 C 章 歴史とライセンス	311
C.1 Python の歴史	311
C.2 Terms and conditions for accessing or otherwise using Python	312
C.3 Licenses and Acknowledgements for Incorporated Software	316
付録 D 章 Copyright	331
索引	333
索引	333

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。

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++ からで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は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。

`Py_UNREACHABLE()`

到達するはずがないと考えているコードパスがあるときに使います。例えば、`case` 節が全ての起こり得る値をカバーしている `switch` 文の `default:` 節です。 `assert(0)` や `abort()` の呼び出しを置きたくなる場所で使ってください。

バージョン 3.7 で追加.

`Py_ABS(x)`

`x` の絶対値を返します。

バージョン 3.3 で追加.

`Py_MIN(x, y)`

`x` と `y` の最小値を返します。

バージョン 3.3 で追加.

`Py_MAX(x, y)`

`x` と `y` の最大値を返します。

バージョン 3.3 で追加.

`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。

バージョン 3.4 で追加.

`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。

バージョン 3.6 で追加.

`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_GETENV(s)`

`getenv(s)` に似ていますが、コマンドラインで `-E` が渡された場合 (つまり `Py_IgnoreEnvironmentFlag` が設定された場合) `NULL` を返します。

`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; }.`

バージョン 3.4 で追加.

`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。

`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 sqlite_row_methods[] = {
    {"keys", (PyCFunction)sqlite_row_keys, METH_NOARGS,
     PyDoc_STR("Returns the keys of the row.")},
    {NULL, NULL}
};
```

1.4 オブジェクト、型および参照カウント

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

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

今日の計算機は有限の (しばしば非常に限られた) メモリサイズしか持たないので、参照カウントは重要な概念です; 参照カウントは、あるオブジェクトに対して参照を行っている場所が何箇所あるかを数える値です。参照を行っている場所とは、別のオブジェクトであったり、グローバルな (あるいは静的な) C 変数であったり、何らかの C 関数内にあるローカルな変数だったりします。あるオブジェクトの参照カウントがゼロになると、そのオブジェクトは解放されます。そのオブジェクトに他のオブジェクトへの参照が入っていれば、他のオブジェクトの参照カウントはデクリメントされます。デクリメントの結果、他のオブジェクトの参照カウントがゼロになると、今度はそのオブジェクトが解放される、といった具合に以後続きます。(言うまでもなく、互いを参照しあうオブジェクトについて問題があります; 現状では、解決策は " 何もしない " です。)

参照カウントは、常に明示的なやり方で操作されます。通常の方法では、`Py_INCREF()` マクロでオブジェクトの参照を 1 インクリメントし、`Py_DECREF()` マクロで 1 デクリメントします。`Py_DECREF()` マクロは、`incrcf` よりもかなり複雑です。というのは、`Py_DECREF` マクロは参照カウントがゼロになったかどうかを調べて、なった場合にはオブジェクトのメモリ解放関数 (`deallocator`) を呼び出さなければならないからです。メモリ解放関数とは、オブジェクトの型を定義している構造体内にある関数へのポインタです。型固有のメモリ解放関数は、その型が複合オブジェクト (`compound object`) 型である場合には、オブジェクト内の他のオブジェクトに対する参照カウントをデクリメントするよう気を配るとともに、その他の必要なファイナライズ (`finalize`) 処理を実行します。参照カウントがオーバフローすることはありません; というのも、仮想メモリ空間には、`(sizeof(Py_ssize_t) >= sizeof(void*))` と仮定した場合) 少なくとも参照カウントの記憶に使用されるビット数と同じだけのメモリ上の位置があるからです。従って、参照カウントのインクリメントは単純な操作になります。

オブジェクトへのポインタが入っているローカルな変数全てについて、オブジェクトの参照カウントを必ずインクリメントしなければならないわけではありません。理論上は、オブジェクトの参照カウントは、オブジェクトを指し示す変数が生成されたときに 1 増やされ、その変数がスコープから出て行った際に 1 減らされます。しかしこの場合、二つの操作は互いに相殺するので、結果的に参照カウントは変化しません。参照カウントを使う真の意義とは、手持ちの何らかの変数がオブジェクトを指している間はオブジェクトがデアロケートされないようにすることにあります。オブジェクトに対して、一つでも別の参照が行われていて、その参照が手持ちの変数と同じ間維持されるのなら、参照カウントを一時的に増やす必要はありません。参照カウント操作の必要性が浮き彫りになる重要な局面とは、Python から呼び出された拡張モジュール内の C 関数にオブジェクトを引数として渡すときです; 呼び出しメカニズムは、呼び出しの間全ての引数に対する参照を保証します。

しかしながら、よく陥る過ちとして、あるオブジェクトをリストから得たときに、参照カウントをインクリメントせずにしばらく放っておくということがあります。他の操作がオブジェクトをリストから除去してしまい、参照カウントがデクリメントされてデアロケートされてしまうことが考えられます。本当に危険なのは、まったく無害そうに見える操作が、上記の動作を引き起こす何らかの Python コードを呼び出しかねないというこ

とです; `Py_DECREF()` からユーザへ制御を戻せるようなコードパスが存在するため、ほとんど全ての操作が潜在的に危険をはらむことになります。

安全に参照カウントを操作するアプローチは、汎用の操作 (関数名が `PyObject_`, `PyNumber_`, `PySequence_`, および `PyMapping_` で始まる関数) の利用です。これらの操作は常に戻り値となるオブジェクトの参照カウントをインクリメントします。ユーザには戻り値が不要になったら `Py_DECREF()` を呼ぶ責任が残されています; とはいえ、すぐにその習慣は身に付くでしょう。

参照カウントの詳細

Python/C API の各関数における参照カウントの振る舞いは、説明するには、**参照の所有権** (*ownership of references*) という言葉でうまく説明できます。所有権は参照に対するもので、オブジェクトに対するものではありません (オブジェクトは誰にも所有されず、常に共有されています)。ある "参照の所有" は、その参照が必要なくなった時点で `Py_DECREF` を呼び出す役割を担うことを意味します。所有権は委譲でき、あるコードが委譲によって所有権を得ると、今度はそのコードが参照が必要なくなった際に最終的に `Py_DECREF()` や `Py_XDECREF()` を呼び出して、参照カウンタを 1 つ減らす役割を担います --- あるいは、その役割を (通常はコードを呼び出した元に) 受け渡します。ある関数が、関数の呼び出し側に対して参照の所有権を渡すと、呼び出し側は **新たな参照** (new reference) を得る、と言います。所有権が渡されない場合、呼び出し側は参照を **借りる** (borrow) と言います。借りた参照に対しては、何もする必要はありません。

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);
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");
```

自作の関数に渡す引数のように、単に参照を借りるだけの要素に対しては、`PyObject_SetItem()` とその仲間を使うのがはるかに一般的です。その場合、参照カウントをインクリメントする必要がなく、参照を引き渡せる (”参照を盗み取らせられる”) ので、参照カウントに関する動作はより健全になります。例えば、以下の関数は与えられた要素をリスト中の全ての要素の値にセットします:

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

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

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;
}
```

```
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

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

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、もし送出的れば、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。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。

例外を検出して渡す例は、上の `sum_sequence()` で示しています。偶然にも、この例ではエラーを検出した際に何ら参照を放棄する必要がありません。以下の関数の例では、エラーに対する後始末について示しています。まず、どうして Python で書くのが好きか思い出してもらうために、等価な Python コードを示します:

```
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 */
}
```

(次のページに続く)

(前のページからの続き)

```

    /* Continue with cleanup code */

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

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

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

```

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) が気かけなければならぬ重要なタスクは、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()` の中では、" スクリプトへの引数リスト " (script argument list, `sys.argv` のこと) を設定しません。この変数が後に実行される Python コード中で必要なら、`Py_Initialize()` の後で `PySys_SetArgvEx(argc, argv, updatepath)` を呼び出して明示的に設定しなければなりません。

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/pythonX.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、`Py_Initialize()` はライブラリが `/usr/local/lib/pythonX.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。

埋め込みを行うアプリケーションでは、`Py_Initialize()` を呼び出す 前に `Py_SetProgramName(file)` を呼び出すことで、上記の検索を操作できます。この埋め込みアプリケーションでの設定は依然として `PYTHONHOME` でオーバーライドでき、標準のモジュール検索パスの前には以前として `PYTHONPATH` が挿入されるので注意してください。アプリケーションでモジュール検索パスを完全に制御したいのなら、独自に `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, および `Py_GetProgramFullPath()` の実装を提供しなければなりません (これらは全て `Modules/getpath.c` で定義されています)。

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트의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。

インタプリタを `Py_DEBUG` マクロを有効にしてコンパイルすると、一般的に「デバッグビルド」といわれる Python ができます。Unix では、`./configure` コマンドに `--with-pydebug` を追加することで、`Py_DEBUG` が有効になります。その場合、暗黙的に Python 専用ではない `_DEBUG` も有効になります。Unix ビルドでは、`Py_DEBUG` が有効な場合、コンパイラの最適化が無効になります。

あとで説明する参照カウンタデバッグの他に、以下の追加チェックも有効になります:

- object allocator に対する追加チェック。
- パーサーとコンパイラに対する追加チェック。
- 情報損失のために、大きい型から小さい型へのダウンキャストに対するチェック。
- 辞書 (dict) と集合 (set) の実装に対する、いくつかの assertion の追加。加えて、集合オブジェクトに `test_c_api()` メソッドが追加されます。
- フレームを作成する時の、引数の健全性チェック。

- 初期化されていない数に対する参照を検出するために、整数のストレージが特定の妥当でないパターンで初期化されます。
- 低レベルトレースと追加例外チェックが VM runtime に追加されます。
- メモリアリーナ (memory arena) の実装に対する追加チェック。
- thread モジュールに対する追加デバッグ機能。

ここで言及されていない追加チェックもあるでしょう。

`Py_TRACE_REFS` を宣言すると、参照トレースが有効になります。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) `Py_TRACE_REFS` は `Py_DEBUG` によって暗黙的に有効になります。

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。

安定 ABI (STABLE APPLICATION BINARY INTERFACE)

伝統的に Python の C API はリリース毎に変更されます。多くの変更はソース互換性を保っていて、既存の API を変更したり取り除いたりすることはありません (ただし、いくつかの API は、一旦非推奨と指定された後に、削除されます)。

残念ながら API の互換性はバイナリレベルの互換性 (ABI) までには適用されません。その理由は主に、構造体フィールドの新規追加や型の変更によって、たとえ API は壊れなくても ABI は壊れてしまうからです。その結果として、拡張モジュールを Python のリリース毎に再コンパイルする必要があります (Unix でその影響を受けるインターフェイスが使用されていない場合は例外かもしれませんが)。また、Windows では、何らかの pythonXY.dll とリンクしている拡張モジュールは、再コンパイルした後に新しい dll とリンクし直す必要があります。

Python 3.2 以降では、安定 ABI を保証するための API サブセットが宣言されています。拡張モジュールでこの API ("limited API" と呼ばれます) を使うには `Py_LIMITED_API` を定義してください。拡張モジュールの細部は大部分隠蔽され、代わりに再コンパイルなしにバージョン 3.x (x>=2) 上で動くモジュールがビルドされます。

いくつかのケースでは、安定 ABI を新しい関数で拡張する必要があります。これらの新しい API を使用したい拡張モジュールは、`Py_LIMITED_API` にサポートしたい最小の Python バージョンの `PY_VERSION_HEX` の値 ([API と ABI のバージョンング](#) を参照) を設定してください。 (例えば Python3.3 なら `0x03030000`)。モジュールは後続のすべての Python リリースで動作しますが、(シンボルが存在しないため) 古いリリースでは動作しません。

Python 3.2 からは limited API は [PEP 384](#) に文書化されています。C API のドキュメントでは、limited API の一部でない API は、"Not part of the limited API" とマークされています。

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。

これらの関数のいくつかは引数として文法の開始記号を受け取ります。使用できる開始記号は `Py_eval_input` と `Py_file_input`、`Py_single_input` です。開始記号の説明はこれらを引数として取る関数の後にあります。

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)`

標準インタプリタのためのメインプログラム。Python を組み込むプログラムのためにこれを利用できるようにしています。`argc` と `argv` 引数を C プログラムの `main()` 関数 (ユーザのロケールに従って `wchar_t` に変換されます) へ渡されるものとまったく同じに作成すべきです。引数リストが変更される可能性があるという点に注意することは重要です。(しかし、引数リストが指している文字列の内容は変更されません)。戻り値はインタプリタが (例外などではなく) 普通に終了した時は 0 に、例外で終了したときには 1 に、引数リストが正しい Python コマンドラインが渡されなかったときは 2 になります。

`Py_InspectFlag` が設定されていない場合、未処理の `SystemExit` 例外が発生すると、この関数は 1 を返すのではなくプロセスを `exit` することに気を付けてください。

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

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

バージョン 3.8 で追加。

`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。

int `PyRun_SimpleString(const char *command)`

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

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

`__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。

`Py_InspectFlag` が設定されていない場合、未処理の `SystemExit` 例外が発生すると、この関数は 1 を返すのではなくプロセスを `exit` することに気を付けてください。

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 (`sys.getfilesystemencoding()`) 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の一つ読み込み、`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 (`sys.getfilesystemencoding()`) 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 (sys.getfilesystemencoding()) 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)
```

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。

```
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。

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

下記の `PyParser_SimpleParseStringFlagsFilename()` の `filename` を NULL に、`flags` を 0 にして単純化したインタフェースです。

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

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

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

開始トークン start を使って str に含まれる Python ソースコードを flags 引数に従ってパースします。効率的に評価可能なコードオブジェクトを作成するためにその結果を使うことができます。コード断片を何度も評価しなければならない場合に役に立ちます。filename はファイルシステムエンコーディング (sys.getfilesystemencoding()) でデコードされます。

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

下記の PyRun_SimpleParseFileFlags() の flags を 0 にして単純化したインタフェースです。

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

PyParser_SimpleParseStringFlagsFilename() に似ていますが、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りません。

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

Return value: New reference. 下記の PyRun_StringFlags() の flags を NULL にして単純化したインタフェースです。

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

Return value: New reference. 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 *lo-
cals)

Return value: New reference. 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フェースです。

PyObject* PyRun_FileEx(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *lo-
cals, int closeit)

Return value: New reference. 下記の PyRun_FileExFlags() の flags を NULL にして単純化したインタフェースです。

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

Return value: New reference. 下記の PyRun_FileExFlags() の closeit を 0 にして単純化したインタフェースです。

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

Return value: New reference. 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 (sys.getfilesystemencoding()) でデコードされます。もし closeit を真にすると、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)

Return value: New reference. 下記の Py_CompileStringFlags() の flags を NULL にして単純化したインタフェースです。

PyObject* Py_CompileStringFlags(const char *str, const char *filename, int start, PyCompiler-
Flags *flags)

Return value: New reference. 下記の Py_CompileStringExFlags() の optimize を -1 にして単純

化したインタフェースです。

*PyObject** **Py_CompileStringObject**(const char *str, *PyObject* *filename, int start, *PyCompiler-Flags* *flags, int optimize)

Return value: New reference. str 内の Python ソースコードをパースしてコンパイルし、作られたコードオブジェクトを返します。開始トークンは start によって与えられます。これはコンパイル可能なコードを制限するために使うことができ、Py_eval_input、Py_file_input もしくは Py_single_input であるべきです。filename で指定されるファイル名はコードオブジェクトを構築するために使われ、トレースバックあるいは SyntaxError 例外メッセージに出てくる可能性があります。コードがパースできなかったりコンパイルできなかったりした場合に、これは NULL を返します。

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。

バージョン 3.4 で追加。

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

Return value: New reference. *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でデコード (os.fsdecode()) されたバイト文字列です。

バージョン 3.2 で追加。

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

Return value: New reference. *PyEval_EvalCodeEx()* のシンプルなインタフェースで、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)

Return value: New reference. 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。

PyFrameObject

フレームオブジェクトを表現するために使われるオブジェクトの C 構造体。この型のフィールドはいつでも変更され得ます。

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

Return value: New reference. 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)

Return value: New reference. Python のインタプリタの主要な、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のように、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を使う記号です。

`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。

`PyCompilerFlags *flags` が `NULL` の場合、`cf_flags` は 0 として扱われ、`from __future__ import` による変更は無視されます。

`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。

参照カウント

この節のマクロは Python オブジェクトの参照カウントを管理するために使われます。

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

オブジェクト *o* に対する参照カウントを一つ増やします。オブジェクトが NULL であってははいけません。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。

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

オブジェクト *o* に対する参照カウントを一つ増やします。オブジェクトが NULL であってもよく、その場合マクロは何の影響も与えません。

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

オブジェクト *o* に対する参照カウントを一つ減らします。オブジェクトが NULL であってははいけません。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。参照カウントがゼロになったら、オブジェクトの型のメモリ解放関数 (NULL であってはならない) が呼ばれます。

警告: (例えば `__del__()` メソッドをもつクラスインスタンスがメモリ解放されたときに) メモリ解放関数は任意の Python コードを呼び出すことができます。このようなコードでは例外は伝播しませんが、実行されたコードはすべての Python グローバル変数に自由にアクセスできます。これが意味するのは、**Py_DECREF()** が呼び出されるより前では、グローバル変数から到達可能などんなオブジェクトも一貫した状態にあるべきであるということです。例えば、リストからオブジェクトを削除するコードは削除するオブジェクトへの参照を一時変数にコピーし、リストデータ構造を更新し、それから一時変数に対して **Py_DECREF()** を呼び出すべきです。

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

オブジェクト *o* への参照カウントを一つ減らします。オブジェクトは NULL でもかまいませんが、その場合マクロは何の影響も与えません。それ以外の場合、結果は **Py_DECREF()** と同じです。また、注意すべきことも同じです。

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

o の参照カウントを減らします。オブジェクトは NULL でもよく、その場合このマクロは何も行いません。オブジェクトが NULL でなければ、引数を NULL にした **Py_DECREF()** と同じ効果をもたらします。このマクロは一時変数を使って、参照カウントをデクリメントする前に引数を NULL にセットしてくれるので、**Py_DECREF()** に使うときの警告を気にしなくて済みます。

ガベージコレクション中に追跡される可能性のあるオブジェクトの参照カウン트의デクリメントを行うには、このマクロを使うのがよいでしょう。

以下の関数: `Py_IncRef(PyObject *o)`, `Py_DecRef(PyObject *o)`, は、実行時の動的な Python 埋め込みで使われる関数です。これらの関数はそれぞれ `Py_XINCREF()` および `Py_XDECREF()` をエクスポートしただけです。

以下の関数やマクロ: `_Py_Dealloc()`, `_Py_ForgetReference()`, `_Py_NewReference()` は、インタプリタのコアの内部においてのみ使用するためのものです。また、グローバル変数 `_Py_RefTotal` も同様です。

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っ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： エラー識別子は `sys.exc_info()` の結果 **ではありません**。エラー識別子はまだ捕捉されていない例外（したがってまだ伝播します）に対応しているのに対し、`sys.exc_info()` の結果は捕捉された後の例外を返します（したがってもう伝播しません）。

5.1 出力とクリア

`void PyErr_Clear()`

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)`

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!

`set_sys_last_vars` が非ゼロであれば、`sys.last_type`, `sys.last_value`, `sys.last_traceback` 変数が、表示される例外のタイプ、値、トレースバックそれぞれに反映されます。

`void PyErr_Print()`

`PyErr_PrintEx(1)` のエイリアスです。

`void PyErr_WriteUnraisable(PyObject *obj)`

現在の例外と `obj` 引数で `sys.unraisablehook()` を呼び出します。

例外が設定されているがインタプリタが実際に例外を発生させることができないときに、このユーティリティ関数は警告メッセージを `sys.stderr` へ出力します。例えば、例外が `__del__()` メソッドで発生したときに使われます。

発生させられない例外が起きたコンテキストを指し示す単一の引数 `obj` で関数が呼び出されます。可能な場合は、`obj` の `repr` 文字列が警告メッセージに出力されます。

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。

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)`

これはエラーインジケータを設定するための最も一般的な方法です。第一引数は例外の型を指定します。通常は標準例外の一つ、例えば `PyExc_RuntimeError` です。その参照カウントを増加させる必要はありません。第二引数はエラーメッセージで、`'utf-8'` からデコードされます。

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

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, ...)`

Return value: Always `NULL`. 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 vars)`

Return value: Always NULL. `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。

バージョン 3.5 で追加。

`void PyErr_SetNone(PyObject *type)`

これは `PyErr_SetObject(type, Py_None)` を省略したものです。

`int PyErr_BadArgument()`

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()`

Return value: Always NULL. 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)`

Return value: Always NULL. C ライブラリ関数がエラーを返して C 変数 `errno` を設定したときに、これは例外を発生させるために便利な関数です。第一要素が整数 `errno` 値で、第二要素が (`strerror()` から得られる) 対応するエラーメッセージであるタプルオブジェクトを構成します。それから、`PyErr_SetObject(type, object)` を呼び出します。Unix では、`errno` 値が `EINTR` であるとき、すなわち割り込まれたシステムコールを表しているとき、これは `PyErr_CheckSignals()` を呼び出し、それがエラーインジケータを設定した場合は設定されたままにしておきます。関数は常に NULL を返します。したがって、システムコールがエラーを返したとき、システムコールのラッパー関数は `return PyErr_SetFromErrno(type);` と書くことができます。

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

Return value: Always NULL. `PyErr_SetFromErrno()` に似ていますが、`filenameObject` が NULL でない場合に、`type` のコンストラクタに第三引数として渡すというふりまが追加されています。OSError 例外の場合では、`filenameObject` が例外インスタンスの `filename` 属性を定義するのに使われます。

*PyObject** `PyErr_SetFromErrnoWithFilenameObjects(PyObject *type, PyObject *filenameObject1, PyObject *filenameObject2)`

Return value: Always NULL. `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。

バージョン 3.4 で追加。

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

Return value: Always NULL. `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 (`os.fsdecode()`) でデコードされます。

*PyObject** `PyErr_SetFromWindowsErr(int ierr)`

Return value: Always NULL. これは `WindowsError` を発生させるために便利な関数です。0 の `ierr` とともに呼び出された場合、`GetLastError()` が返すエラーコードが代りに使われます。 `ierr` あるいは `GetLastError()` によって与えられるエラーコードの Windows 用の説明を取り出すために、Win32 関数 `FormatMessage()` を呼び出します。それから、第一要素が `ierr` 値で第二要素が (`FormatMessage()` から得られる) 対応するエラーメッセージであるタプルオブジェクトを構成します。そして、`PyErr_SetObject(PyExc_WindowsError, object)` を呼び出します。この関数は常に NULL を返します。

利用可能な環境: Windows。

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

Return value: Always NULL. `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。

利用可能な環境: Windows。

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

Return value: Always NULL. `PyErr_SetFromWindowsErrWithFilenameObject()` 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 (`os.fsdecode()`) でデコードされます。

利用可能な環境: Windows。

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

Return value: Always NULL. `PyErr_SetFromWindowsErrWithFilenameObject()` 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。

利用可能な環境: Windows。

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

Return value: Always NULL. `PyErr_SetExcFromWindowsErrWithFilenameObject()` に似ていますが、2 つ目のファイル名オブジェクトを受け取ります。

利用可能な環境: Windows。

バージョン 3.4 で追加。

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

Return value: Always NULL. `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。

利用可能な環境: Windows。

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

Return value: Always NULL. 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。

バージョン 3.3 で追加。

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。

バージョン 3.4 で追加。

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

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でデコードされた (`os.fsdecode()`) バイト文字列です。

バージョン 3.2 で追加。

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

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**()

`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)

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。stack_level 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。stack_level 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。

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

Return value: Always `NULL`. `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。

バージョン 3.6 で追加。

`int PyErr_WarnExplicit(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()` の直接的なラップで、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。

バージョン 3.4 で追加。

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

`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 (`os.fsdecode()`) でデコードされます。

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

`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。

バージョン 3.2 で追加。

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

`PyErr_WarnFormat()` に似た関数ですが、`category` は `ResourceWarning` になり、`source` は `warnings.WarningMessage()` に渡されます。

バージョン 3.6 で追加。

5.4 エラーインジケータの問い合わせ

*PyObject** `PyErr_Occurred()`

Return value: Borrowed reference. 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。

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)`

`PyErr_GivenExceptionMatches(PyErr_Occurred(), exc)` と同じ。例外が実際に設定されたときにだけ、これ呼び出すべきです。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)`

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。

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

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: 通常、この関数は例外を捕捉する必要のあるコードや、エラーインジケータを一時的に保存して復元する必要のあるコードでのみ使います。

```
{
    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)`

三つのオブジェクトからエラーインジケータを設定します。エラーインジケータがすでに設定されている場合は、最初にクリアされます。オブジェクトが NULL ならば、エラーインジケータがクリアされます。NULL の *type* と非 NULL の *value* あるいは *traceback* を渡してはいけません。例外の型 (*type*) はクラスであるべきです。無効な例外の型 (*type*) あるいは値 (*value*) を渡してはいけません。(これらの規則を破ると後で気付にくい問題の原因となるでしょう。) この呼び出しはそれぞれのオブジェクトへの参照を取り除きます: あなたは呼び出しの前にそれぞれのオブジェクトへの参照を持たなければならないのであり、また呼び出しの後にはもはやこれらの参照を持っていません。(これを理解していない場合は、この関数を使ってはいけません。注意しておきます。)

注釈: 通常、この関数はエラーインジケータを一時的に保存し復元する必要のあるコードでのみ使います。現在のエラーインジケータを保存するためには `PyErr_Fetch()` を使ってください。

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

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: この関数は例外値に暗黙的に `__traceback__` 属性を設定 **しません**。トレースバックを適切に設定する必要がある場合は、次の追加のコード片が必要です:

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

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

`sys.exc_info()` で得られる例外情報を取得します。これは **既に捕まえた** 例外を参照するもので、新たに送出された例外への参照は持っていません。新しい3つのオブジェクトへの参照を返しますが、その中には `NULL` があるかもしれません。この関数は例外情報の状態を変更しません。

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。

バージョン 3.3 で追加.

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

`sys.exc_info()` で得られる例外情報を設定します。これは **既に捕まえた** 例外を参照するもので、新たに送出された例外への参照は持っていません。この関数は引数への参照を盗みます。例外の状態をクリアしたい場合は、3つ全ての引数に `NULL` を渡してください。3つの引数についての一般的な規則は、`PyErr_Restore()` を参照してください。

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。

バージョン 3.3 で追加.

5.5 シグナルハンドリング

`int PyErr_CheckSignals()`

この関数は Python のシグナル処理とやりとりすることができます。シグナルがそのプロセスへ送られたかどうかチェックし、そうならば対応するシグナルハンドラを呼び出します。`signal` モジュールがサポートされている場合は、これは Python で書かれたシグナルハンドラを呼び出せます。すべての場合で、SIGINT のデフォルトの効果は `KeyboardInterrupt` 例外を発生させることです。例外が発生した場合、エラーインジケータが設定され、関数は -1 を返します。そうでなければ、関数は 0 を返します。エラーインジケータが以前に設定されている場合は、それがクリアされるかどうか分からない。

`void PyErr_SetInterrupt()`

SIGINT 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、SIGINT 用の Python のシグナルハンドラが呼び出されます。

SIGINT が Python に対処されなかった (`signal.SIG_DFL` または `signal.SIG_IGN` に設定されていた) 場合、この関数は何もしません。

`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)`

Return value: New reference. 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。

新しいクラスの `__module__` 属性は `name` 引数の前半部分 (最後のドットまで) に設定され、クラス名は後半部分 (最後のドットの後) に設定されます。`base` 引数は代わりのベースクラスを指定するために使えます; 一つのクラスでも、クラスのタプルでも構いません。`dict` 引数はクラス変数とメソッドの辞書を指定するために使えます。

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

Return value: New reference. `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。

バージョン 3.2 で追加。

5.7 例外オブジェクト

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

Return value: New reference. 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*)

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*)

Return value: New reference. Python で `__context__` 属性からアクセスできるものと同じ、例外に関するコンテキスト (*ex* が送出されたときに処理していた別の例外インスタンス) の新しい参照を返します。関係するコンテキストが無い場合は、NULL を返します。

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

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*)

Return value: New reference. Python で `__cause__` 属性からアクセスできるものと同じ、例外に関する原因 (`raise ... from ...` によってセットされる例外インスタンス、もしくは None) の新しい参照を返します。

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

例外に關係する原因に *cause* をセットします。クリアするには NULL を使用してください。 *cause* が例外インスタンスか None のどちらかであることを確かめる型チェックは行われません。これは *cause* への参照を盗みます。

この関数によって暗黙的に `__suppress_context__` に True がセットされます。

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*)

Return value: New reference. *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** **PyUnicodeEncodeError_Create**(const char **encoding*, const *Py_UNICODE* **object*, Py_ssize_t *length*, Py_ssize_t *start*, Py_ssize_t *end*, const char **reason*)

Return value: New reference. *encoding*, *object*, *length*, *start*, *end*, *reason* 属性を持った `UnicodeEncodeError` 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。

バージョン 3.3 で非推奨: 3.11

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

*PyObject** `PyUnicodeTranslateError_Create`(const *Py_UNICODE* **object*, *Py_ssize_t* *length*,
Py_ssize_t *start*, *Py_ssize_t* *end*, const
char **reason*)
Return value: New reference. *object*, *length*, *start*, *end*, *reason* 属性を持った `UnicodeTranslateError` オブジェクトを作成します。*reason* は UTF-8 エンコードされた文字列です。

バージョン 3.3 で非推奨: 3.11

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

*PyObject** `PyUnicodeDecodeError_GetEncoding`(*PyObject* **exc*)
*PyObject** `PyUnicodeEncodeError_GetEncoding`(*PyObject* **exc*)
Return value: New reference. 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*)
Return value: New reference. 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*)
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*)
渡された例外オブジェクトの *start* 属性を *start* 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` `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*)
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*)
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)

Return value: New reference. 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)

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

これら 2 つの関数は C レベルの再帰呼び出しを安全に実行する方法を、コアモジュールにも拡張モジュールにも提供します。再帰を使ったコードが必ずしも Python コードを実行するわけではない場合 (Python コードは再帰の深さを自動的に追跡します)、これらの関数が必要となります。

int Py_EnterRecursiveCall(const char *where)

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。

USE_STACKCHECK が定義されている場合、OS のスタックがオーバーフローがしたかどうかを *PyOS_CheckStack()* を使ってチェックします。もしオーバーフローしているなら、*MemoryError* をセットしゼロでない値を返します。

次にこの関数は再帰の上限に達していないかをチェックします。上限に達している場合、*RecursionError* をセットしゼロでない値を返します。そうでない場合はゼロを返します。

where は " in instance check" のような文字列にして、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。

void Py_LeaveRecursiveCall()

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。

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)

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)`

`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

All standard Python exceptions are available as global variables whose names are `PyExc_` followed by the Python exception name. These have the type `PyObject*`; they are all class objects. For completeness, here are all the variables:

C 名	Python 名	注釈
<code>PyExc_BaseException</code>	<code>BaseException</code>	(1)
<code>PyExc_Exception</code>	<code>Exception</code>	(1)
<code>PyExc_ArithmeticError</code>	<code>ArithmeticError</code>	(1)
<code>PyExc_AssertionError</code>	<code>AssertionError</code>	
<code>PyExc_AttributeError</code>	<code>AttributeError</code>	
<code>PyExc_BlockingIOError</code>	<code>BlockingIOError</code>	
<code>PyExc_BrokenPipeError</code>	<code>BrokenPipeError</code>	
<code>PyExc_BufferError</code>	<code>BufferError</code>	
<code>PyExc_ChildProcessError</code>	<code>ChildProcessError</code>	
<code>PyExc_ConnectionAbortedError</code>	<code>ConnectionAbortedError</code>	
<code>PyExc_ConnectionError</code>	<code>ConnectionError</code>	
<code>PyExc_ConnectionRefusedError</code>	<code>ConnectionRefusedError</code>	
<code>PyExc_ConnectionResetError</code>	<code>ConnectionResetError</code>	
<code>PyExc_EOFError</code>	<code>EOFError</code>	
<code>PyExc_FileExistsError</code>	<code>FileExistsError</code>	
<code>PyExc_FileNotFoundError</code>	<code>FileNotFoundError</code>	
<code>PyExc_FloatingPointError</code>	<code>FloatingPointError</code>	
<code>PyExc_GeneratorExit</code>	<code>GeneratorExit</code>	
<code>PyExc_ImportError</code>	<code>ImportError</code>	
<code>PyExc_IndentationError</code>	<code>IndentationError</code>	
<code>PyExc_IndexError</code>	<code>IndexError</code>	
<code>PyExc_InterruptedError</code>	<code>InterruptedError</code>	
<code>PyExc_IsADirectoryError</code>	<code>IsADirectoryError</code>	
<code>PyExc_KeyError</code>	<code>KeyError</code>	
<code>PyExc_KeyboardInterrupt</code>	<code>KeyboardInterrupt</code>	
<code>PyExc_LookupError</code>	<code>LookupError</code>	(1)
<code>PyExc_MemoryError</code>	<code>MemoryError</code>	
<code>PyExc_ModuleNotFoundError</code>	<code>ModuleNotFoundError</code>	

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_NameError	NameError	
PyExc_NotADirectoryError	NotADirectoryError	
PyExc_NotImplementedError	NotImplementedError	
PyExc_OSError	OSError	(1)
PyExc_OverflowError	OverflowError	
PyExc_PermissionError	PermissionError	
PyExc_ProcessLookupError	ProcessLookupError	
PyExc_RecursionError	RecursionError	
PyExc_ReferenceError	ReferenceError	(2)
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	

バージョン 3.3 で追加: 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。

バージョン 3.5 で追加: PyExc_StopAsyncIteration および PyExc_RecursionError 。

バージョン 3.6 で追加: 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	(3)

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:

- (1) これは別の標準例外のためのベースクラスです。
- (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。

5.11 標準警告カテゴリ

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

C 名	Python 名	注釈
PyExc_Warning	Warning	(1)
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	

バージョン 3.2 で追加: `PyExc_ResourceWarning`.

注釈:

- (1) 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。

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)

Return value: *New reference.* path のファイルシステム表現を返します。オブジェクトが str か bytes の場合は、その参照カウンタがインクリメントされます。オブジェクトが os.PathLike インタフェースを実装している場合、__fspath__() が呼び出され、その結果が str が bytes であればその値が返されます。さもなければ、TypeError が送出され、NULL が返されます。

バージョン 3.6 で追加。

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

filename という名前の標準 I/O ファイル fp が対話的 (interactive) であると考えられる場合に真 (非ゼロ) を返します。これは isatty(fileno(fp)) が真になるファイルの場合です。グローバルなフラグ Py_InteractiveFlag が真の場合には、filename ポインタが NULL か、名前が '<stdin>' または '???' のいずれかに等しい場合にも真を返します。

void PyOS_BeforeFork()

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().

バージョン 3.7 で追加。

void PyOS_AfterFork_Parent()

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()`.

バージョン 3.7 で追加.

void `PyOS_AfterFork_Child()`

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

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()`.

バージョン 3.7 で追加.

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()`

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()`

インタプリタがスタック空間を使い尽くしたときに真を返します。このチェック関数には信頼性がありますが、`USE_STACKCHECK` が定義されている場合 (現状では Microsoft Visual C++ コンパイラでビルドした Windows 版) にしか利用できません。 `USE_STACKCHECK` は自動的に定義されます; 自前のコードでこの定義を変更してはなりません。

`PyOS_sighandler_t` `PyOS_getsig(int i)`

シグナル *i* に対する現在のシグナルハンドラを返します。この関数は `sigaction()` または `signal()` のいずれかに対する薄いラップです。 `sigaction()` や `signal()` を直接呼び出してはなりません! `PyOS_sighandler_t` は `void (*)(int)` の typedef による別名です。

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

シグナル *i* に対する現在のシグナルハンドラを *h* に設定します; 以前のシグナルハンドラを返します。

この関数は `sigaction()` または `signal()` のいずれかに対する薄いラッパーです。`sigaction()` や `signal()` を直接呼び出してはなりません! `PyOS_sighandler_t` は `void (*)(int)` の typedef による別名です。

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

surrogateescape エラーハンドラ を使って、ロケールエンコーディングのバイト文字列をデコードします: 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ます。バイト列がサロゲート文字としてデコードされる場合は、そのまま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をエスケープします。

エンコーディング (優先度の高い順から):

- UTF-8 on macOS, Android, and VxWorks;
- UTF-8 on Windows if `Py_LegacyWindowsFSEncodingFlag` is zero;
- UTF-8 Python UTF-8 モードが有効な場合
- ASCII if the LC_CTYPE locale is "C", `nl_langinfo(CODESET)` returns the ASCII encoding (or an alias), and `mbstowcs()` and `wcstombs()` functions uses the ISO-8859-1 encoding.
- 現在のロケールエンコーディング

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。

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。

バージョン 3.5 で追加。

バージョン 3.7 で変更: この関数は、UTF-8 モードでは 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 `Py_LegacyWindowsFSEncodingFlag` is zero;

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

surrogateescape エラーハンドラ を使って、ワイドキャラクター文字列をロケールエンコーディングにエンコードします: 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。

エンコーディング（優先度の高い順から）：

- UTF-8 on macOS, Android, and VxWorks;
- UTF-8 on Windows if `Py_LegacyWindowsFSEncodingFlag` is zero;
- UTF-8 Python UTF-8 モードが有効な場合
- ASCII if the LC_CTYPE locale is "C", `nl_langinfo(CODESET)` returns the ASCII encoding (or an alias), and `mbstowcs()` and `wcstombs()` functions uses the ISO-8859-1 encoding.
- 現在のロケールエンコーディング

この関数は、Python UTF-8 モードでは UTF-8 エンコーディングを使います。

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.

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。

参考：

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

バージョン 3.5 で追加.

バージョン 3.7 で変更: この関数は、UTF-8 モードでは 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 `Py_LegacyWindowsFSEncodingFlag` 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)`

Return value: Borrowed reference. `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)`

`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()`

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

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

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

void **PySys_AddWarnOptionUnicode**(PyObject *unicode)

`sys.warnoptions` に *unicode* を追加します。

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

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

`sys.path` を *path* に含まれるパスの、リストオブジェクトに設定します。 *path* はプラットフォームの検索パスデリミタ (Unix では `:`, Windows では `;`) で区切られたパスのリストでなければなりません。

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

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, ...)

`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, ...)

`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。

バージョン 3.2 で追加。

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

`PySys_FormatStdout()` と同じですが、`sys.stderr` もしくは *stderr* に出力します。

バージョン 3.2 で追加。

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

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

バージョン 3.2 で追加。

PyObject ***PySys_GetXOptions**()

Return value: Borrowed reference. `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。

バージョン 3.2 で追加.

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.

バージョン 3.8 で追加.

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

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

引数無しで 監査イベント `sys.addaudithook` を送出します。

バージョン 3.8 で追加.

6.3 プロセス制御

`void Py_FatalError(const char *message)`

致命的エラーメッセージ (fatal error message) を出力してプロセスを強制終了 (kill) します。後始末処理は行われません。この関数は、Python インタプリタを使い続けるのが危険であるような状況が検出されたとき; 例えば、オブジェクト管理が崩壊していると思われるときにのみ、呼び出されるようにしなければなりません。Unix では、標準 C ライブラリ関数 `abort()` を呼び出して `core` を生成しようと試みます。

`void Py_Exit(int status)`

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)())`

`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)`

Return value: New reference. この関数は下で述べる `PyImport_ImportModuleEx()` を単純化したインタフェースで、`globals` および `locals` 引数を `NULL` のままにし、`level` を 0 にしたものです。`name` 引数にドットが含まれる場合 (あるパッケージのサブモジュールを指定している場合)、`fromlist` 引数がリスト `['*']` に追加され、戻り値がモジュールを含むトップレベルパッケージではなく名前つきモジュール (named module) になるようにします。(残念ながらこのやり方には、`name` が実際にはサブモジュールでなくサブパッケージを指定している場合、パッケージの `__all__` 変数に指定されているサブモジュールがロードされてしまうという副作用があります。) インポートされたモジュールへの新たな参照を返します。失敗した場合には例外をセットし、`NULL` を返します。インポートに失敗したモジュールは `sys.modules` に残りません。

この関数は常に絶対インポートを使用します。

`PyObject* PyImport_ImportModuleNoBlock(const char *name)`

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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。

バージョン 3.3 で追加。

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

Return value: New reference. *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)

Return value: New reference. 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を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)

Return value: New reference. 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)

Return value: Borrowed reference. モジュール名に対応するモジュールオブジェクトを返します。name 引数は package.module の形式でもかまいません。まずモジュール辞書に該当するモジュールがあるかどうか調べ、なければ新たなモジュールを生成してモジュール辞書に挿入します。失敗した場合には例外をセットして NULL を返します。

注釈: この関数はモジュールのインポートやロードを行いません; モジュールがまだロードされていなければ、空のモジュールオブジェクトを得ることになります。 `PyImport_ImportModule()` やその別形式を使ってモジュールをインポートしてください。ドット名表記で指定した `name` が存在しない場合、パッケージ構造は作成されません。

バージョン 3.3 で追加。

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

Return value: Borrowed reference. `PyImport_AddModuleObject()` と似ていますが、`name` が UTF-8 でエンコードされた文字列ではなく Unicode オブジェクトを使用する点で異なります。

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

Return value: New reference. モジュール名 (`package.module` 形式でも構いません) および Python のバイトコードファイルや組み込み関数 `compile()` で得られたコードオブジェクトを元にモジュールをロードします。モジュールオブジェクトへの新たな参照を返します。失敗した場合には例外をセットし、NULL を返します。たとえ `PyImport_ExecCodeModule()` の処理に入った時に `name` が `sys.modules` に入っていたとしても、インポートに失敗したモジュールは `sys.modules` に残りません。初期化の不完全なモジュールを `sys.modules` に残すのは危険であり、そのようなモジュールをインポートするコードにとっては、モジュールの状態がわからない (モジュール作者の意図から外れた壊れた状態かもしれない) からです。

モジュールの `__spec__` と `__loader__` がまだ設定されていなければ、適切な値が設定されます。`spec` のローダーは、モジュールの `__loader__` が (もし設定されていれば) それに設定され、そうでなければ `SourceFileLoader` のインスタンスに設定されます。

モジュールの `__file__` 属性はコードオブジェクトの `co_filename` へ設定されます。もし適切な場合は、`__cached__` へも設定されます。

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。

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

Return value: New reference. `PyImport_ExecCodeModule()` と似ていますが、`pathname` が NULL でない場合にモジュールオブジェクトの `__file__` 属性に `pathname` が設定される点が異なります。

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

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

Return value: New reference. `PyImport_ExecCodeModuleEx()` と似ていますが、`cpathname` が NULL でない場合にモジュールオブジェクトの `__cached__` 属性に `cpathname` が設定される点が異なる

ります。これらの 3 つの関数のうち、この関数の使用が望ましいです。

バージョン 3.3 で追加。

*PyObject** **PyImport_ExecCodeModuleWithPathnames**(const char **name*, *PyObject* **co*, const char **pathname*, const char **cpathname*)
Return value: New reference. *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。

バージョン 3.2 で追加。

バージョン 3.3 で変更: バイトコードのパスが与えられた場合にのみ `imp.source_from_cache()` がソースパスの計算に使用されます。

long **PyImport_GetMagicNumber()**

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()**

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。

バージョン 3.2 で追加。

*PyObject** **PyImport_GetModuleDict()**

Return value: Borrowed reference. 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*)

Return value: New reference. 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。

バージョン 3.7 で追加。

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

Return value: New reference. `sys.path` もしくは `pkg.__path__` の要素である *path* を見付けるためのオブジェクトを返します。場合によっては `sys.path_importer_cache` 辞書から取得することもあります。もしまだオブジェクトがキャッシュされていなかった場合は、*path* 要素を扱えるフックが見付かるまで `sys.path_hooks` を走査します。どのフックも *path* 要素を扱えない場合は None を返します; これにより、*path based finder* がこの *path* 要素を見付けるためのオブジェクトが得られなかったことを呼び出し元に伝えます。最終的に得られたオブジェクトを `sys.path_importer_cache` へキャッシュし、オブジェクトへの新たな参照を返します。

void `_PyImport_Init()`

インポート機構を初期化します。内部使用だけのための関数です。

void `PyImport_Cleanup()`

モジュールテーブルを空にします。内部使用だけのための関数です。

void `_PyImport_Fini()`

インポート機構を終了処理します。内部使用だけのための関数です。

int `PyImport_ImportFrozenModuleObject(PyObject *name)`

Return value: New reference. `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。)

バージョン 3.3 で追加。

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)`

`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립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;
};
```

const struct `__frozen*` `PyImport_FrozenModules`

このポインタは `struct _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))`

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()` のラップ関数で、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`

組み込みモジュールリスト内の一つのエントリを記述している構造体です。リスト内の各構造体には、

インタプリタ内に組み込まれているモジュールの名前と初期化関数が指定されています。Python を埋め込むようなプログラムは、この構造体の配列と `PyImport_ExtendInittab()` を組み合わせて、追加の組み込みモジュールを提供できます。構造体は `Include/import.h` で以下のように定義されています:

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

int `PyImport_ExtendInittab`(struct `__inittab` *`newtab`)

組み込みモジュールのテーブルに一群のモジュールを追加します。配列 `newtab` は `name` フィールドが `NULL` になっているセンチネル (sentinel) エントリで終端されていなければなりません。センチネル値を与えられなかった場合にはメモリ違反になるかもしれません。成功すると 0 を、内部テーブルを拡張するのに十分なメモリを確保できなかった場合には -1 を返します。操作が失敗した場合、モジュールは一切内部テーブルに追加されません。 `Py_Initialize()` よりも前に呼び出さなければなりません。

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。

このモジュールでは、3つのバージョンのデータ形式をサポートしています。バージョン 0 は従来のもので、バージョン 1 は intern 化された文字列をファイル内で共有し、逆マーシャル化の時に共有されるようにします。バージョン 2 は、浮動小数点数に対してバイナリフォーマットを利用します。 `Py_MARSHAL_VERSION` は現在のバージョン (バージョン 2) を示します。

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。

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

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

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。

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

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

Return value: New reference. `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)`

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

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)`

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

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)`

Return value: *New reference.* Return a Python object from the data stream in a `FILE*` opened for reading.

