
The Python/C API

リリース 2.7.17

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

12 月 24, 2019

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

目次

第 1 章	はじめに	3
1.1	インクルードファイル	3
1.2	オブジェクト、型および参照カウント	4
1.3	例外	8
1.4	Python の埋め込み	11
1.5	デバッグ版ビルド (Debugging Builds)	12
第 2 章	超高レベルレイヤ	13
第 3 章	参照カウント	19
第 4 章	例外処理	21
4.1	Unicode 例外オブジェクト	27
4.2	再帰の管理	28
4.3	標準例外	28
4.4	標準警告カテゴリ	30
4.5	文字列の例外	30
第 5 章	ユーティリティ	31
5.1	オペレーティングシステム関連のユーティリティ	31
5.2	システム関数	32
5.3	プロセス制御	33
5.4	モジュールのインポート	33
5.5	データ整列化 (data marshalling) のサポート	37
5.6	引数の解釈と値の構築	38
5.7	文字列の変換と書式化	47
5.8	リフレクション	49
5.9	codec レジストリとサポート関数	50
第 6 章	抽象オブジェクトレイヤ (abstract objects layer)	53
6.1	オブジェクトプロトコル (object protocol)	53
6.2	数値型プロトコル (number protocol)	58
6.3	シーケンス型プロトコル (sequence protocol)	63
6.4	マップ型プロトコル (mapping protocol)	66
6.5	イテレータプロトコル (iterator protocol)	67
6.6	古いバッファプロトコル	68

第 7 章	具象オブジェクト (concrete object) レイヤ	71
7.1	基本オブジェクト (fundamental object)	71
7.2	数値型オブジェクト (numeric object)	73
7.3	シーケンスオブジェクト (sequence object)	82
7.4	マップ型オブジェクト (mapping object)	115
7.5	その他のオブジェクト	118
第 8 章	初期化 (initialization)、終了処理 (finalization)、スレッド	141
8.1	インタプリタの初期化と終了処理	141
8.2	プロセスワイドのパラメータ	142
8.3	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)	145
8.4	サブインタプリタサポート	152
8.5	非同期通知	154
8.6	プロファイルとトレース (profiling and tracing)	154
8.7	高度なデバッガサポート (advanced debugger support)	156
第 9 章	メモリ管理	159
9.1	概要	159
9.2	メモリインタフェース	160
9.3	Object allocators	161
9.4	pymalloc アロケータ	162
9.5	例	162
第 10 章	オブジェクト実装サポート (object implementation support)	165
10.1	オブジェクトをヒープ上にメモリ確保する	165
10.2	共通のオブジェクト構造体 (common object structure)	167
10.3	型オブジェクト	172
10.4	数値オブジェクト構造体	191
10.5	マップ型オブジェクト構造体	193
10.6	シーケンスオブジェクト構造体	193
10.7	バッファオブジェクト構造体 (buffer object structure)	194
10.8	循環参照ガベージコレクションをサポートする	196
付録 A	用語集	199
付録 B	このドキュメントについて	211
B.1	Python ドキュメント 貢献者	211
付録 C	歴史とライセンス	213
C.1	Python の歴史	213
C.2	Terms and conditions for accessing or otherwise using Python	214
C.3	取り入れられているソフトウェアのライセンスと謝辞	218
付録 D	Copyright	231
索引		233

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

第 1 章

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 (*extention 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 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:

```
#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: システムによっては、Python の定義しているプリプロセッサ定義が標準ヘッダに影響をおよぼす可能性があるので、Python.h は必ず他の標準ヘッダファイルよりも前にインクルードしてください。

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、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。

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

ヘッダファイル群は通常 Python と共にインストールされます。Unix では `prefix/include/pythonversion/` および `exec_prefix/include/pythonversion/` に置かれます。`prefix` と `exec_prefix` は Python をビルドする際の `configure` スクリプトに与えたパラメタに対応し、`version` は `sys.version[:3]` 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。

API はすべて C 言語を使って定義していますが、ヘッダファイルはエントリポイントを `extern "C"` で適切に宣言しているので、C++ ユーザは、なんの問題もなく C++ から API を利用できることに気づくはずで

1.2 オブジェクト、型および参照カウント

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

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

1.2.1 参照カウント

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()` でオブジェクトの参照を 1 インクリメントし、`Py_DECREF()` で 1 デクリメントします。`Py_DECREF()` マクロは、`Py_INCREF()` 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デアロケータ (deallocator) を呼び出さなければならないからです。デアロケータとは、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。型固有のデアロケータは、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、 $(\text{sizeof}(\text{Py_ssize_t}) \geq \text{sizeof}(\text{void}^*))$ と仮定した場合) 少なくとも参照カウントの記憶に使わ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みえる操作が、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()` を呼び出す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 `decreef` する役割を担います — 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ことです。

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

```
PyObject *t;

t = PyTuple_New(3);
PyTuple_SetItem(t, 0, PyInt_FromLong(1L));
PyTuple_SetItem(t, 1, PyInt_FromLong(2L));
PyTuple_SetItem(t, 2, PyString_FromString("three"));
```

ここで、`PyInt_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)
{
    int i, n;

    n = PyObject_Length(target);
    if (n < 0)
        return -1;
    for (i = 0; i < n; i++) {
        PyObject *index = PyInt_FromLong(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)
{
    int i, n;
    long total = 0;
    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);
        total += PyLong_AsLong(item);
    }
    return total;
}
```

(次のページに続く)

(前のページからの続き)

```

    for (i = 0; i < n; i++) {
        item = PyList_GetItem(list, i); /* Can't fail */
        if (!PyInt_Check(item)) continue; /* Skip non-integers */
        total += PyInt_AsLong(item);
    }
    return total;
}

```

```

long
sum_sequence(PyObject *sequence)
{
    int i, n;
    long total = 0;
    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 (PyInt_Check(item))
            total += PyInt_AsLong(item);
        Py_DECREF(item); /* Discard reference ownership */
    }
    return total;
}

```

1.2.2 型

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

1.3 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、ある関数は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 オブジェクトの `sys.exc_type`, `sys.exc_value`, および `sys.exc_traceback` 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_type` その他に転送するよう取り計らう Python のバイトコードインタプリタのメインループに到達するまで、例外が関数の間で受け渡しされている間だけです。

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

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

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 = PyInt_FromLong(0L);
        if (item == NULL)
            goto error;
    }
    const_one = PyInt_FromLong(1L);
    if (const_one == NULL)
        goto error;

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

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

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

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

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

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。

1.4 Python の埋め込み

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

基本的な初期化処理を行う関数は `Py_Initialize()` です。この関数はロード済みのモジュールからなるテーブルを作成し、土台となるモジュール `_builtin_`, `_main_`, `sys`, `exceptions` 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` 上にな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 “初期化しない” ように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できるようにしたいかもしれません。`Py_Finalize()` 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_Finalize()` 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。

1.5 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。参照カウンタのトレース、メモリアロケータのデバッグ、インタプリタのメインループの低レベルプロファイリングが利用可能です。よく使われるビルドについてのみ、この節の残りの部分で説明します。

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、長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。

第 2 章

超高レベルレイヤ

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。

これらの関数のいくつかが `FILE*` 引数をとることに注意してください。注意深く扱う必要がある特別な問題の 1 つは、異なる C ライブラリの `FILE` 構造体は異なっていて互換性がない可能性があるということです。実際に (少なくとも) Windows では、動的リンクされる拡張が異なるライブラリを使うことが可能であり、したがって、`FILE*` 引数が Python ランタイムが使っているライブラリと同じライブラリによって作成されたことが確かならば、単にこれらの関数へ渡すだけということに注意すべきです。