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

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

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

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

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

Return value: *New reference.* `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`, *exc:TypeError*) を設定し `NULL` を返します。

6.6 引数の解釈と値の構築

これらの関数は独自の拡張モジュール用の関数やメソッドを作成する際に便利です。詳しい情報や用例は `extending-index` にあります。

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。

一般に、フォーマットがバッファにポインタをセットする時は、そのバッファは対応する Python オブジェクトにより管理され、バッファはそのオブジェクトのライフタイムを共有します。自分自身でメモリを解放する必要はありません。この例外は、`es`, `es#`, `et`, `et#` だけです。

ただし、`Py_buffer` 構造体に格納されたバッファはロックされて、呼び出し側はそのバッファを `Py_BEGIN_ALLOW_THREADS` ブロック内でも、mutable なデータが破棄されたりリサイズされたりするリスク無しに利用することができます。そのかわりに、そのデータに対する処理が終わった場合 (もしくは処理せずに中断した場合) には **必ず** `PyBuffer_Release()` を呼ばなければなりません。

特に言及されていない場合、バッファは NUL 終端されていません。

いくつかのフォーマットは読み込み専用の *bytes-like オブジェクト* を必要とし、バッファ構造体の代わりにポインタをセットします。それらは、オブジェクトの `PyBufferProcs.bf_releasebuffer` フィールドが NULL であることをチェックして動きます。これは `bytearray` などの変更可能オブジェクトを禁止します。

注釈: 全ての # 型のフォーマット (`s#`, `y#`, など) において、`length` 引数の型 (`int` か `Py_ssize_t`) は `Python.h` を include する前に `PY_SSIZE_T_CLEAN` マクロを定義することで制御します。マクロが定義されている場合、`length` は `int` 型ではなく `Py_ssize_t` 型になります。この挙動は将来の Python バージョンで変更され、`Py_ssize_t` のみがサポートされて `int` はサポートされなくなるでしょう。常に `PY_SSIZE_T_CLEAN` を定義したほうが良いです。

`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で埋め込まれてはなりません; 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` を送出处します。

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, 読み出し専用の *bytes-like object*) [`const char *`, `int` or `Py_ssize_t`] `s*` と同じですが、mutable なオブジェクトを受け取りません。結果は 2 つの C 変数に格納されます。1 つ目は C 文字列へのポインターで、2 つ目はその長さです。受け取った文字列は null バイトを含むかもしれません。Unicode オブジェクトは 'utf-8' エンコーディングを利用して C 文字列に変換されます。

`z` (str または `None`) [`const char *`] `s` に似ていますが、Python オブジェクトは `None` でもよく、その場合には C のポインタは `NULL` にセットされます。

`z*` (str, *bytes-like object* または `None`) [`Py_buffer`] `s*` と同じですが、Python の `None` オブジェクトを受け取ることができます。その場合、*Py_buffer* 構造体の `buf` メンバーは `NULL` になります。

`z#` (str, 読み出し専用の *bytes-like object* または `None`) [`const char *`, `int` or `Py_ssize_t`] `s#` に似ていますが、Python オブジェクトは `None` でもよく、その場合には C のポインタは `NULL` にセットされます。

`y` (読み出し専用の *bytes-like object*) [`const char *`] このフォーマットは *bytes-like object* をキャラクタ文字列を指す C のポインタに変換します; Unicode オブジェクトを受け付けません。バイトバッファは null バイトを含むべきではありません; null バイトを含む場合は `ValueError` 例外を送出します。

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 *`, `int` or `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) [`const Py_UNICODE *`] Python Unicode オブジェクトを NUL 終端された Unicode 文字バッファへのポインタに変換します。*Py_UNICODE* ポインタ変数へのアドレスを渡さなければならず、このアドレスに存在する Unicode バッファへのポインタが格納されます。*Py_UNICODE* 文字のバイト幅はコンパイルオプション (16 または 32 ビットのどちらか) に依存することに注意してください。Python 文

文字列は null コードポイントを含んではなりません; null コードポイントを含む場合、`ValueError` 例外が送出されます。

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。

Deprecated since version 3.3, will be removed in version 3.12: 古いスタイルの `Py_UNICODE` API の一部です。 `PyUnicode_AsWideCharString()` を使用するように移行してください。

`u# (str) [const Py_UNICODE *, int または Py_ssize_t]` これは `u` のバリエーションで、値を二つの変数に記録します。一つ目の変数は Unicode データバッファへのポインタで、二つ目はその長さです。このフォーマットは null コードポイントを含むことができます。

Deprecated since version 3.3, will be removed in version 3.12: 古いスタイルの `Py_UNICODE` API の一部です。 `PyUnicode_AsWideCharString()` を使用するように移行してください。

`Z (str または None) [const Py_UNICODE *]` `u` に似ていますが、Python オブジェクトは `None` でもよく、その場合には `Py_UNICODE` ポインタは `NULL` にセットされます。

Deprecated since version 3.3, will be removed in version 3.12: 古いスタイルの `Py_UNICODE` API の一部です。 `PyUnicode_AsWideCharString()` を使用するように移行してください。

`Z# (str または None) [const Py_UNICODE *, int または Py_ssize_t]` `u#` に似ていますが、Python オブジェクトは `None` でもよく、その場合には `Py_UNICODE` ポインタは `NULL` にセットされます。

Deprecated since version 3.3, will be removed in version 3.12: 古いスタイルの `Py_UNICODE` API の一部です。 `PyUnicode_AsWideCharString()` を使用するように移行してください。

`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, int または 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, int または 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。

数

`b (int) [unsigned char]` Python の非負の整数を、C の `unsigned char` 型の小さな符号無し整数に変換します。

`B (int) [unsigned char]` Python の整数を、オーバーフローチェックを行わずに、C の `unsigned char` 型の小さな整数に変換します。

`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、オーバー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、オーバー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、オーバー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 `int` を C `long long` へ変換する。

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] Python の浮動小数点型を、C の `float` 型に変換します。

d (float) [double] Python の浮動小数点型を、C の `double` 型に変換します。

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

0 (object) [PyObject*] Python オブジェクトを (一切変換を行わずに) C の Python オブジェクト型ポインタに保存します。これにより、C プログラムは実際のオブジェクトを受け渡しされます。オブジェクトの参照カウントは増加しません。保存されるポインタが NULL になることはありません。

0! (object) [PyObject*, 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。0 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。

0& (object) [converter, anything] 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。

もし `converter` が `Py_CLEANUP_SUPPORTED` を返すと、引数のパースが失敗した際に、コンバーターをもう一度呼び出し、すでに割り当てたメモリを開放するチャンスを与えます。二度目の呼び出しでは `object` 引数は `NULL` になり、`address` は最初の呼び出しと同じ値になります。

バージョン 3.1 で変更: `Py_CLEANUP_SUPPORTED` の追加。

`p (bool) [int]` 真偽値が求められる箇所 (a boolean predicate) 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いければ、`truth` を参照してください。

バージョン 3.3 で追加。

`(items) (tuple) [matching-items]` オブジェクトは `items` に入っている書式単位の数だけの長さを持つ Python のシーケンス型でなければなりません。各 C 引数は `items` 内の個々の書式単位に対応づけてきなければなりません。シーケンスの書式単位は入れ子構造にできます。

”長” 整数 (プラットフォームの `LONG_MAX` を超える値の整数) を渡すのは可能です; しかしながら、適切な値域チェックはまったく行われません --- 値を受け取るためのフィールドが、値全てを受け取るには小さすぎる場合、上桁のビット群は暗黙のうちに切り詰められます (実際のところ、このセマンティクスは C のダウンキャスト (downcast) から継承しています --- その恩恵は人それぞれかもしれませんが)。

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。

バージョン 3.3 で追加。

: 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 ” 附属値 (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。

呼び出し側に提供されるオブジェクトへの参照はすべて **借用 参照** (borrowed reference) になります; これらのオブジェクトの参照カウントをデクリメントしてはなりません!

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, ...)`

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 vars)`

`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[], ...)`

位置引数とキーワード引数の両者を取る関数の引数を解釈します。`keywords` 引数は NULL で終端されたキーワード名の配列です。空の名前は **位置専用引数** を示します。成功した場合、真を返します。失敗した場合は偽を返し、適切な例外を送出します。

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 vars)`

`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 *)`

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。

バージョン 3.2 で追加。

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

”旧スタイル”の関数における引数リストを分析するために使われる関数です --- 旧スタイルの関数は、引数解釈手法に、Python 3 で削除された `METH_OLDARGS` を使います。新たに書かれるコードでのパラメタ解釈にはこの関数の使用は奨められず、標準のインタプリタにおけるほとんどのコードがもはや引数解釈のためにこの関数を使わないように変更済みです。この関数を残しているのは、この関数が依然として引数以外のタプルを分析する上で便利だからですが、この目的においては将来も使われつづけるかもしれません。

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

パラメータ取得を簡単にした形式で、引数の型を指定する書式文字列を使いません。パラメタの取得にこの手法を使う関数は、関数宣言テーブル、またはメソッド宣言テーブル内で `METH_VARARGS` として宣言しなければなりません。実引数の入ったタプルは `args` に渡します; このタプルは本当のタプルでなければなりません。タプルの長さは少なくとも `min` で、`max` を超えてはなりません; `min` と `max` が等しくてもかまいません。補助引数を関数に渡さなければならず、各補助引数は `PyObject*` 変数へ

のポインタでなければなりません; これらの補助引数には、*args* の値が入ります; 値の参照は借用参照です。オプションのパラメタに対応する変数のうち、*args* に指定していないものには値が入りません; 呼び出し側はそれらの値を初期化しておかなければなりません。この関数は成功すると真を返し、*args* がタプルでない場合や間違った数の要素が入っている場合に偽を返します; 何らかの失敗が起きた場合には例外をセットします。

この関数の使用例を以下に示します。この例は、弱参照のための `_weakref` 補助モジュールのソースコードからとったものです:

```
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, ...)`

Return value: New reference. `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なります。

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 または None) [const char *, int または Py_ssize_t] C 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 *] C 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 *, int または 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 *] s と同じです。

z# (str または None) [const char *, int または 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 *, int または 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 *, int または 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 long long を Python の int オブジェクトへ変換する。

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 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]` C の `double` を Python の浮動小数点数に変換します。
- `f (float) [float]` C の `float` を Python の浮動小数点数に変換します。
- `D (complex) [Py_complex *]` C の `Py_complex` 構造体を Python の複素数型に変換します。
- `0 (object) [PyObject *]` Python オブジェクトを手を加えずに渡します (ただし、参照カウントは 1 インクリメントします)。渡したオブジェクトが `NULL` ポインタの場合、この引数を生成するのに使った何らかの呼び出しがエラーになったのが原因であると仮定して、例外をセットします。従ってこのとき `Py_BuildValue()` は `NULL` を返しますが例外は送出しません。例外をまだ送出していなければ `SystemError` をセットします。
- `S (object) [PyObject *]` `0` と同じです。
- `N (object) [PyObject *]` `0` と同じです。ただし、オブジェクトの参照カウントをインクリメントしません。オブジェクトが引数リスト内のオブジェクトコンストラクタ呼び出しによって生成されている場合に便利です。
- `0& (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 vars)`

Return value: New reference. `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, ...)`

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)`

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` and `format != NULL`.

もし *vsnprintf()* のないプラットフォームで、切り捨てを避けるために必要なバッファサイズが *size* を 512 バイトより大きく超過していれば、Python は *Py_FatalError()* で abort します。

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:

- $0 \leq rv < size$ 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。
- $rv \geq size$ 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。

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

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。

s が `float` に格納し切れないほど大きい値を表現していた場合、(例えば、`"1e500"` は多くのプラットフォームで表現できません) *overflow_exception* が `NULL` なら `Py_HUGE_VAL` に適切な符号を付け

て返します。他の場合は `overflow_exception` は Python の例外オブジェクトへのポインタでなければならず、その例外を発生させて `-1.0` を返します。どちらの場合でも、`*endptr` には変換された値の直後の最初の文字へのポインタが設定されます。

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。

バージョン 3.1 で追加.

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

バージョン 3.1 で追加.

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

大文字/小文字を区別しない文字列比較。大文字/小文字を無視する以外は、`strcmp()` と同じ動作をします。

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

大文字/小文字を区別しない文字列比較。大文字/小文字を無視する以外は、`strncmp()` と同じ動作をします。

6.8 リフレクション

*PyObject** PyEval_GetBuiltins()

Return value: Borrowed reference. 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()

Return value: Borrowed reference. 現在の実行フレーム内のローカル変数の辞書か、実行中のフレームがなければ NULL を返します。

*PyObject** PyEval_GetGlobals()

Return value: Borrowed reference. 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()

Return value: Borrowed reference. 現在のスレッド状態のフレームを返します。現在実行中のフレームがなければ NULL を返します。

int PyFrame_GetLineNumber(*PyFrameObject* *frame)

frame が現在実行している行番号を返します。

const char* PyEval_GetFuncName(*PyObject* *func)

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)

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。

6.9 codec レジストリとサポート関数

int PyCodec_Register(*PyObject* *search_function)

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_KnownEncoding(const char *encoding)

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)

Return value: New reference. 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)

Return value: New reference. 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。

6.9.1 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 与えられた encoding の StreamWriter ファクトリ関数を返します。

6.9.2 Unicode エラーハンドラ用レジストリ API

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

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)

Return value: New reference. 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)

Return value: Always NULL. exc を例外として発生させます。

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

Return value: New reference. 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)

Return value: New reference. unicode エラーを ? か U+FFFD で置き換えます。

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

Return value: New reference. unicode encode エラーを XML 文字参照で置き換えます。

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

Return value: New reference. 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)

Return value: New reference. unicode encode エラーを \N{...} で置き換えます。

バージョン 3.5 で追加.

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

C 関数から *Py_NotImplemented* を返す処理を適切に行います (すなわち、NotImplemented シングルトンの参照カウントを増やし、返却します)。

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

オブジェクト *o* をファイル *fp* に出力します。失敗すると -1 を返します。*flags* 引数は何らかの出力オプションを有効にする際に使います。現在サポートされている唯一のオプションは *Py_PRINT_RAW* です; このオプションを指定すると、*repr()* の代わりに *str()* を使ってオブジェクトを書き込みます。

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

o が属性 *attr_name* を持つときに 1 を、それ以外の場合に 0 を返します。この関数は Python の式 *hasattr(o, attr_name)* と同じです。この関数は常に成功します。

__getattr__() メソッドや *__getattribute__()* メソッド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、代わりに *PyObject_GetAttr()* を使ってください。

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

o が属性 *attr_name* を持つときに 1 を、それ以外の場合に 0 を返します。この関数は Python の式 *hasattr(o, attr_name)* と同じです。この関数は常に成功します。

__getattr__() メソッドや *__getattribute__()* メソッドの呼び出し中や、一時的な文字列オブ

ジェクトの作成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、代わりに `PyObject_GetAttrString()` を使ってください。

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

Return value: *New reference.* 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)`

Return value: *New reference.* 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_GenericGetAttr(PyObject *o, PyObject *name)`

Return value: *New reference.* 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립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립タはインスタンスの属性より優先され、非データデスク립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)`

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_DelAttr()` を使うのが望ましいです。

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

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。

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

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립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)`

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。

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

Return value: *New reference.* `__dict__` デスク립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。

バージョン 3.3 で追加。

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

`__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。

バージョン 3.3 で追加。

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

Return value: New reference. `o1` と `o2` を `opid` に指定した演算によって比較します。`opid` は `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, または `Py_GE`, のいずれかでなければならず、それぞれ `<`, `<=`, `==`, `!=`, `>`, および `>=` に対応します。この関数は Python の式 `o1 op o2` と同じで、`op` が `opid` に対応する演算子です。成功すると比較結果の値を返し失敗すると `NULL` を返します。

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

`o1` と `o2` を `opid` に指定した演算によって比較します。`opid` は `Py_LT`, `Py_LE`, `Py_EQ`, `Py_NE`, `Py_GT`, または `Py_GE`, のいずれかでなければならず、それぞれ `<`, `<=`, `==`, `!=`, `>`, および `>=` に対応します。比較結果が真ならば 1 を、偽ならば 0 を、エラーが発生すると -1 を返します。この関数は Python の式 `o1 op o2` と同じで、`op` が `opid` に対応する演算子です。

注釈: `o1` と `o2` が同一のオブジェクトである場合、`PyObject_RichCompareBool()` は `Py_EQ` に対して常に 1 を返し、`Py_NE` に対して常に 0 を返します。

PyObject* PyObject_Repr(PyObject *o)

Return value: New reference. 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のように、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)

Return value: New reference. `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 Python 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)

Return value: New reference. 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のように、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*)

Return value: *New reference.* 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*)

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、それ以外るとき 0 になります。

cls に `__subclasscheck__()` メソッドがある場合は、子クラスの状態が [PEP 3119](#) にある通りかどうかを判定するために呼ばれます。そうでないとき *derived* が *cls* の子クラスになるのは、直接的あるいは間接的な子クラスである場合、つまり *cls.__mro__* に含まれる場合です。

通常は、クラスオブジェクト、つまり `type` のインスタンスやそこから派生したクラスだけがクラスと見なされます。しかし、オブジェクトに属性 `__bases__` (これは基底クラスのタプルでなければならない) を持たせることで上書きできます。

int **PyObject_IsInstance**(*PyObject *inst*, *PyObject *cls*)

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、それ以外るとき 0 になります。

cls に `__instancecheck__()` メソッドがある場合は、子クラスの状態が [PEP 3119](#) にある通りかどうかを判定するために呼ばれます。そうでないとき *inst* が *cls* のインスタンスになるのは、そのクラスが *cls* の子クラスである場合です。

インスタンス *inst* に属性 `__class__` を持たせることで、そのクラスと見なされるものを上書きできます。

オブジェクト *cls* とその基底クラスがクラスと見なされる場合、属性 `__bases__` (これは基底クラスのタプルでなければならない) を持たせることで上書きできます。

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

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。

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

Return value: *New reference.* 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_CallObject**(*PyObject* *callable, *PyObject* *args)

Return value: *New reference.* 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。引数が不要な場合は、*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, ...)

Return value: *New reference.* 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。

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)`。

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

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, ...)

Return value: *New reference.* 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, ...)`。

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

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, ...)

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

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, ...)

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

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

Call a callable Python object *callable*, using *vectorcall* if possible.

args is a C array with the positional arguments.

nargsf is the number of positional arguments plus optionally the flag `PY_VECTORCALL_ARGUMENTS_OFFSET` (see below). To get actual number of arguments, use *PyVectorcall_NARGS(nargsf)*.

kwnames can be either NULL (no keyword arguments) or a tuple of keyword names. In the latter case, the values of the keyword arguments are stored in *args* after the positional arguments. The number of keyword arguments does not influence *nargsf*.

kwnames must contain only objects of type `str` (not a subclass), and all keys must be unique.

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。

This uses the vectorcall protocol if the callable supports it; otherwise, the arguments are converted to use *tp_call*.

注釈: This function is provisional and expected to become public in Python 3.9, with a different name and, possibly, changed semantics. If you use the function, plan for updating your code for Python 3.9.

バージョン 3.8 で追加.

`PY_VECTORCALL_ARGUMENTS_OFFSET`

If 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.

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) cheaply.

バージョン 3.8 で追加.

Py_ssize_t *PyVectorcall_NARGS*(*size_t* *nargsf*)

Given a vectorcall *nargsf* argument, return the actual number of arguments. Currently equivalent to `nargsf & ~PY_VECTORCALL_ARGUMENTS_OFFSET`.

バージョン 3.8 で追加.

*PyObject** *_PyObject_FastCallDict*(*PyObject* **callable*, *PyObject* **const* **args*, *size_t* *nargsf*, *PyObject* **kwdict*)

Same as *_PyObject_Vectorcall()* except that the keyword arguments are passed as a dictionary in *kwdict*. This may be NULL if there are no keyword arguments.

For callables supporting *vectorcall*, the arguments are internally converted to the vectorcall convention. Therefore, this function adds some overhead compared to *_PyObject_Vectorcall()*. It should only be used if the caller already has a dictionary ready to use.

注釈: This function is provisional and expected to become public in Python 3.9, with a different name and, possibly, changed semantics. If you use the function, plan for updating your code for Python 3.9.

バージョン 3.8 で追加.

`Py_hash_t PyObject_Hash(PyObject *o)`

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: 戻り値の型が `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)`

`type(o)` がハッシュ不可能であることを示す `TypeError` を設定し、-1 を返します。この関数は `tp_hash` スロットに格納されたときには特別な扱いを受け、その `type` がハッシュ不可能であることをインタプリタに明示的に示します。

`int PyObject_IsTrue(PyObject *o)`

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)`

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)`

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

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

オブジェクト *o* が、*type* か *type* のサブタイプであるときに真を返します。どちらのパラメタも NULL であってはなりません。

`Py_ssize_t PyObject_Size(PyObject *o)`

`Py_ssize_t PyObject_Length(PyObject *o)`

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 default)`

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, default)` と同じです。

バージョン 3.4 で追加。

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

Return value: New reference. 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)`

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)`

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。

`PyObject* PyObject_Dir(PyObject *o)`

Return value: New reference. 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)`

Return value: New reference. 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。

7.2 数値型プロトコル (number protocol)

`int PyNumber_Check(PyObject *o)`

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)`

Return value: New reference. 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)`

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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。

バージョン 3.5 で追加。

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

Return value: New reference. 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。”旧仕様の”整数間での除算と同じです。

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

Return value: New reference. 成功すると、数学的な o1 の o2 による除算値に対する妥当な近似 (reasonable approximation) を返し、失敗すると NULL を返します。全ての実数を 2 を基数として表現するのは不可能なため、二進の浮動小数点数は ”近似値” しか表現できません。このため、戻り値も近似になります。この関数に二つの整数を渡した際、浮動小数点の値を返すことがあります。

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

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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。

バージョン 3.5 で追加。

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

Return value: New reference. 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)

Return value: New reference. 成功すると、数学的な o1 の o2 による除算値に対する妥当な近似

(reasonable approximation) を返し、失敗すると NULL を返します。全ての実数を 2 を基数として表現するのは不可能なため、二進の浮動小数点数は ” 近似値 ” しか表現できません。このため、戻り値も近似になります。この関数に二つの整数を渡した際、浮動小数点の値を返すことがあります。o1 が in-place 演算をサポートする場合、in-place 演算を行います。

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

Return value: New reference. 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)

Return value: New reference. 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ccessを引き起こすことがあります)。

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

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. *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。

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

Return value: New reference. *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)

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)

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。

7.3 シーケンス型プロトコル (sequence protocol)

`int` **PySequence_Check**(*PyObject* *o)

オブジェクトがシーケンス型プロトコルを提供している場合は 1 を、そうでない場合は 0 を返します。`__getitem__()` メソッドを持つ Python クラスについては、それらが `dict` のサブクラスでない限り、1 を返すのに注意してください。そうなる理由は、一般的なケースではオブジェクトがどの種類のキーをサポートしているかを判別するのが不可能だからです。この関数は常に成功します。

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

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

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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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*)

Return value: New reference. 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*)

Return value: New reference. 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*)

Return value: New reference. 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*)

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*)

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*)

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*)

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*)

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*)

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*)

o[*i*] == *value* となる最初に見つかったインデックス *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*)

Return value: New reference. 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)`

Return value: *New reference.* 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)`

Return value: *New reference.* 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、**:exec: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)`

Return value: *Borrowed reference.* *o* が NULL でなく、*PySequence_Fast()* が返したオブジェクトであり、かつ *i* がインデッ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)`

Return value: *New reference.* *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。

7.4 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)`

オブジェクトがマップ型プロトコルを提供しているか、スライスをサポートしている場合は 1 を、そうでない場合は 0 を返します。__getitem__() メソッドを持つ Python クラスについては 1 を返すのに注意してください。そうなる理由は、一般的なケースではオブジェクトがどの種類のキーをサポートしているかを判別するのが不可能だからです。この関数は常に成功します。

`Py_ssize_t PyMapping_Size(PyObject *o)`

`Py_ssize_t PyMapping_Length(PyObject *o)`

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)`

Return value: New reference. 文字列 *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。*PyObject_GetItem()* も参照してください。

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

オブジェクト *o* 上で文字列 *key* を値 *v* に対応付けます。失敗すると -1 を返します。Python の文 `o[key] = v` と同じです。*PyObject_SetItem()* も参照してください。この関数は *v* への参照を盗み取り **ません**。

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

オブジェクト *o* から、オブジェクト *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。この関数は *PyObject_DelItem()* の別名です。

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

オブジェクト *o* から、文字列 *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 PyMapping_HasKey(PyObject *o, PyObject *key)`

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。

__getitem__() メソッド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、代わりに *PyObject_GetItem()* を使ってください。

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

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。

__getitem__() メソッドの呼び出し中や、一時的な文字列オブジェクトの作成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、代わりに *PyMapping_GetItemString()* を使ってください。

`PyObject* PyMapping_Keys(PyObject *o)`

Return value: New reference. 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)

Return value: *New reference.* 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)

Return value: *New reference.* 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。

7.5 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)

o がイテレータプロトコルをサポートする場合に真を返します。

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

Return value: *New reference.* 反復処理 *o* における次の値を返します。オブジェクトはイテレータでなければなりません (これをチェックするのは呼び出し側の責任です)。要素が何も残っていない場合は、例外がセットされていない状態で NULL を返します。要素を取り出す際にエラーが生じた場合は、NULL を返し、発生した例外を送出します。

イテレータの返す要素にわたって反復処理を行うループを書くと、C のコードは以下になるはずです:

```
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 */
}
```

(次のページに続く)

(前のページからの続き)

```
else {  
    /* continue doing useful work */  
}
```

7.6 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 は *buffer protocol* の形式で C レベルの仕組みを提供します。このプロトコルには二つの側面があります:

- 提供する側では、ある型は、そのオブジェクトの下層にあるバッファに関する情報を提供できる "buffer インタフェース" をエクスポートすることができます。このインタフェースは [バッファオブジェクト構造体 \(buffer object structure\)](#) の節で説明します。
- 利用する側では、オブジェクトの下層にある生データへのポインタを得るいくつかの手段が利用できます (たとえばメソッド引数)。

`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。

7.6.1 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。

Py_buffer

`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。

`void *obj`

エクスポートされているオブジェクトの新しい参照。参照は消費者によって所有され、*PyBuffer_Release()* によって自動的にデクリメントされて NULL に設定されます。このフィールドは標準的な C-API 関数の戻り値と等価です。

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。

`const char *format`

要素一つ分の内容を指定する、`struct` モジュールスタイル文法の、`NUL` 終端文字列。このポインタの値が `NULL` なら、"B" (符号無しバイト) として扱われます。

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`

メモリが `N` 次元配列を表している時の次元数。0 の場合、`buf` はスカラ値を表す 1 つの要素を指しています。この場合、`shape`、`strides`、`suboffsets` は `NULL` でなければなりません。

`PyBUF_MAX_NDIM` は次元数の最大値を 64 に制限しています。提供側はこの制限を尊重しなければなりません。多次元配列の消費側は `PyBUF_MAX_NDIM` 次元までを扱えるようにする必要があります。

`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。

`shape` (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よう必要があります。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何バイト加えればいいのかを示しています。`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ている)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くは *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。

7.6.2 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

readonly フィールドを制御します。もしこのフラグが設定されている場合、*exporter* は、書き込み可能なバッファを提供するか、さもなければ失敗を報告しなければなりません。フラグが設定されていない場合、*exporter* は、読み出し専用と書き込み可能なバッファのどちらを提供しても構いませんが、どちらで提供するかどうかは全ての消費者に対して一貫性がなければなりません。

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 は、*PyBUF_SIMPLE* 以外のどのフラグとも | を取ってかまいません。後者のフラグは B (符号なしバイト) フォーマットを既に指示しています。

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
<i>PyBUF_ND</i>	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います。

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

7.6.3 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
```

(次のページに続く)

(前のページからの続き)

```

        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];
        }
    }
    return (void*)pointer;
}

```

7.6.4 バッファ関連の関数

`int PyObject_CheckBuffer(PyObject *obj)`

`obj` が buffer インタ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)`

`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、`PyExc_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。チェー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)`

バッファ `view` を解放し、`view->obj` の参照カウントを 1 つ減らします。この関数はバッファが使われることがなくなったときに呼び出さなければならず、そうしないと参照のリークが起こり得ます。

`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 *)`

`itemsz` の値を `format` から計算して返します。この関数はまだ実装されていません。

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

`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(Py_buffer *view, Py_ssize_t *indices)`

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

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, Py_buffer *src, Py_ssize_t len, char order)`

`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。

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

成功したときは、`view->obj` に `exporter` への新しい参照を設定し、0 を返します。失敗したときは、`PyExc_BufferError` を送出し、`view->obj` に NULL を設定し、-1 を返します；

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。

7.7 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)`
 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)`
 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)

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。

この関数は試み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)

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。

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

`PyTypeObject`

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。

`PyObject*` `PyType_Type`

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)`

オブジェクト `o` が型オブジェクトの場合に真を返します。標準型オブジェクトから派生したサブタイプのインスタンスも含みます。その他の場合には偽を返します。

`int` `PyType_CheckExact(PyObject *o)`

オブジェクト `o` が型オブジェクトの場合に真を返します。標準型のサブタイプの場合には含みません。その他の場合には偽を返します。

unsigned int `PyType_ClearCache()`

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)`

type のメンバーである *tp_flags* を返します。この関数は基本的に *Py_LIMITED_API* を定義して使うことを想定しています。それぞれのフラグは python の異なるリリースで安定していることが保証されていますが、*tp_flags* 自体は限定された API の一部ではありません。

バージョン 3.2 で追加。

バージョン 3.4 で変更: 戻り値の型が long ではなく unsigned long になりました。

void `PyType_Modified(PyTypeObject *type)`

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_HasFeature(PyTypeObject *o, int feature)`

型オブジェクト *o* に、型機能 *feature* が設定されている場合に真を返します。型機能は各々単一ビットのフラグで表されます。

int `PyType_IS_GC(PyTypeObject *o)`

型オブジェクトが *o* が循環参照検出をサポートしている場合に真を返します; この関数は型機能フラグ *Py_TPFLAGS_HAVE_GC* の設定状態をチェックします。

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

a が *b* のサブタイプの場合に真を返します。

この関数は実際のサブクラスをチェックするだけです。つまり、`__subclasscheck__()` は *b* に対し呼ばれません。issubclass() と同じチェックをするには *PyObject_IsSubclass()* を呼んでください。

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

Return value: New reference. 型オブジェクトの *tp_alloc* に対するジェネリックハンドラです。Python のデフォルトのメモリアロケートメカニズムを使って新しいインスタンスをアロケートし、すべての内容を NULL で初期化します。

*PyObject** `PyType_GenericNew(PyTypeObject *type, PyObject *args, PyObject *kwargs)`

Return value: New reference. 型オブジェクトの *tp_new* に対するジェネリックハンドラです。型の *tp_alloc* スロットを使って新しいインスタンスを作成します。

int `PyType_Ready(PyTypeObject *type)`

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。

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

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。戻り値が 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、呼び出し側は戻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.

An exception is raised if *type* is not a heap type.

バージョン 3.4 で追加.

Creating Heap-Allocated Types

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

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

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

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

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

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

バージョン 3.3 で追加.

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

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

PyType_Spec

Structure defining a type's behavior.

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

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

`int` **PyType_Spec.basicsize**

`int` **PyType_Spec.itemsize**

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

`int` **PyType_Spec.flags**

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

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

PyType_Slot ***PyType_Spec.slots**

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

PyType_Slot

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

`int` **PyType_Slot.slot**

A slot ID.

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

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

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

- *tp_dict*
- *tp_mro*
- *tp_cache*
- *tp_subclasses*
- *tp_weaklist*
- *tp_print*
- *tp_weaklistoffset*
- *tp_dictoffset*
- *bf_getbuffer*
- *bf_releasebuffer*

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

`void *PyType_Slot.pfunc`

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

May not be NULL.

8.1.2 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`

Python における `None` オブジェクトで、値がないことを表します。このオブジェクトにはメソッドがありません。参照カウントについては、このオブジェクトも他のオブジェクトと同様に扱う必要があります。

Py_RETURN_NONE

C 関数からの *Py_None* の返却を適切に扱います。(これは None の参照カウントをインクリメントして返します。)

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。

PyLongObject

この *PyObject* のサブタイプは整数型を表現します。

PyTypeObject **PyLong_Type**

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* のサブタイプのときに真を返します。

int PyLong_CheckExact(*PyObject* *p)

引数が *PyLongObject* 型で、かつ *PyLongObject* 型のサブタイプでないときに真を返します。

*PyObject** **PyLong_FromLong(long v)**

Return value: New reference. *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するようにしており、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ります。従って、1 の値を変えることすら可能です。変えてしまった場合の Python の挙動は未定義です :-)

*PyObject** **PyLong_FromUnsignedLong(unsigned long v)**

Return value: New reference. 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)**

Return value: New reference. 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)**

Return value: New reference. 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*)

Return value: New reference. 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*)

Return value: New reference. 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*)

Return value: New reference. *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*)

Return value: New reference. *str* の文字列値に基づいて、新たな *PyLongObject* を返します。このとき *base* を基数として文字列を解釈します。 *pend* が NULL でない場合は、**pend* は *str* 中で数が表現されている部分以降の先頭文字のアドレスを指しています。 *base* が 0 の場合は、*str* は integers の定義を使って解釈されます; この場合では、先頭に 0 がある 0 でない十進数は *ValueError* を送出します。 *base* が 0 でなければ、*base* は 2 以上 36 以下の数でなければなりません。先頭の空白、基数の指定の後や数字の間にある単一のアンダースコアは無視されます。数字が全くない場合、*ValueError* が送出されます。

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

Return value: New reference. Convert a sequence of Unicode digits to a Python integer value.

Deprecated since version 3.3, will be removed in version 3.10: 古いスタイルの *Py_UNICODE* API の一部です; *PyLong_FromUnicodeObject()* を使用するように移行してください。

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

Return value: New reference. Convert a sequence of Unicode digits in the string *u* to a Python integer value.

バージョン 3.3 で追加。

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

Return value: New reference. 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*)

obj が表す、C の long 表現を返します。もし *obj* が *PyLongObject* のインスタンスでなければ、まず、その *__index__()* メソッドあるいは *__int__()* メソッドを (もしあれば) 呼び出して、オブジェクトを *PyLongObject* に変換します。

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 3.8 で非推奨: *__int__()* の使用は非推奨です。

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

obj が表す、C の `long` 表現を返します。もし *obj* が *PyLongObject* のインスタンスでなければ、まず、その `__index__()` メソッドあるいは `__int__()` メソッドを (もしあれば) 呼び出して、オブジェクトを *PyLongObject* に変換します。

もし、*obj* の値が `LONG_MAX` より大きい、`LONG_MIN` より小さければ、*overflow* は、それぞれ 1 か -1 に設定され、-1 を返します; さもないと *overflow* は 0 に設定されます。もし、ほかの例外が発生した場合は *overflow* が 0 に設定され -1 を返します。

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 3.8 で非推奨: `__int__()` の使用は非推奨です。

`long long PyLong_AsLongLong(PyObject *obj)`

obj が表す、C の `long long` 表現を返します。もし *obj* が *PyLongObject* のインスタンスでなければ、まず、その `__index__()` メソッドあるいは `__int__()` メソッドを (もしあれば) 呼び出して、オブジェクトを *PyLongObject* に変換します。

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 3.8 で非推奨: `__int__()` の使用は非推奨です。

`long long PyLong_AsLongLongAndOverflow(PyObject *obj, int *overflow)`

obj が表す、C の `long long` 表現を返します。もし *obj* が *PyLongObject* のインスタンスでなければ、まず、その `__index__()` メソッドあるいは `__int__()` メソッドを (もしあれば) 呼び出して、オブジェクトを *PyLongObject* に変換します。

もし、*obj* の値が `PY_LLONG_MAX` より大きい、`PY_LLONG_MIN` より小さければ、*overflow* は、それぞれ 1 か -1 に設定され、-1 を返します; さもないと *overflow* は 0 に設定されます。もし、ほかの例外が発生した場合は *overflow* が 0 に設定され -1 を返します。

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 3.2 で追加.

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 3.8 で非推奨: `__int__()` の使用は非推奨です。

`Py_ssize_t PyLong_AsSsize_t(PyObject *pylong)`

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)`

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)`

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)`

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)`

obj が表す、C の `unsigned long` 表現を返します。もし *obj* が `PyLongObject` のインスタンスでなければ、まず、その `__index__()` メソッドあるいは `__int__()` メソッドを (もしあれば) 呼び出して、オブジェクトを `PyLongObject` に変換します。

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 3.8 で非推奨: `__int__()` の使用は非推奨です。

`unsigned long long PyLong_AsUnsignedLongLongMask(PyObject *obj)`

obj が表す、C の `unsigned long long` 表現を返します。もし *obj* が `PyLongObject` のインスタ

スでなければ、まず、その `__index__()` メソッドあるいは `__int__()` メソッドを (もしあれば) 呼び出して、オブジェクトを *PyLongObject* に変換します。

obj の値が `unsigned long long` の範囲から外れていた場合は、`PY_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 3.8 で非推奨: `__int__()` の使用は非推奨です。

`double PyLong_AsDouble(PyObject *pylong)`

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)`

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。

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。

`int PyBool_Check(PyObject *o)`

o が `PyBool_Type` の場合に真を返します。

*PyObject** `Py_False`

Python における `False` オブジェクトです。このオブジェクトはメソッドを持ちません。参照カウンターの点では、他のオブジェクトと同様に扱う必要があります。

*PyObject** `Py_True`

Python における `True` オブジェクトです。このオブジェクトはメソッドを持ちません。参照カウンターの点では、他のオブジェクトと同様に扱う必要があります。

`Py_RETURN_FALSE`

`Py_False` に適切な参照カウンターのインクリメントを行って、関数から返すためのマクロです。

Py_RETURN_TRUE

`Py_True` に適切な参照カウンターのインクリメントを行って、関数から返すためのマクロです。

*PyObject** **PyBool_FromLong**(long *v*)

Return value: New reference. *v* の値に応じて `Py_True` または `Py_False` への新しい参照を返します。

8.2.3 浮動小数点型オブジェクト (floating point object)

PyFloatObject

この *PyObject* のサブタイプは Python 浮動小数点オブジェクトを表現します。

PyTypeObject **PyFloat_Type**

この *PyTypeObject* のインスタンスは Python 浮動小数点型を表現します。これは Python レイヤにおける `float` と同じオブジェクトです。

int **PyFloat_Check**(*PyObject* **p*)

引数が *PyFloatObject* か *PyFloatObject* のサブタイプのときに真を返します。

int **PyFloat_CheckExact**(*PyObject* **p*)

引数が *PyFloatObject* 型で、かつ *PyFloatObject* 型のサブタイプでないときに真を返します。

*PyObject** **PyFloat_FromString**(*PyObject* **str*)

Return value: New reference. *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*)

Return value: New reference. *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*)

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)

Return value: New reference. `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**()

`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()`

`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。

`int PyFloat_ClearFreeList()`

`float` の free list をクリアします。解放できなかったアイテム数を返します。

8.2.4 複素数オブジェクト (complex number object)

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。

`Py_complex`

Python 複素数オブジェクトの値の部分に対応する C の構造体です。複素数オブジェクトを扱うほとんどの関数は、この型の構造体の場合に応じて入力や出力として使います。構造体は以下のように定義されています:

```
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 complex)`

複素数 `complex` 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。

8.2. 数値型オブジェクト (numeric object)

Python オブジェクトとしての複素数型

`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`

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* 型か *PyComplexObject* 型のサブタイプのときに真を返します。

`int PyComplex_CheckExact(PyObject *p)`

引数が *PyComplexObject* 型で、かつ *PyComplexObject* 型のサブタイプでないときに真を返します。

*PyObject** `PyComplex_FromCComplex(Py_complex v)`

Return value: New reference. C の *Py_complex* 型から Python の複素数値を生成します。

*PyObject** `PyComplex_FromDoubles(double real, double imag)`

Return value: New reference. 新たな *PyComplexObject* オブジェクトを *real* と *imag* から生成します。

`double PyComplex_RealAsDouble(PyObject *op)`

op の実数部分を C の `double` 型で返します。

`double PyComplex_ImagAsDouble(PyObject *op)`

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。

op が Python の複素数オブジェクトではないが、`__complex__()` メソッドを持っていた場合、このメソッドが最初に呼ばれ、*op* が Python の複素数オブジェクトに変換されます。`__complex__()` が定義されていない場合は、`__float__()` にフォールバックされます。`__float__()` が定義されていない場合は、`__index__()` にフォールバックされます。処理が失敗した場合は、このメソッドは実数の `-1.0` を返します。

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。

`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`

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* がバイトオブジェクトか、または `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。

`int PyBytes_CheckExact(PyObject *o)`

オブジェクト *o* がバイトオブジェクトであり、かつ `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。

*PyObject** `PyBytes_FromString(const char *v)`

Return value: *New reference.* 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)`

Return value: *New reference.* 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, ...)`

Return value: *New reference.* 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
%%	<i>n/a</i>	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	<code>printf("%d")</code> と同等です。 ^{*1}
%u	unsigned int	<code>printf("%u")</code> と同等。 ^{*1}
%ld	long	<code>printf("%ld")</code> と同等。 ^{*1}
%lu	unsigned long	<code>printf("%lu")</code> と同等。 ^{*1}
%zd	Py_ssize_t	<code>printf("%zd")</code> と同等。 ^{*1}
%zu	size_t	<code>printf("%zu")</code> と同等。 ^{*1}
%i	int	<code>printf("%i")</code> と同等。 ^{*1}
%x	int	<code>printf("%x")</code> 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。 <code>printf("%p")</code> とほとんど同じですが、プラットフォームにおける <code>printf</code> 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*)

Return value: New reference. 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*)

Return value: New reference. 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*)

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() をマクロで実装したもので、エラーチェックををしません。

char* **PyBytes_AsString**(*PyObject* **o*)

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。

char* **PyBytes_AS_STRING**(*PyObject* **string*)

PyBytes_AsString() をマクロで実装したもので、エラーチェックををしません。

^{*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。

`int PyBytes_AsStringAndSize(PyObject *obj, char **buffer, Py_ssize_t *length)`

obj の null 終端された中身を、出力用の変数 *buffer* と *length* を介して返します。

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)`

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)`

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; この関数は、*newpart* の参照カウントをデクリメントします。

`int _PyBytes_Resize(PyObject **bytes, Py_ssize_t newsize)`

本来 "変更不可能 (immutable)" なバイトオブジェクトをリサイズする方法です。作成されたばかりのバイトオブジェクトにのみこれをしようしてください; 他のコードですでに使用されている可能性のあるバイトオブジェクトに使用してはいけません。入力されたバイトオブジェクトの参照カウントが 1 でない場合、この関数はエラーになります。左辺値として存在する (つまり書き込み可能な) バイトオブジェクトのアドレスを受け取り、新しいサイズを要求します。成功時には、**bytes* はリサイズされたバイトオブジェクトを保持し、0 が返されます; **bytes* のアドレスは入力された際のアドレスと異なるかもしれません。再割り当て (reallocation) が失敗した場合、**bytes* が元々指していたバイトオブジェクトを解放し、**bytes* を NULL に設定し、`MemoryError` を設定し、そして -1 が返されます。

8.3.2 bytearray オブジェクト

`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`

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)

o が bytearray かそのサブタイプのインスタンスだった場合に真を返します。

int PyByteArray_CheckExact(*PyObject* *o)

o が bytearray オブジェクトで、そのサブタイプのインスタンスでは無いときに、真を返します。

ダイレクト API 関数

*PyObject** PyByteArray_FromObject(*PyObject* *o)

Return value: New reference. *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。

*PyObject** PyByteArray_FromStringAndSize(const char *string, Py_ssize_t len)

Return value: New reference. *string* とその長さ *len* から新しい bytearray オブジェクトを返します。失敗した場合は NULL を返します。

*PyObject** PyByteArray_Concat(*PyObject* *a, *PyObject* *b)

Return value: New reference. bytearray *a* と *b* を連結した結果を新しい bytearray として返します。

Py_ssize_t PyByteArray_Size(*PyObject* *bytearray)

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)

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)

bytearray の内部バッファを *len* へリサイズします。

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() のマクロバージョン。

Py_ssize_t PyByteArray_GET_SIZE(*PyObject* *bytearray)

PyByteArray_Size() のマクロバージョン。

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。

`Py_UNICODE*` and UTF-8 representations are created on demand and cached in the Unicode object. The `Py_UNICODE*` representation is deprecated and inefficient.

古い API から新しい API への移行の影響で、Unicode オブジェクトの内部の状態は 2 通りあります。これはオブジェクトの作られ方によって決まります。

- ”正統な” Unicode オブジェクトは、非推奨ではない Unicode API で作成されたすべてのオブジェクトです。これらのオブジェクトは実装が許すかぎり最も効率の良い表現形式を使用します。
- ”legacy” Unicode objects have been created through one of the deprecated APIs (typically `PyUnicode_FromUnicode()`) and only bear the `Py_UNICODE*` representation; you will have to call `PyUnicode_READY()` on them before calling any other API.

注釈: The ”legacy” Unicode object will be removed in Python 3.12 with deprecated APIs. All Unicode objects will be ”canonical” since then. See **PEP 623** for more information.

Unicode 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:

`Py_UCS4`

`Py_UCS2`

`Py_UCS1`

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十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。

バージョン 3.3 で追加.

`Py_UNICODE`

これは、`wchar_t` の typedef で、プラットフォームに依存して 16 ビットか 32 ビットの型になります。

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。

`PyASCIIObject`

`PyCompactUnicodeObject`

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。

バージョン 3.3 で追加.

PyTypeObject **PyUnicode_Type**

この *PyTypeObject* のインスタンスは、Python Unicode 型を表します。これは、Python コードに `str` として露出されます。

以下の API は実際には C マクロで、Unicode オブジェクト内部の読み取り専用データに対するチェックやアクセスを高速に行います:

`int PyUnicode_Check(PyObject *o)`

o が Unicode 文字列型か Unicode 文字列型のサブタイプであるときに真を返します。

`int PyUnicode_CheckExact(PyObject *o)`

o が Unicode 文字列型で、かつ Unicode 文字列型のサブタイプでないときに真を返します。

`int PyUnicode_READY(PyObject *o)`

文字列オブジェクト *o* が " 正統な " 表現形式であることを保証します。このマクロは、下で説明していただくアクセスマクロを使うときも必要となります。

成功のときには 0 を返し、失敗のときには例外を設定し -1 を返します。後者は、メモリ確保に失敗したときに特に起きやすいです。

バージョン 3.3 で追加.

Deprecated since version 3.10, will be removed in version 3.12: This API will be removed with *PyUnicode_FromUnicode()*.

`Py_ssize_t PyUnicode_GET_LENGTH(PyObject *o)`

Unicode 文字列のコードポイントでの長さを返します。*o* 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)。

バージョン 3.3 で追加.

*Py_UCS1** `PyUnicode_1BYTE_DATA(PyObject *o)`

*Py_UCS2** `PyUnicode_2BYTE_DATA(PyObject *o)`

*Py_UCS4** `PyUnicode_4BYTE_DATA(PyObject *o)`

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()* を使って正しいマクロを選んでください。このオブジェクトにアクセスする前に、忘れずに *PyUnicode_READY()* を呼び出してください。

バージョン 3.3 で追加.

PyUnicode_WCHAR_KIND

PyUnicode_1BYTE_KIND

`PyUnicode_2BYTE_KIND`

`PyUnicode_4BYTE_KIND`

`PyUnicode_KIND()` マクロの返り値です。

バージョン 3.3 で追加.

Deprecated since version 3.10, will be removed in version 3.12: `PyUnicode_WCHAR_KIND` is deprecated.

`int PyUnicode_KIND(PyObject *o)`

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。 *o* 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)。

バージョン 3.3 で追加.

`void* PyUnicode_DATA(PyObject *o)`

Return a void pointer to the raw Unicode buffer. *o* has to be a Unicode object in the "canonical" representation (not checked).

バージョン 3.3 で追加.

`void PyUnicode_WRITE(int kind, void *data, Py_ssize_t index, Py_UCS4 value)`

正統な表現形式となっている (`PyUnicode_DATA()` で取得した) *data* に書き込みます。このマクロは正常性のチェックを一切行わない、ループで使われるためのものです。呼び出し側は、他のマクロを呼び出して取得した *kind* 値と *data* ポインタをキャッシュすべきです。*index* は文字列の (0 始まりの) インデックスで、*value* はその場所書き込まれることになる新しいコードポイントの値です。

バージョン 3.3 で追加.

`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。

バージョン 3.3 で追加.

`Py_UCS4 PyUnicode_READ_CHAR(PyObject *o, Py_ssize_t index)`

Unicode オブジェクト *o* 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。

バージョン 3.3 で追加.

`PyUnicode_MAX_CHAR_VALUE(o)`

o 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。

バージョン 3.3 で追加.

int PyUnicode_ClearFreeList()

free list をクリアします。解放された要素数を返します。

Py_ssize_t PyUnicode_GET_SIZE(*PyObject* *o)

非推奨の *Py_UNICODE* 表現形式のサイズをコード単位で返します (サロゲートペアを 2 つとしています)。o は 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)。

Deprecated since version 3.3, will be removed in version 3.12: 古いスタイルの Unicode API の一部なので、*PyUnicode_GET_LENGTH()* を使用するように移行してください。

Py_ssize_t PyUnicode_GET_DATA_SIZE(*PyObject* *o)

非推奨の *Py_UNICODE* 表現形式のサイズをバイト単位で返します。o は 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)。

Deprecated since version 3.3, will be removed in version 3.12: 古いスタイルの Unicode API の一部なので、*PyUnicode_GET_LENGTH()* を使用するように移行してください。

*Py_UNICODE** PyUnicode_AS_UNICODE(*PyObject* *o)

const char* PyUnicode_AS_DATA(*PyObject* *o)

与えられたオブジェクトの *Py_UNICODE* 表現形式へのポインタを返します。返されるバッファは常に終端に null コードポイントが 1 つ余分に付いています。それとは別の null コードポイントがバッファに含まれることもあるかもしれませんが、たいていの C 関数ではそのような文字列は切り詰められてしまうでしょう。AS_DATA の方はポインタを const char * にキャストしています。引数 o は 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)。

バージョン 3.3 で変更: このマクロは今では非効率なものになりました。というのも、多くのケースで *Py_UNICODE* 表現形式が登場せず、作成されず、そして失敗し得ます (例外を設定して NULL を返します)。コードを修正して、PyUnicode_nBYTE_DATA() マクロを使うか *PyUnicode_WRITE()* や *PyUnicode_READ()* を使うようにしてください。

Deprecated since version 3.3, will be removed in version 3.12: 古いスタイルの Unicode API の一部なので、PyUnicode_nBYTE_DATA() 系のマクロを使用するように移行してください。

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_UNICODE* 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_UNICODE* 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_UNICODE* 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_UNICODE 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_UNICODE 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_UNICODE 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_UNICODE 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_UNICODE 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_UNICODE 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_UNICODE 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_UNICODE ch)`

ch が文字が印字可能な文字かどうかに基づいて 1 または 0 を返します。非印字可能文字は、Unicode 文字データベースで "Other" または "Separator" と定義されている文字の、印字可能と見なされる ASCII space (0x20) 以外のものです。(なお、この文脈での印字可能文字は、文字列に `repr()` が呼び出されるときにエスケープすべきでない文字のことです。これは `sys.stdout` や `sys.stderr` に書き込まれる文字列の操作とは関係ありません。)

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_UNICODE Py_UNICODE_TOLOWER(Py_UNICODE 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: この関数は単純な大文字小文字変換を使っています。

`Py_UNICODE Py_UNICODE_TOUPPER(Py_UNICODE 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: この関数は単純な大文字小文字変換を使っています。

`Py_UNICODE Py_UNICODE_TOTITLE(Py_UNICODE 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: この関数は単純な大文字小文字変換を使っています。

`int Py_UNICODE_TODECIMAL(Py_UNICODE ch)`

ch を 10 進の正の整数に変換したものを返します。不可能ならば -1 を返します。このマクロは例外を送出しません。

`int Py_UNICODE_TODIGIT(Py_UNICODE ch)`

ch を一桁の 2 進整数に変換したものを返します。不可能ならば -1 を返します。このマクロは例外を

送出しません。

`double Py_UNICODE_TONUMERIC(Py_UNICODE 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。このマクロ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:

`Py_UNICODE_IS_SURROGATE(ch)`

`ch` がサロゲートかどうか (`0xD800 <= ch <= 0xDFFF`) をチェックします。

`Py_UNICODE_IS_HIGH_SURROGATE(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。

`Py_UNICODE_IS_LOW_SURROGATE(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_UNICODE_JOIN_SURROGATES(high, low)`

2 つのサロゲート文字を組み合わせて単一の `Py_UCS4` 値を返します。`high` と `low` はそれぞれサロゲートペアの前半分と後半分です。

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)`

Return value: *New reference.* 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。

これは新しい Unicode オブジェクトを生成する推奨された方法です。この関数を使って生成されたオブジェクトはサイズ変更は不可能です。

バージョン 3.3 で追加.

*PyObject** `PyUnicode_FromKindAndData(int kind, const void *buffer, Py_ssize_t size)`

Return value: *New reference.* 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。

バージョン 3.3 で追加.

*PyObject** `PyUnicode_FromStringAndSize(const char *u, Py_ssize_t size)`

Return value: *New reference.* `char` 型バッファ `u` から Unicode オブジェクトを生成します。`u` の内容は UTF-8 でエンコードされているものとします。バッファの内容は新たなオブジェクトにコピーされます。バッファが `NULL` でない場合、帰りは共有されたオブジェクトになることがあります。つまり、この関数が返す Unicode オブジェクトの変更は許されていません。

If *u* is NULL, this function behaves like `PyUnicode_FromUnicode()` with the buffer set to NULL. This usage is deprecated in favor of `PyUnicode_New()`, and will be removed in Python 3.12.

*PyObject** `PyUnicode_FromString(const char *u)`

Return value: *New reference.* UTF-8 エンコードされた null 終端の char 型バッファ *u* から Unicode オブジェクトを生成します。

*PyObject** `PyUnicode_FromFormat(const char *format, ...)`

Return value: *New reference.* Take a C `printf()`-style *format* string and a variable number of arguments, calculate the size of the resulting Python Unicode string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* ASCII-encoded string. The following format characters are allowed:

書式指定文字	型	備考
%%	<i>n/a</i>	リテラルの % 文字
%c	int	C の整数型で表現される単一の文字。
%d	int	<code>printf("%d")</code> と同等。 ^{*1}
%u	unsigned int	<code>printf("%u")</code> と同等。 ^{*1}
%ld	long	<code>printf("%ld")</code> と同等。 ^{*1}
%li	long	<code>printf("%li")</code> と同等。 ^{*1}
%lu	unsigned long	<code>printf("%lu")</code> と同等。 ^{*1}
%lld	long long	<code>printf("%lld")</code> と同等。 ^{*1}
%lli	long long	<code>printf("%lli")</code> と同等。 ^{*1}
%llu	unsigned long long	<code>printf("%llu")</code> と同等。 ^{*1}
%zd	Py_ssize_t	<code>printf("%zd")</code> と同等。 ^{*1}
%zi	Py_ssize_t	<code>printf("%zi")</code> と同等。 ^{*1}
%zu	size_t	<code>printf("%zu")</code> と同等。 ^{*1}
%i	int	<code>printf("%i")</code> と同等。 ^{*1}
%x	int	<code>printf("%x")</code> 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。 <code>printf("%p")</code> とほとんど同じですが、プラットフォームにおける <code>printf</code> の定義に関わりなく先頭にリテラル 0x が付きます。
%A	PyObject*	<code>ascii()</code> の戻り値。
%U	PyObject*	Unicode オブジェクト。
%V	PyObject*, const char*	A Unicode object (which may be NULL) and a null-terminated C character array as a second parameter (which will be used, if the first parameter is NULL).
%S	PyObject*	<code>PyObject_Str()</code> の戻り値。
%R	PyObject*	<code>PyObject_Repr()</code> の戻り値。

^{*1} For integer specifiers (d, u, ld, li, lu, lld, lli, llu, zd, zi, zu, i, x): the 0-conversion flag has effect even when a precision

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出力文字列にコピーされ、残りの引数は無視されます。

注釈: The width formatter unit is number of characters rather than bytes. The precision formatter unit is number of bytes for "%s" and "%V" (if the PyObject* argument is NULL), and a number of characters for "%A", "%U", "%S", "%R" and "%V" (if the PyObject* argument is not NULL).

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。

*PyObject** **PyUnicode_FromFormatV**(const char *format, va_list vars)

Return value: New reference. 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_FromEncodedObject**(*PyObject* *obj, const char *encoding, const char *errors)

Return value: New reference. 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 (詳しくは [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。

Py_ssize_t **PyUnicode_GetLength**(*PyObject* *unicode)

Unicode オブジェクトの長さをコードポイントで返します。

バージョン 3.3 で追加。

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。

バージョン 3.3 で追加。

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。

is given.

`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い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。

バージョン 3.3 で追加。

`int PyUnicode_WriteChar(PyObject *unicode, Py_ssize_t index, Py_UCS4 character)`

文字列に文字を書き込みます。文字列は `PyUnicode_New()` で作成しなければなりません。Unicode 文字列は不変とされているので、この文字列は共有されていたり、これまでにハッシュ化されてはいけません。

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、オブジェクトが安全に変更できる (つまり参照カウントが 1 である) ことをチェックします。

バージョン 3.3 で追加。

`Py_UCS4 PyUnicode_ReadChar(PyObject *unicode, Py_ssize_t index)`

文字列から文字を読み取ります。マクロ版の `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。

バージョン 3.3 で追加。

`PyObject* PyUnicode_Substring(PyObject *str, Py_ssize_t start, Py_ssize_t end)`

Return value: New reference. `str` の文字インデックス `start` (端点を含む) から文字インデックス `end` (端点を含まず) までの部分文字列を返します。負のインデックスはサポートされていません。

バージョン 3.3 で追加。

`Py_UCS4* PyUnicode_AsUCS4(PyObject *u, Py_UCS4 *buffer, Py_ssize_t buflen, int copy_null)`

文字列 `u` を UCS4 のバッファへコピーします。`copy_null` が設定されている場合は、ヌル文字も含めます。エラーが起きたときは、NULL を返し、例外を設定します (`buflen` が `u` の長さより短かった場合については、`SystemError` が設定されます)。成功したときは `buffer` を返します。

バージョン 3.3 で追加。

`Py_UCS4* PyUnicode_AsUCS4Copy(PyObject *u)`

文字列 `u` 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。

バージョン 3.3 で追加。

廃止予定の Py_UNICODE API 群

Deprecated since version 3.3, will be removed in version 3.12.

これらの API 関数は **PEP 393** の実装により廃止予定です。Python 3.x では削除されないため、拡張モジュールはこれらの関数を引き続き使えますが、これらの関数の使用はパフォーマンスとメモリに影響があることを念頭に置いてください。

*PyObject** **PyUnicode_FromUnicode**(const *Py_UNICODE* **u*, Py_ssize_t *size*)

Return value: New reference. *size* で指定された長さを持つ *Py_UNICODE* 型バッファ *u* から Unicode オブジェクトを生成します。*u* を NULL にしてもよく、その場合オブジェクトの内容は未定義です。バッファに必要な情報を埋めるのはユーザの責任です。バッファの内容は新たなオブジェクトにコピーされます。

バッファが NULL でない場合、戻り値は共有されたオブジェクトになることがあります。従って、この関数が返す Unicode オブジェクトを変更してよいのは *u* が NULL のときだけです。

バッファが NULL の場合、文字列の内容が埋められたなら *PyUnicode_KIND()* のようなアクセスマクロを使う前に *PyUnicode_READY()* を呼び出さなければなりません。

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using *PyUnicode_FromKindAndData()*, *PyUnicode_FromWideChar()*, or *PyUnicode_New()*.

*Py_UNICODE** **PyUnicode_AsUnicode**(*PyObject* **unicode*)

Unicode オブジェクトの *Py_UNICODE* 型の内部バッファへの読み出し専用のポインタを返すか、失敗のときには NULL を返します。オブジェクトの *Py_UNICODE** 表現形式が無い場合には作成します。結果の *Py_UNICODE* 文字列には null コードポイントが含まれていることがあり、たいていの C 関数では、そのような文字列は切り詰められてしまうことに注意してください。

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using *PyUnicode_AsUCS4()*, *PyUnicode_AsWideChar()*, *PyUnicode_ReadChar()* or similar new APIs.

Deprecated since version 3.3, will be removed in version 3.10.

*PyObject** **PyUnicode_TransformDecimalToASCII**(*Py_UNICODE* **s*, Py_ssize_t *size*)

Return value: New reference. 与えられた長さ *size* を持つ *Py_UNICODE* 型のバッファにある全ての decimal digit を、それらの 10 進の値に対応する 0 から 9 までの ASCII 数字に置き換えた Unicode オブジェクトを生成します。例外が起きた場合は NULL を返します。

Deprecated since version 3.3, will be removed in version 3.11: Part of the old-style *Py_UNICODE* API; please migrate to using *Py_UNICODE_TODECIMAL()*.

*Py_UNICODE** **PyUnicode_AsUnicodeAndSize**(*PyObject* **unicode*, Py_ssize_t **size*)

PyUnicode_AsUnicode() に似てますが、さらに (追加の null 終端子を除いた) *Py_UNICODE()* 配列の長さを *size* に保存します。結果の *Py_UNICODE* 文字列には null コードポイントが含まれていることがあり、たいていの C 関数では、そのような文字列は切り詰められてしまうことに注意してください。

バージョン 3.3 で追加.

Deprecated since version 3.3, will be removed in version 3.12: Part of the old-style Unicode API, please migrate to using `PyUnicode_AsUCS4()`, `PyUnicode_AsWideChar()`, `PyUnicode_ReadChar()` or similar new APIs.

`Py_UNICODE*` `PyUnicode_AsUnicodeCopy(PyObject *unicode)`

終端に null コードポイントが付加された Unicode 文字列のコピーを作成します。メモリ確保に失敗したときは NULL を返し `MemoryError` 例外を送出します。そうでないときは新しくメモリ確保されたバッファを返します (このバッファを解放するときには `PyMem_Free()` を使ってください)。結果の `Py_UNICODE` 文字列には null コードポイントが含まれていることがあり、たいていの C 関数では、そのような文字列は切り詰められてしまうことに注意してください。

バージョン 3.2 で追加.

`PyUnicode_AsUCS4Copy()` や類似の新しい API を使用するように移行してください。

`Py_ssize_t` `PyUnicode_GetSize(PyObject *unicode)`

非推奨の `Py_UNICODE` 表現形式のサイズをコード単位で返します (サロゲートペアを 2 つとしています)。

Deprecated since version 3.3, will be removed in version 3.12: 古いスタイルの Unicode API の一部なので、`PyUnicode_GET_LENGTH()` を使用するように移行してください。

`PyObject*` `PyUnicode_FromObject(PyObject *obj)`

Return value: *New reference.* Unicode のサブタイプのインスタンスを、必要な場合は本物の Unicode オブジェクトにコピーします。obj が (サブタイプではない) 既に本物の Unicode オブジェクトだった場合は、参照カウントを 1 つ増やした参照を返します。

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。

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 len, const char *errors)`

Return value: *New reference.* Decode a string from UTF-8 on Android and VxWorks, or from the current locale encoding on other platforms. The supported error handlers are "strict" and "surrogateescape" ([PEP 383](#)). The decoder uses "strict" error handler if *errors* is NULL. *str* must end with a null character but cannot contain embedded null characters.

`PyUnicode_DecodeFSDefaultAndSize()` を使って (Python の起動時に読み込まれるロケールエンコーディングの) `Py_FileSystemDefaultEncoding` の文字列をデコードします。

この関数は Python の UTF-8 モードを無視します。

参考:

`Py_DecodeLocale()` 関数。

バージョン 3.3 で追加.

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)

Return value: New reference. `PyUnicode_DecodeLocaleAndSize()` と似てますが、`strlen()` を使って文字列の長さを計算します。

バージョン 3.3 で追加.

*PyObject** **PyUnicode_EncodeLocale**(*PyObject* *unicode, const char *errors)

Return value: New reference. Encode a Unicode object to UTF-8 on Android and VxWorks, or to the current locale encoding on other platforms. The supported error handlers are "strict" and "surrogateescape" (**PEP 383**). The encoder uses "strict" error handler if *errors* is NULL. Return a bytes object. *unicode* cannot contain embedded null characters.

`PyUnicode_EncodeFSDefault()` を使って (Python の起動時に読み込まれるロケールエンコーディングの) `Py_FileSystemDefaultEncoding` の文字列へエンコードします。

この関数は Python の UTF-8 モードを無視します。

参考:

`Py_EncodeLocale()` 関数。

バージョン 3.3 で追加.

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

ファイル名や他の環境文字列のエンコードやデコードを行うには、`Py_FileSystemDefaultEncoding` をエンコーディングとして使い、`Py_FileSystemDefaultEncodeErrors` をエラーハンドラとして使うべきです (**PEP 383** および **PEP 529**)。引数の構文解析中にファイル名を bytes にエンコードするには、`"0&"` コンバーターを使い、`PyUnicode_FSConverter()` を変換関数として渡すべきです:

int **PyUnicode_FSConverter**(*PyObject** obj, void* result)

ParseTuple コンバーター: `PyUnicode_EncodeFSDefault()` を使い -- 直接あるいは `os.PathLike` インターフェースを通して取得した -- str オブジェクトを bytes へエンコードします; bytes オブジェクトはそのまま出力されます。result は *PyBytesObject** でなければならず、使われなくなったときには解放されなければなりません。

バージョン 3.1 で追加.

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。

引数の構文解析中にファイル名を `str` にデコードするには、"`0&`" コンバーターを使い、`PyUnicode_FSDecoder()` を変換関数として渡すのがよいです:

```
int PyUnicode_FSDecoder(PyObject* obj, void* result)
```

ParseTuple コンバーター: `PyUnicode_DecodeFSDefaultAndSize()` を使い -- 直接的あるいは間接的に `os.PathLike` インターフェースを通して取得した -- `bytes` オブジェクトを `str` ヘエンコードします; `bytes` オブジェクトはそのまま出力されます。 `result` は `PyUnicodeObject*` でなければならず、使われなくなったときには解放されなければなりません。

バージョン 3.2 で追加。

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 *s, Py_ssize_t size)
```

Return value: *New reference.* `Py_FileSystemDefaultEncoding` と `Py_FileSystemDefaultEncodeErrors` エラーハンドラを使い、文字列をデコードします。

`Py_FileSystemDefaultEncoding` が設定されていない場合は、ロケールエンコーディングに差し戻されます。

`Py_FileSystemDefaultEncoding` は起動時にロケールエンコーディングで初期化され、それ以降は変更できません。現在のロケールエンコーディングで文字列をデコードする必要がある場合は、`PyUnicode_DecodeLocaleAndSize()` を使ってください。

参考:

`Py_DecodeLocale()` 関数。

バージョン 3.6 で変更: `Py_FileSystemDefaultEncodeErrors` エラーハンドラを使うようになりました。

```
PyObject* PyUnicode_DecodeFSDefault(const char *s)
```

Return value: *New reference.* `Py_FileSystemDefaultEncoding` と `Py_FileSystemDefaultEncodeErrors` エラーハンドラを使い、null 終端文字列をデコードします。

`Py_FileSystemDefaultEncoding` が設定されていない場合は、ロケールエンコーディングに差し戻されます。

文字列の長さが分かっている場合は、`PyUnicode_DecodeFSDefaultAndSize()` を使ってください。

バージョン 3.6 で変更: `Py_FileSystemDefaultEncodeErrors` エラーハンドラを使うようになりました。

```
PyObject* PyUnicode_EncodeFSDefault(PyObject *unicode)
```

Return value: *New reference.* `Py_FileSystemDefaultEncoding` エラーハンドラで Unicode オブジェクトを `Py_FileSystemDefaultEncoding` にエンコードし、`bytes` を返します。返される `bytes` オブジェクトは null バイトを含んでいるかもしれないことに注意してください。

`Py_FileSystemDefaultEncoding` が設定されていない場合は、ロケールエンコーディングに差し戻されます。

`Py_FileSystemDefaultEncoding` は起動時にロケールエンコーディングで初期化され、それ以降は変更できません。現在のロケールエンコーディングで文字列をエンコードする必要がある場合は、`PyUnicode_EncodeLocale()` を使ってください。

参考:

`Py_EncodeLocale()` 関数。

バージョン 3.2 で追加。

バージョン 3.6 で変更: `Py_FileSystemDefaultEncodeErrors` エラーハンドラを使うようになりました。

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` **w*, `Py_ssize_t` *size*)

Return value: New reference. *size* の `wchar_t` バッファ *w* から Unicode オブジェクトを生成します。*size* として -1 を渡すことで、`wcslen` を使い自身で長さを計算しなければならないことを関数に指示します。失敗すると NULL を返します。

`Py_ssize_t` `PyUnicode_AsWideChar`(*PyObject* **unicode*, `wchar_t` **w*, `Py_ssize_t` *size*)

Unicode オブジェクトの内容を `wchar_t` バッファ *w* にコピーします。最大で *size* 個の `wchar_t` 文字を (末尾の null 終端文字を除いて) コピーします。コピーした `wchar_t` 文字の個数を返します。エラーの時には -1 を返します。`wchar_t` 文字列は null 終端されている場合も、されていない場合があります。関数の呼び出し側の責任で、アプリケーションの必要に応じて `wchar_t` 文字列を null 終端してください。また、結果の `wchar_t*` 文字列には文字列の途中にも null 文字が含まれていることがあり、たいていの C 関数では、そのような文字列は切り詰められてしまうことに注意してください。

`wchar_t*` `PyUnicode_AsWideCharString`(*PyObject* **unicode*, `Py_ssize_t` **size*)

Convert the Unicode object to a wide character string. The output string always ends with a null character. If *size* is not NULL, write the number of wide characters (excluding the trailing null termination character) into **size*. Note that the resulting `wchar_t` string might contain null characters, which would cause the string to be truncated when used with most C functions. If *size* is NULL and the `wchar_t*` string contains null characters a `ValueError` is raised.

Returns a buffer allocated by `PyMem_Alloc()` (use `PyMem_Free()` to free it) on success. On error, returns NULL and **size* is undefined. Raises a `MemoryError` if memory allocation is failed.

バージョン 3.2 で追加。

バージョン 3.7 で変更: *size* が NULL かつ `wchar_t*` 文字列が null 文字を含んでいた場合、`ValueError` を送出します。

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。

encoding を NULL にすると、デフォルトエンコーディングである ASCII を使います。ファイルシステムに関する関数の呼び出しでは、ファイル名に対するエンコーディングとして `PyUnicode_FSConverter()` を使わねばなりません。これは内部で変数 `Py_FileSystemDefaultEncoding` を使用しています。この変数は読み出し専用の変数として扱わねばなりません: この変数は、あるシステムによっては静的な文字列に対するポインタであったり、また別のシステムでは、(アプリケーションが `setlocale` を呼んだときなどに) 変わったりもします。

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 *s, Py_ssize_t size, const char *encoding, const
                           char *errors)
Return value: New reference. size バイトのエンコードされた文字列 s をデコードして Unicode オブジェクトを生成します。encoding と errors は、組み込み関数 str() の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。
```

```
PyObject* PyUnicode_AsEncodedString(PyObject *unicode, const char *encoding, const
                                     char *errors)
Return value: New reference. 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。
```

```
PyObject* PyUnicode_Encode(const Py_UNICODE *s, Py_ssize_t size, const char *encoding,
                           const char *errors)
Return value: New reference. size で指定されたサイズの Py_UNICODE バッファ s をエンコードした Python の bytes オブジェクトを返します。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。
```

Deprecated since version 3.3, will be removed in version 3.11: 古いスタイルの *Py_UNICODE* API の一部です; `PyUnicode_AsEncodedString()` を使用するよう移行してください。

UTF-8 Codecs

以下は UTF-8 codec の API です:

*PyObject** **PyUnicode_DecodeUTF8**(const char *s, Py_ssize_t size, const char *errors)

Return value: New reference. UTF-8 でエンコードされた *size* バイトの文字列 *s* 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 *s, Py_ssize_t size, const char *errors, Py_ssize_t *consumed)

Return value: New reference. *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、**PyUnicode_DecodeUTF8Stateful()** は末尾の不完全な UTF-8 バイト列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_AsUTF8String**(*PyObject* *unicode)

Return value: New reference. 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)

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)。

エラーが起きた場合は、NULL を返し、例外を設定し、*size* には何も格納しません。

この関数は、Unicode オブジェクトの文字列の UTF-8 表現形式をキャッシュし、それ以降の呼び出しでは同じバッファへのポインタを返します。呼び出し側にバッファを解放する責任はありません。

バージョン 3.3 で追加。

バージョン 3.7 で変更: 戻り値の型が char * ではなく const char * になりました。

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。

バージョン 3.3 で追加。

バージョン 3.7 で変更: 戻り値の型が char * ではなく const char * になりました。

*PyObject** **PyUnicode_EncodeUTF8**(const *Py_UNICODE* *s, Py_ssize_t size, const char *errors)

Return value: New reference. 与えられたサイズの *Py_UNICODE* バッファ *s* を UTF-8 でエンコードして、Python の bytes オブジェクトとして返します。codec が例外を発生させたときは NULL を返します。

Deprecated since version 3.3, will be removed in version 3.11: 古いスタイルの *Py_UNICODE* API の一部です; **PyUnicode_AsUTF8String()**, **PyUnicode_AsUTF8AndSize()**, **PyUnicode_AsEncodedString()** のいずれかを使用するように移行してください。

UTF-32 Codecs

以下は UTF-32 codec API です:

*PyObject** **PyUnicode_DecodeUTF32**(const char *s, Py_ssize_t size, const char *errors, int *byteorder)

Return value: New reference. 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き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 *s, Py_ssize_t size, const char *errors, int *byteorder, Py_ssize_t *consumed)

Return value: New reference. *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)

Return value: New reference. 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。

*PyObject** **PyUnicode_EncodeUTF32**(const *Py_UNICODE* *s, Py_ssize_t size, const char *errors, int byteorder)

Return value: New reference. *s* の Unicode データを UTF-32 にエンコードし、その値を Python の bytes オブジェクトに格納して返します。出力は以下のバイトオーダーで従って書かれます:

```
byteorder == -1: little endian
byteorder == 0:  native byte order (writes a BOM mark)
byteorder == 1:  big endian
```

byteorder が 0 のとき、出力文字列は常に Unicode BOM マーク (U+FEFF) で始まります。それ以外の 2 つのモードでは、先頭に BOM マークは出力されません。

If `Py_UNICODE_WIDE` is not defined, surrogate pairs will be output as a single code point.

codec が例外を発生させたときは `NULL` を返します。

Deprecated since version 3.3, will be removed in version 3.11: 古いスタイルの `Py_UNICODE` API の一部です; `PyUnicode_AsUTF32String()` または `PyUnicode_AsEncodedString()` を使用するように移行してください。

UTF-16 Codecs

以下は UTF-16 codec の API です:

*PyObject** **PyUnicode_DecodeUTF16**(const char *s, Py_ssize_t size, const char *errors, int *byteorder)

Return value: New reference. 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 `\uffeff` 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 *s, Py_ssize_t size, const char *errors, int *byteorder, Py_ssize_t *consumed)

Return value: New reference. *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)

Return value: New reference. 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。

*PyObject** **PyUnicode_EncodeUTF16**(const *Py_UNICODE* *s, Py_ssize_t size, const char *errors, int byteorder)

Return value: New reference. *s* の Unicode データを UTF-16 にエンコードし、その値を Python の bytes オブジェクトに格納して返します。出力は以下のバイトオーダーに従って書かれます:

```
byteorder == -1: little endian
byteorder == 0: native byte order (writes a BOM mark)
byteorder == 1: big endian
```

byteorder が 0 のとき、出力文字列は常に Unicode BOM マーク (U+FEFF) で始まります。それ以外の 2 つのモードでは、先頭に BOM マークは出力されません。

If `Py_UNICODE_WIDE` is defined, a single `Py_UNICODE` value may get represented as a surrogate pair. If it is not defined, each `Py_UNICODE` values is interpreted as a UCS-2 character.

codec が例外を発生させたときは NULL を返します。

Deprecated since version 3.3, will be removed in version 3.11: 古いスタイルの `Py_UNICODE` API の一部です; `PyUnicode_AsUTF16String()` または `PyUnicode_AsEncodedString()` を使用するよう移行してください。

UTF-7 Codecs

以下は UTF-7 codec の API です:

`PyObject*` `PyUnicode_DecodeUTF7`(const char **s*, Py_ssize_t *size*, const char **errors*)

Return value: New reference. UTF-7 でエンコードされた *size* バイトの文字列 *s* 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。

`PyObject*` `PyUnicode_DecodeUTF7Stateful`(const char **s*, Py_ssize_t *size*, const char **errors*,
Py_ssize_t **consumed*)

Return value: New reference. *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す。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。

`PyObject*` `PyUnicode_EncodeUTF7`(const `Py_UNICODE` **s*, Py_ssize_t *size*, int *base64SetO*,
int *base64WhiteSpace*, const char **errors*)

Return value: New reference. 与えられたサイズの `Py_UNICODE` バッファを UTF-7 でエンコードして、Python の bytes オブジェクトとして返します。codec が例外を発生させたときは NULL を返します。

base64SetO がゼロでないとき、"Set O" 文字 (他の場合には何も特別な意味を持たない句読点) を base-64 エンコードします。*base64WhiteSpace* がゼロでないとき、空白文字を base-64 エンコードします。Python の "utf-7" codec では、両方ともゼロに設定されています。

Deprecated since version 3.3, will be removed in version 3.11: 古いスタイルの `Py_UNICODE` API の一部です; `PyUnicode_AsEncodedString()` を使用するよう移行してください。

Unicode-Escape Codecs

以下は "Unicode Escape" codec の API です:

*PyObject** **PyUnicode_DecodeUnicodeEscape**(const char *s, Py_ssize_t size, const char *errors)

Return value: New reference. Unicode-Escape でエンコードされた *size* バイトの文字列 *s* 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_AsUnicodeEscapeString**(*PyObject* *unicode)

Return value: New reference. 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。

*PyObject** **PyUnicode_EncodeUnicodeEscape**(const *Py_UNICODE* *s, Py_ssize_t size)

Return value: New reference. Unicode-Escape を使い *size* で指定された長さを持つ *Py_UNICODE* 型バッファをエンコードし、bytes オブジェクトにして返します。codec が例外を送出した場合には NULL を返します。

Deprecated since version 3.3, will be removed in version 3.11: 古いスタイルの *Py_UNICODE* API の一部です; *PyUnicode_AsUnicodeEscapeString()* を使用するように移行してください。

Raw-Unicode-Escape Codecs

以下は "Raw Unicode Escape" codec の API です:

*PyObject** **PyUnicode_DecodeRawUnicodeEscape**(const char *s, Py_ssize_t size, const char *errors)

Return value: New reference. Raw-Unicode-Escape でエンコードされた *size* バイトの文字列 *s* 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_AsRawUnicodeEscapeString**(*PyObject* *unicode)

Return value: New reference. 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。

*PyObject** **PyUnicode_EncodeRawUnicodeEscape**(const *Py_UNICODE* *s, Py_ssize_t size)

Return value: New reference. Raw-Unicode-Escape を使い *size* で指定された長さを持つ *Py_UNICODE* 型バッファをエンコードし、bytes オブジェクトにして返します。codec が例外を送出した場合には NULL を返します。

Deprecated since version 3.3, will be removed in version 3.11: 古いスタイルの *Py_UNICODE* API の一部です; *PyUnicode_AsRawUnicodeEscapeString()* または *PyUnicode_AsEncodedString()* を使用するように移行してください。

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 *s, Py_ssize_t size, const char *errors)

Return value: New reference. Latin-1 でエンコードされた size バイトの文字列 s 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)

Return value: New reference. 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。

*PyObject** **PyUnicode_EncodeLatin1**(const *Py_UNICODE* *s, Py_ssize_t size, const char *errors)

Return value: New reference. size で指定された長さを持つ *Py_UNICODE* 型バッファを Latin-1 でエンコードし、Python bytes オブジェクトにして返します。codec が例外を送出した場合には NULL を返します。

Deprecated since version 3.3, will be removed in version 3.11: 古いスタイルの *Py_UNICODE* API の一部です; *PyUnicode_AsLatin1String()* または *PyUnicode_AsEncodedString()* を使用するように移行してください。

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 *s, Py_ssize_t size, const char *errors)

Return value: New reference. ASCII でエンコードされた size バイトの文字列 s 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_AsASCIIString**(*PyObject* *unicode)

Return value: New reference. 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。

*PyObject** **PyUnicode_EncodeASCII**(const *Py_UNICODE* *s, Py_ssize_t size, const char *errors)

Return value: New reference. size で指定された長さを持つ *Py_UNICODE* 型バッファを ASCII でエンコードし、Python bytes オブジェクトにして返します。codec が例外を送出した場合には NULL を返します。

Deprecated since version 3.3, will be removed in version 3.11: 古いスタイルの *Py_UNICODE* API の一部です; *PyUnicode_AsASCIIString()* または *PyUnicode_AsEncodedString()* を使用するように移行してください。

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 *data, Py_ssize_t size, *PyObject* *mapping, const char *errors)

Return value: New reference. 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 *s* 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)

Return value: New reference. 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.

*PyObject** **PyUnicode_EncodeCharmap**(const *Py_UNICODE* *s, Py_ssize_t size, *PyObject* *mapping, const char *errors)

Return value: New reference. Encode the *Py_UNICODE* buffer of the given *size* using the given *mapping* object and return the result as a bytes object. Return NULL if an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 3.11: 古いスタイルの *Py_UNICODE* API の一部です; `PyUnicode_AsCharmapString()` または `PyUnicode_AsEncodedString()` を使用するよう移行してください。

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* *str, *PyObject* *table, const char *errors)

Return value: New reference. 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。

対応表が提供する必要があるメソッドは `__getitem__()` インタフェースだけです; 従って、辞書や

シーケンス型を使ってもうまく動作します。対応付けを行っていない (`LookupError` を起こすような) 文字序数に対しては、変換は行わず、そのままコピーします。

`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。

*PyObject** **PyUnicode_TranslateCharmap**(const *Py_UNICODE* *s, Py_ssize_t size, *PyObject*-
ject *mapping, const char *errors)

Return value: New reference. Translate a *Py_UNICODE* buffer of the given size by applying a character mapping table to it and return the resulting Unicode object. Return `NULL` when an exception was raised by the codec.

Deprecated since version 3.3, will be removed in version 3.11: 古いスタイルの *Py_UNICODE* API の一部です; *PyUnicode_Translate()*. または 汎用の codec ベースの API を使用するように移行してください。

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 *s, Py_ssize_t size, const char *errors)

Return value: New reference. MBCS でエンコードされた size バイトの文字列 s 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_DecodeMBCSStateful**(const char *s, Py_ssize_t size, const char *errors,
Py_ssize_t *consumed)

Return value: New reference. 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_AsMBCSString**(*PyObject* *unicode)

Return value: New reference. 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)

Return value: New reference. 指定されたコードページを使い Unicode オブジェクトをエンコードし、Python bytes オブジェクトを返します。codec が例外を送出した場合には `NULL` を返します。CP_ACP コードページを使い MBCS エンコーダを取得してください。

バージョン 3.3 で追加。

*PyObject** **PyUnicode_EncodeMBCS**(const *Py_UNICODE* *s, Py_ssize_t size, const char *errors)

Return value: New reference. size で指定された長さを持つ *Py_UNICODE* 型バッファを MBCS でエ

ンコードし、Python bytes オブジェクトにして返します。codec が例外を送出した場合には NULL を返します。

Deprecated since version 3.3, will be removed in version 4.0: 古いスタイルの `Py_UNICODE` API の一部です; `PyUnicode_AsMBCSString()`, `PyUnicode_EncodeCodePage()`, `PyUnicode_AsEncodedString()` のいずれかを使用するように移行してください。

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)`

Return value: New reference. 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 *s, PyObject *sep, Py_ssize_t maxsplit)`

Return value: New reference. 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。

*PyObject** `PyUnicode_Splitlines(PyObject *s, int keepend)`

Return value: New reference. Unicode 文字列を改行文字で区切り、Unicode 文字列からなるリストを返します。CRLF は一個の改行文字とみなします。keepend が 0 の場合、分割結果のリスト内に改行文字を含めません。

*PyObject** `PyUnicode_Join(PyObject *separator, PyObject *seq)`

Return value: New reference. 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 *str, PyObject *substr, Py_ssize_t start, Py_ssize_t end, int direction)`

substr が str[start:end] の末端 (direction == -1 は先頭一致、direction == 1 は末尾一致) でとマッチする場合に 1 を返し、それ以外の場合には 0 を返します。エラーが発生した時は -1 を返します。

`Py_ssize_t` `PyUnicode_Find(PyObject *str, PyObject *substr, Py_ssize_t start, Py_ssize_t end, int direction)`

str[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 *str, Py_UCS4 ch, Py_ssize_t start, Py_ssize_t end, int direction)`

`str[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。

バージョン 3.3 で追加。

バージョン 3.7 で変更: `start` and `end` are now adjusted to behave like `str[start:end]`.

`Py_ssize_t PyUnicode_Count(PyObject *str, PyObject *substr, Py_ssize_t start, Py_ssize_t end)`
`str[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 *str, PyObject *substr, PyObject *replstr, Py_ssize_t maxcount)`
Return value: New reference. `str` 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)`
 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 *uni, const char *string)`
 Unicode オブジェクト `uni` 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)`
Return value: New reference. 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` を、`left` と `right` のどちらかに対する `PyUnicode_FromObject()` が失敗したときに返します。(原文: in case the type combination is unknown)

`op` に入れられる値は、`Py_GT`, `Py_GE`, `Py_EQ`, `Py_NE`, `Py_LT`, and `Py_LE` のどれかです。

`PyObject* PyUnicode_Format(PyObject *format, PyObject *args)`
Return value: New reference. 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* *container, *PyObject* *element)

element が *container* 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。

element 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* **string)

引数 **string* をインプレースで収容 (intern) します。引数は Python Unicode 文字列オブジェクトを指すポインタ変数のアドレスでなければなりません。*string と等しい、すでに収容済みの文字列が存在する場合は、*string にその文字列を設定します (そして、元の渡された文字列オブジェクトの参照カウントをデクリメントし、すでに収容済みの文字列オブジェクトの参照カウントをインクリメントします)。そうでない場合は、*string は変更せず、そのまま収容します (そして、参照カウントをインクリメントします)。(解説: 参照カウントについては沢山説明して来ましたが、この関数は参照カウント中立 (reference-count-neutral) と考えてください; この関数を呼び出した後にオブジェクトの所有権を持っているのは、関数を呼び出す前にすでに所有権を持っていた場合かつその場合に限ります。)

*PyObject** **PyUnicode_InternFromString**(const char *v)

Return value: New reference. *PyUnicode_FromString()* と *PyUnicode_InternInPlace()* を組み合わせたもので、収容済みの新たな文字列オブジェクトを返すか、同じ値を持つ収容済みの Unicode 文字列オブジェクトに対する新たな ("所有権のある") 参照を返します。

8.3.4 タプルオブジェクト (tuple object)

PyTupleObject

この *PyTupleObject* 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**

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。

int **PyTuple_CheckExact**(*PyObject* *p)

p がタプルオブジェクトで、かつ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。

*PyObject** **PyTuple_New**(Py_ssize_t len)

Return value: New reference. サイズが *len* の新たなタプルオブジェクトを返します。失敗すると NULL を返します。

*PyObject** **PyTuple_Pack**(Py_ssize_t n, ...)

Return value: New reference. サイズが *n* の新たなタプルオブジェクトを返します。失敗すると NULL を返します。タプルの値は後続の *n* 個の Python オブジェクトを指す C 引数になります。PyTuple_Pack(2, a, b) は Py_BuildValue("(OO)", a, b) と同じです。

Py_ssize_t **PyTuple_Size**(*PyObject* *p)

タプルオブジェクトへのポインタを引数にとり、そのタプルのサイズを返します。

`Py_ssize_t PyTuple_GET_SIZE(PyObject *p)`

タプル *p* のサイズを返しますが、*p* は非 NULL でなくてはならず、タプルオブジェクトを指していないければなりません; この関数はエラーチェックを行いません。

`PyObject* PyTuple_GetItem(PyObject *p, Py_ssize_t pos)`

Return value: Borrowed reference. Return the object at position *pos* in the tuple pointed to by *p*. If *pos* is out of bounds, return NULL and set an `IndexError` exception.

`PyObject* PyTuple_GET_ITEM(PyObject *p, Py_ssize_t pos)`

Return value: Borrowed reference. `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)`

Return value: New reference. 成功すると *p* の *low* から *high* までの間を指し示すタプルのスライスを返し、失敗すると NULL を返します。Python の式 `p[low:high]` と同じです。リストの終端からのインデックスはサポートされていません。

`int PyTuple_SetItem(PyObject *p, Py_ssize_t pos, PyObject *o)`

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、変更先のインデッ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: This macro "steals" a reference to *o*, and, unlike `PyTuple_SetItem()`, does *not* discard a reference to any item that is being replaced; any reference in the tuple at position *pos* will be leaked.

`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。

`int PyTuple_ClearFreeList()`

free list をクリアします。解放された要素数を返します。

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。

*PyObject** `PyStructSequence_NewType(PyStructSequence_Desc *desc)`

Return value: New reference. 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。

`void PyStructSequence_InitType(PyObject *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(PyObject *type, PyStructSequence_Desc *desc)`

`PyStructSequence_InitType` と同じですが、成功した場合には 0 を、失敗した場合には -1 を返します。

バージョン 3.4 で追加。

`PyStructSequence_Desc`

生成する struct sequence 型のメタデータを保持します。

フィールド	C の型	意味
<code>name</code>	<code>const char *</code>	生成する struct sequence 型の名前
<code>doc</code>	<code>const char *</code>	生成する型の docstring へのポインタ、または省略する場合は NULL
<code>fields</code>	<code>PyStructSequence_Field *</code>	新しい型のフィールド名を格納した NULL 終端された配列へのポインタ
<code>n_in_sequence</code>	<code>int</code>	Python 側で可視となるフィールドの数 (もしタプルとして使用する場合)

`PyStructSequence_Field`

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.

フィールド	C の型	意味
<code>name</code>	<code>const char *</code>	フィールドの名前。もし名前づけされたフィールドのリストの終端を表す場合は NULL。名前がないままにする場合は、 <code>PyStructSequence_UnnamedField</code> を設定します
<code>doc</code>	<code>const char *</code>	フィールドの docstring、省略する場合は NULL

char* PyStructSequence_UnnamedField

フィールド名を名前がないままするための特殊な値。

PyObject* PyStructSequence_New(PyTypeObject *type)

Return value: New reference. 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。

PyObject* PyStructSequence_GetItem(PyObject *p, Py_ssize_t pos)

Return value: Borrowed reference. p の指す struct sequence 内の、位置 pos にあるオブジェクトを返します。境界チェックを行いません。

PyObject* PyStructSequence_GET_ITEM(PyObject *p, Py_ssize_t pos)

Return value: Borrowed reference. PyStructSequence_GetItem() と同等のマクロです。

void PyStructSequence_SetItem(PyObject *p, Py_ssize_t pos, PyObject *o)

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)

PyStructSequence_SetItem() と同等のマクロ。

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

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

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。

int PyList_CheckExact(PyObject *p)

p がリスト型オブジェクトであり、かつリスト型のサブクラスのインスタンスでない場合に真を返します。

PyObject* PyList_New(Py_ssize_t len)

Return value: New reference. 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: `len` が 0 より大きいとき、返されるリストオブジェクトの要素には NULL がセットされています。なので、`PyList_SetItem()` で本当にオブジェクトをセットするまでは、Python コードにこのオブジェクトを渡したり、`PySequence_SetItem()` のような抽象 API を利用してはいけません。

`Py_ssize_t PyList_Size(PyObject *list)`

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)`

マクロ形式でできた `PyList_Size()` で、エラーチェックをしません。

`PyObject* PyList_GetItem(PyObject *list, Py_ssize_t index)`

Return value: Borrowed reference. `list` の指すリストオブジェクト内の、位置 `index` にあるオブジェクトを返します。位置は非負である必要があり、リスト終端からのインデックスはサポートされていません。`index` が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。

`PyObject* PyList_GET_ITEM(PyObject *list, Py_ssize_t i)`

Return value: Borrowed reference. マクロ形式でできた `PyList_GetItem()` で、エラーチェックをしません。

`int PyList_SetItem(PyObject *list, Py_ssize_t index, PyObject *item)`

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、変更先のインデッ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)`

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)`

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)`

Return value: New reference. *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)`

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)`

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)`

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)`

Return value: New reference. *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。

`int PyList_ClearFreeList()`

free list をクリアします。解放された要素数を返します。

バージョン 3.3 で追加。

8.4 Container オブジェクト

8.4.1 辞書オブジェクト (dictionary object)

`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`

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)`

引数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の場合に真を返します。

`int PyDict_CheckExact(PyObject *p)`

p が辞書オブジェクトであり、かつ辞書型のサブクラスのインスタンスでない場合に真を返します。

`PyObject* PyDict_New()`

Return value: New reference. 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)

Return value: *New reference.* あるマップ型オブジェクトに対して、読み出し専用で制限された `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)

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)

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)

Return value: *New reference.* *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)

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)

Insert *val* into the dictionary *p* using *key* as a key. *key* should be a `const char*`. The key object is created using `PyUnicode_FromString(key)`. Return 0 on success or -1 on failure. This function *does not* steal a reference to *val*.

int PyDict_DelItem(*PyObject* *p, *PyObject* *key)

Remove the entry in dictionary *p* with key *key*. *key* must be hashable; if it isn't, `TypeError` is raised. If *key* is not in the dictionary, `KeyError` is raised. Return 0 on success or -1 on failure.

int PyDict_DelItemString(*PyObject* *p, const char *key)

辞書 *p* から文字列 *key* をキーとするエントリを除去します。*key* が辞書になれば、`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。

*PyObject** PyDict_GetItem(*PyObject* *p, *PyObject* *key)

Return value: *Borrowed reference.* 辞書 *p* 内で *key* をキーとするオブジェクトを返します。キー *key* が存在しない場合には `NULL` を返しますが、例外をセット **しません**。

`__hash__()` メソッドや `__eq__()` メソッド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、代わりに `PyDict_GetItemWithError()` を使ってください。

*PyObject** PyDict_GetItemWithError(*PyObject* *p, *PyObject* *key)

Return value: *Borrowed reference.* `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)

Return value: *Borrowed reference.* This is the same as `PyDict_GetItem()`, but *key* is specified as a `const char*`, rather than a *PyObject**.

`__hash__()` メソッドや `__eq__()` メソッドの呼び出し中や、一時的な文字列オブジェクトの作成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、代わりに `PyDict_GetItemWithError()` を使ってください。

`PyObject*` **PyDict_SetDefault**(`PyObject *`*p*, `PyObject *`*key*, `PyObject *`*defaultobj*)

Return value: Borrowed reference. 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。

バージョン 3.4 で追加。

`PyObject*` **PyDict_Items**(`PyObject *`*p*)

Return value: New reference. 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*)

Return value: New reference. 辞書内の全てのキーが入った `PyListObject` を返します。

`PyObject*` **PyDict_Values**(`PyObject *`*p*)

Return value: New reference. 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*)

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*)

Iterate over all key-value pairs in the dictionary *p*. The `Py_ssize_t` referred to by *ppos* must be initialized to 0 prior to the first call to this function to start the iteration; the function returns true for each pair in the dictionary, and false once all pairs have been reported. The parameters *pkey* and *pvalue* should either point to `PyObject*` variables that will be filled in with each key and value, respectively, or may be NULL. Any references returned through them are borrowed. *ppos* should not be altered during iteration. Its value represents offsets within the internal dictionary structure, and since the structure is sparse, the offsets are not consecutive.

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;
```

(次のページに続く)

(前のページからの続き)

```

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)`

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)`

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)`

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_ClearFreeList()`

free list をクリアします。解放された要素数を返します。

バージョン 3.3 で追加.

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。

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、非公開で変更される可能性があると考えて下さい。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`

この `PyTypeObject` のインスタンスは、Python の `set` 型を表します。

`PyTypeObject` `PyFrozenSet_Type`

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タ可能な 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。

`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。

`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。

`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。

`int PyFrozenSet_CheckExact(PyObject *p)`

`p` が `frozenset` 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。

`PyObject* PySet_New(PyObject *iterable)`

Return value: New reference. `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` がイテレータ可能でない場合は、`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)`

Return value: New reference. *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)`

`set` や `frozenset` のオブジェクトの長さを返します。`len(anyset)` と同じです。*anyset* が `set`、`frozenset` およびそのサブタイプのオブジェクトでない場合は、`PyExc_SystemError` を送出します。

`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)`

見つかったら 1 を、見つからなかったら 0 を、エラーが発生した場合は -1 を返します。Python の `__contains__()` メソッドと違って、この関数は非ハッシュ `set` を一時的な `frozenset` に自動で変換しません。*key* がハッシュ可能で無い場合、`TypeError` を送出します。*anyset* が `set`、`frozenset` 及びそのサブタイプのオブジェクトで無い場合は `PyExc_SystemError` を送出します。

`int PySet_Add(PyObject *set, PyObject *key)`

`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)`

key が見つかって、値を削除したら 1 を返します。見つからなかったら (何もせずに) 0 を返します。エラーが発生した場合は -1 を返します。*key* が無くても `KeyError` を送出しません。*key* がハッシュ可能でない場合は `TypeError` を送出します。Python の `discard()` メソッドと違って、この関数は非ハッシュ `set` を一時的な `frozenset` に変換しません。*set* が `set` かそのサブタイプのインスタンスでないときは、`PyExc_SystemError` を送出します。

`PyObject* PySet_Pop(PyObject *set)`

Return value: New reference. *set* 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)`

set を空にします。

`int PySet_ClearFreeList()`

free list をクリアします。解放された要素数を返します。

バージョン 3.3 で追加。

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。

`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` 型なら) なら `true` を返します。引数は `NULL` であってはなりません。

`PyObject* PyFunction_New(PyObject *code, PyObject *globals)`

Return value: New reference. 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__` は関数名と同じ値に設定されます。

`PyObject* PyFunction_NewWithQualName(PyObject *code, PyObject *globals, PyObject *qualname)`

Return value: New reference. `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 `__name__` 属性と同じ値がセットされます。

バージョン 3.3 で追加。

`PyObject* PyFunction_GetCode(PyObject *op)`

Return value: Borrowed reference. 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)`

Return value: Borrowed reference. 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)`

Return value: Borrowed reference. 関数オブジェクト `op` の `__module__` 属性を返します。これには普通はモジュール名の文字列が入っていますが、Python コードから他のオブジェクトをセットされることもあります。

`PyObject* PyFunction_GetDefaults(PyObject *op)`

Return value: Borrowed reference. 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。

`PyObject* PyFunction_GetClosure(PyObject *op)`

Return value: Borrowed reference. 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)`

Return value: Borrowed reference. 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。

8.5.2 インスタンスメソッドオブジェクト (Instance Method Objects)

インスタンスメソッドとは `PyCFunction` のためのラッパーであり、`PyCFunction` をクラスオブジェクトにバインドするための新しいやり方です。これは以前の `PyMethod_New(func, NULL, class)` の呼び出しを置き換えます。

`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。

`PyObject* PyInstanceMethod_New(PyObject *func)`

Return value: New reference. 任意の呼び出し可能オブジェクト *func* 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ときに呼び出されるオブジェクトです。

`PyObject* PyInstanceMethod_Function(PyObject *im)`

Return value: Borrowed reference. 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)

Return value: Borrowed reference. *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。

*PyObject** PyMethod_New(*PyObject* *func, *PyObject* *self)

Return value: New reference. 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)

Return value: Borrowed reference. 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)

Return value: Borrowed reference. *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)

Return value: Borrowed reference. 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)

Return value: Borrowed reference. *PyMethod_Self()* のマクロ版で、エラーチェックを行いません。

int PyMethod_ClearFreeList()

free list をクリアします。解放された要素数を返します。

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。

PyObject* 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(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。

PyObject* PyCell_New(PyObject *ob)

Return value: New reference. 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)

Return value: New reference. *cell* の内容を返します。

PyObject* PyCell_GET(PyObject *cell)

Return value: Borrowed reference. *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)

セルオブジェクト *cell* の内容を *value* に設定します。この関数は現在のセルの全ての内容に対する参照を解放します。*value* は NULL でもかまいません。*cell* は非 NULL でなければなりません。*cell* がセルオブジェクトでない場合、-1 を返します。成功すると 0 を返します。

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。

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

これは Python の code 型を表現する *PyTypeObject* のインスタンスです。

int *PyCode_Check*(*PyObject* *co)

co が code オブジェクトのときに真を返します。

int *PyCode_GetNumFree*(*PyCodeObject* *co)

co 内の自由変数 (free variables) の数を返します。

*PyCodeObject** *PyCode_New*(int argcount, int kwnonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, int firstlineno, *PyObject* *notab)

Return value: New reference. 新しいコードオブジェクトを返します。フレームを作成するためにダミーのコードオブジェクトが必要な場合は、代わりに *PyCode_NewEmpty()* を利用してください。バイトコードは頻繁に変更されるため、*PyCode_New()* を直接呼び出すと、Python の詳細バージョンに依存してしまうことがあります。

*PyCodeObject** *PyCode_NewWithPosOnlyArgs*(int argcount, int posonlyargcount, int kwnonlyargcount, int nlocals, int stacksize, int flags, *PyObject* *code, *PyObject* *consts, *PyObject* *names, *PyObject* *varnames, *PyObject* *freevars, *PyObject* *cellvars, *PyObject* *filename, *PyObject* *name, int firstlineno, *PyObject* *notab)

Return value: New reference. *PyCode_New()* に似ていますが、位置専用引数のための "posonlyargcount" が追加されています。

バージョン 3.8 で追加。

*PyCodeObject** *PyCode_NewEmpty*(const char *filename, const char *funcname, int firstlineno)

Return value: New reference. 新しい空のコードオブジェクトを、指定されたファイル名、関数名、開始行番号で作成します。返されたコードオブジェクトに対しての *exec()* や *eval()* は許されていません。

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*)
Return value: New reference. 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*)

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; このメソッドの戻り値は、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*)

Return value: New reference. `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。

ハンドラは `PyObject (*)(PyObject *path, void *userData)` 型の関数で、*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。

バージョン 3.8 で追加。

```
int PyFile_WriteObject(PyObject *obj, PyObject *p, int flags)
```

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)
```

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
```

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。

```
int PyModule_CheckExact(PyObject *p)
```

`p` がモジュールオブジェクトで、かつモジュールオブジェクトのサブタイプでないときに真を返します。
`PyModule_Type`.

```
PyObject* PyModule_NewObject(PyObject *name)
```

Return value: New reference. `__name__` 属性に `name` が設定された新しいモジュールオブジェクトを返します。モジュールの `__name__`, `__doc__`, `__package__`, `__loader__` 属性に値が入っています (`__name__` 以外は全て None です); `__file__` 属性に値を入れるのは呼び出し側の責任です。

バージョン 3.3 で追加。

バージョン 3.4 で変更: `__package__` と `__loader__` は None に設定されます。

*PyObject** PyModule_New(const char *name)

Return value: New reference. *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)

Return value: Borrowed reference. 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)

Return value: New reference. module の `__name__` の値を返します。モジュールがこの属性を提供していない場合や文字列型でない場合、`SystemError` を送出して NULL を返します。

バージョン 3.3 で追加。

const char* PyModule_GetName(*PyObject* *module)

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)

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)

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)

Return value: New reference. module の `__file__` 属性をもとに module がロードされたもののファイル名を返します。もしファイル名が定義されていない場合や、Unicode 文字列ではない場合、`SystemError` を発生させて NULL を返します。それ以外の場合は Unicode オブジェクトへの参照を返します。

バージョン 3.2 で追加。

const char* PyModule_GetFilename(*PyObject* *module)

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` をご覧ください。

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。

PyModuleDef

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・インタプリタ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**

A traversal function to call during GC traversal of the module object, or NULL if not needed. This function may be called before module state is allocated (*PyModule_GetState()* may return NULL), and before the *Py_mod_exec* function is executed.

inquiry **m_clear**

A clear function to call during GC clearing of the module object, or NULL if not needed. This function may be called before module state is allocated (*PyModule_GetState()* may return NULL), and before the *Py_mod_exec* function is executed.

freefunc **m_free**

A function to call during deallocation of the module object, or NULL if not needed. This function may be called before module state is allocated (*PyModule_GetState()* may return NULL), and before the *Py_mod_exec* function is executed.

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)

Return value: New reference. *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。これは *PyModule_Create2()* の *module_api_version* に PYTHON_API_VERSION を設定したときのよに振る舞います。

*PyObject** **PyModule_Create2**(*PyModuleDef* *def, int *module_api_version*)

Return value: New reference. 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。

注釈: ほとんどの場合、この関数ではなく *PyModule_Create()* を利用すべきです。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_AddObject()* 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

拡張を直接生成するもう 1 つのやり方は、”多段階初期化”を要求する方法です。この方法で作られる拡張モジュールは、より Python モジュールに近い振る舞いをします: 初期化処理は、モジュールオブジェクトを生成する **生成段階** とメンバを所属させる **実行段階** に分割されます。この区別はクラスの `__new__()` メソッドと `__init__()` メソッドに似ています。

一段階初期化で生成されたモジュールと違い、多段階初期化で生成されたモジュールはシングルトンではありません: `sys.modules` のエントリが削除されモジュールが再インポートされた場合、新しいモジュールオブジェクトが生成され、古いモジュールは Python モジュールと同じように通常のガベージコレクションで処理されることになります。デフォルトでは、同じ定義から作られた複数のモジュールは独立であるべきです: あるインスタンスに加えた変更は別のインスタンスに影響しません。これは、(例えば `PyModule_GetState()` を使って取得できる) 全ての状態や、(モジュールの `__dict__` や `PyType_FromSpec()` で生成された個々のクラスのような) モジュールに所属するものは、特定のモジュールオブジェクト特有のものであるべきということです。

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** 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なりません:

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

Return value: Borrowed reference. 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。

バージョン 3.5 で追加。

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:

`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。

バージョン 3.5 で追加。

`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なりません:

```
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う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様です: 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。

多段階初期化についてより詳しくは **PEP 489** をご覧ください。

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)
```

Return value: New reference. *module* 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。この関数は、`PyModule_FromDefAndSpec2()` 関数の `module_api_version` に `PYTHON_API_VERSION` を指定した時とおなじようにふるまいます。

バージョン 3.5 で追加.

```
PyObject * PyModule_FromDefAndSpec2(PyModuleDef *def, PyObject *spec, int module_api_version)
```

Return value: New reference. API バージョンを *module_api_version* として、*module* 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。

注釈: ほとんどの場合、この関数ではなく *PyModule_FromDefAndSpec()* を利用すべきです。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。

バージョン 3.5 で追加。

```
int PyModule_ExecDef(PyObject *module, PyModuleDef *def)
    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。
```

バージョン 3.5 で追加。

```
int PyModule_SetDocString(PyObject *module, const char *docstring)
    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。
```

バージョン 3.5 で追加。

```
int PyModule_AddFunctions(PyObject *module, PyMethodDef *functions)
    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。
```

バージョン 3.5 で追加。

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_AddObject(PyObject *module, const char *name, PyObject *value)
    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。この関数は成功したときには value への参照を盗みます。エラーのときには -1 を、
    成功したときには 0 を返します。
```

注釈: Unlike other functions that steal references, *PyModule_AddObject()* only decrements the reference count of *value* **on success**.

This means that its return value must be checked, and calling code must *Py_DECREF()* *value* manually on error. Example usage:

```
Py_INCREF(spam);
if (PyModule_AddObject(module, "spam", spam) < 0) {
    Py_DECREF(module);
    Py_DECREF(spam);
    return NULL;
}
```

int **PyModule_AddIntConstant**(*PyObject* *module, const char *name, long value)

module に整数定数を *name* として追加します。この便宜関数はモジュールの初期化関数から利用されています。エラーのときには -1 を、成功したときには 0 を返します。

int **PyModule_AddStringConstant**(*PyObject* *module, const char *name, const char *value)

module に文字列定数を *name* として追加します。この便利関数はモジュールの初期化関数から利用されています。文字列 *value* は NULL 終端されていなければなりません。エラーのときには -1 を、成功したときには 0 を返します。

int **PyModule_AddIntMacro**(*PyObject* *module, macro)

module に int 定数を追加します。名前と値は *macro* から取得されます。例えば、`PyModule_AddIntMacro(module, AF_INET)` とすると、*AF_INET* という名前の int 型定数を *AF_INET* の値で *module* に追加します。エラー時には -1 を、成功時には 0 を返します。

int **PyModule_AddStringMacro**(*PyObject* *module, macro)

文字列定数を *module* に追加します。

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)

Return value: Borrowed reference. 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)

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).

成功したら 0 を、失敗したら -1 を返します。

バージョン 3.3 で追加.

`int PyState_RemoveModule(PyModuleDef *def)`

def から作られたモジュールオブジェクトをインタプリタ state から削除します。成功したら 0 を、失敗したら -1 を返します。

バージョン 3.3 で追加.

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センチネル値 (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`

`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(op)`

`PySeqIter_Type` の型が *op* のときに真を返します。

*PyObject** `PySeqIter_New(PyObject *seq)`

Return value: *New reference.* 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`

`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(op)`

op の型が `PyCallIter_Type` のときに真を返します。

*PyObject** `PyCallIter_New(PyObject *callable, PyObject *sentinel)`

Return value: *New reference.* 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**

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)

Return value: New reference.

*PyObject** **PyDescr_NewMember**(*PyTypeObject* *type, struct *PyMemberDef* *meth)

Return value: New reference.

*PyObject** **PyDescr_NewMethod**(*PyTypeObject* *type, struct *PyMethodDef* *meth)

Return value: New reference.

*PyObject** **PyDescr_NewWrapper**(*PyTypeObject* *type, struct wrapperbase *wrapper,
void *wrapped)

Return value: New reference.

*PyObject** **PyDescr_NewClassMethod**(*PyTypeObject* *type, *PyMethodDef* *method)

Return value: New reference.

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真を、メソッドデスクリプタの場合には偽を返します。 *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* *)

Return value: New reference.

8.6.5 スライスオブジェクト (slice object)

PyTypeObject **PySlice_Type**

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。

*PyObject** **PySlice_New**(*PyObject* *start, *PyObject* *stop, *PyObject* *step)

Return value: New reference. 指定した値から新たなスライスオブジェクトを返します。パラメータ *start*, *stop*, および *step* はスライスオブジェクトにおける同名の属性として用いられます。これらの値はいずれも NULL にでき、対応する値には None が使われます。新たなオブジェクトをアロケーションできない場合には NULL を返します。

int **PySlice_GetIndices**(*PyObject* *slice, Py_ssize_t length, Py_ssize_t *start,
Py_ssize_t *stop, Py_ssize_t *step)

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。

成功のときには 0 を、エラーのときには例外をセットせずに -1 を返します (ただし、指定インデクスのいずれかが None ではなく、かつ整数に変換できなかった場合を除きます。この場合、-1 を返して例外をセットします)。

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)`
PySlice_GetIndicesEx() 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。

成功のときには 0 を、エラーのときには例外をセットして -1 を返します。

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)`

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 on error, 0 on success.

バージョン 3.6.1 で追加.

`Py_ssize_t PySlice_AdjustIndices(Py_ssize_t length, Py_ssize_t *start, Py_ssize_t *stop, Py_ssize_t step)`

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.

バージョン 3.6.1 で追加.

8.6.6 Ellipsis オブジェクト

PyObject *Py_Ellipsis

Python における `Ellipsis` オブジェクトです。このオブジェクトはメソッドを持ちません。参照カウンターの点では、他のオブジェクトと同様に扱う必要があります。*Py_None* のように、これもシングルトンオブジェクトです。

8.6.7 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)

Return value: New reference. 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、エクスポートの分別にもとづいて読み書きが可能となります。

PyObject *PyMemoryView_FromMemory(char *mem, Py_ssize_t size, int flags)

Return value: New reference. *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。

バージョン 3.3 で追加.

PyObject *PyMemoryView_FromBuffer(*Py_buffer* *view)

Return value: New reference. *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)

Return value: New reference. *buffer* インタ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。

`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。

`Py_buffer *PyMemoryView_GET_BUFFER(PyObject *mview)`

書きだされたバッファの memoryview のプライベート コピーに、ポインタを返します。*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。

`Py_buffer *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。

8.6.8 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(ob)`

ob が参照オブジェクトかプロキシオブジェクトの場合に真を返します。

`int PyWeakref_CheckRef(ob)`

ob が参照オブジェクトの場合に真を返します。

`int PyWeakref_CheckProxy(ob)`

ob がプロキシオブジェクトの場合に真を返します。

`PyObject* PyWeakref_NewRef(PyObject *ob, PyObject *callback)`

Return value: *New reference.* *ob* に対する弱参照オブジェクトを返します。この関数は常に新たな参照を返しますが、必ずしも新たなオブジェクトを作る保証はありません; 既存の参照オブジェクトが返されることもあります。第二のパラメタ *callback* は呼び出し可能オブジェクトで、*ob* がガーベジコレクションされた際に通知を受け取ります; *callback* は弱参照オブジェクト自体を単一のパラメタとして受け取ります。*callback* は None や NULL にしてもかまいません。*ob* が弱参照できないオブジェクトの場合や、*callback* が呼び出し可能オブジェクト、None、NULL のいずれでもない場合は、NULL を返して `TypeError` を送出します。

`PyObject* PyWeakref_NewProxy(PyObject *ob, PyObject *callback)`

Return value: *New reference.* *ob* に対する弱参照プロキシオブジェクトを返します。この関数は常に新たな参照を返しますが、必ずしも新たなオブジェクトを作る保証はありません。既存のプロキシオブジェクトが返されることもあります。第二のパラメタ *callback* は呼び出し可能オブジェクトで、*ob* がガーベジコレクションされた際に通知を受け取ります。*callback* は弱参照オブジェクト自体を単一のパラメタとして受け取ります。*callback* は None や NULL にしてもかまいません。*ob* が弱参照できないオ

プロジェクトの場合や、*callback* が呼び出し可能オブジェクト、`None`、`NULL` のいずれでもない場合は、`NULL` を返して `TypeError` を送出します。

*PyObject** `PyWeakref_GetObject(PyObject *ref)`

Return value: Borrowed reference. 弱参照 *ref* が参照しているオブジェクトを返します。被参照オブジェクトがすでに存続していない場合、`Py_None` を返します。

注釈: この関数は参照先オブジェクトの **借り物の参照** を返します。そのため、そのオブジェクトを利用している間そのオブジェクトが破棄されないことが判っている場合を除き、常に `Py_INCREF()` を呼び出すべきです。

*PyObject** `PyWeakref_GET_OBJECT(PyObject *ref)`

Return value: Borrowed reference. `PyWeakref_GetObject()` に似ていますが、マクロで実装されていて、エラーチェックを行いません。

8.6.9 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。

バージョン 3.1 で追加。

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。

PyCapsule_Destructor

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* だったときに `true` を返します。

*PyObject** `PyCapsule_New(void *pointer, const char *name, PyCapsule_Destructor destructor)`

Return value: New reference. *pointer* を格納する *PyCapsule* を作成します。*pointer* 引数は `NULL` であってはいけません。

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)`

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。

`name` 引数はカプセルに保存されている名前と正確に一致しなければなりません。もしカプセルに格納されている `name` が NULL なら、この関数の `name` 引数も同じく NULL でなければなりません。Python は C 言語の `strcmp()` を使ってこの `name` を比較します。

`PyCapsule_Destructor PyCapsule_GetDestructor(PyObject *capsule)`

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 がエラーを指してい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)`

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 がエラーを指してい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)`

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 がエラーを指してい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)`

モジュールのカプセル属性から C オブジェクトへのポインタをインポートします。 `name` 引数はその属性の完全名を `module.attribute` のように指定しなければなりません。カプセルに格納されている `name` はこの文字列に正確に一致しなければなりません。 `no_block` が真の時、モジュールを (`PyImport_ImportModuleNoBlock()` を使って) ブロックせずにインポートします。 `no_block` が偽の時、モジュールは (`PyImport_ImportModule()` を使って) 通常の方法でインポートされます。

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。

`int PyCapsule_IsValid(PyObject *capsule, const char *name)`

`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)

言い換えると、*PyCapsule_IsValid()* が真を返す場合、全てのアクセッサ (*PyCapsule_Get()* で始まる全ての関数) が成功することが保証されます。

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*)

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*)

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*)

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*)

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。

8.6.10 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。

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* が NULL であってはなりません。

int *PyGen_CheckExact*(*PyObject* **ob*)

ob の型が *PyGen_Type* のときに true を返します。*ob* は NULL であってはいけません。

***PyObject** *PyGen_New*(*PyFrameObject* **frame*)**

Return value: New reference. *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*)

Return value: New reference. *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

バージョン 3.5 で追加。

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。

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 の型が *PyCoro_Type* のときに true を返します。*ob* は NULL であってはいけません。

*PyObject** **PyCoro_New**(*PyFrameObject* **frame*, *PyObject* **name*, *PyObject* **qualname*)

Return value: New reference. *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

注釈: バージョン 3.7.1 で変更: 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。

バージョン 3.7 で追加。

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。

PyContext

contextvars.Context オブジェクトを表現するための C 構造体。

PyContextVar

contextvars.ContextVar オブジェクトを表現するための C 構造体。

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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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。

`int PyContext_ClearFreeList()`

コンテキスト変数のフリーリストをクリアします。解放された要素数を返します。この関数は常に成功します。

コンテキスト変数の関数:

`PyObject *PyContextVar_New(const char *name, PyObject *def)`

Return value: *New reference.* 新しい“ContextVar”オブジェクトをさくせいします。*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)`

Return value: *New reference.* 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。

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。

バージョン 3.7 で追加。

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。

`int PyDate_CheckExact(PyObject *ob)`

ob が `PyDateTime_DateType` 型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。

`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。

`int PyDateTime_CheckExact(PyObject *ob)`

ob が `PyDateTime_DateTimeType` 型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。

`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。

`int PyTime_CheckExact(PyObject *ob)`

ob が `PyDateTime_TimeType` 型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。

`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。

`int PyDelta_CheckExact(PyObject *ob)`

ob が `PyDateTime_DeltaType` 型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。

`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。

`int PyTZInfo_CheckExact(PyObject *ob)`

ob が `PyDateTime_TZInfoType` 型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。

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)

Return value: New reference. 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)

Return value: New reference. 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)

Return value: New reference. 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。

バージョン 3.6 で追加.

*PyObject** **PyTime_FromTime**(int hour, int minute, int second, int usecond)

Return value: New reference. 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)

Return value: New reference. 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。

バージョン 3.6 で追加.

*PyObject** **PyDelta_FromDSU**(int days, int seconds, int useconds)

Return value: New reference. 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**(PyDateTime_DeltaType* offset)

Return value: New reference. *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。

バージョン 3.7 で追加.

*PyObject** **PyTimeZone_FromOffsetAndName**(PyDateTime_DeltaType* offset, PyUnicode* name)

Return value: New reference. *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。

バージョン 3.7 で追加.

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)`

Return the fold, as an int from 0 through 1.

バージョン 3.6 で追加.

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)`

Return the fold, as an int from 0 through 1.

バージョン 3.6 で追加.

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。

バージョン 3.3 で追加.

`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。

バージョン 3.3 で追加.

`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。

バージョン 3.3 で追加.

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)`

Return value: New reference. `datetime.datetime.fromtimestamp()` に渡すのに適した引数タプルから新たな `datetime.datetime` オブジェクトを生成して返します。

*PyObject** `PyDate_FromTimestamp(PyObject *args)`

Return value: New reference. `datetime.date.fromtimestamp()` に渡すのに適した引数タプルから新たな `datetime.date` オブジェクトを生成して返します。

初期化 (INITIALIZATION)、終了処理 (FINALIZATION)、スレッド

Python 初期化設定 も参照してください。

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: 次の関数は *Py_Initialize()* より前に 呼び出すべき ではありません: *Py_EncodeLocale()*, *Py_GetPath()*, *Py_GetPrefix()*, *Py_GetExecPrefix()*, *Py_GetProgramFullPath()*, *Py_GetPythonHome()*, *Py_GetProgramName()*, *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*

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。

int *Py_DebugFlag*

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。

int Py_DontWriteBytecodeFlag

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。

int Py_FrozenFlag

`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_importlib` 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。

int Py_HashRandomizationFlag

`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。

int Py_IgnoreEnvironmentFlag

全ての `PYTHON*` 環境変数を見捨てます。例えば、`PYTHONPATH` や `PYTHONHOME` などです。

`-E` オプションと `-I` オプションで設定します。

int Py_InspectFlag

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。

int Py_InteractiveFlag

`-i` オプションで設定します。

int Py_IsolatedFlag

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。

バージョン 3.4 で追加。

int Py_LegacyWindowsFSEncodingFlag

フラグがゼロでない場合は、ファイルシステムエンコーディングに UTF-8 エンコーディングの代わりに `mbcs` エンコーディングが使われます。

`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。

利用可能な環境: Windows。

int `Py_LegacyWindowsStdioFlag`

フラグがゼロでない場合、`WindowsConsoleIO` の代わりに `io.FileIO` を `sys` の標準ストリームとして使います。

`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。

利用可能な環境: Windows。

int `Py_NoSiteFlag`

`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。

int `Py_NoUserSiteDirectory`

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。

int `Py_OptimizeFlag`

`-O` オプションと `PYTHONOPTIMIZE` 環境変数で設定します。

int `Py_QuietFlag`

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。

バージョン 3.2 で追加。

int `Py_UnbufferedStdioFlag`

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。

int `Py_VerboseFlag`

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。

9.3 インタプリタの初期化と終了処理

`void Py_Initialize()`

Python インタ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。

この関数はロード済みモジュールのテーブル (`sys.modules`) を初期化し、基盤となるモジュール群、`builtins`, `__main__`, `sys` を生成します。また、モジュール検索パス (`sys.path`) も初期化します。`sys.argv` の設定は行いません。設定するには、*PySys_SetArgvEx()* を使ってください。この関数を (*Py_FinalizeEx()* を呼ばずに) 再度呼び出しても何も行いません。戻り値はありません。初期化が失敗すれば、それは致命的なエラーです。

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)`

initsigs に 1 を指定した場合、この関数は *Py_Initialize()* と同じように動作します。*initsigs* に 0 を指定した場合、初期化時のシグナルハンドラの登録をスキップすることができ、これは Python の埋め込みで便利でしょう。

`int Py_IsInitialized()`

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()`

Py_Initialize() とそれ以後の Python/C API 関数で行った全ての初期化処理を取り消し、最後の *Py_Initialize()* 呼び出し以後に Python インタプリタが生成した全てのサブインタプリタ (sub-interpreter, 下記の *Py_NewInterpreter()* を参照) を消去します。理想的な状況では、この関数によって Python インタプリタが確保したメモリは全て解放されます。この関数を (*Py_Initialize()* を呼ばずに) 再度呼び出しても何も行いません。通常は返り値は 0 です。終了処理中 (バッファリングされたデータの書き出し) のエラーがあった場合は -1 が返されます。

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。

バグおよび注意事項: モジュールやモジュール内のオブジェクトはランダムな順番で削除されます。このため、他のオブジェクト (関数オブジェクトも含みます) やモジュールに依存するデストラクタ (`__del__()` メソッド) が失敗してしまうことがあります。動的にロードされるようになっている拡張モジュールが Python によってロードされていた場合、アンロードされません。Python が確保した

メモリがわずかながら解放されないかもしれません (メモリリークを発見したら、どうか報告してください)。オブジェクト間の循環参照に捕捉されているメモリは解放されないことがあります。拡張モジュールが確保したメモリは解放されないことがあります。拡張モジュールによっては、初期化ルーチンを 2 度以上呼び出すと正しく動作しないことがあります。こうした状況は、`Py_Initialize()` や `Py_FinalizeEx()` を 2 度以上呼び出すと起こり得ます。

引数無しで 監査イベント `cpython._PySys_ClearAuditHooks` を送出します。

バージョン 3.6 で追加。

`void Py_Finalize()`

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)`

もし `Py_Initialize()` が呼ばれるなら、この関数はその前に呼ばなければなりません。標準の IO において、どのエンコーディングおよびどんなエラー処理を使うかを、`str.encode()` と同様の意味で指定します。

これは、環境変数が働かない時に `PYTHONIOENCODING` の値を上書きし、埋め込みコードが IO エンコーディングをコントロールできるようにします。

`encoding` と `errors` のどちらかまたは両方を `NULL` にすることで、`PYTHONIOENCODING` とデフォルト値のどちらかまたは両方を使うことができます (他の設定に依存します)。

この設定 (あるいは他の設定) に関わらず、`sys.stderr` は常に "backslashreplace" エラーハンドラを使うことに注意してください。

`Py_FinalizeEx()` を呼び出した場合は、`Py_Initialize()` を呼び出す前に、この関数を再度呼び出す必要があるでしょう。

成功したら 0 を、エラーの場合は 0 でない値を返します (例えば、インタプリタが初期化された後に、この関数が呼び出された場合)。

バージョン 3.4 で追加。

`void Py_SetProgramName(const wchar_t *name)`

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ない、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。

バイト文字列を `wchar_t*` 文字列にデコードするには `Py_DecodeLocale()` を使ってください。

wchar* **Py_GetProgramName()**

`Py_SetProgramName()` で設定されたプログラム名か、デフォルトのプログラム名を返します。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。

wchar_t* **Py_GetPrefix()**

プラットフォーム非依存のファイル群がインストールされている場所である *prefix* を返します。この値は `Py_SetProgramName()` でセットされたプログラム名やいくつかの環境変数をもとに、数々の複雑な規則から導出されます。例えば、プログラム名が `'/usr/local/bin/python'` の場合、*prefix* は `'/usr/local'` になります。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。この値はトップレベルの `Makefile` に指定されている変数 `prefix` や、ビルド値に `configure` スクリプトに指定した `--prefix` 引数に対応しています。この値は Python コードからは `sys.prefix` として利用できます。これは Unix でのみ有用です。次に説明する関数も参照してください。

wchar_t* **Py_GetExecPrefix()**

プラットフォーム **依存** のファイルがインストールされている場所である *exec-prefix* を返します。この値は `Py_SetProgramName()` でセットされたプログラム名やいくつかの環境変数をもとに、数々の複雑な規則から導出されます。例えば、プログラム名が `'/usr/local/bin/python'` の場合、*exec-prefix* は `'/usr/local'` になります。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。この値はトップレベルの `Makefile` に指定されている変数 `exec_prefix` や、ビルド値に `configure` スクリプトに指定した `--exec-prefix` 引数に対応しています。この値は Python コードからは `sys.exec_prefix` として利用できます。Unix のみで有用です。

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、メジャーバー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。

wchar_t* **Py_GetProgramFullPath()**

Python 実行可能形式の完全なプログラム名を返します; この値はデフォルトのモジュール検索パスを (前述の `Py_SetProgramName()` で設定された) プログラム名から導出する際に副作用的に計算されます。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.executable` として利用できます。

`wchar_t* Py_GetPath()`

デフォルトのモジュール検索パスを返します; パスは (上の `Py_SetProgramName()` で設定された) プログラム名と、いくつかの環境変数から計算されます。戻り値となる文字列は、プラットフォーム依存のパス区切り文字で分割された一連のディレクトリ名からなります。区切り文字は Unix と Mac OS X では `':'`, Windows では `';'` 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。`sys.path` はインタプリタによってこの値で初期化され、その後モジュールをロードする際の検索パスを変更するために修正することが可能で、たいていそうされます。

`void Py_SetPath(const wchar_t *)`

デフォルトのモジュール検索パスを設定します。この関数が `Py_Initialize()` より前に呼び出された場合、`Py_GetPath()` はデフォルトの検索パスを計算しようとせず、代わりにこの関数が与えた検索パスを返します。これは、Python がアプリケーションに組み込まれていて、そのアプリケーションが全てのモジュールの場所を完全に把握しているときに便利です。パスはプラットフォーム依存の区切り文字で分割されている必要があります。区切り文字は Unix と Mac OS X では `':'` 、Windows では `';'` です。

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()`.

バイト文字列を `wchar_t*` 文字列にデコードするには `Py_DecodeLocale()` を使ってください。

パス引数は内部でコピーされます。したがって、呼び出し完了後に呼び出し元は引数を解放できます。

バージョン 3.8 で変更: プログラムの名前に代わって、プログラムのフルパスが `data:sys.executable` に使われるようになりました。

`const char* Py_GetVersion()`

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; 最初の三文字は、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。

`const char* Py_GetPlatform()`

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。Mac OS X 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()`

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()`

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()`

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)`

`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。

バイト文字列を `wchar_t*` 文字列にデコードするには `Py_DecodeLocale()` を使ってください。

注釈: 単一のスクリプトを実行する以外の目的で Python インタプリタを埋め込んでいるアプリケーションでは、`updatepath` に 0 を渡して、必要な場合は自分で `sys.path` を更新することをおすすめします。CVE-2008-5983 を参照してください。

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");
```

バージョン 3.1.3 で追加.

`void PySys_SetArgv(int argc, wchar_t **argv)`

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。

バイト文字列を `wchar_t *` 文字列にデコードするには `Py_DecodeLocale()` を使ってください。

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。

`void Py_SetPythonHome(const wchar_t *home)`

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。

バイト文字列を `wchar_t *` 文字列にデコードするには `Py_DecodeLocale()` を使ってください。

`w_char* Py_GetPythonHome()`

前回の `Py_SetPythonHome()` 呼び出しで設定されたデフォルトの "home" か、`PYTHONHOME` 環境変数が設定されていればその値を返します。

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)

Python インタプリタは完全にはスレッドセーフではありません。マルチスレッドの Python プログラムをサポートするために、**グローバルインタプリタロック** あるいは *GIL* と呼ばれるグローバルなロックが存在していて、現在のスレッドが Python オブジェクトに安全にアクセスする前に必ずロックを獲得しなければならないくなっています。ロック機構がなければ、単純な操作でさえ、マルチスレッドプログラムの実行に問題を引き起こす可能性があります。たとえば、二つのスレッドが同じオブジェクトの参照カウントを同時にインクリメントすると、結果的に参照カウントは二回でなく一回だけしかインクリメントされないかもしれません。

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。

Python インタプリタはスレッドごとに必要な情報を *PyThreadState* と呼ばれるデータ構造の中に保存します。そしてグローバル変数として現在の *PyThreadState* を指すポインタを 1 つ持ちます。このグローバル変

数は `PyThreadState_Get()` を使って取得できます。

9.5.1 拡張コード内で GIL を解放する

GIL を操作するほとんどのコードは、次のような単純な構造になります:

```
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もので、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);
```

これらの関数の動作を説明します。GIL は現在のスレッド状態を指すポインタを保護するために使われます。ロックを解放してスレッド状態を退避する際、ロックを解放する前に現在のスレッド状態ポインタを取得しておかなければなりません (他のスレッドがすぐさまロックを獲得して、自らのスレッド状態をグローバル変数に保存してしまうかもしれないからです)。逆に、ロックを獲得してスレッド状態を復帰する際には、グローバル変数にスレッド状態ポインタを保存する前にロックを獲得しておかなければなりません。

注釈: GIL を解放するのはほとんどがシステムの I/O 関数を呼び出す時ですが、メモリバッファに対する圧縮や暗号化のように、Python のオブジェクトにアクセスしない長時間かかる計算処理を呼び出すときも GIL を解放することは有益です。例えば、`zlib` や `hashlib` モジュールは圧縮やハッシュ計算の前に GIL を解放します。

9.5.2 Python 以外で作られたスレッド

Python API を通して作られたスレッド (`threading` モジュールなど) では自動的にスレッド状態が割り当てられて、上記のコードは正しく動きます。しかし、(自前でスレッド管理を行う外部のライブラリなどにより)C 言語でスレッドを生成した場合、そのスレッドには GIL がなく、スレッド状態データ構造体もないことに注意する必要があります。

このようなスレッドから Python コードを呼び出す必要がある場合 (外部のライブラリからコールバックする API などがよくある例です)、Python/C API を呼び出す前に、スレッド状態データ構造体を生成し、GIL を獲得し、スレッド状態ポインタを保存することで、スレッドをインタプリタに登録しなければなりません。スレッドが作業を終えたら、スレッド状態ポインタをリセットして、ロックを解放し、最後にスレッド状態データ構造体のメモリを解放しなければなりません。

`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);
```

`PyGILState_*`() 関数は、(`Py_Initialize()` によって自動的に作られる) グローバルインタプリタ 1 つだけが存在すると仮定する事に気をつけて下さい。Python は (`Py_NewInterpreter()` を使って) 追加のインタプリタを作成できることに変わりはありませんが、複数インタプリタと `PyGILState_*`() API を混せて使うことはサポートされていません。

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:

PyInterpreterState

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립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。

PyThreadState

This data structure represents the state of a single thread. The only public data member is `interp` (*PyInterpreterState **), which points to this thread's interpreter state.

void PyEval_InitThreads()

GIL を初期化し、獲得します。この関数は、主スレッドが第二のスレッドを生成する以前や、`PyEval_ReleaseThread(tstate)` といった他のスレッド操作に入るよりも前に呼び出されるようにしておかなければなりません。`PyEval_SaveThread()`、`PyEval_RestoreThread()` の前に呼び出す必要はありません。

二度目に呼び出すと何も行いません。

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。

int PyEval_ThreadsInitialized()

`PyEval_InitThreads()` をすでに呼び出している場合は真 (非ゼロ) を返します。この関数は、GIL を獲得せずに呼び出すことができますので、シングルスレッドで実行している場合にはロック関連の API 呼び出しを避けるために使うことができます。

バージョン 3.7 で変更: `GIL` が `Py_Initialize()` で初期化されるようになりました。

PyThreadState* PyEval_SaveThread()

(GIL が生成されている場合) GIL を解放して、スレッドの状態を NULL にし、以前のスレッド状態 (NULL にはなりません) を返します。ロックがすでに生成されている場合、現在のスレッドがロックを獲得していなければなりません。

void **PyEval_RestoreThread**(*PyThreadState* *tstate)

(GIL が生成されている場合) GIL を獲得して、現在のスレッド状態を *tstate* に設定します。*tstate* は NULL であってはなりません。GIL が生成されていて、この関数を呼び出したスレッドがすでにロックを獲得している場合、デッドロックに陥ります。

注釈: ランタイムが終了処理をしているときに、スレッドからこの関数を呼び出すと、そのスレッドが Python によって作成されたものではなかったとしても終了させられます。`_Py_IsFinalizing()` や `sys.is_finalizing()` を使うと、この関数を呼び出す前にインタプリタが終了される過程の途中なのか確認でき、望まないスレッドの終了が避けられます。

*PyThreadState** **PyThreadState_Get**()

現在のスレッド状態を返します。GIL を保持していなければなりません。現在のスレッド状態が NULL なら、(呼び出し側が NULL チェックをしなくてすむように) この関数は致命的エラーを起こすようになっています。

*PyThreadState** **PyThreadState_Swap**(*PyThreadState* *tstate)

現在のスレッド状態を *tstate* に指定したスレッド状態と入れ変えます。*tstate* は NULL の場合があります。GIL を保持していなければならず、解放しません。

以下の関数はスレッドローカルストレージを利用して、サブインタプリタとの互換性がありません:

PyGILState_STATE **PyGILState_Ensure**()

Python の状態や GIL に関わらず、実行中スレッドで Python C API の呼び出しが可能となるようにします。この関数はスレッド内で何度でも呼び出すことができますが、必ず全ての呼び出しに対応して `PyGILState_Release()` を呼び出す必要があります。通常、`PyGILState_Ensure()` 呼び出しと `PyGILState_Release()` 呼び出しの間でこれ以外のスレッド関連 API を使用することができますが、`Release()` の前にスレッド状態は復元されていなければなりません。例えば、通常の `Py_BEGIN_ALLOW_THREADS` マクロと `Py_END_ALLOW_THREADS` は使用することができます。

戻り値は `PyGILState_Ensure()` 呼び出し時のスレッド状態を隠蔽した”ハンドル”で、`PyGILState_Release()` に渡して Python を同じ状態に保たなければなりません。再起呼び出しも可能ですが、ハンドルを共有することは **できません** - それぞれの `PyGILState_Ensure()` 呼び出しでハンドルを保存し、対応する `PyGILState_Release()` 呼び出しで渡してください。

関数から復帰したとき、実行中のスレッドは GIL を所有していて、任意の Python コードを実行できます。処理の失敗は致命的なエラーです。

注釈: ランタイムが終了処理をしているときに、スレッドからこの関数を呼び出すと、そのスレッドが Python によって作成されたものではなかったとしても終了させられます。`_Py_IsFinalizing()` や `sys.is_finalizing()` を使うと、この関数を呼び出す前にインタプリタが終了される過程の途中なのか確認でき、望まないスレッドの終了が避けられます。

void **PyGILState_Release**(**PyGILState_STATE**)

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()`

このスレッドの現在のスレッドの状態を取得します。これまで現在のスレッドで GILState API を使ったことが無い場合は、NULL が返ります。メインスレッドで自身のスレッド状態に関する呼び出しを全くしないとしても、メインスレッドは常にスレッド状態の情報を持っていることに注意してください。こうなっている目的は主にヘルパ機能もしくは診断機能のためです。

`int` `PyGILState_Check()`

現在のスレッドが GIL を保持しているならば 1 を、そうでなければ 0 を返します。この関数はいつでもどのスレッドからでも呼び出すことができます。Python スレッドの状態が初期化されており、現在 GIL を保持している場合にのみ 1 を返します。これは主にヘルパー/診断用の関数です。この関数は、例えばコールバックのコンテキストやメモリ割り当て機能で有益でしょう。なぜなら、GIL がロックされていると知っていれば、呼び出し元は sensitive な行動を実行することができ、そうでなければ異なるやりかたで振る舞うことができるからです。

バージョン 3.4 で追加。

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`

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`

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`

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`

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()` は `GIL` を初期化するようになりました。

*PyInterpreterState** `PyInterpreterState_New()`

新しいインタプリタ状態オブジェクトを生成します。GIL を保持しておく必要はありませんが、この関数を次々に呼び出す必要がある場合には保持しておいたほうがよいでしょう。

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)`

インタプリタ状態オブジェクト内の全ての情報をリセットします。GIL を保持していなければなりません。

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)`

インタプリタ状態オブジェクトを破壊します。GIL を保持しておく必要はありません。インタプリタ状態は `PyInterpreterState_Clear()` であらかじめリセットしておかなければなりません。

*PyThreadState** `PyThreadState_New(PyInterpreterState *interp)`

指定したインタプリタオブジェクトに属する新たなスレッド状態オブジェクトを生成します。GIL を保持しておく必要はありませんが、この関数を次々に呼び出す必要がある場合には保持しておいたほうがよいでしょう。

`void PyThreadState_Clear(PyThreadState *tstate)`

スレッド状態オブジェクト内の全ての情報をリセットします。GIL を保持していなければなりません。

`void PyThreadState_Delete(PyThreadState *tstate)`

スレッド状態オブジェクトを破壊します。GIL を保持する必要はありません。スレッド状態は `PyThreadState_Clear()` であらかじめリセットしておかなければなりません。

`PY_INT64_T PyInterpreterState_GetID(PyInterpreterState *interp)`

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。

バージョン 3.7 で追加。

*PyObject** `PyInterpreterState_GetDict(PyInterpreterState *interp)`

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、ここまですべて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。

バージョン 3.8 で追加。

*PyObject** `PyThreadState_GetDict()`

Return value: Borrowed reference. 拡張モジュールがスレッド固有の状態情報を保存できるような辞書を返します。各々の拡張モジュールが辞書に状態情報を保存するためには唯一のキーを使わなければなりません。現在のスレッド状態がない時にこの関数を呼び出してもかまいません。この関数が NULL を返す場合、例外はまったく送出されず、呼び出し側は現在のスレッド状態が利用できないと考えなければなりません。

`int PyThreadState_SetAsyncExc(unsigned long id, PyObject *exc)`

スレッド内で非同期的に例外を送出します。*id* 引数はターゲットとなるスレッドのスレッド id です; *exc* は送出する例外オブジェクトです。この関数は *exc* に対する参照を一切盗み取りません。安直な間違いを防ぐため、この関数を呼び出すには独自に C 拡張モジュールを書かなければなりません。GIL を保持した状態で呼び出さなければなりません。状態が変更されたスレッドの数を返します; 通常は 1 ですが、スレッドが見付からなかった場合は 0 になることもあります。*exc* が NULL の場合は、このスレッドのまだ送出されていない例外は (何であれ) 消去されます。この場合は例外を送出しません。

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)`

GIL を獲得し、現在のスレッド状態を *tstate* に設定します。*tstate* は NULL であってはなりません。ロックはあらかじめ作成されていなければなりません。この関数を呼び出したスレッドがすでにロックを獲得している場合、デッドロックに陥ります。

注釈: ランタイムが終了処理をしているときに、スレッドからこの関数を呼び出すと、そのスレッドが Python によって作成されたものではなかったとしても終了させられます。`_Py_IsFinalizing()` や `sys.is_finalizing()` を使うと、この関数を呼び出す前にインタプリタが終了される過程の途中なのか確認でき、望まないスレッドの終了が避けられます。

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ていないときでも) 利用可能な高レベル関数です。

`void PyEval_ReleaseThread(PyThreadState *tstate)`

現在のスレッド状態をリセットして NULL にし、GIL を解放します。ロックはあらかじめ作成されていなければならず、かつ現在のスレッドが保持していなければなりません。*tstate* は NULL であってはなりません、その値が現在のスレッド状態を表現しているかどうかを調べるためにだけ使われます --- もしそうでなければ、致命的エラーが報告されます。

`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()`

GIL を獲得します。ロックは前もって作成されていなければなりません。この関数を呼び出したスレッドがすでにロックを獲得している場合、デッドロックに陥ります。

バージョン 3.2 で非推奨: この関数は現在のスレッドの状態を更新しません。代わりに `PyEval_RestoreThread()` もしくは `PyEval_AcquireThread()` を使用してください。

注釈: ランタイムが終了処理をしているときに、スレッドからこの関数を呼び出すと、そのスレッドが Python によって作成されたものではなかったとしても終了させられます。`_Py_IsFinalizing()` や `sys.is_finalizing()` を使うと、この関数を呼び出す前にインタプリタが終了される過程の途中なのか確認でき、望まないスレッドの終了が避けられます。

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()`

GIL を解放します。ロックは前もって作成されていなければなりません。

バージョン 3.2 で非推奨: この関数は現在のスレッドの状態を更新しません。代わりに `PyEval_SaveThread()` もしくは `PyEval_ReleaseThread()` を使用してください。

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:

`PyThreadState*` `Py_NewInterpreter()`

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)。

戻り値は、新たなサブインタプリタが生成したスレッド状態 (thread state) オブジェクトのうち、最初のものを指しています。このスレッド状態が現在のスレッド状態 (current thread state) になります。

実際のスレッドが生成されるわけではないので注意してください; 下記のスレッド状態に関する議論を参照してください。新たなインタプリタの生成に失敗すると、NULL を返します; 例外状態はセットされませんが、これは例外状態が現在のスレッド状態に保存されることになっていて、現在のスレッド状態なるものが存在しないことがあるからです。(他の Python/C API 関数のように、この関数を呼び出す前には GIL が保持されていなければならず、関数が処理を戻した際にも保持されたままになります; しかし、他の Python/C API 関数とは違い、関数から戻ったときの現在のスレッド状態が関数に入るときと同じとは限らないので注意してください。)

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.

`void Py_EndInterpreter(PyThreadState *tstate)`

指定されたスレッド状態 `tstate` で表現される (サブ) インタプリタを抹消します。`tstate` は現在のスレッド状態でなければなりません。下記のスレッド状態に関する議論を参照してください。関数呼び出しが戻ったとき、現在のスレッド状態は NULL になっています。このインタプリタに関連付けられた全てのスレッド状態は抹消されます。(この関数を呼び出す前には GIL を保持しておかなければならず、ロックは関数が戻ったときも保持されています。) `Py_FinalizeEx()` は、その時点で明示的に抹消されていない全てのサブインタプリタを抹消します。

9.6.1 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)
```

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、
- メインスレッドが *global interpreter lock* を保持している (すなわち *func* が全ての C API を呼び出せる) とき。

成功したら *func* は 0 を返さねばならず、失敗したら -1 を返し例外をセットしなければいけません。*func* は、他の非同期通知を行うために、さらに割り込まれることはありませんが、グローバルインタプリタロックが解放された場合は、スレッドの切り替えによって割り込まれる可能性があります。

この関数は実行するのに現在のスレッド状態を必要とせず、グローバルインタプリタロックも必要としません。

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。

バージョン 3.1 で追加。

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ーヘッドを避け、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。

`int (*Py_tracefunc)(PyObject *obj, PyFrameObject *frame, int what, PyObject *arg)`

`PyEval_SetProfile()` および `PyEval_SetTrace()` を使って登録できるトレース関数の形式です。最初のパラメタはオブジェクトで、登録関数に `obj` として渡されます。`frame` はイベントが属している実行フレームオブジェクトで、`what` は定数 `PyTrace_CALL`, `PyTrace_EXCEPTION`, `PyTrace_LINE`, `PyTrace_RETURN`, `PyTrace_C_CALL`, `PyTrace_C_RETURN`, `PyTrace_OPCODE` のいずれかで、`arg` は `what` の値によって以下のように異なります:

<i>what</i> の値	<i>arg</i> の意味
<code>PyTrace_CALL</code>	常に <code>Py_None</code> 。
<code>PyTrace_EXCEPTION</code>	<code>sys.exc_info()</code> の返す例外情報です。
<code>PyTrace_LINE</code>	常に <code>Py_None</code> 。
<code>PyTrace_RETURN</code>	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 <code>NULL</code> です。
<code>PyTrace_C_CALL</code>	呼び出される関数オブジェクト。
<code>PyTrace_C_EXCEPTION</code>	呼び出される関数オブジェクト。
<code>PyTrace_C_RETURN</code>	呼び出される関数オブジェクト。
<code>PyTrace_OPCODE</code>	常に <code>Py_None</code> 。

`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`

行番号イベントを報告するときに、(プロファイル関数ではなく) `Py_tracefunc` 関数に `what` 引数と

して渡す値です。f_trace_lines が 0 に設定されたフレームでは無効化されていることもあります。

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

新しい opcode が実行されるときに、(プロファイル関数ではなく) *Py_tracefunc* 関数に *what* 引数として渡す値です。デフォルトではイベントは発火しません。フレーム上で f_trace_lines を 1 に設定して、明示的に要請しなければなりません。

void PyEval_SetProfile(*Py_tracefunc func*, *PyObject *obj*)

プロファイル関数を *func* に設定します。 *obj* パラメタは関数の第一引数として渡されるもので、何らかの Python オブジェクトあるいは NULL です。プロファイル関数がスレッド状態を維持する必要があるなら、各々のスレッドに異なる *obj* を使うことで、状態を記憶しておく便利でスレッドセーフな場所を提供できます。プロファイル関数は、モニタされているイベントのうち、PyTrace_LINE PyTrace_OPCODE, PyTrace_EXCEPTION を除く全てのイベントに対して呼び出されます。

void PyEval_SetTrace(*Py_tracefunc func*, *PyObject *obj*)

トレース関数を *func* に設定します。この関数は *PyEval_SetProfile()* と同じですが、トレース関数は行番号イベントおよび opcode イベントを受け取り、呼び出された C 関数オブジェクトと関係する任意のイベントを受け取らないところが異なっています。 *PyEval_SetTrace()* で登録されたトレース関数は、*what* 引数の値として PyTrace_C_CALL, PyTrace_C_EXCEPTION, PyTrace_C_RETURN を受け取りません。

9.9 高度なデバグサポート (advanced debugger support)

以下の関数は高度なデバ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

The Python interpreter provides low-level support for thread-local storage (TLS) which wraps the underlying native TLS implementation to support the Python-level thread local storage API (`threading.local`). The CPython C level APIs are similar to those offered by pthreads and Windows: use a thread key and functions to associate a `void*` value per thread.

API で使われる関数を呼ぶときは、GIL を取得する必要は **ありません**。関数自身のロックがサポートされています。

`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: None of these API functions handle memory management on behalf of the `void*` values. You need to allocate and deallocate them yourself. If the `void*` values happen to be *PyObject**, these functions don't do refcount operations on them either.

9.10.1 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。

バージョン 3.7 で追加。

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**)

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()`

Return a value which is the same state as a value initialized with `Py_tss_NEEDS_INIT`, or `NULL` in the case of dynamic allocation failure.

`void` `PyThread_tss_free(Py_tss_t *key)`

Free the given `key` allocated by `PyThread_tss_alloc()`, after first calling `PyThread_tss_delete()` to ensure any associated thread locals have been unassigned. This is a no-op if the `key` argument is `NULL`.

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)`

Return a non-zero value if the given `Py_tss_t` has been initialized by `PyThread_tss_create()`.

`int` `PyThread_tss_create(Py_tss_t *key)`

Return a zero value on successful initialization of a TSS key. The behavior is undefined if the value pointed to by the `key` argument is not initialized by `Py_tss_NEEDS_INIT`. This function can be called repeatedly on the same key -- calling it on an already initialized key is a no-op and immediately returns success.

`void` `PyThread_tss_delete(Py_tss_t *key)`

Destroy a TSS key to forget the values associated with the key across all threads, and change the key's initialization state to uninitialized. A destroyed key is able to be initialized again by `PyThread_tss_create()`. This function can be called repeatedly on the same key -- calling it on an already destroyed key is a no-op.

`int` `PyThread_tss_set(Py_tss_t *key, void *value)`

Return a zero value to indicate successfully associating a `void*` value with a TSS key in the current thread. Each thread has a distinct mapping of the key to a `void*` value.

`void*` `PyThread_tss_get(Py_tss_t *key)`

Return the `void*` value associated with a TSS key in the current thread. This returns `NULL` if no value is associated with the key in the current thread.

9.10.2 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()

void PyThread_delete_key(int key)

int PyThread_set_key_value(int key, void *value)

void* PyThread_get_key_value(int key)

void PyThread_delete_key_value(int key)

void PyThread_ReInitTLS()
```


PYTHON 初期化設定

バージョン 3.8 で追加.

構造体:

- *PyConfig*
- *PyPreConfig*
- *PyStatus*
- *PyWideStringList*

関数:

- *PyConfig_Clear()*
- *PyConfig_InitIsolatedConfig()*
- *PyConfig_InitPythonConfig()*
- *PyConfig_Read()*
- *PyConfig_SetArgv()*
- *PyConfig_SetBytesArgv()*
- *PyConfig_SetBytesString()*
- *PyConfig_SetString()*
- *PyConfig_SetWideStringList()*
- *PyPreConfig_InitIsolatedConfig()*
- *PyPreConfig_InitPythonConfig()*
- *PyStatus_Error()*
- *PyStatus_Exception()*
- *PyStatus_Exit()*
- *PyStatus_IsError()*

- `PyStatus_IsExit()`
- `PyStatus_NoMemory()`
- `PyStatus_Ok()`
- `PyWideStringList_Append()`
- `PyWideStringList_Insert()`
- `Py_ExitStatusException()`
- `Py_InitializeFromConfig()`
- `Py_PreInitialize()`
- `Py_PreInitializeFromArgs()`
- `Py_PreInitializeFromBytesArgs()`
- `Py_RunMain()`

The preconfiguration (PyPreConfig type) is stored in `_PyRuntime.preconfig` and the configuration (PyConfig type) is stored in `PyInterpreterState.config`.