`int Py_Main (int argc, char** 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` することに気がつけてください。

`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ならば、`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* 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*)

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

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

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

`PyInspectFlag` 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_SimpleFileFlags** (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_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りません。*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。

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を一つ読み込み、*flags* に従って実行します。`sys.ps1` と `sys.ps2` を使って、ユーザにプロンプトを表示します。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。したがって、必要ならば特別に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 に達するまで複数の文を読み込み実行します。使われます。 `sys.ps1` と `sys.ps2` を使って、ユーザにプロンプトを表示します。 EOF に達すると 0 を返します。

```
struct _node* PyParser_SimpleParseString (const char*str, intstart)
```

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, intstart, intflags)
```

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, intstart, intflags)
```

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。

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

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, intstart, intflags)
```

`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, intstart, 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, intstart, PyObject*globals, PyObject*locals, PyCompilerFlags*flags)
```

Return value: New reference. 辞書 `globals` と `locals` で指定されるコンテキストにおいて、`str` に含まれる Python ソースコードをコンパイラフラグ `flags` のもとで実行します。パラメータ `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, intstart, PyObject*globals, PyObject*locals)
```

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, intstart, PyObject*globals, PyObject*locals, intcloseit)
```

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, intstart, PyObject*globals, PyObject*locals, 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***locals*, 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りません。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** **PyCompileString** (const char**str*, const char**filename*, int*start*)

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

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

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。

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

Return value: New reference. *PyEval_EvalCodeEx()* に対する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の辞書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** (*PyCodeObject***co*, *PyObject***globals*, *PyObject***locals*, *PyObject****args*, int*argcount*, *PyObject****kws*, int*kwcount*, *PyObject****defs*, int*defcount*, *PyObject***closure*)

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。環境はグローバルとローカルの辞書、引き数の配列、キーワードとデフォルト値、クロージャーのためのセルのタプルで構成されています。

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

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*)

Python のインタープリタの主要な、直接的な関数です。この関数には 2000 行ほどあります。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。

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、 *PyCompileString()* 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。

```
struct PyCompilerFlags {  
    int cf_flags;  
}
```

int `CO_FUTURE_DIVISION`

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

第 3 章

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変数の参照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。

バージョン 2.4 で追加.

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。

第 4 章

例外処理

この章で説明する関数を使うと、Python の例外の処理や送出力ができるようになります。Python の例外処理の基本をいくつ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すが、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)。

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。

エラーインジケータは Python 変数 `sys.exc_type`, `sys.exc_value` および `sys.exc_traceback` に対応する三つの Python オブジェクトからなります。いろいろな方法でエラーインジケータとやりとりするために API 関数が存在します。各スレッドに別々のエラーインジケータがあります。

`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。

*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、これを呼び出すべきです。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)`

`given` 例外が `exc` の例外と一致するなら真を返します。これは `exc` がクラスオブジェクトである場合も真を返します。これは `given` がサブクラスのインスタンスであるときも真を返します。`exc` がタプルならば、タプル内 (と再帰的にサブタプル内) のすべての例外が一致するか調べられます。

`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。

`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_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、この関数は例外を扱う必要のあるコードあるいはエラーインジケータを一時的に保存して元に戻す必要のあるコードによってのみ使用されます。

`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元に戻す必要のあるコードによってのみに使われます。現在の例外状態を保存するためには `PyErr_Fetch()` を使ってください。

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、文字列オブジェクトへ変換されます。

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、 `PyString_FromFormat()` の引数と同じ意味を持っています。

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**
*PyObject***filenameObject*)

PyErr_SetFromErrno() に似ていますが、*filenameObject* が *NULL* でない場合にそれを *type* のコンストラクタに第三引数として渡す、というふるまいが追加されています。IOError と OSError のような例外の場合では、これが例外インスタンスの *filename* 属性を定義するために使われます。

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

Return value: Always *NULL*. *PyErr_SetFromErrnoWithFilenameObject*() と似ていますが、ファイル名は C 文字列として与えられます。

*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。利用可能範囲: 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。利用可能範囲: Windows。

バージョン 2.3 で追加。

*PyObject** **PyErr_SetFromWindowsErrWithFilenameObject** (int*ierr*, *PyObject**
*PyObject***filenameObject*)

PyErr_SetFromWindowsErr() に似ていますが、*filenameObject* が *NULL* でない場合には *WindowsError* のコンストラクタに第三引数として渡されるというふるまいが追加されています。利用可能範囲: Windows。

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

Return value: Always *NULL*. *PyErr_SetFromWindowsErrWithFilenameObject*() と似ていますが、ファイル名は C 文字列として与えられます。利用可能範囲: Windows。

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

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。利用可能範囲: Windows。

バージョン 2.3 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。利用可能範囲: Windows。

バージョン 2.3 で追加。

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。

int `PyErr_WarnEx` (*PyObject**category, char*message, intstacklevel)

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 引数はメッセージ文字列です。stacklevel はスタックフレームの数を示す正の整数です; 警告はそのスタックフレームの中の実行している行から発行されます。stacklevel が 1 だと `PyErr_WarnEx()` を呼び出している関数が、2 だとその上の関数が Warning の発行元になります。

この関数は通常警告メッセージを `sys.stderr` へプリントします。けれども、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可能です。そのような場合には、これは例外を発生させます。警告機構がもつ問題のためにその関数が例外を発生させるということも可能です。(実装ではその厄介な仕事を行うために `warnings` モジュールをインポートします)。例外が発生させられなければ、戻り値は 0 です。あるいは、例外が発生させられると -1 です。(警告メッセージが実際にプリントされるかどうかを決定することはできず、また何がその例外の原因なのかを決定することもできない。これは意図的なものです。) 例外が発生した場合、呼び出し元は通常の例外処理を行います (例えば、`Py_DECREF()` は参照を持っており、エラー値を返します)。

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

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

int `PyErr_Warn` (*PyObject**category, char*message)

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 引数はメッセージ文字列です。警告は、`PyErr_WarnEx()` を stacklevel に 1 を指定した時と同じく、`PyErr_Warn()` を呼び出した関数から発行されます。

非推奨; `PyErr_WarnEx()` を使って下さい。

int `PyErr_WarnExplicit` (*PyObject**category, const char*message, const char*filename, intlineno, const char*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。

int `PyErr_WarnPy3k` (char*message, intstacklevel)

`PyPy3kWarningFlag` フラグが有効な場合、与えられた message と stacklevel に応じて `DeprecationWarning` を発生します。

バージョン 2.6 で追加。

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かわからない。

void **PyErr_SetInterrupt** ()

この関数は廃止されています。 `SIGINT` シグナルが到達した影響をシミュレートします — 次に `PyErr_CheckSignals()` が呼ばれるとき、 `KeyboardInterrupt` は送出されるでしょう。インタプリタロックを保持することなく呼び出すことができます。

int **PySignal_SetWakeupFd** (intfd)

このユーティリティ関数は、シグナルを受信したときに `'\0'` バイトを書き込むファイルディスクリプタを指定します。戻り値は、それまで設定されていたファイルディスクリプタです。 `-1` はこの機能を無効にします。これは初期状態です。これは Python の `signal.set_wakeup_fd()` と同じものですが、エラーチェックを行いません。 `fd` は有効なファイルディスクリプタであるべきです。この関数の呼び出しはメインスレッドのみから行われるべきです。