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 PyWideStringList

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 は、0 以上でなければなりません。

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。

10.2 PyStatus

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。

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;
}
```

10.3 PyPreConfig

PyPreConfig

Structure used to preinitialize Python:

- Set the Python memory allocator
- Configure the LC_CTYPE locale
- Set the UTF-8 mode

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 memory allocator:

- `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): force usage of `malloc()`
- `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*.

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` to 0.

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.

int coerce_c_locale_warn

If non-zero, emit a warning if the C locale is coerced.

int dev_mode

See *PyConfig.dev_mode*.

int isolated

See *PyConfig.isolated*.

int legacy_windows_fs_encoding(Windows *only*)

If non-zero, disable UTF-8 Mode, set the Python filesystem encoding to `mbcs`, set the filesystem error handler to `replace`.

Only available on Windows. `#ifdef MS_WINDOWS` macro can be used for Windows specific code.

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*.

`int use_environment`
See *PyConfig.use_environment*.

`int utf8_mode`
If non-zero, enable the UTF-8 mode.

10.4 Preinitialization with PyPreConfig

Functions to preinitialize Python:

PyStatus **Py_PreInitialize**(const *PyPreConfig* **preconfig*)

Preinitialize Python from *preconfig* preconfiguration.

PyStatus **Py_PreInitializeFromBytesArgs**(const *PyPreConfig* **preconfig*, int *argc*, char *
const **argv*)

Preinitialize Python from *preconfig* preconfiguration and command line arguments (bytes strings).

PyStatus **Py_PreInitializeFromArgs**(const *PyPreConfig* **preconfig*, int *argc*, wchar_t * const
* *argv*)

Preinitialize Python from *preconfig* preconfiguration and command line arguments (wide strings).

The caller is responsible to handle exceptions (error or exit) using *PyStatus_Exception()* and *Py_ExitStatusException()*.

For *Python Configuration* (*PyPreConfig_InitPythonConfig()*), if Python is initialized with command line arguments, the command line arguments must also be passed to preinitialize Python, since they have an effect on the pre-configuration like encodings. For example, the `-X utf8` command line option enables the UTF-8 Mode.

PyMem_SetAllocator() can be called after *Py_PreInitialize()* and before *Py_InitializeFromConfig()* to install a custom memory allocator. It can be called before *Py_PreInitialize()* if *PyPreConfig.allocator* is set to `PYMEM_ALLOCATOR_NOT_SET`.

Python memory allocation functions like *PyMem_RawMalloc()* must not be used before Python preinitialization, whereas calling directly `malloc()` and `free()` is always safe. *Py_DecodeLocale()* must not be called before the preinitialization.

Example using the preinitialization to enable the UTF-8 Mode:

```
PyStatus status;
PyPreConfig preconfig;
PyPreConfig_InitPythonConfig(&preconfig);

preconfig.utf8_mode = 1;

status = Py_PreInitialize(&preconfig);
if (PyStatus_Exception(status)) {
    Py_ExitStatusException(status);
}
```

(次のページに続く)

(前のページからの続き)

```

/* at this point, Python will speak UTF-8 */

Py_Initialize();
/* ... use Python API here ... */
Py_Finalize();

```

10.5 PyConfig

PyConfig

Structure containing most parameters to configure Python.

Structure methods:

`void PyConfig_InitPythonConfig(PyConfig *config)`

Initialize configuration with *Python Configuration*.

`void PyConfig_InitIsolatedConfig(PyConfig *config)`

Initialize configuration with *Isolated Configuration*.

PyStatus `PyConfig_SetString(PyConfig *config, wchar_t * const *config_str, const wchar_t *str)`

Copy the wide character string *str* into **config_str*.

Preinitialize Python if needed.

PyStatus `PyConfig_SetBytesString(PyConfig *config, wchar_t * const *config_str, const char *str)`

Decode *str* using `Py_DecodeLocale()` and set the result into **config_str*.

Preinitialize Python if needed.

PyStatus `PyConfig_SetArgv(PyConfig *config, int argc, wchar_t * const *argv)`

Set command line arguments from wide character strings.

Preinitialize Python if needed.

PyStatus `PyConfig_SetBytesArgv(PyConfig *config, int argc, char * const *argv)`

Set command line arguments: decode bytes using `Py_DecodeLocale()`.

Preinitialize Python if needed.

PyStatus `PyConfig_SetWideStringList(PyConfig *config, PyWideStringList *list, Py_ssize_t length, wchar_t **items)`

Set the list of wide strings *list* to *length* and *items*.

Preinitialize Python if needed.

PyStatus `PyConfig_Read(PyConfig *config)`

Read all Python configuration.

Fields which are already initialized are left unchanged.

Preinitialize Python if needed.

`void PyConfig_Clear(PyConfig *config)`

Release configuration memory.

Most `PyConfig` methods preinitialize Python if needed. In that case, the Python preinitialization configuration is based on the `PyConfig`. If configuration fields which are in common with `PyPreConfig` are tuned, they must be set before calling a `PyConfig` method:

- `dev_mode`
- `isolated`
- `parse_argv`
- `use_environment`

Moreover, if `PyConfig_SetArgv()` or `PyConfig_SetBytesArgv()` is used, this method must be called first, before other methods, since the preinitialization configuration depends on command line arguments (if `parse_argv` is non-zero).

The caller of these methods is responsible to handle exceptions (error or exit) using `PyStatus_Exception()` and `Py_ExitStatusException()`.

構造体フィールド:

`PyWideStringList argv`

Command line arguments, `sys.argv`. See `parse_argv` to parse `argv` the same way the regular Python parses Python command line arguments. If `argv` is empty, an empty string is added to ensure that `sys.argv` always exists and is never empty.

`wchar_t* base_exec_prefix`

`sys.base_exec_prefix`.

`wchar_t* base_executable`

`sys._base_executable`: `__PYENV_LAUNCHER__` environment variable value, or copy of `PyConfig.executable`.

`wchar_t* base_prefix`

`sys.base_prefix`.

`int buffered_stdio`

If equals to 0, enable unbuffered mode, making the stdout and stderr streams unbuffered.

stdin is always opened in buffered mode.

`int bytes_warning`

If equals to 1, issue a warning when comparing `bytes` or `bytearray` with `str`, or comparing `bytes` with `int`. If equal or greater to 2, raise a `BytesWarning` exception.

wchar_t* **check_hash_pycs_mode**

Control the validation behavior of hash-based .pyc files (see [PEP 552](#)):
--check-hash-based-pycs command line option value.

Valid values: `always`, `never` and `default`.

The default value is: `default`.

int **configure_c_stdio**

If non-zero, configure C standard streams (`stdio`, `stdout`, `stderr`). For example, set their mode to `O_BINARY` on Windows.

int **dev_mode**

Development mode: see `-X dev`.

int **dump_refs**

If non-zero, dump all objects which are still alive at exit.

Require a debug build of Python (`Py_REF_DEBUG` macro must be defined).

wchar_t* **exec_prefix**

`sys.exec_prefix`.

wchar_t* **executable**

`sys.executable`.

int **faulthandler**

If non-zero, call `faulthandler.enable()` at startup.

wchar_t* **filesystem_encoding**

Filesystem encoding, `sys.getfilesystemencoding()`.

wchar_t* **filesystem_errors**

Filesystem encoding errors, `sys.getfilesystemencodeerrors()`.

unsigned long **hash_seed**

int **use_hash_seed**

Randomized hash function seed.

If `use_hash_seed` is zero, a seed is chosen randomly at Python startup, and `hash_seed` is ignored.

wchar_t* **home**

Python home directory.

Initialized from `PYTHONHOME` environment variable value by default.

int **import_time**

If non-zero, profile import time.

int **inspect**

Enter interactive mode after executing a script or a command.

int **install_signal_handlers**

Install signal handlers?

int **interactive**

Interactive mode.

int **isolated**

If greater than 0, enable isolated mode:

- `sys.path` contains neither the script's directory (computed from `argv[0]` or the current directory) nor the user's site-packages directory.
- Python REPL doesn't import `readline` nor enable default readline configuration on interactive prompts.
- Set `use_environment` and `user_site_directory` to 0.

int **legacy_windows_stdio**

If non-zero, use `io.FileIO` instead of `io.WindowsConsoleIO` for `sys.stdin`, `sys.stdout` and `sys.stderr`.

Only available on Windows. `#ifdef MS_WINDOWS` macro can be used for Windows specific code.

int **malloc_stats**

If non-zero, dump statistics on *Python pymalloc memory allocator* at exit.

The option is ignored if Python is built using `--without-pymalloc`.

wchar_t* **pythonpath_env**

Module search paths as a string separated by `DELIM` (`os.path.pathsep`).

Initialized from `PYTHONPATH` environment variable value by default.

PyWideStringList **module_search_paths**

int **module_search_paths_set**

`sys.path`. If `module_search_paths_set` is equal to 0, the `module_search_paths` is overridden by the function calculating the *Path Configuration*.

int **optimization_level**

Compilation optimization level:

- 0: Peephole optimizer (and `__debug__` is set to `True`)
- 1: Remove assertions, set `__debug__` to `False`
- 2: Strip docstrings

int **parse_argv**

If non-zero, parse *argv* the same way the regular Python command line arguments, and strip Python arguments from *argv*: see Command Line Arguments.

int **parser_debug**

If non-zero, turn on parser debugging output (for expert only, depending on compilation options).

int **pathconfig_warnings**

If equal to 0, suppress warnings when calculating the *Path Configuration* (Unix only, Windows does not log any warning). Otherwise, warnings are written into `stderr`.

wchar_t* **prefix**

`sys.prefix`.

wchar_t* **program_name**

Program name. Used to initialize *executable*, and in early error messages.

wchar_t* **pycache_prefix**

`sys.pycache_prefix`: `.pyc` cache prefix.

If NULL, `sys.pycache_prefix` is set to None.

int **quiet**

Quiet mode. For example, don't display the copyright and version messages in interactive mode.

wchar_t* **run_command**

`python3 -c COMMAND` argument. Used by *Py_RunMain()*.

wchar_t* **run_filename**

`python3 FILENAME` argument. Used by *Py_RunMain()*.

wchar_t* **run_module**

`python3 -m MODULE` argument. Used by *Py_RunMain()*.

int **show_alloc_count**

Show allocation counts at exit?

Set to 1 by `-X showalloccount` command line option.

Need a special Python build with `COUNT_ALLOCS` macro defined.

int **show_ref_count**

Show total reference count at exit?

Set to 1 by `-X showrefcount` command line option.

Need a debug build of Python (`Py_REF_DEBUG` macro must be defined).

int **site_import**

Import the `site` module at startup?

`int skip_source_first_line`

Skip the first line of the source?

`wchar_t* stdio_encoding`

`wchar_t* stdio_errors`

Encoding and encoding errors of `sys.stdin`, `sys.stdout` and `sys.stderr`.

`int tracemalloc`

If non-zero, call `tracemalloc.start()` at startup.

`int use_environment`

If greater than 0, use environment variables.

`int user_site_directory`

If non-zero, add user site directory to `sys.path`.

`int verbose`

If non-zero, enable verbose mode.

PyWideStringList `warnoptions`

`sys.warnoptions`: options of the `warnings` module to build warnings filters: lowest to highest priority.

The `warnings` module adds `sys.warnoptions` in the reverse order: the last *PyConfig.warnoptions* item becomes the first item of `warnings.filters` which is checked first (highest priority).

`int write_bytecode`

If non-zero, write `.pyc` files.

`sys.dont_write_bytecode` is initialized to the inverted value of *write_bytecode*.

PyWideStringList `xoptions`

`sys._xoptions`.

If `parse_argv` is non-zero, `argv` arguments are parsed the same way the regular Python parses command line arguments, and Python arguments are stripped from `argv`: see Command Line Arguments.

The `xoptions` options are parsed to set other options: see `-X` option.

10.6 Initialization with PyConfig

Function to initialize Python:

PyStatus `Py_InitializeFromConfig`(const *PyConfig* **config*)

Initialize Python from *config* configuration.

The caller is responsible to handle exceptions (error or exit) using *PyStatus_Exception()* and *Py_ExitStatusException()*.

If `PyImport_FrozenModules`, `PyImport_AppendInittab()` or `PyImport_ExtendInittab()` are used, they must be set or called after Python preinitialization and before the Python initialization.

Example setting the program name:

```
void init_python(void)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name. Implicitly preinitialize Python. */
    status = PyConfig_SetString(&config, &config.program_name,
                               L"/path/to/my_program");
    if (PyStatus_Exception(status)) {
        goto fail;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto fail;
    }
    PyConfig_Clear(&config);
    return;

fail:
    PyConfig_Clear(&config);
    Py_ExitStatusException(status);
}
```

More complete example modifying the default configuration, read the configuration, and then override some parameters:

```
PyStatus init_python(const char *program_name)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);

    /* Set the program name before reading the configuration
       (decode byte string from the locale encoding).

       Implicitly preinitialize Python. */
    status = PyConfig_SetBytesString(&config, &config.program_name,
                                     program_name);
    if (PyStatus_Exception(status)) {
        goto done;
    }

    /* Read all configuration at once */
```

(次のページに続く)

```

status = PyConfig_Read(&config);
if (PyStatus_Exception(status)) {
    goto done;
}

/* Append our custom search path to sys.path */
status = PyWideStringList_Append(&config.module_search_paths,
                                L"/path/to/more/modules");
if (PyStatus_Exception(status)) {
    goto done;
}

/* Override executable computed by PyConfig_Read() */
status = PyConfig_SetString(&config, &config.executable,
                           L"/path/to/my_executable");
if (PyStatus_Exception(status)) {
    goto done;
}

status = Py_InitializeFromConfig(&config);

done:
    PyConfig_Clear(&config);
    return status;
}

```

10.7 Isolated Configuration

PyPreConfig_InitIsolatedConfig() and *PyConfig_InitIsolatedConfig()* functions create a configuration to isolate Python from the system. For example, to embed Python into an application.

This configuration ignores global configuration variables, environments variables, command line arguments (*PyConfig.argv* is not parsed) and user site directory. The C standard streams (ex: `stdout`) and the `LC_CTYPE` locale are left unchanged. Signal handlers are not installed.

Configuration files are still used with this configuration. Set the *Path Configuration* ("output fields") to ignore these configuration files and avoid the function computing the default path configuration.

10.8 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 UTF-8 Mode (**PEP 540**) depending on the

LC_CTYPE locale, PYTHONUTF8 and PYTHONCOERCECLOCALE environment variables.

Example of customized Python always running in isolated mode:

```
int main(int argc, char **argv)
{
    PyStatus status;

    PyConfig config;
    PyConfig_InitPythonConfig(&config);
    config.isolated = 1;

    /* Decode command line arguments.
       Implicitly preinitialize Python (in isolated mode). */
    status = PyConfig_SetBytesArgv(&config, argc, argv);
    if (PyStatus_Exception(status)) {
        goto fail;
    }

    status = Py_InitializeFromConfig(&config);
    if (PyStatus_Exception(status)) {
        goto fail;
    }
    PyConfig_Clear(&config);

    return Py_RunMain();
fail:
    PyConfig_Clear(&config);
    if (PyStatus_IsExit(status)) {
        return status.exitcode;
    }
    /* Display the error message and exit the process with
       non-zero exit code */
    Py_ExitStatusException(status);
}
```

10.9 Path Configuration

PyConfig contains multiple fields for the path configuration:

- Path configuration inputs:
 - *PyConfig.home*
 - *PyConfig.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*)
- *__PYENV_LAUNCHER__* environment variable
- (Windows only) Application paths in the registry under "SoftwarePythonPython-CoreX.YPythonPath" of HKEY_CURRENT_USER and HKEY_LOCAL_MACHINE (where X.Y is the Python version).
- Path configuration output fields:
 - *PyConfig.base_exec_prefix*
 - *PyConfig.base_executable*
 - *PyConfig.base_prefix*
 - *PyConfig.exec_prefix*
 - *PyConfig.executable*
 - *PyConfig.module_search_paths_set*, *PyConfig.module_search_paths*
 - *PyConfig.prefix*

If at least one "output field" is not set, Python calculates the path configuration to fill unset fields. If *module_search_paths_set* is equal to 0, *module_search_paths* is overridden and *module_search_paths_set* is set to 1.

It is possible to completely ignore the function calculating the default path configuration by setting explicitly all path configuration output fields listed above. A string is considered as set even if it is non-empty. *module_search_paths* is considered as set if *module_search_paths_set* is set to 1. In this case, path configuration input fields are ignored as well.

Set *pathconfig_warnings* to 0 to suppress warnings when calculating the path configuration (Unix only, Windows does not log any warning).

If *base_prefix* or *base_exec_prefix* fields are not set, they inherit their value from *prefix* and *exec_prefix* respectively.

PyRunMain() and *Py_Main()* modify *sys.path*:

- If *run_filename* is set and is a directory which contains a *__main__.py* script, prepend *run_filename* to *sys.path*.
- If *isolated* is zero:
 - If *run_module* is set, prepend the current directory to *sys.path*. Do nothing if the current directory cannot be read.
 - If *run_filename* is set, prepend the directory of the filename to *sys.path*.
 - Otherwise, prepend an empty string to *sys.path*.

If *site_import* is non-zero, `sys.path` can be modified by the `site` module. If *user_site_directory* is non-zero and the user's site-package directory exists, the `site` module appends the user's site-package directory to `sys.path`.

The following configuration files are used by the path configuration:

- `pyvenv.cfg`
- `python._pth` (Windows only)
- `pybuilddir.txt` (Unix only)

The `__PYENVV_LAUNCHER__` environment variable is used to set *PyConfig.base_executable*

10.10 Py_RunMain()

int `Py_RunMain`(void)

Execute the command (*PyConfig.run_command*), the script (*PyConfig.run_filename*) or the module (*PyConfig.run_module*) specified on the command line or in the configuration.

By default and when if `-i` option is used, run the REPL.

Finally, finalizes Python and returns an exit status that can be passed to the `exit()` function.

See *Python Configuration* for an example of customized Python always running in isolated mode using *Py_RunMain()*.

10.11 Multi-Phase Initialization Private Provisional API

This section is a private provisional API introducing multi-phase initialization, the core feature of the [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);
}
}
```


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ーを実装しています。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 メモリマネー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 メモリマネー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 生メモリインタフェース

以下の関数群はシステムのアロケータをラップします。これらの関数はスレッドセーフで、[GIL](#) を保持していても呼び出すことができます。

デフォルトの **生メモリアロケータ** は次の関数を利用します: `malloc()`, `calloc()`, `realloc()`, `free()` 0 バイトを要求されたときには `malloc(1)` (あるいは `calloc(1, 1)`) を呼びます。

バージョン 3.4 で追加。

`void* PyMem_RawMalloc(size_t n)`

Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

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)`

Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

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。

バージョン 3.5 で追加.

`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。

11.3 メモリインタフェース

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： これらの関数を呼ぶときには、*GIL* を保持しておく必要があります。

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)`

Allocates `n` bytes and returns a pointer of type `void*` to the allocated memory, or NULL if the request fails.

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)`

Allocates `nelem` elements each whose size in bytes is `elsize` and returns a pointer of type `void*` to the allocated memory, or NULL if the request fails. The memory is initialized to zeros.

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。

バージョン 3.5 で追加.

`void* PyMem_Realloc(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_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` は前のメモリアreaをさす有効なポインタのままです。

`void PyMem_Free(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_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。

`TYPE* PyMem_New(TYPE, size_t 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.

`TYPE* PyMem_Resize(void *p, TYPE, size_t 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)`

11.4 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:

The *default object allocator* uses the *pymalloc memory allocator*.

警告: これらの関数を呼ぶときには、*GIL* を保持しておく必要があります。

`void* PyObject_Malloc(size_t n)`

Allocates *n* bytes and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails.

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)`

Allocates *nelem* elements each whose size in bytes is *elsize* and returns a pointer of type `void*` to the allocated memory, or `NULL` if the request fails. The memory is initialized to zeros.

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。

バージョン 3.5 で追加.

`void* PyObject_Realloc(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 `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)`

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。

11.5 Default Memory Allocators

Default memory allocators:

Configuration	名前	PyMem_Raw-Malloc	PyMem_Malloc	PyObject_Mal-loc
リリースビルド	"pymalloc"	malloc	pymalloc	pymalloc
デバッグビルド	"pymalloc_debug"	malloc + debug	pymalloc + de- bug	pymalloc + de- bug
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 + de- bug	malloc + de- bug

説明:

- 名前: PYTHONMALLOC 環境変数の値
- malloc: 標準 C ライブラリのシステムアロケータ、C 関数: malloc(), calloc(), realloc(), free()
- pymalloc: *pymalloc* メモリアロケータ
- "+ debug": *PyMem_SetupDebugHooks()* で設置されるデバッグフックとの組み合わせ

11.6 メモリアロケータをカスタマイズする

バージョン 3.4 で追加.

PyMemAllocatorEx

メモリブロックアロケータを記述するための構造体です。4 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* 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: `PyMemAllocator` 構造体が `PyMemAllocatorEx` にリネームされた上で `calloc` フィールドが追加されました。

`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。

`PYMEM_DOMAIN_RAW` ドメインでは、アロケータはスレッドセーフでなければなりません: アロケータが呼び出されたとき `GIL` は保持されていません。

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。

void `PyMem_SetupDebugHooks`(void)

Python メモリアロケータ関数のバグを検出するためのフックを設定します。

Newly allocated memory is filled with the byte 0xCD (CLEANBYTE), freed memory is filled with the byte 0xDD (DEADBYTE). Memory blocks are surrounded by "forbidden bytes" (FORBIDDENBYTE: byte 0xFD).

実行時チェック:

- API 違反を検出します。例: `PyMem_Malloc()` が割り当てたバッファに対して `PyObject_Free()` を呼び出した。
- バッファの開始前の書き込み (バッファアンダーフロー) を検出します
- バッファ終了後の書き込み (バッファオーバーフロー) を検出します
- Check that the `GIL` is held when allocator functions of `PYMEM_DOMAIN_OBJ` (ex: `PyObject_Malloc()`) and `PYMEM_DOMAIN_MEM` (ex: `PyMem_Malloc()`) domains are called

On error, the debug hooks use the `tracemalloc` module to get the traceback where a memory block was allocated. The traceback is only displayed if `tracemalloc` is tracing Python memory allocations and the memory block was traced.

These hooks are *installed by default* if Python is compiled in debug mode. The `PYTHONMALLOC` environment variable can be used to install debug hooks on a Python compiled in release mode.

バージョン 3.6 で変更: This function now also works on Python compiled in release mode. On error, the debug hooks now use `tracemalloc` to get the traceback where a memory block was allocated. The debug hooks now also check if the `GIL` is held when functions of `PYMEM_DOMAIN_OBJ` and `PYMEM_DOMAIN_MEM` domains are called.

バージョン 3.8 で変更: Byte patterns 0xCB (CLEANBYTE), 0xDB (DEADBYTE) and 0xFB (FORBIDDENBYTE) have been replaced with 0xCD, 0xDD and 0xFD to use the same values than Windows CRT debug `malloc()` and `free()`.

11.7 pymalloc アロケータ

Python には、寿命の短い小さな (512 バイト以下の) オブジェクトに最適化された `pymalloc` アロケータがあります。 `pymalloc` は、256 KiB の固定サイズの "アリーナ" と呼ばれるメモリマッピングを使います。 512 バイトよりも大きな割り当てでは、`PyMem_RawMalloc()` と `PyMem_RawRealloc()` にフォールバックします。

`pymalloc` は、 `PYMEM_DOMAIN_MEM` (ex: `PyMem_Malloc()`) と `PYMEM_DOMAIN_OBJ` (ex: `PyObject_Malloc()`) ドメインの **既定のアロケータ** です。

アリーナアロケータは、次の関数を使います:

- Windows では `VirtualAlloc()` と `VirtualFree()`、
- 利用できる場合、`mmap()` と `munmap()`、

- それ以外の場合は `malloc()` と `free()`。

11.7.1 pymalloc アリーナアロケータのカスタマイズ

バージョン 3.4 で追加。

`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
<code>void *ctx</code>	第一引数として渡されるユーザコンテキスト
<code>void* alloc(void *ctx, size_t size)</code>	<code>size</code> バイトのアリーナを割り当てます
<code>void free(void *ctx, void *ptr, size_t size)</code>	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。

11.8 tracemalloc C API

バージョン 3.7 で追加。

`int PyTraceMalloc_Track(unsigned int domain, uintptr_t ptr, size_t size)`

Track an allocated memory block in the `tracemalloc` module.

Return 0 on success, return -1 on error (failed to allocate memory to store the trace). Return -2 if `tracemalloc` is disabled.

If memory block is already tracked, update the existing trace.

`int PyTraceMalloc_Untrack(unsigned int domain, uintptr_t ptr)`

Untrack an allocated memory block in the `tracemalloc` module. Do nothing if the block was not tracked.

Return -2 if `tracemalloc` is disabled, otherwise return 0.

11.9 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() */
```

素のメモリブロックを Python ヒープ上で操作する関数に加え、*PyObject_New()*、*PyObject_NewVar()*、および *PyObject_Del()* を使うと、Python におけるオブジェクトをメモリ確保したり解放したりできます。

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。

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*)

Return value: New reference.

*PyVarObject** **_PyObject_NewVar**(*PyTypeObject* **type*, *Py_ssize_t* *size*)

Return value: New reference.

*PyObject** **PyObject_Init**(*PyObject* **op*, *PyTypeObject* **type*)

Return value: Borrowed reference. 新たにメモリ確保されたオブジェクト *op* に対し、型と初期状態での参照 (initial reference) を初期化します。初期化されたオブジェクトを返します。*type* からそのオブジェクトが循環参照ガベージ検出の機能を有する場合、検出機構が監視対象とするオブジェクトのセットに追加されます。オブジェクトの他のフィールドには影響を及ぼしません。

*PyVarObject** **PyObject_InitVar**(*PyVarObject* **op*, *PyTypeObject* **type*, *Py_ssize_t* *size*)

Return value: Borrowed reference. **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。

*TYPE** **PyObject_New**(*TYPE*, *PyTypeObject* **type*)

Return value: New reference. C 構造体型 *TYPE* と Python 型オブジェクト *type* を使って、新しい Python オブジェクトをメモリ上に確保します。Python オブジェクトヘッダで定義されていないフィールドは初期化されません; オブジェクトの参照カウントは 1 になります。確保するメモリのサイズは型オブジェクトの *tp_basicsize* フィールドで決まります。

*TYPE** **PyObject_NewVar**(*TYPE*, *PyTypeObject* **type*, *Py_ssize_t* *size*)

Return value: New reference. C 構造体型 *TYPE* と Python 型オブジェクト *type* を使って新しい Python オブジェクトをメモリ上に確保します。Python オブジェクトヘッダで定義されていないフィールドは初期化されません。確保されたメモリは、*TYPE* 構造体に加え、*type* の *tp_itemsize* フィールドで指定されているサイズを *size* 個分の大きさを格納できます。この関数は、例えばタブルのように生成時にサイズを決定できるオブジェクトを実装する際に便利です。一連の複数のフィールドのメモリ割り当てを一度で行うことでアロケーション回数を減らし、メモリ管理の効率が向上します。

`void PyObject_Del(void *op)`

`PyObject_New()` や `PyObject_NewVar()` で確保したメモリを解放します。通常、この関数はオブジェクトの型に指定されている `tp_dealloc` ハンドラから呼び出されます。この関数を呼び出した後は、メモリ領域はもはや有効な Python オブジェクトを表現していないので、オブジェクトのフィールドに対してアクセスしてはなりません。

`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。

全ての Python オブジェクトは、オブジェクトのメモリ内表現の先頭部分にある少数のフィールドを完全に共有しています。このフィールドは `PyObject` 型および `PyVarObject` 型で表現されます。これらの型もまた、他の全ての Python オブジェクトの定義で直接または間接的に使われているマクロを使って定義されています。

PyObject

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*` ヘキャストできます。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。

PyVarObject

`PyObject` を拡張して、`ob_size` フィールドを追加したものです。この構造体は、長さ (*length*) の概念を持つオブジェクトだけに対して使います。この型が Python/C API で使われることはほとんどありません。メンバにアクセスするには `Py_REFCNT` マクロ、`Py_TYPE` マクロ、`Py_SIZE` マクロを使わなければなりません。

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。

Py_TYPE(o)

Python オブジェクトの `ob_type` メンバにアクセスするのに使うマクロです。これは次のように展開されます:

```
((PyObject*)(o))->ob_type)
```

Py_REFCNT(o)

Python オブジェクトの `ob_refcnt` メンバにアクセスするのに使うマクロです。これは次のように展開されます:

```
((PyObject*)(o))->ob_refcnt)
```

Py_SIZE(o)

Python オブジェクトの `ob_size` メンバにアクセスするのに使うマクロです。これは次のように展開されます:

```
((PyVarObject*)(o))->ob_size)
```

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)

新しい、`ob_size` フィールドを含む *PyVarObject* 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, size,
```

PyCFunction

Type of the functions used to implement most Python callables in C. Functions of this type take two *PyObject** parameters and return one such value. If the return value is NULL, an exception shall have been set. If not NULL, the return value is interpreted as the return value of the function as exposed in Python. The function must return a new reference.

PyCFunctionWithKeywords

Type of the functions used to implement Python callables in C with signature `METH_VARARGS | METH_KEYWORDS`.

_PyCFunctionFast

Type of the functions used to implement Python callables in C with signature *METH_FASTCALL*.

_PyCFunctionFastWithKeywords

Type of the functions used to implement Python callables in C with signature *METH_FASTCALL* | *METH_KEYWORDS*.

PyMethodDef

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:

フィールド	C の型	意味
ml_name	const char *	メソッド名
ml_meth	PyCFunction	C 実装へのポインタ
ml_flags	int	呼び出しをどのように行うかを示すフラグビット
ml_doc	const char *	docstring の内容を指すポインタ

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 four basic calling conventions for positional arguments and two of them can be combined with *METH_KEYWORDS* to support also keyword arguments. So there are a total of 6 calling conventions:

METH_VARARGS

This is the typical calling convention, where the methods have the type *PyCFunction*. The function expects two *PyObject** values. The first one is the *self* object for methods; for module functions, it is the module object. The second parameter (often called *args*) is a tuple object representing all arguments. This parameter is typically processed using *PyArg_ParseTuple()* or *PyArg_UnpackTuple()*.

METH_VARARGS | METH_KEYWORDS

Methods with these flags must be of type *PyCFunctionWithKeywords*. The function expects three parameters: *self*, *args*, *kwargs* where *kwargs* is a dictionary of all the keyword arguments or possibly NULL if there are no keyword arguments. The parameters are typically processed using *PyArg_ParseTupleAndKeywords()*.

METH_FASTCALL

Fast calling convention supporting only positional arguments. The methods have the type *_PyCFunctionFast*. The first parameter is *self*, the second parameter is a C array of *PyObject** values indicating the arguments and the third parameter is the number of arguments (the length of the array).

This is not part of the *limited API*.

バージョン 3.7 で追加.

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 or possibly NULL if there are no keywords. The values of the keyword arguments are stored in the *args* array, after the positional arguments.

This is not part of the *limited API*.

バージョン 3.7 で追加.

METH_NOARGS

引数のないメソッドは、*METH_NOARGS* フラグをつけた場合、必要な引数が指定されているかをチェックしなくなります。こうしたメソッドは *PyCFunction* 型でなくてはなりません。第一のパラメタは *self* になり、モジュールかオブジェクトインスタンスへの参照を保持することになります。いずれにせよ、第二のパラメタは NULL になります。

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

メソッドを既存の定義を置き換える形でロードします。*METH_COEXIST* を指定しなければ、デフォルトの設定にしたがって、定義が重複しないようスキップします。スロットラップはメソッドテーブルよりも前にロードされるので、例えば *sq_contains* スロットはラップしているメソッド *__contains__()* を生成し、同名の *PyCFunction* のロードを阻止します。このフラグを定義すると、*PyCFunction* はラップオブジェクトを置き換える形でロードされ、スロットと連立します。*PyCFunctions* の呼び出しはラップオブジェクトの呼び出しよりも最適化されているので、こうした仕様が便利になります。

PyMemberDef

`type` の C 構造体のメンバとして格納されている、ある型の属性を表す構造体です。この構造体のフィールドは以下のとおりです:

フィールド	C の型	意味
<code>name</code>	<code>const char *</code>	メンバ名
<code>type</code>	<code>int</code>	C 構造体の中のメンバの型
<code>offset</code>	<code>Py_ssize_t</code>	そのメンバの <code>type object</code> 構造体中の場所の <code>offset</code> バイト数
<code>flags</code>	<code>int</code>	フィールドが読み出し専用か書き込み可能なのかを示すビットフラグ
<code>doc</code>	<code>const char *</code>	<code>docstring</code> の内容を指すポインタ

`type` はたくさんの C の型を意味する `T_` マクロのうちの 1 つです。メンバが Python からアクセスされるとき、そのメンバは対応する Python の型に変換されます。

マクロ名	C の型
<code>T_SHORT</code>	<code>short</code>
<code>T_INT</code>	<code>int</code>
<code>T_LONG</code>	<code>long</code>
<code>T_FLOAT</code>	浮動小数点数
<code>T_DOUBLE</code>	<code>double</code>
<code>T_STRING</code>	<code>const char *</code>
<code>T_OBJECT</code>	<code>PyObject *</code>
<code>T_OBJECT_EX</code>	<code>PyObject *</code>
<code>T_CHAR</code>	<code>char</code>
<code>T_BYTE</code>	<code>char</code>
<code>T_UBYTE</code>	<code>unsigned char</code>
<code>T_UINT</code>	<code>unsigned int</code>
<code>T_USHORT</code>	<code>unsigned short</code>
<code>T_ULONG</code>	<code>unsigned long</code>
<code>T_BOOL</code>	<code>char</code>
<code>T_LONGLONG</code>	<code>long long</code>
<code>T_ULONGLONG</code>	<code>unsigned long long</code>
<code>T_PYSSIZET</code>	<code>Py_ssize_t</code>

`T_OBJECT` と `T_OBJECT_EX` が異なっているのは、`T_OBJECT` はメンバが `NULL` だったときに `None` を返すのに対し、`T_OBJECT_EX` は `AttributeError` を送出する点です。`T_OBJECT_EX` は `T_OBJECT` より属性に対する `del` 文を正しくあつかうので、できれば `T_OBJECT` ではなく `T_OBJECT_EX` を使ってください。

`flags` can be 0 for write and read access or `READONLY` for read-only access. Using `T_STRING` for `type` implies `READONLY`. `T_STRING` data is interpreted as UTF-8. Only `T_OBJECT` and `T_OBJECT_EX` members can be deleted. (They are set to `NULL`).

PyGetSetDef

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。c:member:PyTypeObject.tp_getset スロットの説明も参照してください。

フィールド	C の型	意味
名前	const char *	属性名
get	getter	C Function to get the attribute
集合	setter	属性をセット/削除する任意の C 言語関数。省略された場合、属性は読み取り専用になります。
doc	const char *	任意のドキュメンテーション文字列
closure	void *	getter と setter の追加データを提供する、オプションの関数ポインタ。

The `get` function takes one *PyObject** parameter (the instance) and a function pointer (the associated `closure`):

```
typedef PyObject *(*getter)(PyObject *, void *);
```

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。

`set` functions take two *PyObject** parameters (the instance and the value to be set) and a function pointer (the associated `closure`):

```
typedef int (*setter)(PyObject *, PyObject *, void *);
```

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 型オブジェクト

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定める基盤ですから、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 slots”

PyTypeObject Slot* ¹	型	特殊メソッド/特殊属性	Info* ²				
			O	T	D	I	
<R> <i>tp_name</i>	const char *	__name__	X	X			
<i>tp_basicsize</i>	Py_ssize_t		X	X			X
<i>tp_itemsize</i>	Py_ssize_t			X			X
<i>tp_dealloc</i>	<i>destructor</i>		X	X			X
<i>tp_vectorcall_offset</i>	Py_ssize_t						?
(<i>tp_getattr</i>)	<i>getattrfunc</i>	__getattribute__, __getattr__					G
(<i>tp_setattr</i>)	<i>setattrfunc</i>	__setattr__, __delattr__					G
<i>tp_as_async</i>	<i>PyAsyncMethods</i> *	<i>sub-slots</i>					%
<i>tp_repr</i>	<i>reprfunc</i>	__repr__	X	X			X
<i>tp_as_number</i>	<i>PyNumberMethods</i> *	<i>sub-slots</i>					%
<i>tp_as_sequence</i>	<i>PySequenceMethods</i> *	<i>sub-slots</i>					%
<i>tp_as_mapping</i>	<i>PyMappingMethods</i> *	<i>sub-slots</i>					%
<i>tp_hash</i>	<i>hashfunc</i>	__hash__	X				G
<i>tp_call</i>	<i>ternaryfunc</i>	__call__		X			X
<i>tp_str</i>	<i>reprfunc</i>	__str__	X				X
<i>tp_getattro</i>	<i>getattrofunc</i>	__getattribute__, __getattr__	X	X			G
<i>tp_setattro</i>	<i>setattrofunc</i>	__setattr__, __delattr__	X	X			G
<i>tp_as_buffer</i>	<i>PyBufferProcs</i> *						%
<i>tp_flags</i>	unsigned long		X	X			?
<i>tp_doc</i>	const char *	__doc__	X	X			
<i>tp_traverse</i>	<i>traverseproc</i>			X			G
<i>tp_clear</i>	<i>inquiry</i>			X			G
<i>tp_richcompare</i>	<i>richcmpfunc</i>	__lt__, __le__, __eq__, __ne__, __gt__, __ge__	X				G
<i>tp_weaklistoffset</i>	Py_ssize_t			X			?
<i>tp_iter</i>	<i>getiterfunc</i>	__iter__					X
<i>tp_iternext</i>	<i>iternextfunc</i>	__next__					X
<i>tp_methods</i>	<i>PyMethodDef</i> []		X	X			
<i>tp_members</i>	<i>PyMemberDef</i> []			X			
<i>tp_getset</i>	<i>PyGetSetDef</i> []		X	X			
<i>tp_base</i>	<i>PyTypeObject</i> *	__base__				X	
<i>tp_dict</i>	<i>PyObject</i> *	__dict__				?	

次のページに続く

表 1 – 前のページからの続き

PyTypeObject Slot* ¹	型	特殊メソッド/特殊属性	Info* ²			
			O	T	D	I
<i>tp_descr_get</i>	<i>descrgetfunc</i>	<code>__get__</code>				X
<i>tp_descr_set</i>	<i>descrsetfunc</i>	<code>__set__</code> , <code>__delete__</code>				X
<i>tp_dictoffset</i>	<code>Py_ssize_t</code>			X		?
<i>tp_init</i>	<i>initproc</i>	<code>__init__</code>	X	X		X
<i>tp_alloc</i>	<i>allocfunc</i>		X		?	?
<i>tp_new</i>	<i>newfunc</i>	<code>__new__</code>	X	X	?	?
<i>tp_free</i>	<i>freefunc</i>		X	X	?	?
<i>tp_is_gc</i>	<i>inquiry</i>			X		X
<code><tp_bases></code>	<i>PyObject *</i>	<code>__bases__</code>			~	
<code><tp_mro></code>	<i>PyObject *</i>	<code>__mro__</code>			~	
<code>[tp_cache]</code>	<i>PyObject *</i>					
<code>[tp_subclasses]</code>	<i>PyObject *</i>	<code>__subclasses__</code>				
<code>[tp_weaklist]</code>	<i>PyObject *</i>					
<code>(tp_del)</code>	<i>destructor</i>					
<code>[tp_version_tag]</code>	unsigned int					
<i>tp_finalize</i>	<i>destructor</i>	<code>__del__</code>				X

COUNT_ALLOCS が定義されている場合、次の (内部専用) フィールドも使えます:

- *tp_allocs*
- *tp_frees*
- *tp_maxalloc*
- *tp_prev*

*¹ 丸括弧で囲われているスロット名は、それが (実質的に) 非推奨であることを示しています。山括弧で囲われているスロット名は、読み出し専用として扱われるべきです。角括弧で囲われているスロット名は、内部でのみ使われます。(接頭辞としての) "`<R>`" は、このフィールドが必須であること (NULL でないこと) を意味します。

*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.

- `tp_next`

sub-slots

Slot	型	特殊メソッド
<code>am_await</code>	<code>unaryfunc</code>	<code>__await__</code>
<code>am_aiter</code>	<code>unaryfunc</code>	<code>__aiter__</code>
<code>am_anext</code>	<code>unaryfunc</code>	<code>__anext__</code>
<code>nb_add</code>	<code>binaryfunc</code>	<code>__add__</code> <code>__radd__</code>
<code>nb_inplace_add</code>	<code>binaryfunc</code>	<code>__iadd__</code>
<code>nb_subtract</code>	<code>binaryfunc</code>	<code>__sub__</code> <code>__rsub__</code>
<code>nb_inplace_subtract</code>	<code>binaryfunc</code>	<code>__isub__</code>
<code>nb_multiply</code>	<code>binaryfunc</code>	<code>__mul__</code> <code>__rmul__</code>
<code>nb_inplace_multiply</code>	<code>binaryfunc</code>	<code>__mul__</code>
<code>nb_remainder</code>	<code>binaryfunc</code>	<code>__mod__</code> <code>__rmod__</code>
<code>nb_inplace_remainder</code>	<code>binaryfunc</code>	<code>__mod__</code>
<code>nb_divmod</code>	<code>binaryfunc</code>	<code>__divmod__</code> <code>__rdivmod__</code>
<code>nb_power</code>	<code>ternaryfunc</code>	<code>__pow__</code> <code>__rpow__</code>
<code>nb_inplace_power</code>	<code>ternaryfunc</code>	<code>__pow__</code>
<code>nb_negative</code>	<code>unaryfunc</code>	<code>__neg__</code>
<code>nb_positive</code>	<code>unaryfunc</code>	<code>__pos__</code>
<code>nb_absolute</code>	<code>unaryfunc</code>	<code>__abs__</code>
<code>nb_bool</code>	<code>inquiry</code>	<code>__bool__</code>
<code>nb_invert</code>	<code>unaryfunc</code>	<code>__invert__</code>
<code>nb_lshift</code>	<code>binaryfunc</code>	<code>__lshift__</code> <code>__rlshift__</code>
<code>nb_inplace_lshift</code>	<code>binaryfunc</code>	<code>__lshift__</code>
<code>nb_rshift</code>	<code>binaryfunc</code>	<code>__rshift__</code> <code>__rrshift__</code>
<code>nb_inplace_rshift</code>	<code>binaryfunc</code>	<code>__rshift__</code>
<code>nb_and</code>	<code>binaryfunc</code>	<code>__and__</code> <code>__rand__</code>
<code>nb_inplace_and</code>	<code>binaryfunc</code>	<code>__and__</code>

次のページに続く

表 2 – 前のページからの続き

Slot	型	特殊メソッド
<i>nb_xor</i>	<i>binaryfunc</i>	<code>__xor__</code> <code>__rxor__</code>
<i>nb_inplace_xor</i>	<i>binaryfunc</i>	<code>__xor__</code>
<i>nb_or</i>	<i>binaryfunc</i>	<code>__or__</code> <code>__ror__</code>
<i>nb_inplace_or</i>	<i>binaryfunc</i>	<code>__or__</code>
<i>nb_int</i>	<i>unaryfunc</i>	<code>__int__</code>
<i>nb_reserved</i>	void *	
<i>nb_float</i>	<i>unaryfunc</i>	<code>__float__</code>
<i>nb_floor_divide</i>	<i>binaryfunc</i>	<code>__floordiv__</code>
<i>nb_inplace_floor_divide</i>	<i>binaryfunc</i>	<code>__floordiv__</code>
<i>nb_true_divide</i>	<i>binaryfunc</i>	<code>__truediv__</code>
<i>nb_inplace_true_divide</i>	<i>binaryfunc</i>	<code>__truediv__</code>
<i>nb_index</i>	<i>unaryfunc</i>	<code>__index__</code>
<i>nb_matrix_multiply</i>	<i>binaryfunc</i>	<code>__matmul__</code> <code>__rmatmul__</code>
<i>nb_inplace_matrix_multiply</i>	<i>binaryfunc</i>	<code>__matmul__</code>
<i>mp_length</i>	<i>lenfunc</i>	<code>__len__</code>
<i>mp_subscript</i>	<i>binaryfunc</i>	<code>__getitem__</code>
<i>mp_ass_subscript</i>	<i>objobjargproc</i>	<code>__setitem__</code> , <code>__delitem__</code>
<i>sq_length</i>	<i>lenfunc</i>	<code>__len__</code>
<i>sq_concat</i>	<i>binaryfunc</i>	<code>__add__</code>
<i>sq_repeat</i>	<i>ssizeargfunc</i>	<code>__mul__</code>
<i>sq_item</i>	<i>ssizeargfunc</i>	<code>__getitem__</code>
<i>sq_ass_item</i>	<i>ssizeobjargproc</i>	<code>__setitem__</code> <code>__delitem__</code>
<i>sq_contains</i>	<i>objobjproc</i>	<code>__contains__</code>
<i>sq_inplace_concat</i>	<i>binaryfunc</i>	<code>__iadd__</code>
<i>sq_inplace_repeat</i>	<i>ssizeargfunc</i>	<code>__imul__</code>
<i>bf_getbuffer</i>	<i>getbufferproc()</i>	
<i>bf_releasebuffer</i>	<i>releasebufferproc()</i>	

スロットの定義型 (typedef)

定義型 (typedef)	引数型	返り値型
<i>allocfunc</i>	<i>PyObject</i> * <i>PyTypeObject</i> * Py_ssize_t	<i>PyObject</i> *
<i>destructor</i>	void *	void
<i>freefunc</i>	void *	void
<i>traverseproc</i>	void * <i>visitproc</i> void *	int
<i>newfunc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
<i>initproc</i>	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	int
<i>reprfunc</i>	<i>PyObject</i> *	<i>PyObject</i> *
<i>getattrfunc</i>	<i>PyObject</i> * const char *	<i>PyObject</i> *
<i>setattrfunc</i>	<i>PyObject</i> * const char * <i>PyObject</i> *	int
<i>getattrofunc</i>	<i>PyObject</i> * <i>PyObject</i> *	<i>PyObject</i> *
<i>setattrofunc</i>	<i>PyObject</i> * <i>PyObject</i> *	int
<i>setattrofunc</i> 12.3. 型オブジェクト	<i>PyObject</i> * <i>PyObject</i> * <i>PyObject</i> *	int

See *Slot Type typedefs* below for more detail.

12.3.2 PyObject 定義

PyObject の構造体定義は `Include/object.h` で見つけられるはずです。参照の手間を省くために、ここでは定義を繰り返します:

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    Py_ssize_t tp_vectorcall_offset;
    getattrofunc tp_getattr;
    setattrofunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                   or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

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

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    unsigned long tp_flags;

    const char *tp_doc; /* Documentation string */

    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* rich comparisons */
```

(次のページに続く)

(前のページからの続き)

```

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;
struct _typeobject *tp_base;
PyObject *tp_dict;
descrgetfunc tp_descr_get;
descrsetfunc tp_descr_set;
Py_ssize_t tp_dictoffset;
initproc tp_init;
allocfunc tp_alloc;
newfunc tp_new;
freefunc tp_free; /* Low-level free-memory routine */
inquiry tp_is_gc; /* For PyObject_IS_GC */
PyObject *tp_bases;
PyObject *tp_mro; /* method resolution order */
PyObject *tp_cache;
PyObject *tp_subclasses;
PyObject *tp_weaklist;
destructor tp_del;

/* Type attribute cache version tag. Added in version 2.6 */
unsigned int tp_version_tag;

destructor tp_finalize;
} PyTypeObject;

```

12.3.3 PyObject スロット

型オブジェクト構造体は *PyVarObject* 構造体を拡張したものです。ob_size フィールドは、(通常 class 文が呼び出す `type_new()` で生成される) 動的な型に使用します。*PyType_Type* (メタタイプ) は `tp_itemsize` を初期化するので注意してください。すなわち、インスタンス (つまり型オブジェクト) には ob_size フィールドが存在しなければなりません。

*PyObject** PyObject._ob_next

*PyObject** PyObject._ob_prev

これらのフィールドはマクロ `Py_TRACE_REFS` が定義されている場合のみ存在します。

`PyObject_HEAD_INIT` マクロを使うと、フィールドを NULL に初期化します。静的にメモリ確保されているオブジェクトでは、これらのフィールドは常に NULL のままです。動的にメモリ確保される

オブジェクトの場合、これら二つのフィールドは、ヒープ上の **全ての** 存続中のオブジェクトからなる二重リンクリストでオブジェクトをリンクする際に使われます。このことは様々なデバッグ目的に利用できます; 現状では、環境変数 `PYTHONDUMPPREFS` が設定されているときに、プログラムの実行終了時点で存続しているオブジェクトを出力するのが唯一の用例です。

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 PyObject.ob_refcnt

型オブジェクトの参照カウントで、`PyObject_HEAD_INIT` はこの値を 1 に初期化します。静的にメモリ確保された型オブジェクトでは、型のインスタンス (`ob_type` が該当する型を指しているオブジェクト) は参照をカウントする対象には **なりません**。動的にメモリ確保される型オブジェクトの場合、インスタンスは参照カウントの対象に **なります**。

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

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;
```

上の操作は、該当する型のいかなるインスタンス生成よりも前にしておかなければなりません。`PyType_Ready()` は `ob_type` が `NULL` かどうか調べ、`NULL` の場合には基底クラスの `ob_type` フィールドで初期化します。`ob_type` フィールドがゼロでない場合、`PyType_Ready()` はこのフィールドを変更しません。

継承:

サブタイプはこのフィールドを継承します。

12.3.4 PyVarObject スロット

Py_ssize_t PyVarObject.ob_size

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。

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*

型の名前が入っている NUL 終端された文字列へのポインタです。モジュールのグローバル変数としてアクセスできる型の場合、この文字列は完全なモジュール名、ドット、そして型の名前と続く文字列になります; 組み込み型の場合、ただの型の名前です。モジュールがあるパッケージのサブモジュールの場合、完全なパッケージ名が完全なモジュール名の一部になっています。例えば、パッケージ P 内のサブモジュール Q に入っているモジュール M 内で定義されている T は、*tp_name* を "P.Q.M.T" に初期化します。

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。

静的にメモリ確保される型オブジェクトの場合、*tp_name* フィールドにはドットが含まれているはずで、最後のドットよりも前にある部分文字列全体は *__module__* 属性として、またドットよりも後ろにある部分は *__name__* 属性としてアクセスできます。

ドットが入っていない場合、*tp_name* フィールドの内容全てが *__name__* 属性になり、*__module__* 属性は (前述のように型の辞書内で明示的にセットしないかぎり) 未定義になります。このため、その型は pickle 化できないことになります。さらに、pydoc が作成するモジュールドキュメントのリストにも載らなくなります。

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。

型には二つの種類があります: 固定長インスタンスの型は、*tp_itemsize* フィールドがゼロで、可変長インスタンスの方は *tp_itemsize* フィールドが非ゼロの値になります。固定長インスタンスの型の場合、全てのインスタンスは等しく *tp_basicsize* で与えられたサイズになります。

可変長インスタンスの型の場合、インスタンスには *ob_size* フィールドがなくではなく、インスタンスのサイズは N をオブジェクトの "長さ" として、*tp_basicsize* と *tp_itemsize* の N 倍を足したものになります。N の値は通常、インスタンスの *ob_size* フィールドに記憶されます。ただし例外がいくつかあります: 例えば、整数では負の値を *ob_size* に使って、インスタンスの表す値が負であることを示し、N 自体は *abs(ob_size)* になります。また、*ob_size* フィールドがあるからといって、必ずしもインスタンスが可変長であることを意味しません (例えば、リスト型の構造体は固定長のインスタンスになるにもかかわらず、インスタンスにはちゃんと意味を持った *ob_size* フィールドがあり

ます)。

基本サイズには、`PyObject_HEAD` マクロまたは `PyObject_VAR_HEAD` マクロ (インスタンス構造体を宣言するのに使ったどちらかのマクロ) で宣言されているフィールドが入っています。さらに、`_ob_prev` および `_ob_next` フィールドがある場合、これらのフィールドもサイズに加算されます。従って、`tp_basicsize` の正しい初期化値を得るには、インスタンスデータのレイアウトを宣言するのに使う構造体に対して `sizeof` 演算子を使うしかありません。基本サイズには、GC ヘッダサイズは入っていません。

アラインメントに関する注釈: 変数の各要素を配置する際に特定のアラインメントが必要となる場合、`tp_basicsize` の値に気をつけなければなりません。例: ある型が `double` の配列を実装しているとします。`tp_itemsize` は `sizeof(double)` です。`tp_basicsize` が `sizeof(double)` (ここではこれを `double` のアラインメントが要求するサイズと仮定する) の個数分のサイズになるようにするのはプログラマの責任です。

For any type with variable-length instances, this field must not be NULL.

継承:

これらのフィールドはサブタイプに別々に継承されます。基底タイプが 0 でない `tp_itemsize` を持っていた場合、基底タイプの実装に依存しますが、一般的にはサブタイプで別の 0 で無い値を `tp_itemsize` に設定するのは安全ではありません。

destructor `PyTypeObject.tp_dealloc`

インスタンスのデストラクタ関数へのポインタです。この関数は (単量子 `None` や `Ellipsis` の場合のように) インスタンスが決してメモリ解放されない型でない限り必ず定義しなければなりません。シグネチャは次の通りです:

```
void tp_dealloc(PyObject *self);
```

デストラクタ関数は、参照カウントが新たにゼロになった際に `Py_DECREF()` や `Py_XDECREF()` マクロから呼び出されます。呼び出された時点では、インスタンスはまだ存在しますが、インスタンスに対する参照は全くない状態です。デストラクタ関数はインスタンスが保持している全ての参照を解放し、インスタンスが確保している全てのメモリバッファを (バッファの確保時に使った関数に対応するメモリ解放関数を使って) 解放し、その型の `tp_free` 関数を呼び出します。ある型がサブタイプを作成できない (`Py_TPFLAGS_BASETYPE` フラグがセットされていない) 場合、`tp_free` の代わりにオブジェクトのメモリ解放関数 (deallocators) を直接呼び出してもかまいません。オブジェクトのメモリ解放関数は、インスタンスのメモリ確保を行う際に使った関数に対応したものでなければなりません; インスタンスを `PyObject_New()` や `PyObject_VarNew()` でメモリ確保した場合には、通常 `PyObject_Del()` を使い、`PyObject_GC_New()` や `PyObject_GC_NewVar()` で確保した場合には `PyObject_GC_Del()` を使います。

Finally, if the type is heap allocated (`Py_TPFLAGS_HEAPTYPE`), the deallocator should decrement the reference count for its type object after calling the type deallocator. In order to avoid dangling pointers, the recommended way to achieve this is:

```
static void foo_dealloc(foo_object *self) {
    PyTypeObject *tp = Py_TYPE(self);
    // free references and buffers here
    tp->tp_free(self);
    Py_DECREF(tp);
}
```

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 signature is the same as for `_PyObject_Vectorcall()`:

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

The *vectorcallfunc* pointer may be zero, 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`:

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

Call *callable*'s *vectorcallfunc* with positional and keyword arguments given in a tuple and dict, respectively.

This function is intended to be used in the *tp_call* slot. It does not fall back to *tp_call* and it currently does not check the `_Py_TPFLAGS_HAVE_VECTORCALL` flag. To call an object, use one of the *PyObject_Call* functions instead.

注釈: It is not recommended for *heap types* to implement the vectorcall protocol. When a user sets `__call__` in Python code, only *tp_call* is updated, possibly making it inconsistent with the vectorcall function.

注釈: The semantics of the `tp_vectorcall_offset` slot are provisional and expected to be finalized in Python 3.9. If you use vectorcall, plan for updating your code for Python 3.9.

バージョン 3.8 で変更: This slot was used for print formatting in Python 2.x. In Python 3.0 to

3.7, it was reserved and named `tp_print`.

継承:

This field is inherited by subtypes together with `tp_call`: a subtype inherits `tp_vectorcall_offset` from its base type when the subtype's `tp_call` is NULL.

Note that *heap types* (including subclasses defined in Python) do not inherit the `_Py_TPFLAGS_HAVE_VECTORCALL` flag.

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。

バージョン 3.5 で追加: 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 *);
```

この関数は C の long 型の値を返さねばなりません。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いいての場合は便利です。

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()*:

```
PyObject *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いいての場合は便利です。

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:

このフィールドの継承は込み入っています。ほとんどのフラグは個別に継承されます。すなわち、基底タイプのフラグビットが設定されていたら、サブタイプのフラグビットもそれを引き継ぎます。拡張機能構造体が継承される場合は、拡張機能構造体に関するフラグビットは厳密に継承されます。すなわち、基底タイプのフラグビットの値は、拡張機能構造体へのポインタと共に、サブタイプにコピーされます。`Py_TPFLAGS_HAVE_GC` フラグビットは `tp_traverse` フィールドと `tp_clear` フィールドと共に継承されます。すなわち、サブタイプにおいて、`Py_TPFLAGS_HAVE_GC` フラグビットがクリアされていて、`tp_traverse` フィールドと `tp_clear` フィールドが存在し `NULL` になっている場合に継承されます。

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 INCREMENTED when a new instance is created, and DECREMENTED when an instance is destroyed (this does not apply to instances of subtypes; only the type referenced by the instance's `ob_type` gets INCREMENTED or DECREMENTED).

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

オブジェクトがガベージコレクション (GC) をサポートする場合にセットされるビットです。このビットがセットされている場合、インスタンスは *PyObject_GC_New()* を使って生成し、*PyObject_GC_Del()* を使って破棄しなければなりません。詳しい情報は [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。このビットは、GC に関連するフィールド *tp_traverse* および *tp_clear* が型オブジェクト内に存在することも示しています。

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

型オブジェクトおよび拡張機能構造体の特定のフィールドの存在の有無に関連する全てのビットからなるビットマスクです。現状では、このビットマスクには以下のビット: *Py_TPFLAGS_HAVE_STACKLESS_EXTENSION* および *Py_TPFLAGS_HAVE_VERSION_TAG* が入っています。

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, **kwargs)` (with `obj` not `None`) must be equivalent to `meth(obj, *args, **kwargs)`.
- `meth.__get__(None, cls)(*args, **kwargs)` must be equivalent to `meth(*args, **kwargs)`.

This flag enables an optimization for typical method calls like `obj.meth()`: it avoids creating a temporary "bound method" object for `obj.meth`.

バージョン 3.8 で追加.

継承:

This flag is never inherited by heap types. For extension types, it is inherited whenever `tp_descr_get` is inherited.

`Py_TPFLAGS_LONG_SUBCLASS`

`Py_TPFLAGS_LIST_SUBCLASS`

`Py_TPFLAGS_TUPLE_SUBCLASS`

`Py_TPFLAGS_BYTES_SUBCLASS`

`Py_TPFLAGS_UNICODE_SUBCLASS`

`Py_TPFLAGS_DICT_SUBCLASS`

`Py_TPFLAGS_BASE_EXC_SUBCLASS`

`Py_TPFLAGS_TYPE_SUBCLASS`

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に関わるコードでは、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。

バージョン 3.4 で追加.

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 set on *static* subtypes if `tp_flags` is not overridden: a subtype inherits `_Py_TPFLAGS_HAVE_VECTORCALL` from its base type when the subtype's `tp_call` is `NULL`

and the subtype's `Py_TPFLAGS_HEAPTYPE` is not set.

Heap types do not inherit `_Py_TPFLAGS_HAVE_VECTORCALL`.

注釈: This flag is provisional and expected to become public in Python 3.9, with a different name and, possibly, changed semantics. If you use `vectorcall`, plan for updating your code for Python 3.9.

バージョン 3.8 で追加.

`const char* PyTypeObject.tp_doc`

オプションのポインタで、この型オブジェクトの docstring を与える NUL 終端された C の文字列を指します。この値は型オブジェクトと型のインスタンスにおける `__doc__` 属性として公開されます。

継承:

サブタイプはこのフィールドを継承 **しません**。

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).

`Py_VISIT()` は `local_traverse()` が `visit` と `arg` という決まった名前の引数を持つことを要求します。

継承:

Group: `Py_TPFLAGS_HAVE_GC`, `tp_traverse`, `tp_clear`

このフィールドは `tp_clear` および `Py_TPFLAGS_HAVE_GC` フラグビット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_traverse` および `tp_clear` が両方ともゼロの場合、サブタイプは基底タイプから `tp_traverse` と `tp_clear` を両方とも継承します。

inquiry `PyObject.tp_clear`

オプションのポインタで、ガベージコレクタにおける消去関数 (clear function) を指します。`Py_TPFLAGS_HAVE_GC` がセットされている場合にのみ使われます。シグネチャは次の通りです:

```
int tp_clear(PyObject *);
```

`tp_clear` メンバ関数は GC が検出した循環しているゴミの循環参照を壊すために用いられます。総合的な視点で考えると、システム内の全ての `tp_clear` 関数が連携して、全ての循環参照を破壊しなければなりません。(訳注: ある型が `tp_clear` を実装しなくても全ての循環参照が破壊できるのであれば実装しなくても良い) これはとても繊細で、もし少しでも不確かな部分があるのであれば、`tp_clear` 関数を提供すべきです。例えば、タプルは `tp_clear` を実装しません。なぜなら、タプルだけで構成された循環参照が見つかることは無いからです。従って、タプル以外の型の `tp_clear` 関数だけで、タプルを含むどんな循環参照も必ず破壊できることになります。これは簡単に判ることではなく、`tp_clear` の実装を避ける良い理由はめったにありません。

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;
}
```

参照のクリアはデリケートなので、`Py_CLEAR()` マクロを使うべきです: ポインタを `NULL` にセットするまで、そのオブジェクトの参照カウントをデクリメントしてはいけません。参照カウントのデクリメントすると、そのオブジェクトが破棄されるかもしれず、(そのオブジェクトに関連付けられたファイナライザ、弱参照のコールバックにより) 任意の Python コードの実行を含む後片付け処理が実行されるかもしれないからです。もしそういったコードが再び `self` を参照することがあれば、すでに持っていたオブジェクトへのポインタは `NULL` になっているので、`self` は所有していたオブジェクトをもう利用できないことを認識できます。`Py_CLEAR()` マクロはその手続きを安全な順番で実行します。

`tp_clear` 関数の目的は参照カウントを破壊することなので、Python 文字列や Python 整数のような、循環参照に含むことのできないオブジェクトをクリアする必要はありません。一方、所有する全ての Python オブジェクトをクリアするようにし、その型の `tp_dealloc` 関数が `tp_clear` 関数を実行するようにすると実装が楽になるでしょう。

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`

このフィールドは `tp_traverse` および `Py_TPFLAGS_HAVE_GC` フラグビット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_traverse` および `tp_clear` が両方ともゼロの場合、サブタイプは基底タイプから `tp_traverse` と `tp_clear` を両方とも継承します。

richcmpfunc `PyObject.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 *PyObject*.

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
<code>Py_LT</code>	<code><</code>
<code>Py_LE</code>	<code><=</code>
<code>Py_EQ</code>	<code>==</code>
<code>Py_NE</code>	<code>!=</code>
<code>Py_GT</code>	<code>></code>
<code>Py_GE</code>	<code>>=</code>

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。

返り値の参照カウントは適切にインクリメントされます。

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。

バージョン 3.7 で追加。

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

`PyObject_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 PyObject.tp_weaklistoffset`

If the instances of this type are weakly referenceable, this field is greater than zero and contains the offset in the instance structure of the weak reference list head (ignoring the GC header, if present); this offset is used by `PyObject_ClearWeakRefs()` and the `PyWeakref_*()` functions. The instance structure needs to include a field of type `PyObject*` which is initialized to NULL.

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。

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ー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。

`class` 文で定義された型に `__slots__` 宣言が全くなく、かつ基底タイプが弱参照可能でない場合、その型を弱参照可能にするには弱参照リストの先頭を表すスロットをインスタンスデータレイアウト構造体に追加し、スロットのオフセットを `tp_weaklistoffset` に設定します。

型の `__slots__` の宣言に `__weakref__` という名前のスロットが含まれているとき、スロットはその型のインスタンスにおける弱参照リストの先頭を表すスロットになり、スロットのオフセットが型の `tp_weaklistoffset` に入ります。

型の `__slots__` 宣言が `__weakref__` という名前のスロットを含んでいないとき、その型は基底タイプから `tp_weaklistoffset` を継承します。

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ない、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립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립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。

このフィールドは通常、*PyType_Ready()* を呼び出す前に NULL に初期化しておかなければなりません; あるいは、型の初期属性の入った辞書で初期化しておいてもかまいません。*PyType_Ready()* が型をひとたび初期化すると、型の新たな属性をこの辞書に追加できるのは、属性が (*__add__*) のような オーバロード用演算でないときだけです。

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*

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。

このフィールドの値がゼロより大きければ、値はインスタンス構造体の先頭からの オフセットを表します。値がゼロより小さければ、インスタンス構造体の **末尾** からのオフセットを表します。負のオフセットを使うコストは比較的高くつくので、インスタンス構造体に可変長部分があるときのみ使うべきです。例えば、*str* や *tuple* のサブタイプにインスタンス変数の辞書を追加する場合には、負のオフセットを使います。この場合、たとえ辞書が基本のオブジェクトレイアウトに含まれていなくても、*tp_basicsize* フィールドは追加された辞書を考慮にいれなければならないことに注意してください。

ポインタサイズが 4 バイトのシステムでは、構造体の最後尾に辞書が宣言されていることを示す場合、`tp_dictoffset` を -4 にしなければなりません。

負の `tp_dictoffset` から、インスタンスでの実際のオフセットを計算するには以下のようにします:

```
dictoffset = tp_basicsize + abs(ob_size)*tp_itemsize + tp_dictoffset
if dictoffset is not aligned on sizeof(void*):
    round up to sizeof(void*)
```

ここで、`tp_basicsize`、`tp_itemsize` および `tp_dictoffset` は型オブジェクトから取り出され、`ob_size` はインスタンスから取り出されます。絶対値を取っているのは、整数は符号を記憶するのに `ob_size` の符号を使うためです。(この計算を自分で行う必要はまったくありません; 計算は `_PyObject_GetDictPtr()` がやってくれます。)

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ーライドすることがあります; 従って、サブタイプでは辞書のオフセットが基底タイプとは異なる場合があります。辞書のオフセットは常に `tp_dictoffset` 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。

`class` 文で定義された型に `__slots__` 宣言がなく、かつ基底タイプの全てにインスタンス変数辞書がない場合、辞書のスロットをインスタンスデータレイアウト構造体に追加し、スロットのオフセットを `tp_dictoffset` に設定します。

`class` 文で定義された型に `__slots__` 宣言がある場合、この型は基底タイプから `tp_dictoffset` を継承します。

(`__dict__` という名前のスロットを `__slots__` 宣言に追加しても、期待どおりの効果は得られず、単に混乱を招くだけになります。とはいえ、これは将来 `__weakref__` のように追加されるはずです。)

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.

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。

この関数はクラスにおける `__init__()` メソッドに対応します。`__init__()` と同様、`__init__()` を呼び出さずにインスタンスを作成できます。また、`__init__()` を再度呼び出してインスタンスの再初期化もできます。

関数のシグネチャは次のとおりです

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwargs);
```

`self` 引数は初期化するインスタンスです; `args` および `kwargs` 引数は、`__init__()` を呼び出す際の位置引数およびキーワード引数です。

`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`

オブ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`

オブ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 *kwargs);
```

`subtype` 引数は生成するオブジェクトの型です; `args` および `kwargs` 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。サブタイプは `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、変更可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。

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);
```

このシグネチャと互換性のある初期化子は `PyObject_Free()` です。

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。

ガベージコレクタは、オブジェクトを回収して良いかどうかを知る必要があります。通常は、オブジェクトの型の `tp_flags` フィールドを見て、`Py_TPFLAGS_HAVE_GC` フラグビットを調べるだけで十分です。しかし、ある型では静的にメモリ確保されたインスタンスと動的にメモリ確保されたインスタンスが混じっていて、静的にメモリ確保されたインスタンスは回収できません。こうした型では、関数を定義しなければなりません; 関数はインスタンスが回収可能の場合には 1 を、回収不能の場合には 0 を返さねばなりません。シグネチャは:

```
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。

`class` 文で生成されたクラスの場合このフィールドがセットされます。静的に定義されている型の場合には、このフィールドは NULL になります。

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。

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。

*PyObject** `PyTypeObject.tp_subclasses`

サブクラスへの弱参照からなるリストです。内部で使用するためだけのものです。

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。

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`

オプションのポインタで、インスタンスの終了処理関数を指します。シグネチャは以下の通りです:

```
void tp_finalize(PyObject *self);
```

`tp_finalize` が設定されている場合、インスタンスをファイナライズするときに、インタプリタがこの関数を 1 回呼び出します。ガベージコレクタ (このインスタンスが孤立した循環参照の一部だった場合) やオブジェクトが破棄される直前にもこの関数は呼び出されます。どちらの場合でも、循環参照を破壊しようとする前に呼び出されることが保証されていて、確実にオブジェクトが正常な状態にあるようにします。

`tp_finalize` は現在の例外状態を変更すべきではありません; 従って、単純でないファイナライザを書くには次の方法が推奨されます:

```
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);
}
```

このフィールドを (継承した場合も含めて) 考慮から漏らさないように、`Py_TPFLAGS_HAVE_FINALIZE` フラグビットも設定し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。

バージョン 3.4 で追加.

参考:

” オブジェクトの安全な終了処理” ([PEP 442](#))

残りのフィールドは、機能テスト用のマクロである `COUNT_ALLOCS` が定義されている場合のみ利用でき、内部で使用するためだけのものです。これらのフィールドについて記述するのは単に完全性のためです。サブタイプはこれらのフィールドを継承しません。

`Py_ssize_t PyObject.tp_allocs`

メモリ確保の回数です。

`Py_ssize_t PyObject.tp_frees`

メモリ解放の回数です。

`Py_ssize_t PyObject.tp_maxalloc`

同時にメモリ確保できる最大オブジェクト数です。

*PyTypeObject** `PyTypeObject.tp_prev`

Pointer to the previous type object with a non-zero *tp_allocs* field.

*PyTypeObject** `PyTypeObject.tp_next`

次のゼロでない *tp_allocs* フィールドを持つ型オブジェクトへのポインタです。

また、Python のガベージコレクションでは、*tp_dealloc* を呼び出すのはオブジェクトを生成したスレッドだけではなく、任意の Python スレッドかもしれないという点にも注意して下さい。(オブジェクトが循環参照の一部の場合、任意のスレッドのガベージコレクションによって解放されてしまうかもしれません)。Python API 側からみれば、*tp_dealloc* を呼び出すスレッドはグローバルインタプリタロック (GIL: Global Interpreter Lock) を獲得するので、これは問題ではありません。しかしながら、削除されようとしているオブジェクトが何らかの C や C++ ライブラリ由来のオブジェクトを削除する場合、*tp_dealloc* を呼び出すスレッドのオブジェクトを削除することで、ライブラリの仮定している何らかの規約に違反しないように気を付ける必要があります。

12.3.6 Heap Types

Traditionally, types defined in C code are *static*, that is, a static *PyTypeObject* structure is defined directly in code and initialized using *PyType_Ready()*.

This results in types that are limited relative to types defined in Python:

- Static types are limited to one base, i.e. they cannot use multiple inheritance.
- Static type objects (but not necessarily their instances) are immutable. It is not possible to add or modify the type object's attributes from Python.
- Static type objects are shared across *sub-interpreters*, so they should not include any subinterpreter-specific state.

Also, since *PyTypeObject* is not part of the *stable ABI*, any extension modules using static types must be compiled for a specific Python minor version.

An alternative to static types is *heap-allocated types*, or *heap types* for short, which correspond closely to classes created by Python's `class` statement.

This is done by filling a *PyType_Spec* structure and calling *PyType_FromSpecWithBases()*.

12.4 数値オブジェクト構造体

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;
    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、他のエラーが起きた場合は、`NULL` を返して例外を設定しなければなりません。

注釈: `nb_reserved` フィールドは常に `NULL` でなければなりません。以前は `nb_long` と呼ばれていて、Python 3.0.1 で名前が変更されました。

binaryfunc `PyNumberMethods.nb_add`

binaryfunc `PyNumberMethods.nb_subtract`

binaryfunc `PyNumberMethods.nb_multiply`

binaryfunc `PyNumberMethods.nb_remainder`

binaryfunc `PyNumberMethods.nb_divmod`

ternaryfunc `PyNumberMethods.nb_power`

unaryfunc `PyNumberMethods.nb_negative`

unaryfunc `PyNumberMethods.nb_positive`

unaryfunc `PyNumberMethods.nb_absolute`

inquiry `PyNumberMethods.nb_bool`

unaryfunc `PyNumberMethods.nb_invert`

binaryfunc `PyNumberMethods.nb_lshift`

binaryfunc `PyNumberMethods.nb_rshift`

binaryfunc `PyNumberMethods.nb_and`

binaryfunc `PyNumberMethods.nb_xor`

binaryfunc `PyNumberMethods.nb_or`

unaryfunc `PyNumberMethods.nb_int`

`void *PyNumberMethods.nb_reserved`

unaryfunc `PyNumberMethods.nb_float`

binaryfunc `PyNumberMethods.nb_inplace_add`

binaryfunc `PyNumberMethods.nb_inplace_subtract`

binaryfunc `PyNumberMethods.nb_inplace_multiply`

binaryfunc `PyNumberMethods.nb_inplace_remainder`

ternaryfunc `PyNumberMethods.nb_inplace_power`

binaryfunc `PyNumberMethods.nb_inplace_lshift`

binaryfunc `PyNumberMethods.nb_inplace_rshift`

binaryfunc `PyNumberMethods.nb_inplace_and`

binaryfunc `PyNumberMethods.nb_inplace_xor`

binaryfunc `PyNumberMethods.nb_inplace_or`

binaryfunc `PyNumberMethods.nb_floor_divide`

binaryfunc `PyNumberMethods.nb_true_divide`

binaryfunc `PyNumberMethods.nb_inplace_floor_divide`

binaryfunc `PyNumberMethods.nb_inplace_true_divide`

unaryfunc `PyNumberMethods.nb_index`

binaryfunc `PyNumberMethods.nb_matrix_multiply`

binaryfunc `PyNumberMethods.nb_inplace_matrix_multiply`

12.5 マップオブジェクト構造体

`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、`PySequence_GetSlice()` 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`

この関数は `PyObject_SetItem()`、`PyObject_DelItem()`、`PyObject_SetSlice()` および `PyObject_DelSlice()` から利用されます。`PyObject_SetItem()` と同じシグネチャを持ちますが、`v` に `NULL` を設定して要素の削除もできます。このスロットが `NULL` の場合は、このオブジェクトはアイテムの代入と削除をサポートしません。

12.6 シーケンスオブジェクト構造体

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.

負のインデックスは次のように処理されます: *sq_length* スロットが埋められていれば、それを出してシーケンスの長さから正のインデックスを計算し、*sq_item* に渡します。*sq_length* が NULL の場合は、インデックスはそのままこの関数に渡されます。

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 aug-

mented assignment `*`, after trying numeric in-place multiplication via the `nb_inplace_multiply` slot.

12.7 バッファオブジェクト構造体 (buffer object structure)

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、エクスポート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) リクエストが合致するか確認します。合致しない場合は、`PyExc_BufferError` を送出し、`view->obj` に `NULL` を設定し `-1` を返します。
- (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ー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トに属し、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。

エクスポートは、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フェースです。

12.8 async オブジェクト構造体

バージョン 3.5 で追加.

`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;
} 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);
```

awaitable オブジェクトを返さなければなりません。詳しいことは `__anext__()` を参照してください。

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);
```

awaitable オブジェクトを返さなければなりません。詳しいことは `__anext__()` を参照してください。
このスロットは `NULL` に設定されていることもあります。

12.9 Slot Type `typedefs`

*PyObject **(**allocfunc*)(*PyTypeObject *cls*, *Py_ssize_t nitems*)

この関数の目的は、メモリ確保をメモリ初期化から分離することにあります。この関数は、インスタンス用の的確なサイズ、適切なアラインメント、ゼロによる初期化がなされ、`ob_refcnt` を 1 に、`ob_type` を型引数 (type argument) にセットしたメモリブロックへのポインタを返さねばなりません。型の `tp_itemsize` がゼロでない場合、オブジェクトの `ob_size` フィールドは `nitems` に初期化され、確保されるメモリブロックの長さは `tp_basicsize + nitems*tp_itemsize` を `sizeof(void*)` の倍数に切り上げた値になるはずです; それ以外の場合、`nitems` の値は使われず、メモリブロックの長さは `tp_basicsize` になるはずです。

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。

`void (*destructor)(PyObject *)`

*PyObject **(**vectorcallfunc*)(*PyObject *callable*, *PyObject *const *args*, *size_t nargsf*, *PyObject *kwnames*)
See `tp_vectorcall_offset`.

Arguments to `vectorcallfunc` are the same as for `_PyObject_Vectorcall()`.

バージョン 3.8 で追加。

`void (*freefunc)(void *)`

`tp_free` を参照してください。

*PyObject **(**newfunc*)(*PyObject **, *PyObject **, *PyObject **)

`tp_new` を参照してください。

```

int (*initproc)(PyObject *, PyObject *, PyObject *)
    tp_init を参照してください。

PyObject *(*reprfunc)(PyObject *)
    tp_repr を参照してください。

PyObject *(*getattrfunc)(PyObject *self, char *attr)
    オブジェクトの属性の値を返します。

int (*setattrfunc)(PyObject *self, char *attr, PyObject *value)
    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 引数に NULL を設定します。

PyObject *(*getattrofunc)(PyObject *self, PyObject *attr)
    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。

int (*setattrofunc)(PyObject *self, PyObject *attr, PyObject *value)
    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 引数に NULL を設定します。

    tp_setattro を参照してください。

PyObject *(*descrgetfunc)(PyObject *, PyObject *, PyObject *)
    tp_descrget を参照してください。

int (*descrsetfunc)(PyObject *, PyObject *, PyObject *)
    tp_descrset を参照してください。

Py_hash_t (*hashfunc)(PyObject *)
    tp_hash を参照してください。

PyObject *(*richcmpfunc)(PyObject *, PyObject *, int)
    tp_richcompare を参照してください。

PyObject *(*getiterfunc)(PyObject *)
    tp_iter を参照してください。

PyObject *(*iternextfunc)(PyObject *)
    tp_iternext を参照してください。

Py_ssize_t (*lenfunc)(PyObject *)

int (*getbufferproc)(PyObject *, Py_buffer *, int)

void (*releasebufferproc)(PyObject *, Py_buffer *)

PyObject *(*unaryfunc)(PyObject *)

PyObject *(*binaryfunc)(PyObject *, PyObject *)

PyObject *(*ternaryfunc)(PyObject *, PyObject *, PyObject *)

PyObject *(*ssizeargfunc)(PyObject *, Py_ssize_t)

```

```
int (*ssizeobjargproc)(PyObject *, Py_ssize_t)

int (*objobjproc)(PyObject *, PyObject *)

int (*objobjargproc)(PyObject *, PyObject *, PyObject *)
```

12.10 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。

基本的な静的型:

```
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 = "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 */
};
```

(次のページに続く)

(前のページからの続き)

```

0,                /* tp_setattro */
0,                /* tp_as_buffer */
0,                /* tp_flags */
"My objects",     /* tp_doc */
0,                /* tp_traverse */
0,                /* tp_clear */
0,                /* tp_richcompare */
0,                /* tp_weaklistoffset */
0,                /* tp_iter */
0,                /* tp_iternext */
0,                /* tp_methods */
0,                /* tp_members */
0,                /* tp_getset */
0,                /* tp_base */
0,                /* tp_dict */
0,                /* tp_descr_get */
0,                /* tp_descr_set */
0,                /* tp_dictoffset */
0,                /* tp_init */
0,                /* tp_alloc */
myobj_new,        /* tp_new */
};

```

弱酸庄屋、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;
    PyObject *inst_dict;
    PyObject *weakreflist;
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
    .tp_basicsize = sizeof(MyObject),
    .tp_doc = "My objects",
    .tp_weaklistoffset = offsetof(MyObject, weakreflist),
    .tp_dictoffset = offsetof(MyObject, inst_dict),
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
    .tp_new = myobj_new,
    .tp_traverse = (traverseproc)myobj_traverse,
    .tp_clear = (inquiry)myobj_clear,
    .tp_alloc = PyType_GenericNew,
    .tp_dealloc = (destructor)myobj_dealloc,
    .tp_repr = (reprfunc)myobj_repr,
    .tp_hash = (hashfunc)myobj_hash,
    .tp_richcompare = PyBaseObject_Type.tp_richcompare,
};

```

サブクラス化を許可せず、インスタンスを生成するのに呼び出せない（つまり、別のファクトリー関数を呼び

出す必要のある) str のサブクラス:

```
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 = "my custom str",
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = NULL,
    .tp_repr = (reprfunc)myobj_repr,
};
```

もっとも単純な静的型 (固定長のインスタンス):

```
typedef struct {
    PyObject_HEAD
} MyObject;

static PyTypeObject MyObject_Type = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "mymod.MyObject",
};
```

もっとも単純な静的型 (可変長インスタンス)

```
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 *),
};
```


12.11 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まれます) 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。

コンテナ型を作るには、型オブジェクトの `tp_flags` フィールドに `Py_TPFLAGS_HAVE_GC` フラグが立っており、`tp_traverse` ハンドラの実装を提供しなければなりません。実装する型のインスタンスが変更可能な場合は、`tp_clear` の実装も提供しなければなりません。

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. オブジェクトのメモリは `PyObject_GC_New()` または `PyObject_GC_NewVar()` で確保しなければなりません。
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。

TYPE* `PyObject_GC_New`(TYPE, *PyTypeObject* *type)

`PyObject_New()` に似ていますが、`Py_TPFLAGS_HAVE_GC` のセットされたコンテナオブジェクト用です。

TYPE* `PyObject_GC_NewVar`(TYPE, *PyTypeObject* *type, Py_ssize_t size)

`PyObject_NewVar()` に似ていますが、`Py_TPFLAGS_HAVE_GC` のセットされたコンテナオブジェクト用です。

TYPE* `PyObject_GC_Resize`(TYPE, *PyVarObject* *op, Py_ssize_t newsize)

`PyObject_NewVar()` が確保したオブジェクトのメモリをリサイズします。リサイズされたオブジェクトを返します。失敗すると NULL を返します。op はコレクタに追跡されてはなりません。

void `PyObject_GC_Track`(*PyObject* *op)

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。

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。

void `PyObject_GC_Del`(void **op*)

`PyObject_GC_New()` や `PyObject_GC_NewVar()` を使って確保されたメモリを解放します。

void `PyObject_GC_UnTrack`(void **op*)

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: `_PyObject_GC_TRACK()` マクロと `_PyObject_GC_UNTRACK()` マクロは公開 C API から外されました。

`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:

int (**visitproc*)(*PyObject* **object*, void **arg*)

`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いとなりません:

int (**traverseproc*)(*PyObject* **self*, *visitproc* *visit*, void **arg*)

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* へのパラメー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、一文字も変わらず *visit* と *arg* でなければなりません:

void `Py_VISIT`(*PyObject* **o*)

o が NULL でなければ、*o* と *arg* を引数にして *visit* コールバックを呼び出します。*visit* がゼロでない値を返した場合、その値を返します。このマクロを使うと、`tp_traverse` ハンドラは次のようになります:

```
static int
my_traverse(Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT(self->foo);
    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。

int (**inquiry*)(*PyObject* **self*)

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いないので、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。

API と ABI のバージョニング

PY_VERSION_HEX は、Python のバージョン番号を単一の整数に符号化したものです。

例えば、PY_VERSION_HEX に 0x030401a2 が設定されていれば、その値を下記のように 32 ビットの値として扱うことで、バージョン情報を得ることができます：

bytes	ビット (ビッグ エンディアン オーダ)	意味
1	1-8	PY_MAJOR_VERSION (3.4.1a2 中の 3)
2	9-16	PY_MINOR_VERSION (3.4.1a2 中の 4)
3	17-24	PY_MICRO_VERSION (3.4.1a2 中の 1)
4	25-28	PY_RELEASE_LEVEL (アルファ版では 0xA、ベータ版では 0xB、リリース候補版では 0xC、そして最終版は 0xF)、この例ではアルファ版を意味しています。
	29-32	PY_RELEASE_SERIAL (3.4.1a2 中の 2、最終リリースでは 0)

従って、3.4.1a2 は hexversion で 0x030401a2 です。

これらのマクロは `Include/patchlevel.h` で定義されています。

用語集

>>> インタラクティブシェルにおけるデフォルトの Python プロンプトです。インタプリタでインタラクティブに実行されるコード例でよく出てきます。

... 次のものが考えられます:

- インタラクティブシェルにおいて、インデントされたコードブロック、対応する左右の区切り文字の組 (丸括弧、角括弧、波括弧、三重引用符) の内側、デコレーターの後に、コードを入力する際に表示されるデフォルトの Python プロンプトです。
- 組み込みの定数 `Ellipsis`。

2to3 Python 2.x のコードを Python 3.x のコードに変換するツールです。ソースコードを解析してその解析木を巡回 (traverse) することで検知できる、非互換性の大部分を処理します。

2to3 は標準ライブラリの `lib2to3` として利用できます。単体のツールとしての使えるスクリプトが `Tools/scripts/2to3` として提供されています。2to3-reference を参照してください。

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。

ローカル変数のアノテーションは実行時にはアクセスできませんが、グローバル変数、クラス属性、関数のアノテーションはそれぞれモジュール、クラス、関数の `__annotations__` 特殊属性に保持されています。

機能の説明がある *variable annotation*, *function annotation*, **PEP 484**, **PEP 526** を参照してください。

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を書くことができ、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、`__anext__()` メソッドを使って呼び出されたときに `awaitable` オブジェクトを返す *asynchronous iterator* です。この `awaitable` オブジェクトは、次の `yield` 式までの非同期ジェネレータ関数の本体を実行します。

`yield` にくるたびに、その位置での実行状態 (ローカル変数と保留状態の `try` 文) 処理は一時停止されます。`__anext__()` で返された他の `awaitable` によって **非同期ジェネレータイテレータ** が実際に再開されたとき、中断した箇所を取得します。**PEP 492** および **PEP 525** を参照してください。

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。

attribute (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。

awaitable (待機可能) `await` 式で使う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。

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 contiguous バッファ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時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。

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

class (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。

coercion (型強制) 同じ型の 2 引数を伴う演算の最中に行われる、ある型のインスタンスの別の型への暗黙の変換です。例えば、`int(3.15)` は浮動小数点数を整数 3 に変換します。しかし `3+4.5` では、各引数は型が異なり (一つは整数、一つは浮動小数点数)、加算をする前に同じ型に変換できなければ `TypeError` 例外が投げられます。型強制がなかったら、すべての引数は、たとえ互換な型であっても、単に `3+4.5` ではなく `float(3)+4.5` というように、プログラマーが同じ型に正規化しなければいけません。

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 (コンテキストマネージャ) `__enter__()` と `__exit__()` メソッドを定義することで `with` 文内の環境を管理するオブジェクトです。[PEP 343](#) を参照してください。

context variable (コンテキスト変数) コンテキストに依存して異なる値を持つ変数。これは、ある変数の値が各々の実行スレッドで異なり得るスレッドローカルストレージに似ています。しかしコンテキスト変数では、1 つの実行スレッドにいくつかのコンテキストがあり得、コンテキスト変数の主な用途は並列な非同期タスクの変数の追跡です。`contextvars` を参照してください。

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。

coroutine (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あ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(...):
    ...
f = staticmethod(f)
```

(次のページに続く)

(前のページからの続き)

```
@staticmethod
def f(...):
    ...
```

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 (デスクリプタ) メソッド `__get__()`、`__set__()`、あるいは `__delete__()` を定義しているオブジェクトです。あるクラス属性がデスクリプタであるとき、属性探索によって、束縛されている特別な動作が呼び出されます。通常、`get`、`set`、`delete` のために `a.b` と書くと、`a` のクラス辞書内でオブジェクト `b` を検索しますが、`b` がデスクリプタであればそれぞれのデスクリプタメソッドが呼び出されます。デスクリプタの理解は、Python を深く理解する上で鍵となります。というのは、デスクリプタこそが、関数、メソッド、プロパティ、クラスメソッド、静的メソッド、そしてスーパークラスの参照といった多くの機能の基盤だからです。

デスクリプタのメソッドに関して詳しくは、`descriptors` を参照してください。

dictionary (辞書) 任意のキーを値に対応付ける連想配列です。`__hash__()` メソッドと `__eq__()` メソッドを実装した任意のオブジェクトをキーにできます。Perl ではハッシュ (hash) と呼ばれています。

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 クラス、関数、モジュールの最初の式である文字列リテラルです。そのスイートの実行時には無視されますが、コンパイラによって識別され、そのクラス、関数、モジュールの `__doc__` 属性として保存されます。イントロスペクションできる (訳注: 属性として参照できる) ので、オブジェクトのドキュメントを書く標準的な場所です。

duck-typing あるオブジェクトが正しいインタ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 (ファイルオブジェクト) 下位のリソースへのファイル志向 API (read() や write() メソッドを持つもの) を公開しているオブジェクトです。ファイルオブジェクトは、作成された手段によって、実際のディスク上のファイルや、その他のタイプのストレージや通信デバイス (例えば、標準入出力、インメモリバッファ、ソケット、パイプ、等) へのアクセスを媒介できます。ファイルオブジェクトは *file-like objects* や *streams* と呼ばれます。

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 **バイナリファイル**、バッファされた **バイナリファイル**、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。

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。

Python 3.3 以降では 2 種類のファインダがあります。sys.meta_path で使用される *meta path finder* と、sys.path_hooks で使用される *path entry finder* です。

詳細については PEP 302、PEP 420 および PEP 451 を参照してください。

floor division 一番近い小さい整数に丸める数学除算。floor division 演算子は // です。例えば、11 // 4 は 2 になり、float の true division の結果 2.75 と異なります。(-11) // 4 は -2.75 を **小さい方に丸める**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。

function (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戻り値の *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 型の戻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* と PEP 484 を参照してください。

`__future__` 互換性のない新たな言語機能を現在のインタプリタで有効にするためにプログラマが利用できる擬似モジュールです。

`__future__` モジュールを `import` してその変数を評価すれば、新たな機能が初めて追加されたのと同じで、いつ言語デフォルトの機能になるかわかります:

```
>>> 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。

generator (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。

`yield` のたびに局所実行状態 (局所変数や未処理の `try` 文などを含む) を記憶して、処理は一時的に中断されます。**ジェネレータイテレータ** が再開されると、中断した位置を取得します (通常の関数が実行のたびに新しい状態から開始するのと同所的対照的)。)

generator expression (ジェネレータ式) イテレータを返す式です。普通の式に、ループ変数を定義する `for` 節、範囲、そして省略可能な `if` 節がつづいているように見えます。こうして構成された式は、外側の関数に向けて値を生成します:

```
>>> 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。

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。

過去に ” 自由なマルチスレッド化 ” したインタプリタ (供用されるデータを細かい粒度でロックする) が開発されましたが、一般的なシングルスプロセッサの場合のパフォーマンスが悪かったので成功しませんでした。このパフォーマンスの問題を克服しようとする、実装がより複雑になり保守コストが増加すると考えられています。

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。

hashable (ハッシュ可能) **ハッシュ可能** なオブジェクトとは、生存期間中変わらないハッシュ値を持ち (`__hash__()` メソッドが必要)、他のオブジェクトと比較ができる (`__eq__()` メソッドが必要) オブジェクトです。同値なハッシュ可能オブジェクトは必ず同じハッシュ値を持つ必要があります。

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) 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。

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 (対話的) Python には対話的インタプリタがあり、文や式をインタプリタのプロンプトに入力すると即座に実行されて結果を見ることができます。`python` と何も引数を与えずに実行してください。(コンピュータのメインメニューから Python の対話的インタプリタを起動できるかもしれません。) 対話的インタプリタは、新しいアイデアを試してみたり、モジュールやパッケージの中を覗いてみる (`help(x)` を覚えておいてください) のに非常に便利なツールです。

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 (反復可能オブジェクト) 要素を一度に 1 つずつ返せるオブジェクトです。反復可能オブジェクトの例には、(list, str, tuple といった) 全てのシーケンス型や、dict や [ファイルオブジェクト](#) といった幾つかの非シーケンス型、あるいは [Sequence](#) 意味論を実装した `__iter__()` メソッドか `__getitem__()` メソッドを持つ任意のクラスのインスタンスが含まれます。

反復可能オブジェクトは for ループ内やその他多くのシーケンス (訳注: ここでのシーケンスとは、シーケンス型ではなくただの列という意味) が必要となる状況 (`zip()`, `map()`, ...) で利用できます。反復可能オブジェクトを組み込み関数 `iter()` の引数として渡すと、オブジェクトに対するイテレータを返します。このイテレータは一連の値を引き渡す際に便利です。通常は反復可能オブジェクトを使う際には、`iter()` を呼んだりイテレータオブジェクトを自分で操作する必要はありません。for 文ではこの操作を自動的にに行い、一時的な無名の変数を作成してループを回している間イテレータを保持します。[イテレータ](#)、[シーケンス](#)、[ジェネレータ](#) も参照してください。

iterator (イテレータ) 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 out 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います。

詳細な情報は `typeiter` にあります。

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()` をキー関数に使用すれば、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できます。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` モジュールは `attrgetter()`、`itemgetter()`、`methodcaller()` という 3 つのキー関数コンストラクタを提供しています。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。

`list` (リスト) Python の組み込みの [シーケンス](#) です。リストという名前ですが、リンクリストではなく、他の言語で言う配列 (array) と同種のもので、要素へのアクセスは $O(1)$ です。

`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` モジュールをロードするオブジェクト。`load_module()` という名前のメソッドを定義していなければなりません。ローダーは一般的に [finder](#) から返されます。詳細は [PEP 302](#) を、[abstract base class](#) については `importlib.abc.Loader` を参照してください。

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、`Mapping` か `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とって、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。

method (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 (メソッド解決順序) 探索中に基底クラスが構成要素を検索される順番です。2.3 以降の Python インタープリタが使用するアルゴリズムの詳細については [The Python 2.3 Method Resolution Order](#) を参照してください。

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。

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应用されています。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()` の戻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
```

(上の例のように) いくつかの名前付きタプルは組み込み型になっています。その他にも名前付きタプルは、通常のクラス定義で `tuple` を継承し、名前のフィールドを定義して作成できます。そのようなクラスは手動で書いたり、`collections.namedtuple()` ファクトリ関数で作成したりできます。後者の方法は、手動で書いた名前付きタプルや組み込みの名前付きタプルには無い付加的なメソッドを追加できます。

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

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 (名前空間パッケージ) サブパッケージのコンテナとして提供される [PEP 420 package](#)。Namespace package はおそらく物理表現を持たず、`__init__.py` ファイルがないため、[regular package](#) と異なります。

[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みます。

new-style class (新スタイルクラス) 今では全てのクラスオブジェクトに使われている味付けの古い名前です。以前の Python のバージョンでは、新スタイルクラスのみが `__slots__`、デスクリプタ、`__getattr__()`、クラスメソッド、そして静的メソッド等の Python の新しい、多様な機能を利用できました。

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 `sys.path_hook` リストにある callable で、指定された *path entry* にあるモジュールを見つける方法を知っている場合に *path entry finder* を返します。

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。

positional argument (位置引数) **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 (参照カウント) あるオブジェクトに対する参照の数。参照カウントが 0 になったとき、そのオブジェクトは破棄されます。参照カウントは通常は Python のコード上には現れませんが、*CPython* 実装の重要な要素です。`sys` モジュールは、プログラマーが任意のオブジェクトの参照カウントを知るための `getrefcount()` 関数を提供しています。

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 (シーケンス) 整数インデクスによる効率的な要素アクセスを `__getitem__()` 特殊メソッドを通じてサポートし、長さを返す `__len__()` メソッドを定義した *iterable* です。組み込みシーケンス型には、`list`, `str`, `tuple`, `bytes` などがあります。`dict` は `__getitem__()` と `__len__()` もサポートしますが、検索の際に整数ではなく任意の *immutable* なキーを使うため、シーケンスではなくマッピング (*mapping*) とみなされているので注意してください。

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()`.

set comprehension A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See comprehensions.

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。

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。

text encoding ユニコード文字列をエンコードするコーデックです。

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](#) を自動的にを行います。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。

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のできる、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 (型) Python オブジェクトの型はオブジェクトがどのようなものかを決めます。あらゆるオブジェクトは型を持っています。オブジェクトの型は `__class__` 属性でアクセスしたり、`type(obj)` で取得したり出来ます。

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:

```
from typing import List, Tuple

def remove_gray_shades(
    colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

これは次のように読みやすくなります:

```
from typing import List, Tuple

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戻り値の期待される型を指定する *annotation* です。

型ヒントは必須ではなく Python では強制ではありませんが、静的型解析ツールにとって有用であり、IDE のコード補完とリファクタリングの手助けになります。

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。

この機能について解説している *function annotation*, **PEP 484**, **PEP 526** を参照してください。

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。

このドキュメントについて

このドキュメントは、Python のドキュメントを主要な目的として作られた ドキュメントプロセッサの [Sphinx](#) を利用して、[reStructuredText](#) 形式のソースから生成されました。

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](#) ページをご覧ください。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. [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 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テーションは生まれませんでした -- ありがとう!

歴史とライセンス

C.1 Python の歴史

Python は 1990 年代の始め、オランダにある Stichting Mathematisch Centrum (CWI, <https://www.cwi.nl/> 参照) で Guido van Rossum によって ABC と呼ばれる言語の後継言語として生み出されました。その後多くの人々が Python に貢献していますが、Guido は今日でも Python 製作者の先頭に立っています。

1995 年、Guido は米国ヴァージニア州レストンにある Corporation for National Reserch Initiatives (CNRI, <https://www.cnri.reston.va.us/> 参照) で Python の開発に携わり、いくつかのバージョンをリリースしました。

2000 年 3 月、Guido と Python のコア開発チームは BeOpen.com に移り、BeOpen PythonLabs チームを結成しました。同年 10 月、PythonLabs チームは Digital Creations (現在の Zope Corporation, <https://www.zope.org/> 参照) に移りました。そして 2001 年、Python に関する知的財産を保有するための非営利組織 Python Software Foundation (PSF, <https://www.python.org/psf/> 参照) を立ち上げました。このとき Zope Corporation は PSF の賛助会員になりました。

Python のリリースは全てオープンソース (オープンソースの定義は <https://opensource.org/> を参照してください) です。歴史的にみて、ごく一部を除くほとんどの Python リリースは GPL 互換になっています; 各リリースについては下表にまとめてあります。

リリース	ベース	西暦年	権利	GPL 互換
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	no
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: 「GPL 互換」という表現は、Python が GPL で配布されているという意味ではありません。Python のライセンスは全て、GPL と違い、変更したバージョンを配布する際に変更をオープンソースにしなくてもかまいません。GPL 互換のライセンスの下では、GPL でリリースされている他のソフトウェアと Python を組み合わせられますが、それ以外のライセンスではそうではありません。

Guido の指示の下、これらのリリースを可能に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 *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement 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 PSF LICENSE AGREEMENT FOR PYTHON 3.8.20

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 3.8.20 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 3.8.20 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All Rights Reserved" are retained in Python 3.8.20 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 3.8.20 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 3.8.20.
4. PSF is making Python 3.8.20 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR

WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 3.8.20 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.8.20 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.8.20, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 3.8.20, 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 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.

(次のページに続く)

(前のページからの続き)

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, 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.

(次のページに続く)

(前のページからの続き)

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.4 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 3.8.20 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` 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

(次のページに続く)

(前のページからの続き)

```
permission.
```

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

```
Any feedback is very welcome.
```

```
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
```

```
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 ソケット

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, <http://www.wide.ad.jp/>.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
```

```
All rights reserved.
```

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

```
THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
```

(次のページに続く)

(前のページからの続き)

SUCH DAMAGE.

C.3.3 Asynchronous socket services

The `asyncchat` and `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 AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR

(次のページに続く)

(前のページからの続き)

```
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
```

(次のページに続く)

(前のページからの続き)

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 all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.8 test_epoll

test_epoll モジュールは次の告知を含んでいます:

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

(次のページに続く)

(前のページからの続き)

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
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 <http://www.netlib.org/fp/>. 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
```

(次のページに続く)

(前のページからの続き)

```
* 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 Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

LICENSE ISSUES

=====

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

OpenSSL License

```
/* =====
 * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 *    software must display the following acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
```

(次のページに続く)

(前のページからの続き)

```

*   endorse or promote products derived from this software without
*   prior written permission. For written permission, please contact
*   openssl-core@openssl.org.
*
* 5. Products derived from this software may not be called "OpenSSL"
*   nor may "OpenSSL" appear in their names without prior written
*   permission of the OpenSSL Project.
*
* 6. Redistributions of any form whatsoever must retain the following
*   acknowledgment:
*   "This product includes software developed by the OpenSSL Project
*   for use in the OpenSSL Toolkit (http://www.openssl.org/)"
*
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com).  This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

```

Original SSLeay License

```

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*
* This library is free for commercial and non-commercial use as long as
* the following conditions are aheared to.  The following conditions
* apply to all code found in this distribution, be it the RC4, RSA,
* lhash, DES, etc., code; not just the SSL code.  The SSL documentation
* included with this distribution is covered by the same copyright terms
* except that the holder is Tim Hudson (tjh@cryptsoft.com).
*
* Copyright remains Eric Young's, and as such any Copyright notices in

```

(次のページに続く)

(前のページからの続き)

```
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
*   notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*   notice, this list of conditions and the following disclaimer in the
*   documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
*   must display the following acknowledgement:
*   "This product includes cryptographic software written by
*     Eric Young (eay@cryptsoft.com)"
*   The word 'cryptographic' can be left out if the routines from the library
*   being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
*   the apps directory (application code) you must include an acknowledgement:
*   "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/
```

C.3.13 expat

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` extension 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,
```

(次のページに続く)

(前のページからの続き)

```
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.
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
```

(次のページに続く)

(前のページからの続き)

notice, this list of conditions and the following disclaimer.