バージョン 2.6 で追加。

*PyObject** **PyErr_NewException** (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` 引数は代わりに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** (char*name, 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 2.7 で追加。

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

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` ヘブプリントします。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。

4.1 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*tend*,
const char**reason*)
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。

*PyObject** **PyUnicodeEncodeError_Create** (const char**encoding*, const *Py_UNICODE***object*,
Py_ssize_t*length*, Py_ssize_t*start*, Py_ssize_t*tend*, const
char**reason*)
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。

*PyObject** **PyUnicodeTranslateError_Create** (const *Py_UNICODE***object*, Py_ssize_t*length*,
Py_ssize_t*start*, Py_ssize_t*tend*, const
char**reason*)
object, *length*, *start*, *end*, *reason* 属性を持った UnicodeTranslateError オブジェクトを作成し
ます。

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

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

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*)

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_tend)
```

```
int PyUnicodeEncodeError_SetEnd (PyObject*exc, Py_ssize_tend)
```

```
int PyUnicodeTranslateError_SetEnd (PyObject*exc, Py_ssize_tend)
```

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)
```

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。

4.2 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、`RuntimeError` 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。

再帰の深さの上限に達して送出される `RuntimeError` のメッセージに連結できるよう *where* は "in instance check" のような文字列にしてください。

```
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。

4.3 標準例外

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

C 名	Python 名	注釈
PyExc.BaseException	BaseException	(1), (4)
PyExc.Exception	Exception	(1)
PyExc.StandardError	StandardError	(1)
PyExc.ArithmeticError	ArithmeticError	(1)
PyExc.AssertionError	AssertionError	
PyExc.AttributeError	AttributeError	
PyExc.BufferError	BufferError	
PyExc.EnvironmentError	EnvironmentError	(1)
PyExc.EOFError	EOFError	
PyExc.FloatingPointError	FloatingPointError	
PyExc.GeneratorExit	GeneratorExit	
PyExc.ImportError	ImportError	
PyExc.IndentationError	IndentationError	
PyExc.IndexError	IndexError	
PyExc.IOError	IOError	
PyExc.KeyError	KeyError	
PyExc.KeyboardInterrupt	KeyboardInterrupt	
PyExc.LookupError	LookupError	(1)
PyExc.MemoryError	MemoryError	
PyExc.NameError	NameError	
PyExc.NotImplementedError	NotImplementedError	
PyExc.OSError	OSError	
PyExc.OverflowError	OverflowError	
PyExc.ReferenceError	ReferenceError	(2)
PyExc.RuntimeError	RuntimeError	
PyExc.StopIteration	StopIteration	
PyExc.SyntaxError	SyntaxError	
PyExc.SystemError	SystemError	
PyExc.SystemExit	SystemExit	
PyExc.TabError	TabError	
PyExc.TypeError	TypeError	
PyExc.UnboundLocalError	UnboundLocalError	
PyExc.UnicodeDecodeError	UnicodeDecodeError	
PyExc.UnicodeEncodeError	UnicodeEncodeError	
PyExc.UnicodeError	UnicodeError	
PyExc.UnicodeTranslateError	UnicodeTranslateError	
PyExc.VMSError	VMSError	(5)
PyExc.ValueError	ValueError	
PyExc.WindowsError	WindowsError	(3)
PyExc.ZeroDivisionError	ZeroDivisionError	

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) これは `weakref.ReferenceError` と同じです。
- (3) 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。
- (4) バージョン 2.5 で追加。
- (5) Only defined on VMS; protect code that uses this by testing that the preprocessor macro `__VMS` is defined.

4.4 標準警告カテゴリ

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

C 名	Python 名	注釈
<code>PyExc_Warning</code>	<code>Warning</code>	(1)
<code>PyExc_BytesWarning</code>	<code>BytesWarning</code>	
<code>PyExc_DeprecationWarning</code>	<code>DeprecationWarning</code>	
<code>PyExc_FutureWarning</code>	<code>FutureWarning</code>	
<code>PyExc_ImportWarning</code>	<code>ImportWarning</code>	
<code>PyExc_PendingDeprecationWarning</code>	<code>PendingDeprecationWarning</code>	
<code>PyExc_RuntimeWarning</code>	<code>RuntimeWarning</code>	
<code>PyExc_SyntaxWarning</code>	<code>SyntaxWarning</code>	
<code>PyExc_UnicodeWarning</code>	<code>UnicodeWarning</code>	
<code>PyExc_UserWarning</code>	<code>UserWarning</code>	

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。

4.5 文字列の例外

バージョン 2.6 で変更: 発生したりキャッチされる全ての例外は `BaseException` を継承しなければなりません。文字列例外を発生させようとすると `TypeError` が発生されます。

第 5 章

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。

5.1 オペレーティングシステム関連のユーティリティ

`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_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。

`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 (inti)`

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 (inti, 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 type-

def による別名です。

5.2 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** (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。

FILE ***PySys_GetFile** (char*name, FILE*def)

`sys` モジュール内のオブジェクト `name` に関連付けられた `FILE*` を返します。 `name` がなかった場合や `FILE*` に関連付けられていなかった場合は `def` を返します。

int **PySys_SetObject** (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** ()

`sys.warnoptions` を、空リストにリセットします。

void **PySys_AddWarnOption** (char*s)

`sys.warnoptions` に `s` を追加します。

void **PySys_SetPath** (char*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, ...)

上と同じですが、 `sys.stderr` が `stderr` に出力します。

5.3 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 (intstatus)`

現在のプロセスを終了 (exit) します。この関数は `Py_Finalize()` を呼び出し、次いで標準 C ライブラリ関数 `exit(status)` を呼び出します。

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

`Py_Finalize()` 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。

5.4 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。) `import` 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。Python 2.4 より前のバージョンでは、失敗した場合でもモジュールは生成されていることがあります — `sys.modules` を使って調べてください。Python 2.4 以降では、`import` に失敗したモジュールは `sys.modules` に残りません。

バージョン 2.4 で変更: `import` に失敗した場合、不完全なモジュールを除去するようになりました。

バージョン 2.6 で変更: 常に、絶対 `import` を使うようになりました。

`PyObject* PyImport_ImportModuleNoBlock (const char*name)`

このバージョンの `PyImport_ImportModule()` はブロックしません。関数を実行するために他のモジュールをインポートする C 関数から使われることを意図しています。インポート処理は他のスレッドがインポートロックを持っている場合はブロックします。この関数はブロックしません。まず `sys.modules` からモジュールのフェッチを試み、失敗したら、ロックが取られていなければ `PyImport_ImportModule()` を実行します。ロックが取られていた場合は `ImportError` を発生

させます。

バージョン 2.6 で追加。

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