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

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

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.17 libmpdec

The `_decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

Copyright (c) 2008-2016 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)

(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.3.18 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.

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

- ..., 291
- 2to3, 291
- >>>, 291
- __all__ (package variable), 49
- __dict__ (module attribute), 156
- __doc__ (module attribute), 155
- __file__ (module attribute), 155, 156
- __future__, 297
- __import__
 - 組み込み関数, 50
- __loader__ (module attribute), 155
- __main__
 - モジュール, 13, 183, 196
- __name__ (module attribute), 155, 156
- __package__ (module attribute), 155
- __slots__, 305
- _frozen (*C* のデータ型), 53
- _inittab (*C* のデータ型), 53
- _Py_c_diff (*C* の関数), 107
- _Py_c_neg (*C* の関数), 107
- _Py_c_pow (*C* の関数), 107
- _Py_c_prod (*C* の関数), 107
- _Py_c_quot (*C* の関数), 107
- _Py_c_sum (*C* の関数), 107
- _Py_InitializeMain (*C* の関数), 222
- _Py_NoneStruct (*C* の変数), 236
- _Py_TPFLAGS_HAVE_VECTORCALL (組み込み変数), 260
- _PyBytes_Resize (*C* の関数), 111
- _PyCFunctionFast (*C* のデータ型), 237
- _PyCFunctionFastWithKeywords (*C* のデータ型), 238
- _PyImport_Fini (*C* の関数), 53
- _PyImport_Init (*C* の関数), 52
- _PyObject_FastCallDict (*C* の関数), 76
- _PyObject_New (*C* の関数), 235
- _PyObject_NewVar (*C* の関数), 235
- _PyObject_Vectorcall (*C* の関数), 76
- _PyTuple_Resize (*C* の関数), 139
- _thread
 - モジュール, 191
- オブジェクト
 - bytearray, 111
 - bytes, 109
 - Capsule, 168
 - complex number, 107
 - dictionary, 143
 - file, 154
 - floating point, 106
 - frozenset, 147
 - function, 149
 - instancemethod, 150
 - integer, 101
 - list, 141
 - long integer, 101
 - mapping, 143
 - memoryview, 166
 - method, 151

- module, 155
- None, 100
- numeric, 101
- sequence, 108
- set, 147
- tuple, 138
- type, 6, 97
- モジュール
 - __main__, 13, 183, 196
 - _thread, 191
 - builtins, 13, 183, 196
 - signal, 35
 - sys, 13, 183, 196
- 環境変数
 - exec_prefix, 4
 - PATH, 13
 - prefix, 4
 - PYTHON*, 181
 - PYTHONCOERCECLOCALE, 219
 - PYTHONDEBUG, 180
 - PYTHONDONTWRITEBYTECODE, 181
 - PYTHONDUMPREFS, 250
 - PYTHONHASHSEED, 181
 - PYTHONHOME, 13, 14, 181, 188, 213
 - PYTHONINSPECT, 181
 - PYTHONIOENCODING, 184
 - PYTHONLEGACYWINDOWSFSENCODING, 181
 - PYTHONLEGACYWINDOWSSSTDIO, 182
 - PYTHONMALLOC, 226, 230, 232
 - PYTHONMALLOCSTATS, 226
 - PYTHONNOUSERSITE, 182
 - PYTHONOPTIMIZE, 182
 - PYTHONPATH, 13, 14, 181, 214
 - PYTHONUNBUFFERED, 182
 - PYTHONUTF8, 219
 - PYTHONVERBOSE, 182
- 組み込み関数
 - __import__, 50
 - abs, 79
 - ascii, 73
 - bytes, 74
 - classmethod, 239
 - compile, 51
 - divmod, 79
 - float, 81
 - hash, 77, 255
 - int, 81
 - len, 77, 82, 85, 142, 145, 148
 - pow, 79, 81
 - repr, 73, 255
 - staticmethod, 239
 - tuple, 84, 143
 - type, 77

A

- abort(), 49
- abs

組み込み関数, 79
 abstract base class, 291
 allocfunc (*C のデータ型*), 280
 annotation, 291
 argument, 291
 argv (*in module sys*), 187
 ascii
 組み込み関数, 73
 asynchronous context manager, 292
 asynchronous generator, 292
 asynchronous generator iterator, 292
 asynchronous iterable, 292
 asynchronous iterator, 293
 attribute, 293
 awaitable, 293

B

BDFL, 293
 binary file, 293
 binaryfunc (*C のデータ型*), 281
 buffer interface
 (see buffer protocol), 87
 buffer object
 (see buffer protocol), 87
 buffer protocol, 87
 builtins
 モジュール, 13, 183, 196
 bytearray
 オブジェクト, 111
 bytecode, 293
 bytes
 オブジェクト, 109
 組み込み関数, 74
 bytes-like object, 293

C

callback, 293
 calloc(), 225
 Capsule
 オブジェクト, 168
 C-contiguous, 91, 294
 class, 293
 class variable, 293
 classmethod
 組み込み関数, 239
 cleanup functions, 49
 close() (*in module os*), 197
 CO_FUTURE_DIVISION (*C の変数*), 24
 code object, 152
 coercion, 294
 compile
 組み込み関数, 51
 complex number, 294
 オブジェクト, 107
 context manager, 294
 context variable, 294
 contiguous, 91, 294
 copyright (*in module sys*), 187
 coroutine, 294
 coroutine function, 294
 CPython, 294
 create_module (*C の関数*), 160

D

decorator, 294
 descrgetfunc (*C のデータ型*), 281
 descriptor, 295
 descrsetfunc (*C のデータ型*), 281
 destructor (*C のデータ型*), 280
 dictionary, 295
 オブジェクト, 143

dictionary comprehension, 295
 dictionary view, 295
 divmod
 組み込み関数, 79
 docstring, 295
 duck-typing, 295

E

EAFP, 295
 EOFError (*built-in exception*), 154
 exc_info() (*in module sys*), 11
 exec_module (*C の関数*), 160
 exec_prefix, 4
 executable (*in module sys*), 185
 exit(), 49
 expression, 296
 extension module, 296

F

f-string, 296
 file
 オブジェクト, 154
 file object, 296
 file-like object, 296
 finder, 296
 float
 組み込み関数, 81
 floating point
 オブジェクト, 106
 floor division, 296
 Fortran contiguous, 91, 294
 free(), 225
 freefunc (*C のデータ型*), 280
 freeze utility, 53
 frozenset
 オブジェクト, 147
 function, 296
 オブジェクト, 149
 function annotation, 296

G

garbage collection, 297
 generator, 297
 generator expression, 297
 generator iterator, 297
 generic function, 297
 getattrfunc (*C のデータ型*), 281
 getattrofunc (*C のデータ型*), 281
 getbufferproc (*C のデータ型*), 281
 getiterfunc (*C のデータ型*), 281
 GIL, 297
 global interpreter lock, 188, 297

H

hash
 組み込み関数, 77, 255
 hash-based pyc, 298
 hashable, 298
 hashfunc (*C のデータ型*), 281

I

IDLE, 298
 immutable, 298
 import path, 298
 importer, 298
 importing, 298
 incr_item(), 12, 13
 initproc (*C のデータ型*), 280
 inquiry (*C のデータ型*), 286
 instancemethod

オブジェクト, 150
 int
 組み込み関数, 81
 integer
 オブジェクト, 101
 interactive, 298
 interpreted, 298
 interpreter lock, 188
 interpreter shutdown, 299
 iterable, 299
 iterator, 299
 iternextfunc (*C のデータ型*), 281

K

key function, 299
 KeyboardInterrupt (*built-in exception*), 35
 keyword argument, 300

L

lambda, 300
 LBYL, 300
 len
 組み込み関数, 77, 82, 85, 142, 145, 148
 lenfunc (*C のデータ型*), 281
 list, 300
 オブジェクト, 141
 list comprehension, 300
 loader, 300
 lock, interpreter, 188
 long integer
 オブジェクト, 101
 LONG_MAX, 102

M

magic
 method, 300
 magic method, 300
 main(), 184, 187
 malloc(), 225
 mapping, 300
 オブジェクト, 143
 memoryview
 オブジェクト, 166
 meta path finder, 300
 metaclass, 300
 METH_CLASS (組み込み変数), 239
 METH_COEXIST (組み込み変数), 239
 METH_FASTCALL (組み込み変数), 238
 METH_NOARGS (組み込み変数), 239
 METH_O (組み込み変数), 239
 METH_STATIC (組み込み変数), 239
 METH_VARARGS (組み込み変数), 238
 method, 301
 magic, 300
 special, 305
 オブジェクト, 151
 method resolution order, 301
 MethodType (*in module types*), 149, 151
 module, 301
 search path, 13, 183, 186
 オブジェクト, 155
 module spec, 301
 modules (*in module sys*), 49, 183
 ModuleType (*in module types*), 155
 MRO, 301
 mutable, 301

N

named tuple, 301
 namespace, 301

namespace package, 302
 nested scope, 302
 new-style class, 302
 newfunc (*C のデータ型*), 280
 None
 オブジェクト, 100
 numeric
 オブジェクト, 101

O

object, 302
 code, 152
 objobjargproc (*C のデータ型*), 282
 objobjproc (*C のデータ型*), 282
 OverflowError (*built-in exception*), 102104

P

package, 302
 package variable
 __all__, 49
 parameter, 302
 PATH, 13
 path
 module search, 13, 183, 186
 path (*in module sys*), 13, 183, 186
 path based finder, 303
 path entry, 303
 path entry finder, 303
 path entry hook, 303
 path-like object, 303
 PEP, 303
 platform (*in module sys*), 186
 portion, 303
 positional argument (位置引数), 303
 pow
 組み込み関数, 79, 81
 prefix, 4
 provisional API, 304
 provisional package, 304
 Py_ABS (*C のマクロ*), 5
 Py_AddPendingCall (*C の関数*), 198
 Py_AddPendingCall(), 198
 Py_AtExit (*C の関数*), 49
 Py_BEGIN_ALLOW_THREADS, 189
 Py_BEGIN_ALLOW_THREADS (*C のマクロ*), 193
 Py_BLOCK_THREADS (*C のマクロ*), 193
 Py_buffer (*C のデータ型*), 88
 Py_buffer.buf (*C のメンバ変数*), 88
 Py_buffer.format (*C のメンバ変数*), 89
 Py_buffer.internal (*C のメンバ変数*), 90
 Py_buffer.itemsize (*C のメンバ変数*), 88
 Py_buffer.len (*C のメンバ変数*), 88
 Py_buffer.ndim (*C のメンバ変数*), 89
 Py_buffer.obj (*C のメンバ変数*), 88
 Py_buffer.readonly (*C のメンバ変数*), 88
 Py_buffer.shape (*C のメンバ変数*), 89
 Py_buffer.strides (*C のメンバ変数*), 89
 Py_buffer.suboffsets (*C のメンバ変数*), 89
 Py_BuildValue (*C の関数*), 63
 Py_BytesMain (*C の関数*), 19
 Py_BytesWarningFlag (*C の変数*), 180
 Py_CHARMASK (*C のマクロ*), 5
 Py_CLEAR (*C の関数*), 25
 Py_CompileString (*C の関数*), 22
 Py_CompileString(), 24
 Py_CompileStringExFlags (*C の関数*), 23
 Py_CompileStringFlags (*C の関数*), 22
 Py_CompileStringObject (*C の関数*), 23
 Py_complex (*C のデータ型*), 107
 Py_DebugFlag (*C の変数*), 180
 Py_DecodeLocale (*C の関数*), 45
 Py_DECREF (*C の関数*), 25

- Py_DECREF(), 7
- Py_DEPRECATED (C のマクロ), 5
- Py_DontWriteBytecodeFlag (C の変数), 180
- Py_Ellipsis (C の変数), 166
- Py_EncodeLocale (C の関数), 45
- Py_END_ALLOW_THREADS, 189
- Py_END_ALLOW_THREADS (C のマクロ), 193
- Py_EndInterpreter (C の関数), 197
- Py_EnterRecursiveCall (C の関数), 38
- Py_eval_input (C の変数), 24
- Py_Exit (C の関数), 49
- Py_ExitStatusException (C の関数), 207
- Py_False (C の変数), 105
- Py_FatalError (C の関数), 49
- Py_FatalError(), 187
- Py_FdIsInteractive (C の関数), 43
- Py_file_input (C の変数), 24
- Py_Finalize (C の関数), 184
- Py_FinalizeEx (C の関数), 183
- Py_FinalizeEx(), 49, 183, 197
- Py_FrozenFlag (C の変数), 181
- Py_GetBuildInfo (C の関数), 187
- Py_GetCompiler (C の関数), 187
- Py_GetCopyright (C の関数), 186
- Py_GETENV (C のマクロ), 5
- Py_GetExecPrefix (C の関数), 185
- Py_GetExecPrefix(), 14
- Py_GetPath (C の関数), 186
- Py_GetPath(), 14, 184, 186
- Py_GetPlatform (C の関数), 186
- Py_GetPrefix (C の関数), 185
- Py_GetPrefix(), 14
- Py_GetProgramFullPath (C の関数), 185
- Py_GetProgramFullPath(), 14
- Py_GetProgramName (C の関数), 184
- Py_GetPythonHome (C の関数), 188
- Py_GetVersion (C の関数), 186
- Py_HashRandomizationFlag (C の変数), 181
- Py_IgnoreEnvironmentFlag (C の変数), 181
- Py_INCREF (C の関数), 25
- Py_INCREF(), 7
- Py_Initialize (C の関数), 183
- Py_Initialize(), 13, 184, 197
- Py_InitializeEx (C の関数), 183
- Py_InitializeFromConfig (C の関数), 216
- Py_InspectFlag (C の変数), 181
- Py_InteractiveFlag (C の変数), 181
- Py_IsInitialized (C の関数), 183
- Py_IsInitialized(), 14
- Py_IsolatedFlag (C の変数), 181
- Py_LeaveRecursiveCall (C の関数), 38
- Py_LegacyWindowsFSEncodingFlag (C の変数), 181
- Py_LegacyWindowsStdioFlag (C の変数), 181
- Py_Main (C の関数), 19
- Py_MAX (C のマクロ), 5
- Py_MEMBER_SIZE (C のマクロ), 5
- Py_MIN (C のマクロ), 5
- Py_mod_create (C のマクロ), 159
- Py_mod_exec (C のマクロ), 160
- Py_NewInterpreter (C の関数), 196
- Py_None (C の変数), 100
- Py_NoSiteFlag (C の変数), 182
- Py_NotImplemented (C の変数), 71
- Py_NoUserSiteDirectory (C の変数), 182
- Py_OptimizeFlag (C の変数), 182
- Py_PreInitialize (C の関数), 210
- Py_PreInitializeFromArgs (C の関数), 210
- Py_PreInitializeFromBytesArgs (C の関数), 210
- Py_PRINT_RAW, 155
- Py_QuietFlag (C の変数), 182
- Py_REFCNT (C のマクロ), 237
- Py_ReprEnter (C の関数), 38
- Py_ReprLeave (C の関数), 39
- Py_RETURN_FALSE (C のマクロ), 105
- Py_RETURN_NONE (C のマクロ), 100
- Py_RETURN_NOTIMPLEMENTED (C のマクロ), 71
- Py_RETURN_RICHCOMPARE (C のマクロ), 263
- Py_RETURN_TRUE (C のマクロ), 105
- Py_RunMain (C の関数), 221
- Py_SetPath (C の関数), 186
- Py_SetPath(), 186
- Py_SetProgramName (C の関数), 184
- Py_SetProgramName(), 14, 183, 185
- Py_SetPythonHome (C の関数), 188
- Py_SetStandardStreamEncoding (C の関数), 184
- Py_single_input (C の変数), 24
- Py_SIZE (C のマクロ), 237
- PY_SSIZE_T_MAX, 103
- Py_STRINGIFY (C のマクロ), 5
- Py_TPFLAGS_BASE_EXC_SUBCLASS (組み込み変数), 260
- Py_TPFLAGS_BASETYPE (組み込み変数), 258
- Py_TPFLAGS_BYTES_SUBCLASS (組み込み変数), 260
- Py_TPFLAGS_DEFAULT (組み込み変数), 259
- Py_TPFLAGS_DICT_SUBCLASS (組み込み変数), 260
- Py_TPFLAGS_HAVE_FINALIZE (組み込み変数), 260
- Py_TPFLAGS_HAVE_GC (組み込み変数), 259
- Py_TPFLAGS_HEAPTYPE (組み込み変数), 258
- Py_TPFLAGS_LIST_SUBCLASS (組み込み変数), 260
- Py_TPFLAGS_LONG_SUBCLASS (組み込み変数), 260
- Py_TPFLAGS_METHOD_DESCRIPTOR (組み込み変数), 259
- Py_TPFLAGS_READY (組み込み変数), 259
- Py_TPFLAGS_READYING (組み込み変数), 259
- Py_TPFLAGS_TUPLE_SUBCLASS (組み込み変数), 260
- Py_TPFLAGS_TYPE_SUBCLASS (組み込み変数), 260
- Py_TPFLAGS_UNICODE_SUBCLASS (組み込み変数), 260
- Py_tracefunc (C のデータ型), 199
- Py_True (C の変数), 105
- Py_tss_NEEDS_INIT (C のマクロ), 201
- Py_tss_t (C のデータ型), 201
- Py_TYPE (C のマクロ), 237
- Py_UCS1 (C のデータ型), 113
- Py_UCS2 (C のデータ型), 113
- Py_UCS4 (C のデータ型), 113
- Py_UNBLOCK_THREADS (C のマクロ), 193
- Py_UnbufferedStdioFlag (C の変数), 182
- Py_UNICODE (C のデータ型), 113
- Py_UNICODE_IS_HIGH_SURROGATE (C のマクロ), 118
- Py_UNICODE_IS_LOW_SURROGATE (C のマクロ), 118
- Py_UNICODE_IS_SURROGATE (C のマクロ), 118
- Py_UNICODE_ISALNUM (C の関数), 117
- Py_UNICODE_ISALPHA (C の関数), 117
- Py_UNICODE_ISDECIMAL (C の関数), 117
- Py_UNICODE_ISDIGIT (C の関数), 117
- Py_UNICODE_ISLINEBREAK (C の関数), 117
- Py_UNICODE_ISLOWER (C の関数), 116
- Py_UNICODE_ISNUMERIC (C の関数), 117
- Py_UNICODE_ISPRINTABLE (C の関数), 117
- Py_UNICODE_ISSPACE (C の関数), 116
- Py_UNICODE_ISTITLE (C の関数), 116
- Py_UNICODE_ISUPPER (C の関数), 116
- Py_UNICODE_JOIN_SURROGATES (C のマクロ), 118
- Py_UNICODE_TODECIMAL (C の関数), 117
- Py_UNICODE_TODIGIT (C の関数), 117
- Py_UNICODE_TOLOWER (C の関数), 117
- Py_UNICODE_TONUMERIC (C の関数), 118
- Py_UNICODE_TOTITLE (C の関数), 117
- Py_UNICODE_TOUPPER (C の関数), 117
- Py_UNREACHABLE (C のマクロ), 5
- Py_UNUSED (C のマクロ), 5
- Py_VaBuildValue (C の関数), 65
- PY_VECTORCALL_ARGUMENTS_OFFSET (C のマクロ), 76
- Py_VerboseFlag (C の変数), 182
- Py_VISIT (C の関数), 286
- Py_XDECREF (C の関数), 25
- Py_XDECREF(), 13
- Py_XINCRREF (C の関数), 25

PyAnySet_Check (*C* の関数), 147
 PyAnySet_CheckExact (*C* の関数), 147
 PyArg_Parse (*C* の関数), 62
 PyArg_ParseTuple (*C* の関数), 62
 PyArg_ParseTupleAndKeywords (*C* の関数), 62
 PyArg_UnpackTuple (*C* の関数), 62
 PyArg_ValidateKeywordArguments (*C* の関数), 62
 PyArg_VaParse (*C* の関数), 62
 PyArg_VaParseTupleAndKeywords (*C* の関数), 62
 PyASCIIObject (*C* のデータ型), 113
 PyAsyncMethods (*C* のデータ型), 279
 PyAsyncMethods.am_aiter (*C* のメンバ変数), 279
 PyAsyncMethods.am_anext (*C* のメンバ変数), 280
 PyAsyncMethods.am_await (*C* のメンバ変数), 279
 PyBool_Check (*C* の関数), 105
 PyBool_FromLong (*C* の関数), 106
 PyBUF_ANY_CONTIGUOUS (*C* のマクロ), 91
 PyBUF_C_CONTIGUOUS (*C* のマクロ), 91
 PyBUF_CONTIG (*C* のマクロ), 92
 PyBUF_CONTIG_RO (*C* のマクロ), 92
 PyBUF_F_CONTIGUOUS (*C* のマクロ), 91
 PyBUF_FORMAT (*C* のマクロ), 90
 PyBUF_FULL (*C* のマクロ), 92
 PyBUF_FULL_RO (*C* のマクロ), 92
 PyBUF_INDIRECT (*C* のマクロ), 91
 PyBUF_ND (*C* のマクロ), 91
 PyBUF_RECORDS (*C* のマクロ), 92
 PyBUF_RECORDS_RO (*C* のマクロ), 92
 PyBUF_SIMPLE (*C* のマクロ), 91
 PyBUF_STRIDED (*C* のマクロ), 92
 PyBUF_STRIDED_RO (*C* のマクロ), 92
 PyBUF_STRIDES (*C* のマクロ), 91
 PyBUF_WRITABLE (*C* のマクロ), 90
 PyBuffer_FillContiguousStrides (*C* の関数), 95
 PyBuffer_FillInfo (*C* の関数), 95
 PyBuffer_FromContiguous (*C* の関数), 94
 PyBuffer_GetPointer (*C* の関数), 94
 PyBuffer_IsContiguous (*C* の関数), 94
 PyBuffer_Release (*C* の関数), 94
 PyBuffer_SizeFromFormat (*C* の関数), 94
 PyBuffer_ToContiguous (*C* の関数), 94
 PyBufferProcs, 87
 PyBufferProcs (*C* のデータ型), 278
 PyBufferProcs.bf_getbuffer (*C* のメンバ変数), 278
 PyBufferProcs.bf_releasebuffer (*C* のメンバ変数), 278
 PyByteArray_AS_STRING (*C* の関数), 112
 PyByteArray_AsString (*C* の関数), 112
 PyByteArray_Check (*C* の関数), 112
 PyByteArray_CheckExact (*C* の関数), 112
 PyByteArray_Concat (*C* の関数), 112
 PyByteArray_FromObject (*C* の関数), 112
 PyByteArray_FromStringAndSize (*C* の関数), 112
 PyByteArray_GET_SIZE (*C* の関数), 112
 PyByteArray_Resize (*C* の関数), 112
 PyByteArray_Size (*C* の関数), 112
 PyByteArray_Type (*C* の変数), 111
 PyByteArrayObject (*C* のデータ型), 111
 PyBytes_AS_STRING (*C* の関数), 110
 PyBytes_AsString (*C* の関数), 110
 PyBytes_AsStringAndSize (*C* の関数), 110
 PyBytes_Check (*C* の関数), 109
 PyBytes_CheckExact (*C* の関数), 109
 PyBytes_Concat (*C* の関数), 111
 PyBytes_ConcatAndDel (*C* の関数), 111
 PyBytes_FromFormat (*C* の関数), 109
 PyBytes_FromFormatV (*C* の関数), 110
 PyBytes_FromObject (*C* の関数), 110
 PyBytes_FromString (*C* の関数), 109
 PyBytes_FromStringAndSize (*C* の関数), 109
 PyBytes_GET_SIZE (*C* の関数), 110
 PyBytes_Size (*C* の関数), 110
 PyBytes_Type (*C* の変数), 109
 PyBytesObject (*C* のデータ型), 109
 PyCallable_Check (*C* の関数), 74
 PyCallIter_Check (*C* の関数), 163
 PyCallIter_New (*C* の関数), 163
 PyCallIter_Type (*C* の変数), 163
 PyCapsule (*C* のデータ型), 168
 PyCapsule_CheckExact (*C* の関数), 168
 PyCapsule_Destructor (*C* のデータ型), 168
 PyCapsule_GetContext (*C* の関数), 169
 PyCapsule_GetDestructor (*C* の関数), 169
 PyCapsule_GetName (*C* の関数), 169
 PyCapsule_GetPointer (*C* の関数), 169
 PyCapsule_Import (*C* の関数), 169
 PyCapsule_IsValid (*C* の関数), 169
 PyCapsule_New (*C* の関数), 168
 PyCapsule_SetContext (*C* の関数), 170
 PyCapsule_SetDestructor (*C* の関数), 170
 PyCapsule_SetName (*C* の関数), 170
 PyCapsule_SetPointer (*C* の関数), 170
 PyCell_Check (*C* の関数), 152
 PyCell_GET (*C* の関数), 152
 PyCell_Get (*C* の関数), 152
 PyCell_New (*C* の関数), 152
 PyCell_SET (*C* の関数), 152
 PyCell_Set (*C* の関数), 152
 PyCell_Type (*C* の変数), 152
 PyCellObject (*C* のデータ型), 152
 PyCFunction (*C* のデータ型), 237
 PyCFunctionWithKeywords (*C* のデータ型), 237
 PyCode_Check (*C* の関数), 153
 PyCode_GetNumFree (*C* の関数), 153
 PyCode_New (*C* の関数), 153
 PyCode_NewEmpty (*C* の関数), 153
 PyCode_NewWithPosOnlyArgs (*C* の関数), 153
 PyCode_Type (*C* の変数), 153
 PyCodec_BackslashReplaceErrors (*C* の関数), 70
 PyCodec_Decompile (*C* の関数), 69
 PyCodec_Decoder (*C* の関数), 69
 PyCodec_Encode (*C* の関数), 68
 PyCodec_Encoder (*C* の関数), 69
 PyCodec_IgnoreErrors (*C* の関数), 70
 PyCodec_IncrementalDecoder (*C* の関数), 69
 PyCodec_IncrementalEncoder (*C* の関数), 69
 PyCodec_KnownEncoding (*C* の関数), 68
 PyCodec_LookupError (*C* の関数), 69
 PyCodec_NameReplaceErrors (*C* の関数), 70
 PyCodec_Register (*C* の関数), 68
 PyCodec_RegisterError (*C* の関数), 69
 PyCodec_ReplaceErrors (*C* の関数), 70
 PyCodec_StreamReader (*C* の関数), 69
 PyCodec_StreamWriter (*C* の関数), 69
 PyCodec_StrictErrors (*C* の関数), 70
 PyCodec_XMLCharRefReplaceErrors (*C* の関数), 70
 PyCodeObject (*C* のデータ型), 153
 PyCompactUnicodeObject (*C* のデータ型), 113
 PyCompilerFlags (*C* のデータ型), 24
 PyCompilerFlags.cf_feature_version (*C* のメンバ変数), 24
 PyCompilerFlags.cf_flags (*C* のメンバ変数), 24
 PyComplex_AsCComplex (*C* の関数), 108
 PyComplex_Check (*C* の関数), 108
 PyComplex_CheckExact (*C* の関数), 108
 PyComplex_FromCComplex (*C* の関数), 108
 PyComplex_FromDoubles (*C* の関数), 108
 PyComplex_ImagAsDouble (*C* の関数), 108
 PyComplex_RealAsDouble (*C* の関数), 108
 PyComplex_Type (*C* の変数), 108
 PyComplexObject (*C* のデータ型), 108
 PyConfig (*C* のデータ型), 211
 PyConfig_Clear (*C* の関数), 212
 PyConfig_InitIsolatedConfig (*C* の関数), 211
 PyConfig_InitPythonConfig (*C* の関数), 211
 PyConfig_Read (*C* の関数), 211
 PyConfig_SetArgv (*C* の関数), 211

- PyConfig_SetBytesArgv (*C* の関数), 211
- PyConfig_SetBytesString (*C* の関数), 211
- PyConfig_SetString (*C* の関数), 211
- PyConfig_SetWideStringList (*C* の関数), 211
- PyConfig_argv (*C* のメンバ変数), 212
- PyConfig.base_exec_prefix (*C* のメンバ変数), 212
- PyConfig.base_executable (*C* のメンバ変数), 212
- PyConfig.base_prefix (*C* のメンバ変数), 212
- PyConfig.buffered_stdio (*C* のメンバ変数), 212
- PyConfig.bytes_warning (*C* のメンバ変数), 212
- PyConfig.check_hash_pycs_mode (*C* のメンバ変数), 212
- PyConfig.configure_c_stdio (*C* のメンバ変数), 213
- PyConfig.dev_mode (*C* のメンバ変数), 213
- PyConfig.dump_refs (*C* のメンバ変数), 213
- PyConfig.exec_prefix (*C* のメンバ変数), 213
- PyConfig.executable (*C* のメンバ変数), 213
- PyConfig.faulthandler (*C* のメンバ変数), 213
- PyConfig.filesystem_encoding (*C* のメンバ変数), 213
- PyConfig.filesystem_errors (*C* のメンバ変数), 213
- PyConfig.hash_seed (*C* のメンバ変数), 213
- PyConfig.home (*C* のメンバ変数), 213
- PyConfig.import_time (*C* のメンバ変数), 213
- PyConfig.inspect (*C* のメンバ変数), 213
- PyConfig.install_signal_handlers (*C* のメンバ変数), 214
- PyConfig.interactive (*C* のメンバ変数), 214
- PyConfig.isolated (*C* のメンバ変数), 214
- PyConfig.legacy_windows_stdio (*C* のメンバ変数), 214
- PyConfig.malloc_stats (*C* のメンバ変数), 214
- PyConfig.module_search_paths (*C* のメンバ変数), 214
- PyConfig.module_search_paths_set (*C* のメンバ変数), 214
- PyConfig.optimization_level (*C* のメンバ変数), 214
- PyConfig.parse_argv (*C* のメンバ変数), 214
- PyConfig.parser_debug (*C* のメンバ変数), 215
- PyConfig.pathconfig_warnings (*C* のメンバ変数), 215
- PyConfig.prefix (*C* のメンバ変数), 215
- PyConfig.program_name (*C* のメンバ変数), 215
- PyConfig.pycache_prefix (*C* のメンバ変数), 215
- PyConfig.pythonpath_env (*C* のメンバ変数), 214
- PyConfig.quiet (*C* のメンバ変数), 215
- PyConfig.run_command (*C* のメンバ変数), 215
- PyConfig.run_filename (*C* のメンバ変数), 215
- PyConfig.run_module (*C* のメンバ変数), 215
- PyConfig.show_alloc_count (*C* のメンバ変数), 215
- PyConfig.show_ref_count (*C* のメンバ変数), 215
- PyConfig.site_import (*C* のメンバ変数), 215
- PyConfig.skip_source_first_line (*C* のメンバ変数), 215
- PyConfig.stdio_encoding (*C* のメンバ変数), 216
- PyConfig.stdio_errors (*C* のメンバ変数), 216
- PyConfig.tracemalloc (*C* のメンバ変数), 216
- PyConfig.use_environment (*C* のメンバ変数), 216
- PyConfig.use_hash_seed (*C* のメンバ変数), 213
- PyConfig.user_site_directory (*C* のメンバ変数), 216
- PyConfig.verbose (*C* のメンバ変数), 216
- PyConfig.warnoptions (*C* のメンバ変数), 216
- PyConfig.write_bytecode (*C* のメンバ変数), 216
- PyConfig.xoptions (*C* のメンバ変数), 216
- PyContext (*C* のデータ型), 171
- PyContext_CheckExact (*C* の関数), 172
- PyContext_ClearFreeList (*C* の関数), 173
- PyContext_Copy (*C* の関数), 172
- PyContext_CopyCurrent (*C* の関数), 172
- PyContext_Enter (*C* の関数), 172
- PyContext_Exit (*C* の関数), 172
- PyContext_New (*C* の関数), 172
- PyContext_Type (*C* の変数), 172
- PyContextToken (*C* のデータ型), 172
- PyContextToken_CheckExact (*C* の関数), 172
- PyContextToken_Type (*C* の変数), 172
- PyContextVar (*C* のデータ型), 172
- PyContextVar_CheckExact (*C* の関数), 172
- PyContextVar_Get (*C* の関数), 173
- PyContextVar_New (*C* の関数), 173
- PyContextVar_Reset (*C* の関数), 173
- PyContextVar_Set (*C* の関数), 173
- PyContextVar_Type (*C* の変数), 172
- PyCoro_CheckExact (*C* の関数), 171
- PyCoro_New (*C* の関数), 171
- PyCoro_Type (*C* の変数), 171
- PyCoroObject (*C* のデータ型), 171
- PyDate_Check (*C* の関数), 174
- PyDate_CheckExact (*C* の関数), 174
- PyDate_FromDate (*C* の関数), 174
- PyDate_FromTimestamp (*C* の関数), 177
- PyDateTime_Check (*C* の関数), 174
- PyDateTime_CheckExact (*C* の関数), 174
- PyDateTime_DATE_GET_FOLD (*C* の関数), 176
- PyDateTime_DATE_GET_HOUR (*C* の関数), 176
- PyDateTime_DATE_GET_MICROSECOND (*C* の関数), 176
- PyDateTime_DATE_GET_MINUTE (*C* の関数), 176
- PyDateTime_DATE_GET_SECOND (*C* の関数), 176
- PyDateTime_DELTA_GET_DAYS (*C* の関数), 176
- PyDateTime_DELTA_GET_MICROSECONDS (*C* の関数), 177
- PyDateTime_DELTA_GET_SECONDS (*C* の関数), 177
- PyDateTime_FromDateAndTime (*C* の関数), 175
- PyDateTime_FromDateAndTimeAndFold (*C* の関数), 175
- PyDateTime_FromTimestamp (*C* の関数), 177
- PyDateTime_GET_DAY (*C* の関数), 176
- PyDateTime_GET_MONTH (*C* の関数), 175
- PyDateTime_GET_YEAR (*C* の関数), 175
- PyDateTime_TIME_GET_FOLD (*C* の関数), 176
- PyDateTime_TIME_GET_HOUR (*C* の関数), 176
- PyDateTime_TIME_GET_MICROSECOND (*C* の関数), 176
- PyDateTime_TIME_GET_MINUTE (*C* の関数), 176
- PyDateTime_TIME_GET_SECOND (*C* の関数), 176
- PyDateTime_TimeZone_UTC (*C* の変数), 173
- PyDelta_Check (*C* の関数), 174
- PyDelta_CheckExact (*C* の関数), 174
- PyDelta_FromDSU (*C* の関数), 175
- PyDescr_IsData (*C* の関数), 164
- PyDescr_NewClassMethod (*C* の関数), 164
- PyDescr_NewGetSet (*C* の関数), 164
- PyDescr_NewMember (*C* の関数), 164
- PyDescr_NewMethod (*C* の関数), 164
- PyDescr_NewWrapper (*C* の関数), 164
- PyDict_Check (*C* の関数), 143
- PyDict_CheckExact (*C* の関数), 143
- PyDict_Clear (*C* の関数), 144
- PyDict_ClearFreeList (*C* の関数), 146
- PyDict_Contains (*C* の関数), 144
- PyDict_Copy (*C* の関数), 144
- PyDict_DelItem (*C* の関数), 144
- PyDict_DelItemString (*C* の関数), 144
- PyDict_GetItem (*C* の関数), 144
- PyDict_GetItemString (*C* の関数), 144
- PyDict_GetItemWithError (*C* の関数), 144
- PyDict_Items (*C* の関数), 145
- PyDict_Keys (*C* の関数), 145
- PyDict_Merge (*C* の関数), 146
- PyDict_MergeFromSeq2 (*C* の関数), 146
- PyDict_New (*C* の関数), 143
- PyDict_Next (*C* の関数), 145
- PyDict_SetDefault (*C* の関数), 145
- PyDict_SetItem (*C* の関数), 144
- PyDict_SetItemString (*C* の関数), 144
- PyDict_Size (*C* の関数), 145
- PyDict_Type (*C* の変数), 143
- PyDict_Update (*C* の関数), 146
- PyDict_Values (*C* の関数), 145
- PyDictObject (*C* のデータ型), 143
- PyDictProxy_New (*C* の関数), 143
- PyDoc_STR (*C* のマクロ), 6
- PyDoc_STRVAR (*C* のマクロ), 6
- PyErr_BadArgument (*C* の関数), 29
- PyErr_BadInternalCall (*C* の関数), 31
- PyErr_CheckSignals (*C* の関数), 35
- PyErr_Clear (*C* の関数), 28

PyErr_Clear(), 11, 13
 PyErr_ExceptionMatches (C の関数), 33
 PyErr_ExceptionMatches(), 13
 PyErr_Fetch (C の関数), 33
 PyErr_Format (C の関数), 28
 PyErr_FormatV (C の関数), 29
 PyErr_GetExcInfo (C の関数), 34
 PyErr_GivenExceptionMatches (C の関数), 33
 PyErr_NewException (C の関数), 35
 PyErr_NewExceptionWithDoc (C の関数), 35
 PyErr_NoMemory (C の関数), 29
 PyErr_NormalizeException (C の関数), 33
 PyErr_Occurred (C の関数), 32
 PyErr_Occurred(), 11
 PyErr_Print (C の関数), 28
 PyErr_PrintEx (C の関数), 28
 PyErr_ResourceWarning (C の関数), 32
 PyErr_Restore (C の関数), 33
 PyErr_SetExcFromWindowsErr (C の関数), 30
 PyErr_SetExcFromWindowsErrWithFilename (C の関数), 30
 PyErr_SetExcFromWindowsErrWithFilenameObject (C の関数), 30
 PyErr_SetExcFromWindowsErrWithFilenameObjects (C の関数), 30
 PyErr_SetExcInfo (C の関数), 34
 PyErr_SetFromErrno (C の関数), 29
 PyErr_SetFromErrnoWithFilename (C の関数), 29
 PyErr_SetFromErrnoWithFilenameObject (C の関数), 29
 PyErr_SetFromErrnoWithFilenameObjects (C の関数), 29
 PyErr_SetFromWindowsErr (C の関数), 30
 PyErr_SetFromWindowsErrWithFilename (C の関数), 30
 PyErr_SetImportError (C の関数), 30
 PyErr_SetImportErrorSubclass (C の関数), 32
 PyErr_SetInterrupt (C の関数), 35
 PyErr_SetNone (C の関数), 29
 PyErr_SetObject (C の関数), 28
 PyErr_SetString (C の関数), 28
 PyErr_SetString(), 11
 PyErr_SyntaxLocation (C の関数), 31
 PyErr_SyntaxLocationEx (C の関数), 31
 PyErr_SyntaxLocationObject (C の関数), 31
 PyErr_WarnEx (C の関数), 31
 PyErr_WarnExplicit (C の関数), 32
 PyErr_WarnExplicitObject (C の関数), 32
 PyErr_WarnFormat (C の関数), 32
 PyErr_WriteUnraisable (C の関数), 28
 PyEval_AcquireLock (C の関数), 195
 PyEval_AcquireThread (C の関数), 195
 PyEval_AcquireThread(), 191
 PyEval_EvalCode (C の関数), 23
 PyEval_EvalCodeEx (C の関数), 23
 PyEval_EvalFrame (C の関数), 23
 PyEval_EvalFrameEx (C の関数), 23
 PyEval_GetBuiltins (C の関数), 68
 PyEval_GetFrame (C の関数), 68
 PyEval_GetFuncDesc (C の関数), 68
 PyEval_GetFuncName (C の関数), 68
 PyEval_GetGlobals (C の関数), 68
 PyEval_GetLocals (C の関数), 68
 PyEval_InitThreads (C の関数), 191
 PyEval_InitThreads(), 183
 PyEval_MergeCompilerFlags (C の関数), 24
 PyEval_ReleaseLock (C の関数), 196
 PyEval_ReleaseThread (C の関数), 195
 PyEval_ReleaseThread(), 191
 PyEval_RestoreThread (C の関数), 191
 PyEval_RestoreThread(), 189, 191
 PyEval_SaveThread (C の関数), 191
 PyEval_SaveThread(), 189, 191
 PyEval_SetProfile (C の関数), 200
 PyEval_SetTrace (C の関数), 200
 PyEval_ThreadsInitialized (C の関数), 191
 PyExc_ArithmeticError, 39
 PyExc_AssertionError, 39
 PyExc_AttributeError, 39
 PyExc_BaseException, 39
 PyExc_BlockingIOError, 39
 PyExc_BrokenPipeError, 39
 PyExc_BufferError, 39
 PyExc_BytesWarning, 41
 PyExc_ChildProcessError, 39
 PyExc_ConnectionAbortedError, 39
 PyExc_ConnectionError, 39
 PyExc_ConnectionRefusedError, 39
 PyExc_ConnectionResetError, 39
 PyExc_DeprecationWarning, 41
 PyExc_EnvironmentError, 40
 PyExc_EOFError, 39
 PyExc_Exception, 39
 PyExc_FileExistsError, 39
 PyExc_FileNotFoundError, 39
 PyExc_FloatingPointError, 39
 PyExc_FutureWarning, 41
 PyExc_GeneratorExit, 39
 PyExc_ImportError, 39
 PyExc_ImportWarning, 41
 PyExc_IndentationError, 39
 PyExc_IndexError, 39
 PyExc_InterruptedError, 39
 PyExc_IOError, 40
 PyExc_IsADirectoryError, 39
 PyExc_KeyboardInterrupt, 39
 PyExc_KeyError, 39
 PyExc_LookupError, 39
 PyExc_MemoryError, 39
 PyExc_ModuleNotFoundError, 39
 PyExc_NameError, 39
 PyExc_NotADirectoryError, 39
 PyExc_NotImplementedError, 39
 PyExc_OSError, 39
 PyExc_OverflowError, 39
 PyExc_PendingDeprecationWarning, 41
 PyExc_PermissionError, 39
 PyExc_ProcessLookupError, 39
 PyExc_RecursionError, 39
 PyExc_ReferenceError, 39
 PyExc_ResourceWarning, 41
 PyExc_RuntimeError, 39
 PyExc_RuntimeWarning, 41
 PyExc_StopAsyncIteration, 39
 PyExc_StopIteration, 39
 PyExc_SyntaxError, 39
 PyExc_SyntaxWarning, 41
 PyExc_SystemError, 39
 PyExc_SystemExit, 39
 PyExc_TabError, 39
 PyExc_TimeoutError, 39
 PyExc_TypeError, 39
 PyExc_UnboundLocalError, 39
 PyExc_UnicodeDecodeError, 39
 PyExc_UnicodeEncodeError, 39
 PyExc_UnicodeError, 39
 PyExc_UnicodeTranslateError, 39
 PyExc_UnicodeWarning, 41
 PyExc_UserWarning, 41
 PyExc_ValueError, 39
 PyExc_Warning, 41
 PyExc_WindowsError, 40
 PyExc_ZeroDivisionError, 39
 PyException_GetCause (C の関数), 36
 PyException_GetContext (C の関数), 36
 PyException_GetTraceback (C の関数), 36
 PyException_SetCause (C の関数), 36
 PyException_SetContext (C の関数), 36
 PyException_SetTraceback (C の関数), 36
 PyFile_FromFd (C の関数), 154