Return value: New reference. モジュールを import します。モジュールの import 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 import されたモジュールが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 (Python 2.4 よりも前のバージョンでは、モジュールは生成されている場合があります) `__import__()` と同じく、パッケージに対してサブモジュールを要求した場合の戻り値は通常、空でない *fromlist* を指定しない限りトップレベルパッケージになります。

バージョン 2.4 で変更: import に失敗した場合、不完全なモジュールを除去するようになりました。

バージョン 2.6 で変更: この関数は *PyImport_ImportModuleLevel()* のエイリアスです。level には相対インポートを意味する `-1` が渡されます。

*PyObject** **PyImport_ImportModuleLevel** (char*name, *PyObject**globals, *PyObject**locals, *PyObject**fromlist, intlevel)

Return value: New reference. モジュールを import します。モジュールの import 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が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 2.5 で追加。

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

Return value: New reference. 現在の "import 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。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 import フック、例えば `rexec` や `ihooks` を使って import を行います。

バージョン 2.6 で変更: 常に、絶対 import を使うようになりました。

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

Return value: New reference. モジュールを再ロード (reload) します。モジュールの再ロードについては組み込みの Python 関数 `reload()` を読むとよく分かります。というのも、標準の `reload()` はこの関数を直接呼び出しているからです。戻り値は再ロードしたモジュールが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_AddModule** (const char*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 `import` 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いければ、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 `import` 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。

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

Return value: New reference. モジュール名 (package.module 形式でもかまいません) 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。Python 2.4 以前では、失敗した場合でもモジュールは生成されていることがありました。Python 2.4 以降では、たとえば `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、`import` 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 `import` 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 `__file__` 属性が、コードオブジェクトの `co_filename` に設定されます。

この関数は、すでに `import` 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。

バージョン 2.4 で変更: エラーが発生した場合に `name` を `sys.modules` から除去するようになりました。

*PyObject** **PyImport_ExecCodeModuleEx** (char*name, *PyObject**co, 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。

long **PyImport_GetMagicNumber** ()

Python バイトコードファイル (いわゆる .pyc および .pyo ファイル) のマジックナンバを返します。マジックナンバはバイトコードファイルの先頭 4 バイトにリトルエンディアン整列で配置されています。

*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_GetImporter** (*PyObject**path)

`sys.path/pkg.__path__` の要素 `path` の、`importer` オブジェクトを返します。可能なら、`sys.path_importer_cache` からフェッチします。まだキャッシュされていない場合、そのパスを扱える `hook` が見つかるまで `sys.path.hooks` を巡回します。hook が見つからない場合、`None` を

返します。この場合、呼び出し元はビルトインのインポート機構にフォールバックします。結果は `sys.path_importer_cache` にキャッシュされます。 `importer` オブジェクトへの新しい参照を返します。

バージョン 2.6 で追加。

`void PyImport_Init()`

`import` 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()`

`import` 機構を終了処理します。内部使用だけのための関数です。

*PyObject** `PyImport_FindExtension(char*, char*)`

内部使用だけのための関数です。

*PyObject** `PyImport_FixupExtension(char*, char*)`

内部使用だけのための関数です。

`int PyImport_ImportFrozenModule(char*name)`

`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 この関数名はいささか誤称めいています — この関数はすでに `import` 済みのモジュールをリロードしてしまいます。)

`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、以下のようにになっています:

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

`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 `import` 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, void (*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 `import` でき、最初に `import` 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 {
    char *name;
    void (*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。

5.5 データ整列化 (data marshallng) 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、(Python 2.4 で新たに追加された) 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、逆マーシャルの時にも共有されるようにします。(Python 2.5 で新たに追加された) 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。

バージョン 2.4 で変更: ファイル形式を示す `version` が追加されました。

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。

バージョン 2.4 で変更: ファイル形式を示す `version` が追加されました。

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

Return `value`: New reference. `value` の整列化表現が入った文字列オブジェクトを返します。

バージョン 2.4 で変更: ファイル形式を示す *version* が追加されました。

以下の関数を使うと、整列化された値を読み戻せます。

XXX What about error detection? It appears that reading past the end of the file will always result in a negative numeric value (where that's relevant), but it's not clear that negative values won't be handled properly when there's no error. What's the right way to tell? Should only non-negative values be written using these routines?

long PyMarshal_ReadLongFromFile (FILE*file)

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

int PyMarshal_ReadShortFromFile (FILE*file)

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

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

Return value: New reference. 読み出し用に開かれた FILE* 内のデータストリームから、Python オブジェクトを読み出して返します。エラーが生じた場合、適切な例外 (EOFError または TypeError) を送出して NULL を返します。

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

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

*PyObject** **PyMarshal_ReadObjectFromString (char*string, Py_ssize_tlen)**

Return value: New reference. string が指している len バイトの文字列バッファに納められたデータストリームから Python オブジェクトを読み出して返します。エラーが生じた場合、適切な例外 (EOFError または TypeError) を送出して NULL を返します。

バージョン 2.5 で変更: この関数は以前は len の型に int を利用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

5.6 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() はいずれも書式化文字列 (format string) を使います。書式化文字列は、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化文字列も、同じ書式を使っています。

書式化文字列は、ゼロ個またはそれ以上の ”書式化単位 (format unit)” から成り立ちます。1 つの書式化単位は 1 つの Python オブジェクトを表します; 通常は単一の文字が、書式化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化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化単位です; (丸) 括弧で囲った部分は書式化単位に対応する 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。

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。また、`es`, `es#`, `et`, `et#` フォーマットの場合を除いて自分でメモリを解放する必要もありません。

`s` (文字列型または Unicode 型) [`const char *`] Python の文字列または 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。変換先の文字列自体の記憶領域を提供する必要はありません; 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、NUL バイトが途中に埋め込まれていてはなりません; もし埋め込まれていれば `TypeError` 例外を送出します。Unicode オブジェクトはデフォルトエンコーディングを使って C 文字列に変換されます。変換に失敗すると `UnicodeError` を送じます。

`s#` (文字列型、Unicode 型または任意の読み出しバッファ互換型) [`const char *`, `int` (または `Py_ssize_t`, 下記参照)] これは `s` の変化形で、値を二つの変数に記録します。一つ目の変数はキャラクタ文字列へのポインタで、二つ目はその長さです。この書式化単位の場合には、Python 文字列に null バイトが埋め込まれていてもかまいません。Unicode オブジェクトの場合、デフォルトエンコーディングでの変換が可能ならば、変換したオブジェクトから文字列へのポインタを返します。その他の読み出しバッファ互換オブジェクトは生の内部データ表現への参照を返します。

Python 2.5 から、長さの引数の型を、`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できるようになりました。もしこのマクロが定義されていた場合、長さは `int` ではなく `Py_ssize_t` になります。

`s*` (文字列型、Unicode 型、または任意のバッファ互換オブジェクト) [`Py_buffer`] `s#` と似ていますが、呼び出し元から渡された `Py_buffer` 構造体書き込みます。バッファはロックされるので、呼び出し元はそのバッファを `Py_BEGIN_ALLOW_THREADS` したブロック内でさえも利用することができます。その代わりに、呼び出し元には、データを処理した後にその構造体に対して `PyBuffer_Release` する責任があります。

バージョン 2.6 で追加.

`z` (文字列型または Unicode 型または `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#` (文字列型、`None`、または任意の読み出しバッファ互換型) [`const char *`, `int`] `s#` の `s` を `z` にしたような意味です。

`z*` (文字列型、Unicode 型、`None` または任意のバッファ互換型) [`Py_buffer`] `s*` の `s` を `z` にしたような意味です。

バージョン 2.6 で追加.

`u` (Unicode 型) [`Py_UNICODE *`] Python の Unicode オブジェクトを、NUL で終端された 16 ビットの

Unicode (UTF-16) データに変換します。 `s` と同様に、Unicode データバッファ用に記憶領域を提供する必要はありません; `Py_UNICODE` 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 Unicode データへのポインタをその変数に記録します。

`u#` (Unicode 型) [`Py_UNICODE *`, `int`] これは `u` の変化形で、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。非 Unicode のオブジェクトの場合、読み出しバッファのポインタを `Py_UNICODE` 型シーケンスへのポインタと解釈して扱います。

`es` (文字列型、Unicode 型またはキャラクタバッファ互換型) [`const char *encoding`, `char **buffer`] これは `s` の変化形で、Unicode オブジェクトや Unicode に変換可能なオブジェクト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文字列でのみ動作します。

この書式化単位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` (文字列型、Unicode 型またはキャラクタバッファ互換型) [`const char *encoding`, `char **buffer`] `es` と同じです。ただし、8 ビット幅の文字列オブジェクトをエンコードし直さずに渡します。その代わりに、実装では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#` (文字列型、Unicode 型またはキャラクタバッファ互換型) [`const char *encoding`, `char **buffer`, `int *buffer_length`] `s#` の変化形で、Unicode オブジェクトや Unicode に変換可能なオブジェクト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` 書式化単位と違って、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。

この書式化単位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*` 型が `NULL` でなければなりません。 `NULL` の場合にはデフォルトエンコーディングを使います。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。

この書式化単位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 `TypeError` がセットされます。注意: Python 3.6 からは `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#` (文字列型、Unicode 型またはキャラクタバッファ互換型) [`const char *encoding, char **buffer, int *buffer_length`] `es#` と同じです。ただし、文字列オブジェクトをエンコードし直さずに渡します。その代わりに、実装では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。

`b` (整数型) [`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` (整数型) [`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。

バージョン 2.3 で追加。

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

`H` (整数型) [`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。

バージョン 2.3 で追加。

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

`I` (整数型) [`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。

バージョン 2.3 で追加。

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

`k` (整数型) [`unsigned long`] Python の整数もしくは長整数を、オーバーフローチェックを行わずに、C の `unsigned long int` 型に変換します。

バージョン 2.3 で追加。

`L` (整数型) [`PY_LONG_LONG`] Python の整数を、C の `long long` 型に変換します。この書式化単位は、`long long` 型 (または Windows の `_int64` 型) がサポートされているプラットフォームでのみ利用できます。

`K` (整数型) [`unsigned PY_LONG_LONG`] Python の整数もしくは長整数を、オーバーフローチェックを行わずに、C の `unsigned long long` 型に変換します。この書式化単位は、`unsigned long long` 型 (または Windows の `unsigned _int64` 型) がサポートされているプラットフォームでのみ利用できます。

バージョン 2.3 で追加。

`n` (整数型) [`Py_ssize_t`] Python の整数もしくは長整数を C の `Py_ssize_t` 型に変換します。

バージョン 2.5 で追加。

- c (長さ 1 の文字列型) [char] 長さ 1 の文字列として表現されている Python キャラクタを C の `char` 型に変換します。
- f (浮動小数点数型) [float] Python の浮動小数点型を、C の `float` 型に変換します。
- d (浮動小数点数型) [double] Python の浮動小数点型を、C の `double` 型に変換します。
- D (複素数型) [Py_complex] Python の複素数型を、C の `Py_complex` 構造体に変換します。
- o (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。
- o! (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。o に似ていますが、二つの C の引数をとります: 一つ目の引数は Python の型オブジェクトへのアドレスで、二つ目の引数はオブジェクトへのポインタが保存されている (`PyObject*` の) C の変数へのアドレスです。Python オブジェクトが指定した型ではない場合、`TypeError` を送出します。
- o& (object) [converter, 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。

- s (文字列型) [PyStringObject*] o に似ていますが、Python オブジェクトは文字列オブジェクトでなければなりません。オブジェクトが文字列オブジェクトでない場合には `TypeError` を送出します。C 変数は `PyObject*` で宣言しておいてもかまいません。
- U (Unicode 型) [PyUnicodeObject*] o に似ていますが、Python オブジェクトは Unicode オブジェクトでなければなりません。オブジェクトが Unicode オブジェクトでない場合には `TypeError` を送出します。C 変数は `PyObject*` で宣言しておいてもかまいません。
- t# (読み出し専用キャラクタバッファ) [char*, int] s# に似ていますが、読み出し専用バッファインタフェースを実装している任意のオブジェクトを受理します。`char*` 変数はバッファの最初のバイトを指すようにセットされ、`int` はバッファの長さにセットされます。単一セグメントからなるバッファオブジェクトだけを受理します; それ以外の場合には `TypeError` を送出します。
- w (読み書き可能なキャラクタバッファ) [char*] s と同様ですが、読み書き可能なバッファインタフェースを実装している任意のオブジェクトを受理します。呼び出し側は何らかの別の手段でバッファの長さを決定するか、あるいは `w#` を使わねばなりません。単一セグメントからなるバッファオブジェクトだけを受理します; それ以外の場合には `TypeError` を送出します。
- w# (読み書き可能なキャラクタバッファ) [char*, Py_ssize_t] s# に似ていますが、読み書き可能なバッファインタフェースを実装している任意のオブジェクトを受理します。`char*` 変数はバッファの最初のバイトを指すようにセットされ、`Py_ssize_t` はバッファの長さにセットされます。単一セグメントからなるバッファオブジェクトだけを受理します; それ以外の場合には `TypeError` を送出します。

`w*` (読み書き可能なバイト列バッファ) [`Py.buffer`] `s` に対する `s*` と同じ、`w` のバージョンです。

バージョン 2.6 で追加。

(`items`) (タプル型) [`matching-items`] オブジェクトは `items` に入っている書式化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` 内の個々の書式化単位に対応づけできなければなりません。シーケンスの書式化単位は入れ子構造にできます。

注釈: Python のバージョン 1.5.2 より以前は、この書式化指定文字列はパラメタ列ではなく、個別のパラメタが入ったタプルでなければなりませんでした。このため、以前は `TypeError` を引き起こしていたようなコードが現在は例外を出さずに処理されるかもしれません。とはいえ、既存のコードにとってこれは問題ないと思われます。

Python 整数型を要求している場所に Python 長整数型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、書式化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ねばなりません — 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。
- : この文字があると、書式化単位の記述はそこで終わります; コロン以降の文字列は、エラーメッセージにおける関数名 (`PyArg_ParseTuple()` が送出する例外の ”付属値 (associated value)”) として使われます。
- ; この文字があると、書式化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 ; は相互に排他的文字です。

呼び出し側に提供される Python オブジェクトの参照は全て 借りた (*borrowed*) ものです; 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、書式化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。上の書式化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、対応する書式化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化文字に一致しなければならず、かつ書式化文字列内の書式化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。書式化単位のどれかの変換失敗により `PyArg_Parse*`() が失敗した場合、失敗した書式化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。