- PyFile_GetLine (*C* の関数), 154
- PyFile_SetOpenCodeHook (*C* の関数), 154
- PyFile_WriteObject (*C* の関数), 155
- PyFile_WriteString (*C* の関数), 155
- PyFloat_AS_DOUBLE (*C* の関数), 106
- PyFloat_AsDouble (*C* の関数), 106
- PyFloat_Check (*C* の関数), 106
- PyFloat_CheckExact (*C* の関数), 106
- PyFloat_ClearFreeList (*C* の関数), 107
- PyFloat_FromDouble (*C* の関数), 106
- PyFloat_FromString (*C* の関数), 106
- PyFloat_GetInfo (*C* の関数), 106
- PyFloat_GetMax (*C* の関数), 106
- PyFloat_GetMin (*C* の関数), 106
- PyFloat_Type (*C* の変数), 106
- PyFloatObject (*C* のデータ型), 106
- PyFrame_GetLineNumber (*C* の関数), 68
- PyFrameObject (*C* のデータ型), 23
- PyFrozenSet_Check (*C* の関数), 147
- PyFrozenSet_CheckExact (*C* の関数), 147
- PyFrozenSet_New (*C* の関数), 147
- PyFrozenSet_Type (*C* の変数), 147
- PyFunction_Check (*C* の関数), 149
- PyFunction_GetAnnotations (*C* の関数), 150
- PyFunction_GetClosure (*C* の関数), 150
- PyFunction_GetCode (*C* の関数), 149
- PyFunction_GetDefaults (*C* の関数), 149
- PyFunction_GetGlobals (*C* の関数), 149
- PyFunction_GetModule (*C* の関数), 149
- PyFunction_New (*C* の関数), 149
- PyFunction_NewWithQualName (*C* の関数), 149
- PyFunction_SetAnnotations (*C* の関数), 150
- PyFunction_SetClosure (*C* の関数), 150
- PyFunction_SetDefaults (*C* の関数), 149
- PyFunction_Type (*C* の変数), 149
- PyFunctionObject (*C* のデータ型), 149
- PyGen_Check (*C* の関数), 170
- PyGen_CheckExact (*C* の関数), 170
- PyGen_New (*C* の関数), 170
- PyGen_NewWithQualName (*C* の関数), 171
- PyGen_Type (*C* の変数), 170
- PyGenObject (*C* のデータ型), 170
- PyGetSetDef (*C* のデータ型), 240
- PyGILState_Check (*C* の関数), 193
- PyGILState_Ensure (*C* の関数), 192
- PyGILState_GetThisThreadState (*C* の関数), 193
- PyGILState_Release (*C* の関数), 192
- PyImport_AddModule (*C* の関数), 51
- PyImport_AddModuleObject (*C* の関数), 50
- PyImport_AppendInittab (*C* の関数), 53
- PyImport_Cleanup (*C* の関数), 53
- PyImport_ExecCodeModule (*C* の関数), 51
- PyImport_ExecCodeModuleEx (*C* の関数), 51
- PyImport_ExecCodeModuleObject (*C* の関数), 51
- PyImport_ExecCodeModuleWithPathnames (*C* の関数), 52
- PyImport_ExtendInittab (*C* の関数), 54
- PyImport_FrozenModules (*C* の変数), 53
- PyImport_GetImporter (*C* の関数), 52
- PyImport_GetMagicNumber (*C* の関数), 52
- PyImport_GetMagicTag (*C* の関数), 52
- PyImport_GetModule (*C* の関数), 52
- PyImport_GetModuleDict (*C* の関数), 52
- PyImport_Import (*C* の関数), 50
- PyImport_ImportFrozenModule (*C* の関数), 53
- PyImport_ImportFrozenModuleObject (*C* の関数), 53
- PyImport_ImportModule (*C* の関数), 49
- PyImport_ImportModuleEx (*C* の関数), 50
- PyImport_ImportModuleLevel (*C* の関数), 50
- PyImport_ImportModuleLevelObject (*C* の関数), 50
- PyImport_ImportModuleNoBlock (*C* の関数), 49
- PyImport_ReloadModule (*C* の関数), 50
- PyIndex_Check (*C* の関数), 82
- PyInstanceMethod_Check (*C* の関数), 150
- PyInstanceMethod_Function (*C* の関数), 150
- PyInstanceMethod_GET_FUNCTION (*C* の関数), 150
- PyInstanceMethod_New (*C* の関数), 150
- PyInstanceMethod_Type (*C* の変数), 150
- PyInterpreterState (*C* のデータ型), 191
- PyInterpreterState_Clear (*C* の関数), 194
- PyInterpreterState_Delete (*C* の関数), 194
- PyInterpreterState_GetDict (*C* の関数), 194
- PyInterpreterState_GetID (*C* の関数), 194
- PyInterpreterState_Head (*C* の関数), 200
- PyInterpreterState_Main (*C* の関数), 200
- PyInterpreterState_New (*C* の関数), 194
- PyInterpreterState_Next (*C* の関数), 200
- PyInterpreterState_ThreadHead (*C* の関数), 200
- PyIter_Check (*C* の関数), 86
- PyIter_Next (*C* の関数), 86
- PyList_Append (*C* の関数), 142
- PyList_AsTuple (*C* の関数), 143
- PyList_Check (*C* の関数), 141
- PyList_CheckExact (*C* の関数), 141
- PyList_ClearFreeList (*C* の関数), 143
- PyList_GET_ITEM (*C* の関数), 142
- PyList_GET_SIZE (*C* の関数), 142
- PyList_GetItem (*C* の関数), 142
- PyList_GetItem(), 9
- PyList_GetSlice (*C* の関数), 142
- PyList_Insert (*C* の関数), 142
- PyList_New (*C* の関数), 141
- PyList_Reverse (*C* の関数), 143
- PyList_SET_ITEM (*C* の関数), 142
- PyList_SetItem (*C* の関数), 142
- PyList_SetItem(), 8
- PyList_SetSlice (*C* の関数), 143
- PyList_Size (*C* の関数), 142
- PyList_Sort (*C* の関数), 143
- PyList_Type (*C* の変数), 141
- PyListObject (*C* のデータ型), 141
- PyLong_AsDouble (*C* の関数), 105
- PyLong_AsLong (*C* の関数), 102
- PyLong_AsLongAndOverflow (*C* の関数), 102
- PyLong_AsLongLong (*C* の関数), 103
- PyLong_AsLongLongAndOverflow (*C* の関数), 103
- PyLong_AsSize_t (*C* の関数), 104
- PyLong_AsSsize_t (*C* の関数), 103
- PyLong_AsUnsignedLong (*C* の関数), 104
- PyLong_AsUnsignedLongLong (*C* の関数), 104
- PyLong_AsUnsignedLongLongMask (*C* の関数), 104
- PyLong_AsVoidPtr (*C* の関数), 105
- PyLong_Check (*C* の関数), 101
- PyLong_CheckExact (*C* の関数), 101
- PyLong_FromDouble (*C* の関数), 102
- PyLong_FromLong (*C* の関数), 101
- PyLong_FromLongLong (*C* の関数), 101
- PyLong_FromSize_t (*C* の関数), 101
- PyLong_FromSsize_t (*C* の関数), 101
- PyLong_FromString (*C* の関数), 102
- PyLong_FromUnicode (*C* の関数), 102
- PyLong_FromUnicodeObject (*C* の関数), 102
- PyLong_FromUnsignedLong (*C* の関数), 101
- PyLong_FromUnsignedLongLong (*C* の関数), 102
- PyLong_FromVoidPtr (*C* の関数), 102
- PyLong_Type (*C* の変数), 101
- PyLongObject (*C* のデータ型), 101
- PyMapping_Check (*C* の関数), 85
- PyMapping_DelItem (*C* の関数), 85
- PyMapping_DelItemString (*C* の関数), 85
- PyMapping_GetItemString (*C* の関数), 85
- PyMapping_HasKey (*C* の関数), 85
- PyMapping_HasKeyString (*C* の関数), 85
- PyMapping_Items (*C* の関数), 86
- PyMapping_Keys (*C* の関数), 85
- PyMapping_Length (*C* の関数), 85

- PyMapping_SetItemString (*C* の関数), 85
- PyMapping_Size (*C* の関数), 85
- PyMapping_Values (*C* の関数), 86
- PyMappingMethods (*C* のデータ型), 276
- PyMappingMethods.mp_ass_subscript (*C* のメンバ変数), 276
- PyMappingMethods.mp_length (*C* のメンバ変数), 276
- PyMappingMethods.mp_subscript (*C* のメンバ変数), 276
- PyMarshal_ReadLastObjectFromFile (*C* の関数), 55
- PyMarshal_ReadLongFromFile (*C* の関数), 54
- PyMarshal_ReadObjectFromFile (*C* の関数), 55
- PyMarshal_ReadObjectFromString (*C* の関数), 55
- PyMarshal_ReadShortFromFile (*C* の関数), 55
- PyMarshal_WriteLongToFile (*C* の関数), 54
- PyMarshal_WriteObjectToFile (*C* の関数), 54
- PyMarshal_WriteObjectToString (*C* の関数), 54
- PyMem_Calloc (*C* の関数), 227
- PyMem_Del (*C* の関数), 228
- PYMEM_DOMAIN_MEM (*C* のマクロ), 231
- PYMEM_DOMAIN_OBJ (*C* のマクロ), 231
- PYMEM_DOMAIN_RAW (*C* のマクロ), 231
- PyMem_Free (*C* の関数), 228
- PyMem_GetAllocator (*C* の関数), 231
- PyMem_Malloc (*C* の関数), 227
- PyMem_New (*C* の関数), 228
- PyMem_RawCalloc (*C* の関数), 226
- PyMem_RawFree (*C* の関数), 227
- PyMem_RawMalloc (*C* の関数), 226
- PyMem_RawRealloc (*C* の関数), 227
- PyMem_Realloc (*C* の関数), 228
- PyMem_Resize (*C* の関数), 228
- PyMem_SetAllocator (*C* の関数), 231
- PyMem_SetupDebugHooks (*C* の関数), 231
- PyMemAllocatorDomain (*C* のデータ型), 231
- PyMemAllocatorEx (*C* のデータ型), 230
- PyMemberDef (*C* のデータ型), 239
- PyMemoryView_Check (*C* の関数), 167
- PyMemoryView_FromBuffer (*C* の関数), 166
- PyMemoryView_FromMemory (*C* の関数), 166
- PyMemoryView_FromObject (*C* の関数), 166
- PyMemoryView_GET_BASE (*C* の関数), 167
- PyMemoryView_GET_BUFFER (*C* の関数), 167
- PyMemoryView_GetContiguous (*C* の関数), 166
- PyMethod_Check (*C* の関数), 151
- PyMethod_ClearFreeList (*C* の関数), 151
- PyMethod_Function (*C* の関数), 151
- PyMethod_GET_FUNCTION (*C* の関数), 151
- PyMethod_GET_SELF (*C* の関数), 151
- PyMethod_New (*C* の関数), 151
- PyMethod_Self (*C* の関数), 151
- PyMethod_Type (*C* の変数), 151
- PyMethodDef (*C* のデータ型), 238
- PyModule_AddFunctions (*C* の関数), 161
- PyModule_AddIntConstant (*C* の関数), 162
- PyModule_AddIntMacro (*C* の関数), 162
- PyModule_AddObject (*C* の関数), 161
- PyModule_AddStringConstant (*C* の関数), 162
- PyModule_AddStringMacro (*C* の関数), 162
- PyModule_Check (*C* の関数), 155
- PyModule_CheckExact (*C* の関数), 155
- PyModule_Create (*C* の関数), 158
- PyModule_Create2 (*C* の関数), 158
- PyModule_ExecDef (*C* の関数), 161
- PyModule_FromDefAndSpec (*C* の関数), 160
- PyModule_FromDefAndSpec2 (*C* の関数), 160
- PyModule_GetDef (*C* の関数), 156
- PyModule_GetDict (*C* の関数), 156
- PyModule_GetFilename (*C* の関数), 156
- PyModule_GetFilenameObject (*C* の関数), 156
- PyModule_GetName (*C* の関数), 156
- PyModule_GetNameObject (*C* の関数), 156
- PyModule_GetState (*C* の関数), 156
- PyModule_New (*C* の関数), 155
- PyModule_NewObject (*C* の関数), 155
- PyModule_SetDocString (*C* の関数), 161
- PyModule_Type (*C* の変数), 155
- PyModuleDef (*C* のデータ型), 157
- PyModuleDef_Init (*C* の関数), 159
- PyModuleDef_Slot (*C* のデータ型), 159
- PyModuleDef_Slot.slot (*C* のメンバ変数), 159
- PyModuleDef_Slot.value (*C* のメンバ変数), 159
- PyModuleDef.m_base (*C* のメンバ変数), 157
- PyModuleDef.m_clear (*C* のメンバ変数), 158
- PyModuleDef.m_doc (*C* のメンバ変数), 157
- PyModuleDef.m_free (*C* のメンバ変数), 158
- PyModuleDef.m_methods (*C* のメンバ変数), 157
- PyModuleDef.m_name (*C* のメンバ変数), 157
- PyModuleDef.m_reload (*C* のメンバ変数), 158
- PyModuleDef.m_size (*C* のメンバ変数), 157
- PyModuleDef.m_slots (*C* のメンバ変数), 157
- PyModuleDef.m_traverse (*C* のメンバ変数), 158
- PyNumber_Absolute (*C* の関数), 79
- PyNumber_Add (*C* の関数), 78
- PyNumber_And (*C* の関数), 80
- PyNumber_AsSsize_t (*C* の関数), 82
- PyNumber_Check (*C* の関数), 78
- PyNumber_Divmod (*C* の関数), 79
- PyNumber_Float (*C* の関数), 81
- PyNumber_FloorDivide (*C* の関数), 79
- PyNumber_Index (*C* の関数), 82
- PyNumber_InPlaceAdd (*C* の関数), 80
- PyNumber_InPlaceAnd (*C* の関数), 81
- PyNumber_InPlaceFloorDivide (*C* の関数), 80
- PyNumber_InPlaceLshift (*C* の関数), 81
- PyNumber_InPlaceMatrixMultiply (*C* の関数), 80
- PyNumber_InPlaceMultiply (*C* の関数), 80
- PyNumber_InPlaceOr (*C* の関数), 81
- PyNumber_InPlacePower (*C* の関数), 81
- PyNumber_InPlaceRemainder (*C* の関数), 81
- PyNumber_InPlaceRshift (*C* の関数), 81
- PyNumber_InPlaceSubtract (*C* の関数), 80
- PyNumber_InPlaceTrueDivide (*C* の関数), 80
- PyNumber_InPlaceXor (*C* の関数), 81
- PyNumber_Invert (*C* の関数), 79
- PyNumber_Long (*C* の関数), 81
- PyNumber_Lshift (*C* の関数), 79
- PyNumber_MatrixMultiply (*C* の関数), 79
- PyNumber_Multiply (*C* の関数), 78
- PyNumber_Negative (*C* の関数), 79
- PyNumber_Or (*C* の関数), 80
- PyNumber_Positive (*C* の関数), 79
- PyNumber_Power (*C* の関数), 79
- PyNumber_Remainder (*C* の関数), 79
- PyNumber_Rshift (*C* の関数), 80
- PyNumber_Subtract (*C* の関数), 78
- PyNumber_ToBase (*C* の関数), 82
- PyNumber_TrueDivide (*C* の関数), 79
- PyNumber_Xor (*C* の関数), 80
- PyNumberMethods (*C* のデータ型), 273
- PyNumberMethods.nb_absolute (*C* のメンバ変数), 275
- PyNumberMethods.nb_add (*C* のメンバ変数), 275
- PyNumberMethods.nb_and (*C* のメンバ変数), 275
- PyNumberMethods.nb_bool (*C* のメンバ変数), 275
- PyNumberMethods.nb_divmod (*C* のメンバ変数), 275
- PyNumberMethods.nb_float (*C* のメンバ変数), 275
- PyNumberMethods.nb_floor_divide (*C* のメンバ変数), 276
- PyNumberMethods.nb_index (*C* のメンバ変数), 276
- PyNumberMethods.nb_inplace_add (*C* のメンバ変数), 275
- PyNumberMethods.nb_inplace_and (*C* のメンバ変数), 275
- PyNumberMethods.nb_inplace_floor_divide (*C* のメンバ変数), 276
- PyNumberMethods.nb_inplace_lshift (*C* のメンバ変数), 275
- PyNumberMethods.nb_inplace_matrix_multiply (*C* のメンバ変数), 276

- PyNumberMethods.nb_inplace_multiply (*C* のメンバ変数), 275
- PyNumberMethods.nb_inplace_or (*C* のメンバ変数), 276
- PyNumberMethods.nb_inplace_power (*C* のメンバ変数), 275
- PyNumberMethods.nb_inplace_remainder (*C* のメンバ変数), 275
- PyNumberMethods.nb_inplace_rshift (*C* のメンバ変数), 275
- PyNumberMethods.nb_inplace_subtract (*C* のメンバ変数), 275
- PyNumberMethods.nb_inplace_true_divide (*C* のメンバ変数), 276
- PyNumberMethods.nb_inplace_xor (*C* のメンバ変数), 276
- PyNumberMethods.nb_int (*C* のメンバ変数), 275
- PyNumberMethods.nb_invert (*C* のメンバ変数), 275
- PyNumberMethods.nb_lshift (*C* のメンバ変数), 275
- PyNumberMethods.nb_matrix_multiply (*C* のメンバ変数), 276
- PyNumberMethods.nb_multiply (*C* のメンバ変数), 275
- PyNumberMethods.nb_negative (*C* のメンバ変数), 275
- PyNumberMethods.nb_or (*C* のメンバ変数), 275
- PyNumberMethods.nb_positive (*C* のメンバ変数), 275
- PyNumberMethods.nb_power (*C* のメンバ変数), 275
- PyNumberMethods.nb_remainder (*C* のメンバ変数), 275
- PyNumberMethods.nb_reserved (*C* のメンバ変数), 275
- PyNumberMethods.nb_rshift (*C* のメンバ変数), 275
- PyNumberMethods.nb_subtract (*C* のメンバ変数), 275
- PyNumberMethods.nb_true_divide (*C* のメンバ変数), 276
- PyNumberMethods.nb_xor (*C* のメンバ変数), 275
- PyObject (*C* のデータ型), 236
- PyObject_AsCharBuffer (*C* の関数), 95
- PyObject_ASCII (*C* の関数), 73
- PyObject_AsFileDescriptor (*C* の関数), 154
- PyObject_AsReadBuffer (*C* の関数), 95
- PyObject_AsWriteBuffer (*C* の関数), 96
- PyObject_Bytes (*C* の関数), 73
- PyObject_Call (*C* の関数), 74
- PyObject_CallFunction (*C* の関数), 75
- PyObject_CallFunctionObjArgs (*C* の関数), 75
- PyObject_CallMethod (*C* の関数), 75
- PyObject_CallMethodObjArgs (*C* の関数), 75
- PyObject_CallObject (*C* の関数), 75
- PyObject_Calloc (*C* の関数), 229
- PyObject_CheckBuffer (*C* の関数), 94
- PyObject_CheckReadBuffer (*C* の関数), 95
- PyObject_Del (*C* の関数), 235
- PyObject_DelAttr (*C* の関数), 72
- PyObject_DelAttrString (*C* の関数), 72
- PyObject_DelItem (*C* の関数), 78
- PyObject_Dir (*C* の関数), 78
- PyObject_Free (*C* の関数), 229
- PyObject_GC_Del (*C* の関数), 285
- PyObject_GC_New (*C* の関数), 285
- PyObject_GC_NewVar (*C* の関数), 285
- PyObject_GC_Resize (*C* の関数), 285
- PyObject_GC_Track (*C* の関数), 285
- PyObject_GC_UnTrack (*C* の関数), 286
- PyObject_GenericGetAttr (*C* の関数), 72
- PyObject_GenericGetDict (*C* の関数), 72
- PyObject_GenericSetAttr (*C* の関数), 72
- PyObject_GenericSetDict (*C* の関数), 73
- PyObject_GetArenaAllocator (*C* の関数), 233
- PyObject_GetAttr (*C* の関数), 72
- PyObject_GetAttrString (*C* の関数), 72
- PyObject_GetBuffer (*C* の関数), 94
- PyObject_GetItem (*C* の関数), 78
- PyObject_GetIter (*C* の関数), 78
- PyObject_HasAttr (*C* の関数), 71
- PyObject_HasAttrString (*C* の関数), 71
- PyObject_Hash (*C* の関数), 77
- PyObject_HashNotImplemented (*C* の関数), 77
- PyObject_HEAD (*C* のマクロ), 236
- PyObject_HEAD_INIT (*C* のマクロ), 237
- PyObject_Init (*C* の関数), 235
- PyObject_InitVar (*C* の関数), 235
- PyObject_IsInstance (*C* の関数), 74
- PyObject_IsSubclass (*C* の関数), 74
- PyObject_IsTrue (*C* の関数), 77
- PyObject_Length (*C* の関数), 77
- PyObject_LengthHint (*C* の関数), 77
- PyObject_Malloc (*C* の関数), 229
- PyObject_New (*C* の関数), 235
- PyObject_NewVar (*C* の関数), 235
- PyObject_Not (*C* の関数), 77
- PyObject._ob_next (*C* のメンバ変数), 249
- PyObject._ob_prev (*C* のメンバ変数), 249
- PyObject_Print (*C* の関数), 71
- PyObject_Realloc (*C* の関数), 229
- PyObject_Repr (*C* の関数), 73
- PyObject_RichCompare (*C* の関数), 73
- PyObject_RichCompareBool (*C* の関数), 73
- PyObject_SetArenaAllocator (*C* の関数), 233
- PyObject_SetAttr (*C* の関数), 72
- PyObject_SetAttrString (*C* の関数), 72
- PyObject_SetItem (*C* の関数), 78
- PyObject_Size (*C* の関数), 77
- PyObject_Str (*C* の関数), 73
- PyObject_Type (*C* の関数), 77
- PyObject_TypeCheck (*C* の関数), 77
- PyObject_VAR_HEAD (*C* のマクロ), 236
- PyObjectArenaAllocator (*C* のデータ型), 233
- PyObject.ob_refcnt (*C* のメンバ変数), 250
- PyObject.ob_type (*C* のメンバ変数), 250
- PyOS_AfterFork (*C* の関数), 44
- PyOS_AfterFork_Child (*C* の関数), 44
- PyOS_AfterFork_Parent (*C* の関数), 43
- PyOS_BeforeFork (*C* の関数), 43
- PyOS_CheckStack (*C* の関数), 44
- PyOS_double_to_string (*C* の関数), 67
- PyOS_FSPath (*C* の関数), 43
- PyOS_getsig (*C* の関数), 44
- PyOS_InputHook (*C* の変数), 21
- PyOS_ReadlineFunctionPointer (*C* の変数), 21
- PyOS_setsig (*C* の関数), 44
- PyOS_snprintf (*C* の関数), 66
- PyOS_stricmp (*C* の関数), 67
- PyOS_string_to_double (*C* の関数), 66
- PyOS_strnicmp (*C* の関数), 67
- PyOS_vsnprintf (*C* の関数), 66
- PyParser_SimpleParseFile (*C* の関数), 21
- PyParser_SimpleParseFileFlags (*C* の関数), 22
- PyParser_SimpleParseString (*C* の関数), 21
- PyParser_SimpleParseStringFlags (*C* の関数), 21
- PyParser_SimpleParseStringFlagsFilename (*C* の関数), 21
- PyPreConfig (*C* のデータ型), 208
- PyPreConfig_InitIsolatedConfig (*C* の関数), 208
- PyPreConfig_InitPythonConfig (*C* の関数), 208
- PyPreConfig.allocator (*C* のメンバ変数), 208
- PyPreConfig.coerce_c_locale (*C* のメンバ変数), 209
- PyPreConfig.coerce_c_locale_warn (*C* のメンバ変数), 209
- PyPreConfig.configure_locale (*C* のメンバ変数), 209
- PyPreConfig.dev_mode (*C* のメンバ変数), 209
- PyPreConfig.isolated (*C* のメンバ変数), 209
- PyPreConfig.legacy_windows_fs_encoding (*C* のメンバ変数), 209
- PyPreConfig.parse_argv (*C* のメンバ変数), 209
- PyPreConfig.use_environment (*C* のメンバ変数), 209
- PyPreConfig.utf8_mode (*C* のメンバ変数), 210
- PyProperty_Type (*C* の変数), 164
- PyRun_AnyFile (*C* の関数), 19
- PyRun_AnyFileEx (*C* の関数), 19
- PyRun_AnyFileExFlags (*C* の関数), 20
- PyRun_AnyFileFlags (*C* の関数), 19
- PyRun_File (*C* の関数), 22
- PyRun_FileEx (*C* の関数), 22

PyRun_FileExFlags (*C* の関数), 22
 PyRun_FileFlags (*C* の関数), 22
 PyRun_InteractiveLoop (*C* の関数), 21
 PyRun_InteractiveLoopFlags (*C* の関数), 21
 PyRun_InteractiveOne (*C* の関数), 20
 PyRun_InteractiveOneFlags (*C* の関数), 20
 PyRun_SimpleFile (*C* の関数), 20
 PyRun_SimpleFileEx (*C* の関数), 20
 PyRun_SimpleFileExFlags (*C* の関数), 20
 PyRun_SimpleString (*C* の関数), 20
 PyRun_SimpleStringFlags (*C* の関数), 20
 PyRun_String (*C* の関数), 22
 PyRun_StringFlags (*C* の関数), 22
 PySeqIter_Check (*C* の関数), 163
 PySeqIter_New (*C* の関数), 163
 PySeqIter_Type (*C* の変数), 163
 PySequence_Check (*C* の関数), 82
 PySequence_Concat (*C* の関数), 82
 PySequence_Contains (*C* の関数), 83
 PySequence_Count (*C* の関数), 83
 PySequence_DelItem (*C* の関数), 83
 PySequence_DelSlice (*C* の関数), 83
 PySequence_Fast (*C* の関数), 84
 PySequence_Fast_GET_ITEM (*C* の関数), 84
 PySequence_Fast_GET_SIZE (*C* の関数), 84
 PySequence_Fast_ITEMS (*C* の関数), 84
 PySequence_GetItem (*C* の関数), 83
 PySequence_GetItem(), 9
 PySequence_GetSlice (*C* の関数), 83
 PySequence_Index (*C* の関数), 83
 PySequence_InPlaceConcat (*C* の関数), 82
 PySequence_InPlaceRepeat (*C* の関数), 83
 PySequence_ITEM (*C* の関数), 84
 PySequence_Length (*C* の関数), 82
 PySequence_List (*C* の関数), 83
 PySequence_Repeat (*C* の関数), 82
 PySequence_SetItem (*C* の関数), 83
 PySequence_SetSlice (*C* の関数), 83
 PySequence_Size (*C* の関数), 82
 PySequence_Tuple (*C* の関数), 84
 PySequenceMethods (*C* のデータ型), 277
 PySequenceMethods.sq_ass_item (*C* のメンバ変数), 277
 PySequenceMethods.sq_concat (*C* のメンバ変数), 277
 PySequenceMethods.sq_contains (*C* のメンバ変数), 277
 PySequenceMethods.sq_inplace_concat (*C* のメンバ変数), 277
 PySequenceMethods.sq_inplace_repeat (*C* のメンバ変数), 277
 PySequenceMethods.sq_item (*C* のメンバ変数), 277
 PySequenceMethods.sq_length (*C* のメンバ変数), 277
 PySequenceMethods.sq_repeat (*C* のメンバ変数), 277
 PySet_Add (*C* の関数), 148
 PySet_Check (*C* の関数), 147
 PySet_Clear (*C* の関数), 148
 PySet_ClearFreeList (*C* の関数), 148
 PySet_Contains (*C* の関数), 148
 PySet_Discard (*C* の関数), 148
 PySet_GET_SIZE (*C* の関数), 148
 PySet_New (*C* の関数), 147
 PySet_Pop (*C* の関数), 148
 PySet_Size (*C* の関数), 148
 PySet_Type (*C* の変数), 147
 PySetObject (*C* のデータ型), 147
 PySignal_SetWakeupFd (*C* の関数), 35
 PySlice_AdjustIndices (*C* の関数), 166
 PySlice_Check (*C* の関数), 164
 PySlice_GetIndices (*C* の関数), 164
 PySlice_GetIndicesEx (*C* の関数), 165
 PySlice_New (*C* の関数), 164
 PySlice_Type (*C* の変数), 164
 PySlice_Unpack (*C* の関数), 165
 PyState_AddModule (*C* の関数), 162
 PyState_FindModule (*C* の関数), 162
 PyState_RemoveModule (*C* の関数), 163
 PyStatus (*C* のデータ型), 207
 PyStatus_Error (*C* の関数), 207
 PyStatus_Exception (*C* の関数), 207
 PyStatus_Exit (*C* の関数), 207
 PyStatus_IsError (*C* の関数), 207
 PyStatus_IsExit (*C* の関数), 207
 PyStatus_NoMemory (*C* の関数), 207
 PyStatus_Ok (*C* の関数), 207
 PyStatus.err_msg (*C* のメンバ変数), 207
 PyStatus.exitcode (*C* のメンバ変数), 207
 PyStatus.func (*C* のメンバ変数), 207
 PyStructSequence_Desc (*C* のデータ型), 140
 PyStructSequence_Field (*C* のデータ型), 140
 PyStructSequence_GET_ITEM (*C* の関数), 141
 PyStructSequence_GetItem (*C* の関数), 141
 PyStructSequence_InitType (*C* の関数), 140
 PyStructSequence_InitType2 (*C* の関数), 140
 PyStructSequence_New (*C* の関数), 141
 PyStructSequence_NewType (*C* の関数), 140
 PyStructSequence_SET_ITEM (*C* の関数), 141
 PyStructSequence_SetItem (*C* の関数), 141
 PyStructSequence_UnnamedField (*C* の変数), 140
 PySys_AddAuditHook (*C* の関数), 48
 PySys_AddWarnOption (*C* の関数), 46
 PySys_AddWarnOptionUnicode (*C* の関数), 47
 PySys_AddXOption (*C* の関数), 47
 PySys_Audit (*C* の関数), 48
 PySys_FormatStderr (*C* の関数), 47
 PySys_FormatStdout (*C* の関数), 47
 PySys_GetObject (*C* の関数), 46
 PySys_GetXOptions (*C* の関数), 47
 PySys_ResetWarnOptions (*C* の関数), 46
 PySys_SetArgv (*C* の関数), 188
 PySys_SetArgv(), 183
 PySys_SetArgvEx (*C* の関数), 187
 PySys_SetArgvEx(), 13, 183
 PySys_SetObject (*C* の関数), 46
 PySys_SetPath (*C* の関数), 47
 PySys_WriteStderr (*C* の関数), 47
 PySys_WriteStdout (*C* の関数), 47
 Python 3000, 304
 Python Enhancement Proposals
 PEP 1, 303
 PEP 7, 3, 6
 PEP 238, 24, 296
 PEP 278, 306
 PEP 302, 296, 300
 PEP 343, 294
 PEP 362, 292, 303
 PEP 383, 123, 124
 PEP 384, 17
 PEP 393, 113, 122
 PEP 411, 304
 PEP 420, 296, 302, 303
 PEP 432, 221, 222
 PEP 442, 272
 PEP 443, 297
 PEP 451, 160, 296
 PEP 484, 291, 296, 306, 307
 PEP 489, 160
 PEP 492, 292294
 PEP 498, 296
 PEP 519, 303
 PEP 525, 292
 PEP 526, 291, 307
 PEP 528, 182
 PEP 529, 124, 181
 PEP 538, 218
 PEP 539, 201
 PEP 540, 218
 PEP 552, 213
 PEP 578, 48

PEP 587, 206
PEP 623, 113
PEP 3116, 307
PEP 3119, 74
PEP 3121, 157
PEP 3147, 52
PEP 3151, 40
PEP 3155, 304
PYTHON*, 181
PYTHONCOERCECLOCALE, 219
PYTHONDEBUG, 180
PYTHONDONTWRITEBYTECODE, 181
PYTHONDUMPPREFS, 250
PYTHONHASHSEED, 181
PYTHONHOME, 13, 14, 181, 188, 213
Pythonic, 304
PYTHONINSPECT, 181
PYTHONIOENCODING, 184
PYTHONLEGACYWINDOWSFSENCODING, 181
PYTHONLEGACYWINDOWSSTDIO, 182
PYTHONMALLOC, 226, 230, 232
PYTHONMALLOCSTATS, 226
PYTHONNOUSERSITE, 182
PYTHONOPTIMIZE, 182
PYTHONPATH, 13, 14, 181, 214
PYTHONUNBUFFERED, 182
PYTHONUTF8, 219
PYTHONVERBOSE, 182
PyThread_create_key (*C* の関数), 203
PyThread_delete_key (*C* の関数), 203
PyThread_delete_key_value (*C* の関数), 203
PyThread_get_key_value (*C* の関数), 203
PyThread_ReInitTLS (*C* の関数), 203
PyThread_set_key_value (*C* の関数), 203
PyThread_tss_alloc (*C* の関数), 202
PyThread_tss_create (*C* の関数), 202
PyThread_tss_delete (*C* の関数), 202
PyThread_tss_free (*C* の関数), 202
PyThread_tss_get (*C* の関数), 202
PyThread_tss_is_created (*C* の関数), 202
PyThread_tss_set (*C* の関数), 202
PyThreadState, 188
PyThreadState (*C* のデータ型), 191
PyThreadState_Clear (*C* の関数), 194
PyThreadState_Delete (*C* の関数), 194
PyThreadState_Get (*C* の関数), 192
PyThreadState_GetDict (*C* の関数), 194
PyThreadState_New (*C* の関数), 194
PyThreadState_Next (*C* の関数), 201
PyThreadState_SetAsyncExc (*C* の関数), 195
PyThreadState_Swap (*C* の関数), 192
PyTime_Check (*C* の関数), 174
PyTime_CheckExact (*C* の関数), 174
PyTime_FromTime (*C* の関数), 175
PyTime_FromTimeAndFold (*C* の関数), 175
PyTimeZone_FromOffset (*C* の関数), 175
PyTimeZone_FromOffsetAndName (*C* の関数), 175
PyTrace_C_CALL (*C* の変数), 200
PyTrace_C_EXCEPTION (*C* の変数), 200
PyTrace_C_RETURN (*C* の変数), 200
PyTrace_CALL (*C* の変数), 199
PyTrace_EXCEPTION (*C* の変数), 199
PyTrace_LINE (*C* の変数), 199
PyTrace_OPCODE (*C* の変数), 200
PyTrace_RETURN (*C* の変数), 200
PyTraceMalloc_Track (*C* の関数), 233
PyTraceMalloc_Untrack (*C* の関数), 233
PyTuple_Check (*C* の関数), 138
PyTuple_CheckExact (*C* の関数), 138
PyTuple_ClearFreeList (*C* の関数), 139
PyTuple_GET_ITEM (*C* の関数), 139
PyTuple_GET_SIZE (*C* の関数), 138
PyTuple_GetItem (*C* の関数), 139

PyTuple_GetSlice (*C* の関数), 139
PyTuple_New (*C* の関数), 138
PyTuple_Pack (*C* の関数), 138
PyTuple_SET_ITEM (*C* の関数), 139
PyTuple_SetItem (*C* の関数), 139
PyTuple_SetItem(), 8
PyTuple_Size (*C* の関数), 138
PyTuple_Type (*C* の変数), 138
PyTupleObject (*C* のデータ型), 138
PyType_Check (*C* の関数), 97
PyType_CheckExact (*C* の関数), 97
PyType_ClearCache (*C* の関数), 97
PyType_FromSpec (*C* の関数), 99
PyType_FromSpecWithBases (*C* の関数), 99
PyType_GenericAlloc (*C* の関数), 98
PyType_GenericNew (*C* の関数), 98
PyType_GetFlags (*C* の関数), 98
PyType_GetSlot (*C* の関数), 98
PyType_HasFeature (*C* の関数), 98
PyType_IS_GC (*C* の関数), 98
PyType_IsSubtype (*C* の関数), 98
PyType_Modified (*C* の関数), 98
PyType_Ready (*C* の関数), 98
PyType_Slot (*C* のデータ型), 99
PyType_Slot.PyType_Slot.pfunc (*C* のメンバ変数), 100
PyType_Slot.PyType_Slot.slot (*C* のメンバ変数), 99
PyType_Spec (*C* のデータ型), 99
PyType_Spec.PyType_Spec.basicsize (*C* のメンバ変数), 99
PyType_Spec.PyType_Spec.flags (*C* のメンバ変数), 99
PyType_Spec.PyType_Spec.itemsize (*C* のメンバ変数), 99
PyType_Spec.PyType_Spec.name (*C* のメンバ変数), 99
PyType_Spec.PyType_Spec.slots (*C* のメンバ変数), 99
PyType_Type (*C* の変数), 97
PyTypeObject (*C* のデータ型), 97
PyTypeObject.tp_alloc (*C* のメンバ変数), 269
PyTypeObject.tp_allocs (*C* のメンバ変数), 272
PyTypeObject.tp_as_async (*C* のメンバ変数), 254
PyTypeObject.tp_as_buffer (*C* のメンバ変数), 257
PyTypeObject.tp_as_mapping (*C* のメンバ変数), 255
PyTypeObject.tp_as_number (*C* のメンバ変数), 255
PyTypeObject.tp_as_sequence (*C* のメンバ変数), 255
PyTypeObject.tp_base (*C* のメンバ変数), 266
PyTypeObject.tp_bases (*C* のメンバ変数), 271
PyTypeObject.tp_basicsize (*C* のメンバ変数), 251
PyTypeObject.tp_cache (*C* のメンバ変数), 271
PyTypeObject.tp_call (*C* のメンバ変数), 256
PyTypeObject.tp_clear (*C* のメンバ変数), 262
PyTypeObject.tp_dealloc (*C* のメンバ変数), 252
PyTypeObject.tp_del (*C* のメンバ変数), 271
PyTypeObject.tp_descr_get (*C* のメンバ変数), 267
PyTypeObject.tp_descr_set (*C* のメンバ変数), 267
PyTypeObject.tp_dict (*C* のメンバ変数), 266
PyTypeObject.tp_dictoffset (*C* のメンバ変数), 267
PyTypeObject.tp_doc (*C* のメンバ変数), 261
PyTypeObject.tp_finalize (*C* のメンバ変数), 271
PyTypeObject.tp_flags (*C* のメンバ変数), 258
PyTypeObject.tp_free (*C* のメンバ変数), 270
PyTypeObject.tp_frees (*C* のメンバ変数), 272
PyTypeObject.tp_getattr (*C* のメンバ変数), 254
PyTypeObject.tp_getattro (*C* のメンバ変数), 257
PyTypeObject.tp_getset (*C* のメンバ変数), 266
PyTypeObject.tp_hash (*C* のメンバ変数), 255
PyTypeObject.tp_init (*C* のメンバ変数), 268
PyTypeObject.tp_is_gc (*C* のメンバ変数), 270
PyTypeObject.tp_itemsize (*C* のメンバ変数), 251
PyTypeObject.tp_iter (*C* のメンバ変数), 265
PyTypeObject.tp_iternext (*C* のメンバ変数), 265
PyTypeObject.tp_maxalloc (*C* のメンバ変数), 272
PyTypeObject.tp_members (*C* のメンバ変数), 265
PyTypeObject.tp_methods (*C* のメンバ変数), 265
PyTypeObject.tp_mro (*C* のメンバ変数), 271
PyTypeObject.tp_name (*C* のメンバ変数), 251
PyTypeObject.tp_new (*C* のメンバ変数), 269

PyTypeObject.tp_next (*C* のメンバ変数), 273
 PyTypeObject.tp_prev (*C* のメンバ変数), 272
 PyTypeObject.tp_repr (*C* のメンバ変数), 254
 PyTypeObject.tp_richcompare (*C* のメンバ変数), 263
 PyTypeObject.tp_setattr (*C* のメンバ変数), 254
 PyTypeObject.tp_setattro (*C* のメンバ変数), 257
 PyTypeObject.tp_str (*C* のメンバ変数), 256
 PyTypeObject.tp_subclasses (*C* のメンバ変数), 271
 PyTypeObject.tp_traverse (*C* のメンバ変数), 261
 PyTypeObject.tp_vectorcall_offset (*C* のメンバ変数), 253
 PyTypeObject.tp_version_tag (*C* のメンバ変数), 271
 PyTypeObject.tp_weaklist (*C* のメンバ変数), 271
 PyTypeObject.tp_weaklistoffset (*C* のメンバ変数), 264
 PyTZInfo.Check (*C* の関数), 174
 PyTZInfo.CheckExact (*C* の関数), 174
 PyUnicode_1BYTE_DATA (*C* の関数), 114
 PyUnicode_1BYTE_KIND (*C* のマクロ), 114
 PyUnicode_2BYTE_DATA (*C* の関数), 114
 PyUnicode_2BYTE_KIND (*C* のマクロ), 114
 PyUnicode_4BYTE_DATA (*C* の関数), 114
 PyUnicode_4BYTE_KIND (*C* のマクロ), 114
 PyUnicode_AS_DATA (*C* の関数), 116
 PyUnicode_AS_UNICODE (*C* の関数), 116
 PyUnicode_AsASCIIString (*C* の関数), 133
 PyUnicode_AsCharmapString (*C* の関数), 134
 PyUnicode_AsEncodedString (*C* の関数), 127
 PyUnicode_AsLatin1String (*C* の関数), 133
 PyUnicode_AsMBCSString (*C* の関数), 135
 PyUnicode_AsRawUnicodeEscapeString (*C* の関数), 132
 PyUnicode_AsUCS4 (*C* の関数), 121
 PyUnicode_AsUCS4Copy (*C* の関数), 121
 PyUnicode_AsUnicode (*C* の関数), 122
 PyUnicode_AsUnicodeAndSize (*C* の関数), 122
 PyUnicode_AsUnicodeCopy (*C* の関数), 123
 PyUnicode_AsUnicodeEscapeString (*C* の関数), 132
 PyUnicode_AsUTF8 (*C* の関数), 128
 PyUnicode_AsUTF8AndSize (*C* の関数), 128
 PyUnicode_AsUTF8String (*C* の関数), 128
 PyUnicode_AsUTF16String (*C* の関数), 130
 PyUnicode_AsUTF32String (*C* の関数), 129
 PyUnicode_AsWideChar (*C* の関数), 126
 PyUnicode_AsWideCharString (*C* の関数), 126
 PyUnicode_Check (*C* の関数), 114
 PyUnicode_CheckExact (*C* の関数), 114
 PyUnicode_ClearFreeList (*C* の関数), 115
 PyUnicode_Compare (*C* の関数), 137
 PyUnicode_CompareWithASCIIString (*C* の関数), 137
 PyUnicode_Concat (*C* の関数), 136
 PyUnicode_Contains (*C* の関数), 137
 PyUnicode_CopyCharacters (*C* の関数), 120
 PyUnicode_Count (*C* の関数), 137
 PyUnicode_DATA (*C* の関数), 115
 PyUnicode_Decode (*C* の関数), 127
 PyUnicode_DecodeASCII (*C* の関数), 133
 PyUnicode_DecodeCharmap (*C* の関数), 134
 PyUnicode_DecodeFSDefault (*C* の関数), 125
 PyUnicode_DecodeFSDefaultAndSize (*C* の関数), 125
 PyUnicode_DecodeLatin1 (*C* の関数), 133
 PyUnicode_DecodeLocale (*C* の関数), 124
 PyUnicode_DecodeLocaleAndSize (*C* の関数), 123
 PyUnicode_DecodeMBCS (*C* の関数), 135
 PyUnicode_DecodeMBCSStateful (*C* の関数), 135
 PyUnicode_DecodeRawUnicodeEscape (*C* の関数), 132
 PyUnicode_DecodeUnicodeEscape (*C* の関数), 132
 PyUnicode_DecodeUTF7 (*C* の関数), 131
 PyUnicode_DecodeUTF7Stateful (*C* の関数), 131
 PyUnicode_DecodeUTF8 (*C* の関数), 128
 PyUnicode_DecodeUTF8Stateful (*C* の関数), 128
 PyUnicode_DecodeUTF16 (*C* の関数), 130
 PyUnicode_DecodeUTF16Stateful (*C* の関数), 130
 PyUnicode_DecodeUTF32 (*C* の関数), 129
 PyUnicode_DecodeUTF32Stateful (*C* の関数), 129
 PyUnicode_Encode (*C* の関数), 127
 PyUnicode_EncodeASCII (*C* の関数), 133
 PyUnicode_EncodeCharmap (*C* の関数), 134
 PyUnicode_EncodeCodePage (*C* の関数), 135
 PyUnicode_EncodeFSDefault (*C* の関数), 125
 PyUnicode_EncodeLatin1 (*C* の関数), 133
 PyUnicode_EncodeLocale (*C* の関数), 124
 PyUnicode_EncodeMBCS (*C* の関数), 135
 PyUnicode_EncodeRawUnicodeEscape (*C* の関数), 132
 PyUnicode_EncodeUnicodeEscape (*C* の関数), 132
 PyUnicode_EncodeUTF7 (*C* の関数), 131
 PyUnicode_EncodeUTF8 (*C* の関数), 128
 PyUnicode_EncodeUTF16 (*C* の関数), 130
 PyUnicode_EncodeUTF32 (*C* の関数), 129
 PyUnicode_Fill (*C* の関数), 120
 PyUnicode_Find (*C* の関数), 136
 PyUnicode_FindChar (*C* の関数), 136
 PyUnicode_Format (*C* の関数), 137
 PyUnicode_FromEncodedObject (*C* の関数), 120
 PyUnicode_FromFormat (*C* の関数), 119
 PyUnicode_FromFormatV (*C* の関数), 120
 PyUnicode_FromKindAndData (*C* の関数), 118
 PyUnicode_FromObject (*C* の関数), 123
 PyUnicode_FromString (*C* の関数), 119
 PyUnicode_FromString(), 144
 PyUnicode_FromStringAndSize (*C* の関数), 118
 PyUnicode_FromUnicode (*C* の関数), 122
 PyUnicode_FromWideChar (*C* の関数), 126
 PyUnicode_FSConverter (*C* の関数), 124
 PyUnicode_FSDecoder (*C* の関数), 125
 PyUnicode_GET_DATA_SIZE (*C* の関数), 116
 PyUnicode_GET_LENGTH (*C* の関数), 114
 PyUnicode_GET_SIZE (*C* の関数), 116
 PyUnicode_GetLength (*C* の関数), 120
 PyUnicode_GetSize (*C* の関数), 123
 PyUnicode_InternFromString (*C* の関数), 138
 PyUnicode_InternInPlace (*C* の関数), 138
 PyUnicode_Join (*C* の関数), 136
 PyUnicode_KIND (*C* の関数), 115
 PyUnicode_MAX_CHAR_VALUE (*C* のマクロ), 115
 PyUnicode_New (*C* の関数), 118
 PyUnicode_READ (*C* の関数), 115
 PyUnicode_READ_CHAR (*C* の関数), 115
 PyUnicode_ReadChar (*C* の関数), 121
 PyUnicode_READY (*C* の関数), 114
 PyUnicode_Replace (*C* の関数), 137
 PyUnicode_RichCompare (*C* の関数), 137
 PyUnicode_Split (*C* の関数), 136
 PyUnicode_Splitlines (*C* の関数), 136
 PyUnicode_Substring (*C* の関数), 121
 PyUnicode_Tailmatch (*C* の関数), 136
 PyUnicode_TransformDecimalToASCII (*C* の関数), 122
 PyUnicode_Translate (*C* の関数), 134
 PyUnicode_TranslateCharmap (*C* の関数), 135
 PyUnicode_Type (*C* の変数), 114
 PyUnicode_WCHAR_KIND (*C* のマクロ), 114
 PyUnicode_WRITE (*C* の関数), 115
 PyUnicode_WriteChar (*C* の関数), 121
 PyUnicodeDecodeError_Create (*C* の関数), 36
 PyUnicodeDecodeError_GetEncoding (*C* の関数), 37
 PyUnicodeDecodeError_GetEnd (*C* の関数), 37
 PyUnicodeDecodeError_GetObject (*C* の関数), 37
 PyUnicodeDecodeError_GetReason (*C* の関数), 37
 PyUnicodeDecodeError_GetStart (*C* の関数), 37
 PyUnicodeDecodeError_SetEnd (*C* の関数), 37
 PyUnicodeDecodeError_SetReason (*C* の関数), 38
 PyUnicodeDecodeError_SetStart (*C* の関数), 37
 PyUnicodeEncodeError_Create (*C* の関数), 36
 PyUnicodeEncodeError_GetEncoding (*C* の関数), 37
 PyUnicodeEncodeError_GetEnd (*C* の関数), 37
 PyUnicodeEncodeError_GetObject (*C* の関数), 37
 PyUnicodeEncodeError_GetReason (*C* の関数), 37
 PyUnicodeEncodeError_GetStart (*C* の関数), 37

PyUnicodeEncodeError_SetEnd (*C* の関数), 37
 PyUnicodeEncodeError_SetReason (*C* の関数), 38
 PyUnicodeEncodeError_SetStart (*C* の関数), 37
 PyUnicodeObject (*C* のデータ型), 113
 PyUnicodeTranslateError_Create (*C* の関数), 37
 PyUnicodeTranslateError_GetEnd (*C* の関数), 37
 PyUnicodeTranslateError_GetObject (*C* の関数), 37
 PyUnicodeTranslateError_GetReason (*C* の関数), 37
 PyUnicodeTranslateError_GetStart (*C* の関数), 37
 PyUnicodeTranslateError_SetEnd (*C* の関数), 37
 PyUnicodeTranslateError_SetReason (*C* の関数), 38
 PyUnicodeTranslateError_SetStart (*C* の関数), 37
 PyVarObject (*C* のデータ型), 236
 PyVarObject_HEAD_INIT (*C* のマクロ), 237
 PyVarObject.ob_size (*C* のメンバ変数), 250
 PyVectorcall_Call (*C* の関数), 253
 PyVectorcall_NARGS (*C* の関数), 76
 PyWeakref_Check (*C* の関数), 167
 PyWeakref_CheckProxy (*C* の関数), 167
 PyWeakref_CheckRef (*C* の関数), 167
 PyWeakref_GET_OBJECT (*C* の関数), 168
 PyWeakref_GetObject (*C* の関数), 168
 PyWeakref_NewProxy (*C* の関数), 167
 PyWeakref_NewRef (*C* の関数), 167
 PyWideStringList (*C* のデータ型), 206
 PyWideStringList_Append (*C* の関数), 206
 PyWideStringList_Insert (*C* の関数), 206
 PyWideStringList.items (*C* のメンバ変数), 207
 PyWideStringList.length (*C* のメンバ変数), 207
 PyWrapper_New (*C* の関数), 164

Q

qualified name, 304

R

realloc(), 225
 reference count, 305
 regular package, 305
 releasebufferproc (*C* のデータ型), 281
 repr
 組み込み関数, 73, 255
 reprfunc (*C* のデータ型), 281
 richcmpfunc (*C* のデータ型), 281

S

sdterr
 stdin stdout, 184
 search
 path, module, 13, 183, 186
 sequence, 305
 オブジェクト, 108
 set
 オブジェクト, 147
 set comprehension, 305
 set_all(), 9
 setattrfunc (*C* のデータ型), 281
 setattrfunc (*C* のデータ型), 281
 setswitchinterval() (*in module sys*), 188
 SIGINT, 35
 signal
 モジュール, 35
 single dispatch, 305
 SIZE_MAX, 104
 slice, 305
 special
 method, 305
 special method, 305
 ssizeargfunc (*C* のデータ型), 281
 ssizeobjargproc (*C* のデータ型), 281
 statement, 305
 staticmethod

組み込み関数, 239
 stderr (*in module sys*), 196
 stdin
 stdout sdterr, 184
 stdin (*in module sys*), 196
 stdout
 sdterr, stdin, 184
 stdout (*in module sys*), 196
 strerror(), 29
 string
 PyObject_Str (*C function*), 73
 sum_list(), 10
 sum_sequence(), 10, 12
 sys
 モジュール, 13, 183, 196
 SystemError (*built-in exception*), 156

T

ternaryfunc (*C* のデータ型), 281
 text encoding, 305
 text file, 305
 traverseproc (*C* のデータ型), 286
 triple-quoted string, 306
 tuple
 オブジェクト, 138
 組み込み関数, 84, 143
 type, 306
 オブジェクト, 6, 97
 組み込み関数, 77
 type alias, 306
 type hint, 306

U

ULONG_MAX, 104
 unaryfunc (*C* のデータ型), 281
 universal newlines, 306

V

variable annotation, 307
 vectorcallfunc (*C* のデータ型), 280
 version (*in module sys*), 186, 187
 virtual environment, 307
 virtual machine, 307
 visitproc (*C* のデータ型), 286

Z

Zen of Python, 307