```
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_listvargs)

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[], ...)

位置引数およびキーワード引数を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_VaParseTupleAndKeywords** (*PyObject**args, *PyObject**kw, const char*format,
char*keywords[], va_listvargs)

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_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 *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_tmin, Py_ssize_tmax, ...)

パラメタ取得を簡単にした形式で、引数の型を指定する書式化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)
```

バージョン 2.2 で追加.

バージョン 2.5 で変更: この関数は `min` と `max` に `int` を利用していました。この変更により、64 bit システムを正しくサポートするためには修正が必要になるでしょう。

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化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、書式化文字列に二つ以上の書式化単位が入っているときだけです。書式化文字列が空の場合、`None` を返します; 書式化単位が厳密に一つだけ入っている場合、書式化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 0 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化文字列を使います。

書式化単位 `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化単位です; (丸) 括弧で囲った部分は書式化単位が返す 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。

書式化文字列内では、(`s#` のような書式化単位を除いて) 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、長い書式化文字列をちょっとだけ読みやすくなります。

`s` (文字列型) [char *] null 終端された C 文字列から 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、`None` になります。

`s#` (文字列型) [char *, int] 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` (文字列型または `None`) [char *] `s` と同じです。

`z#` (文字列型または `None`) [char *, int] `s#` と同じです。

`u` (Unicode 文字列型) [Py_UNICODE *] null 終端された Unicode (UCS-2 または UCS-4) データのバッファから 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#` (Unicode 文字列型) [Py_UNICODE *, int] Unicode (UCS-2 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。

- i (整数型) [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 (整数型) [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 (整数型) [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 (整数型) [long int] C の `long int` を Python の整数オブジェクトに変換します。
- B (整数型) [unsigned char] C の `unsigned char` を Python の整数オブジェクトに変換します。
- H (整数型) [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 (整数型/長整数型) [unsigned int] C の `unsigned int` を Python の整数オブジェクト、あるいは、値が `sys.maxint` より大きければ長整数オブジェクトに変換します。
- k (整数型/長整数型) [unsigned long] C の `unsigned long` を Python の整数オブジェクト、あるいは、値が `sys.maxint` より大きければ長整数オブジェクトに変換します。
- L (長整数型) [PY_LONG_LONG] C の `long long` を Python の長整数オブジェクトに変換します。
`long long` をサポートしているプラットフォームでのみ利用可能です。
- K (長整数型) [unsigned PY_LONG_LONG] C の `unsigned long long` を Python の長整数オブジェクトに変換します。`long long` をサポートしているプラットフォームでのみ利用可能です。
- n (整数型) [Py_ssize_t] C の `Py_ssize_t` を Python の整数オブジェクト、あるいは長整数オブジェクトに変換します。

バージョン 2.5 で追加。
- c (長さ 1 の文字列型) [char] 文字を表す通常の C の `int` を、長さ 1 の Python の文字列オブジェクトに変換します。
- d (浮動小数点数型) [double] C の `double` を Python の浮動小数点数に変換します。
- f (浮動小数点数型) [float] d と同じです。
- D (複素数型) [Py_complex *] C の `Py_complex` 構造体を Python の複素数型に変換します。
- o (object) [PyObject *] 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 `Py_BuildValue()` 自体は例外を送出しません。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 *] o と同じです。
- N (object) [PyObject *] o 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。
- O& (object) [converter, anything] *anything* を *converter* 関数を介して Python オブジェクトに変換します。この関数は *anything* (`void *` と互換の型でなければなりません) を引数にして呼び出さ

れ、”新たな” オブジェクトを返すか、失敗した場合には *NULL* を返すようにしなければなりません。

(items) (タプル型) [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] (リスト型) [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} (辞書型) [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。

書式化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_listvargs)

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。

5.7 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_tsize, const char*format, ...)

書式化文字列 *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(2)* を参照してください。

int **PyOS_vsnprintf** (char*str, size_tsize, const char*format, va_listva)

書式化文字列 *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(2)* 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。

これらのラップ関数は、戻るときに *str*[*size-1] が常に '\0' であることを保証します。(str の末尾の '\0' を含めて) *size* バイト以上を書き込みません。両関数とも *str != NULL, size > 0, format != NULL* を要求します。

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 2.7 で追加.

double **PyOS_ascii_strtod** (const char*nptr, char**endptr)

文字列を double へ変換します。この関数は、C locale における C 標準の strtod() と同じように動作します。スレッドセーフのために、この関数は現在の locale を変更せずに実装されています。

PyOS_ascii_strtod() は通常、設定ファイルを読み込むときや、ロケール独立な非ユーザーからの入力を読み込むときに使われるべきです。

詳細は Unix man page の strtod(2) を参照してください。

バージョン 2.4 で追加.

バージョン 2.7 で非推奨: 代わりに *PyOS_string_to_double()* を使ってください。

char* **PyOS_ascii_formatd** (char*buffer, size_tbuf_len, const char*format, doubleval)

double を '.' を小数点記号に利用して文字列に変換します。format は数値のフォーマットを指定する printf() スタイルの文字列です。利用できる変換文字は 'e', 'E', 'f', 'F', 'g', 'G' です。

戻り値は、変換された文字列が格納された buffer へのポインタか、失敗した場合は NULL です。

バージョン 2.4 で追加.

バージョン 2.7 で非推奨: この関数は Python 2.7 と 3.1 では削除されました。代わりに *PyOS_double_to_string()* を使ってください。

char* **PyOS_double_to_string** (doubleval, charformat_code, intprecision, intflags, 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_DTST_SIGN`, `Py_DTST_ADD_DOT_0`, `Py_DTST_ALT` かこれらの or を取ったものです:

- `Py_DTST_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_DTST_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_DTST_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。

pctype 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 2.7 で追加.

double **PyOS_ascii_atof** (const char**nptr*)

文字列を、locale 非依存な方法で double へ変換します。

詳細は Unix man page の `atof(2)` を参照してください。

バージョン 2.4 で追加.

バージョン 3.1 で非推奨: 代わりに `PyOS_string_to_double()` を使ってください。

char* **PyOS_stricmp** (char*s1, 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。

バージョン 2.6 で追加.

char* **PyOS_strnicmp** (char*s1, 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。

バージョン 2.6 で追加.

5.8 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が、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が、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。

int **PyEval_GetRestricted** ()

現在のフレームがありそれが制限モードで実行していた場合、真を返します。それ以外の場合は偽を返します。

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。

5.9 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。

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

汎用の codec ベースの encode API.

encoding に応じて見つかったエンコード関数に対して 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。エンコード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)

汎用の codec ベースの decode API.

encoding に応じて見つかったデコード関数に対して 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。デコードが見つからなかった場合は *LookupError* を発生させます。

5.9.1 コーデック検索 API

以下の関数では、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*)

与えられた *encoding* のエンコード関数を返します。

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

与えられた *encoding* のデコード関数を返します。

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

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*)

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*)

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*)

与えられた *encoding* の `StreamWriter` ファクトリ関数を返します。

5.9.2 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*)

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*)

exc を例外として発生させます。

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

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*)

unicode エラーを ? か U+FFFD で置き換えます。

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

unicode encode エラーを XML 文字参照で置き換えます。

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

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。

第 6 章

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。

6.1 オブジェクトプロトコル (object protocol)

`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。

`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。

`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)`

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 `__dict__` 属性に加え、オブジェクトの MRO にあるクラスの辞書にあるデスクリプタを探します。 `descriptors` で概要が述べられている通り、データのデスクリプタはインスタンスの属性より優先され、非データデスクリプタは後回しにされます。見付からなかった場合は `AttributeError` を送出します。

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

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スクリプタを探し、見付かった場合はインスタンスの辞書にある属性の設定や削除よりも優先されます。そうでない場合は、(もし存在すれば) オブジェクトの `__dict__` に属性を設定もしくは削除します。成功すると `0` が返され、そうでない場合は `AttributeError` が送出され `-1` が返されます。

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

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

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

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_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。

int PyObject_Cmp (*PyObject**o1, *PyObject**o2, *int**result)

o1 と *o2* の値を比較します。このとき *o1* が比較ルーチンを持っていればそれを使い、なければ *o2* のルーチンを使います。比較結果は *result* に返されます。失敗すると -1 を返します。Python 文 `result = cmp(o1, o2)` と同じです。

int PyObject_Compare (*PyObject**o1, *PyObject**o2)

o1 と *o2* の値を比較します。このとき *o1* が比較ルーチンを持っていればそれを使い、なければ *o2* のルーチンを使います。成功すると比較結果を返します。エラーが生じた場合の戻り値は未定義です; `PyErr_Occurred()` を使ってエラー検出を行って下さい。Python 式 `cmp(o1, o2)` と同じです。

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

Return value: New reference. *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()` や逆クオート表記の処理で呼び出されます。

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

Return value: New reference. *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` 文の処理で呼び出されます。

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

o オブジェクトの bytes 表現を計算します。2.x では、単なる `PyObject_Str()` のエイリアスです。

***PyObject** PyObject_Unicode** (*PyObject**o)

Return value: New reference. *o* の Unicode 文字列表現を計算します。成功すると Unicode 文字列表現を返し失敗すると `NULL` を返します。Python 式 `unicode(o)` と同じです。この関数は組み込み関数 `unicode()` の処理で呼び出されます。

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

inst が *cls* のインスタンスか、*cls* のサブクラスのインスタンスの場合に 1 を返し、そうでなければ 0 を返します。エラーの時には -1 を返し、例外をセットします。*cls* がクラスオブジェクトではなく型オブジェクトの場合、`PyObject_IsInstance()` は *inst* が *cls* であるときに 1 を返します。*cls* をタプルで指定した場合、*cls* に指定した全てのエントリについてチェックを行います。少なくとも一つのエントリに対するチェックが 1 を返せば結果は 1 になり、そうでなければ 0 になります。*inst* がクラスインスタンスでなく、かつ *cls* が型オブジェクトでもクラスオブジェクトでもタプルでもない場合、*inst* には `__class__` 属性がなくてはなりません

バージョン 2.1 で追加。

バージョン 2.2 で変更: 二つ目の引数にタプルのサポートを追加しました。

サブクラスの決定はかなり素直な方法で行いますが、クラスシステムの拡張を実装する人たちに知っておいて欲しいちょっとした問題点があります。A と B がクラスオブジェクトの場合、B が A のサブクラスとなるのは、B が A を直接的あるいは間接的に継承 (inherit) している場合です。両方がクラスオブジェクトでない場合、二つのオブジェクト間のクラス関係を決めるには、より汎用的な機構を使います。B が A のサブクラスで

あるか調べたとき、*A* が *B* と等しければ、`PyObject_IsSubclass()` は真を返します。*A* および *B* が異なるオブジェクトなら、*B* の `__bases__` 属性から深さ優先探索 (depth-first search) で *A* を探索します

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

クラス *derived* が *cls* と同じクラスか、*cls* の派生クラスの場合に 1 を返し、それ以外の場合には 0 を返します。エラーが生じると -1 を返します。*cls* をタプルで指定した場合、*cls* に指定した全てのエントリについてチェックを行います。少なくとも一つのエントリに対するチェックが 1 を返せば結果は 1 になり、そうでなければ 0 になります。*derived* または *cls* のいずれかが実際のクラスオブジェクト (あるいはタプル) でない場合、上で述べた汎用アルゴリズムを使います。

バージョン 2.1 で追加。

バージョン 2.3 で変更: 以前の Python のバージョンは、二つ目の引数にタプルをサポートしていませんでした。

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_object, *PyObject**args, *PyObject**kw)

Return value: New reference. 呼び出し可能な Python オブジェクト *callable_object* をタプルで指定された引数 *args* および辞書で指定された名前つき引数 (named argument) *kw* とともに呼び出します。名前つき引数を必要としない場合、*kw* を `NULL` にしてもかまいません。*args* は `NULL` であってはなりません。引数が全く必要ない場合には空のタプルを使ってください。成功すると呼び出し結果として得られたオブジェクトを返し、失敗すると `NULL` を返します。Python の式 `apply(callable_object, args, kw)` あるいは `callable_object(*args, **kw)` と同じです。

バージョン 2.2 で追加。

*PyObject** **PyObject_CallObject** (*PyObject**callable_object, *PyObject**args)

Return value: New reference. 呼び出し可能な Python オブジェクト *callable_object* をタプルで指定された引数 *args* とともに呼び出します。引数を必要としない場合、*args* を `NULL` にしてもかまいません。成功すると呼び出し結果として得られたオブジェクトを返し、失敗すると `NULL` を返します。Python の式 `apply(callable_object, args)` あるいは `callable_object(*args)` と同じです。

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

Return value: New reference. 呼び出し可能な Python オブジェクト *callable_object* を可変数個の C 引数とともに呼び出します。C 引数は `Py_BuildValue()` 形式のフォーマット文字列を使って記述します。*format* は `NULL` にしてもよく、与える引数がないことを表します。成功すると呼び出し結果として得られたオブジェクトを返し、失敗すると `NULL` を返します。Python の式 `apply(callable, args)` あるいは `callable(*args)` と同じです。もしも、`PyObject *args` だけを引数に渡す場合は、`PyObject_CallFunctionObjArgs()` がより速い方法であることを覚えておいてください。

*PyObject** **PyObject_CallMethod** (*PyObject**o, char*method, char*format, ...)

Return value: New reference. オブジェクト *o* の *method* という名前のメソッドを、可変数個の 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ような `Py_BuildValue()` 形式のフォーマット文字列を使って記述します。*format* は `NULL` にしてもよく、与える引数がないことを表します。成功すると呼び出し結果として得られたオブジェクトを返し、失敗すると `NULL` を返します。

Python の式 `o.method(args)` と同じです。もしも、`PyObject *args` だけを引数に渡す場合は、`PyObject_CallMethodObjArgs()` がより速い方法であることを覚えておいてください。

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

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 `PyObject*` 引数とともに呼び出します。引数列は末尾に `NULL` がついた可変数個のパラメタとして与えます。成功すると呼び出し結果として得られたオブジェクトを返し失敗すると `NULL` を返します。

バージョン 2.2 で追加。

`PyObject* PyObject_CallMethodObjArgs (PyObject*o, PyObject*name, ..., NULL)`

Return value: New reference. オブジェクト `o` のメソッドを呼び出します、メソッド名は Python 文字列オブジェクト `name` で与えます。可変数個の `PyObject*` 引数と共に呼び出されます。引数列は末尾に `NULL` がついた可変数個のパラメタとして与えます。成功すると呼び出し結果として得られたオブジェクトを返し失敗すると `NULL` を返します。

バージョン 2.2 で追加。

`long 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。

`long 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。

バージョン 2.6 で追加。

`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. `o` が `NULL` でない場合、オブジェクト `o` のオブジェクト型に相当する型オブジェクトを返します。失敗すると `SystemError` を送出して `NULL` を返します。Python の式 `type(o)` と同じです。この関数は戻り値の参照カウントをインクリメントします。参照カウントのインクリメントが必要でない限り、広く使われていて `PyTypeObject*` 型のポインタを返す表記法 `o->ob.type` の代わりに使う理由は全くありません。

`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。

バージョン 2.2 で追加。

`Py_ssize_t PyObject_Length (PyObject*o)`

`Py_ssize_t PyObject_Size (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。

バージョン 2.5 で変更: これらの関数は以前は `int` を返していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`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。Python の文 `o[key] = v` と同じです。

`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。

`int PyObject_AsFileDescriptor (PyObject*o)`

Python オブジェクトからファイル記述子を取り出します。オブジェクトが整数か長整数なら、その値を返します。(長) 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; この場合、`fileno()` メソッドは整数または長整数をファイル記述子の値として返さなければなりません。失敗すると `-1` を返します。

`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。

6.2 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。

`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_Divide** (*PyObject***o1*, *PyObject***o2*)

Return value: New reference. 成功すると *o1* を *o2* で除算した結果を返し、失敗すると *NULL* を返します。Python の式 $o1 / o2$ と同じです。

*PyObject** **PyNumber_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。

バージョン 2.2 で追加。

*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。

バージョン 2.2 で追加。

*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_InPlaceDivide** (*PyObject***o1*, *PyObject***o2*)

Return value: *New reference.* 成功すると *o1* を *o2* で除算した結果を返し、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_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。

バージョン 2.2 で追加。

***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。

バージョン 2.2 で追加。

***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、不正なメモリアクセス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。

int PyNumber_Coerce (*PyObject**p1, PyObject**p2*)

この関数は *PyObject** 型の二つの変数のアドレスを引数にとります。 `*p1` と `*p2` が指すオブジェクトが同じ型の場合、それぞれの参照カウントをインクリメントして 0 (成功) を返します。オブジェクトを変換して共通の数値型にできる場合、 `*p1` と `*p2` を変換後の値に置き換えて (参照カウントを '新しく' して)、 0 を返します。変換が不可能な場合や、その他何らかのエラーが生じた場合、 -1 (失敗) を返し、参照カウントをインクリメントしません。 `PyNumber_Coerce(&o1, &o2)` の呼び出しは Python 文 `o1, o2 = coerce(o1, o2)` と同じです。

int PyNumber_CoerceEx (*PyObject**p1, PyObject**p2*)

この関数は `PyNumber_Coerce()` と似ていますが、変換が失敗した場合にはエラーを発生させず、 1 (訳注: -1 の間違いではありません、 1 です) を返します。この場合、参照カウントはインクリメントされません。

*PyObject** **PyNumber_Int** (*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。引数の値が整数の範囲外の場合、長整数を代わりに返します。Python の式 `int(o)` と同じです。

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

Return value: *New reference.* 成功すると `o` を長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 `long(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*)

`o` を Python の `int` もしくは `long` 型に変換し、成功したらその値を返します。失敗すると `NULL` が返され、`TypeError` 例外が送出されます。

バージョン 2.5 で追加。

*PyObject** **PyNumber_ToBase** (*PyObject*n, intbase*)

整数 `n` を、基数 `base` 進数の文字列に変換し、適切であれば `'0b'`, `'0o'`, `'0x'` の基数マーカーをつけます。 `base` が 2, 8, 10, 16 のいずれでも無い場合、フォーマットは `x` を基数として `'x#num'` となります。もし `n` が整数オブジェクトでない場合、まず `PyNumber_Index()` を使って変換されます。

バージョン 2.6 で追加。

`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。もし `o` が Python の `int` もしくは `long` 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。

バージョン 2.5 で追加。

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。

バージョン 2.5 で追加。

6.3 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。

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。

バージョン 2.5 で変更: これらの関数は以前は `int` を返していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

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。

バージョン 2.5 で変更: この関数は以前は *count* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

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。

バージョン 2.5 で変更: この関数は以前は *count* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

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。

バージョン 2.5 で変更: この関数は以前は *i* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*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。

バージョン 2.5 で変更: この関数は以前は *i1*, *i2* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`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。

バージョン 2.5 で変更: この関数は以前は *i* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`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。

バージョン 2.5 で変更: この関数は以前は *i* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

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

シーケンスオブジェクト *o* の *i1* から *i2* までのスライスに、シーケンスオブジェクト *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[i1:i2] = v` と同じです。

v が `NULL` の場合はスライスは削除されますが、この機能は非推奨であり `PySequence_DelSlice()` を使うのが望ましいです。

バージョン 2.5 で変更: この関数は以前は *i1*, *i2* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`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。

バージョン 2.5 で変更: この関数は以前は *i1*, *i2* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

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。

バージョン 2.5 で変更: この関数は以前は `int` を返していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

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。

バージョン 2.5 で変更: この関数は以前は `int` を返していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

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

Return value: *New reference.* 任意のシーケンス *o* と同じ内容を持つリストオブジェクトを返します。返されるリストは必ず新しいリストオブジェクトになります。

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

Return value: *New reference.* 任意のシーケンス *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* がタブルの場合、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)` と同じです。

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

Return value: *New reference.* シーケンス *o* がすでにタプルやリストであれば *o* を返し、そうでなければ *o* をタプルで返します。返されるタプルのメンバにアクセスするには `PySequence_Fast_GET_ITEM()` を使ってください。失敗すると `NULL` を返します。オブジェクトがシーケンスでなければ、*m* がメッセージテキストになっている `TypeError` を送出します。

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。

バージョン 2.5 で変更: この関数は以前は *i* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

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。

バージョン 2.4 で追加。

PyObject* **PySequence_ITEM** (*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。`PySequence_GetItem()` ですが、*o* への `PySequence_Check()` が真になるかチェックせず、負のインデクスに対する調整を行いません。

バージョン 2.3 で追加。

バージョン 2.5 で変更: この関数は以前は *i* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`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。

6.4 マップ型プロトコル (mapping protocol)

`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場合に 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。

`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。マップ型プロトコルを提供していないオブジェクトに対しては、Python の式 `len(o)` と同じになります。

バージョン 2.5 で変更: これらの関数は以前は `int` を返していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`int PyMapping_DelItemString (PyObject*o, char*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。

`int PyMapping_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。

`int PyMapping_HasKeyString (PyObject*o, 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 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。

`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 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。

`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。Python の式 `o.keys()` と同じです。

`PyObject* PyMapping_Values (PyObject*o)`

Return value: New reference. 成功するとオブジェクト *o* のキーに対応する値からなるリストを返します。失敗すると `NULL` を返します。Python の式 `o.values()` と同じです。

*PyObject** **PyMapping_Items** (*PyObject** *o*)

Return value: *New reference.* 成功するとオブジェクト *o* の要素対、すなわちキーと値のペアが入ったタプルからなるリストを返します。失敗すると *NULL* を返します。Python の式 `o.items()` と同じです。

*PyObject** **PyMapping_GetItemString** (*PyObject** *o*, *char** *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 **PyMapping_SetItemString** (*PyObject** *o*, *char** *key*, *PyObject** *v*)

オブジェクト *o* で *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。

6.5 イテレータプロトコル (iterator protocol)

バージョン 2.2 で追加.

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。

この関数は旧スタイルクラスの場合に虚偽の真値を返します。というのもそれらのクラスは `next()` メソッドを呼び出すか `TypeError` を投げる実装を持った `tp_iternext` スロットを常に定義しているからです。

*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);
```

(次のページに続く)

```

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

```

6.6 古いバッファプロトコル

このセクションは Python 1.6 で導入された古いバッファプロトコルについて解説します。このプロトコルは、Python 2.x 系ではサポートされていますが廃止予定扱いです。Python 3 から、このプロトコルの弱点や欠点を克服した新しいバッファプロトコルが導入され、Python 2.6 へと逆移植されました。詳細は [buffer オブジェクト](#) と [memoryview オブジェクト](#) を参照してください。

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。

バージョン 1.6 で追加。

バージョン 2.5 で変更: この関数は以前は buffer_len の型に int * を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

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。

バージョン 1.6 で追加。

バージョン 2.5 で変更: この関数は以前は buffer_len の型に int * を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

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。

バージョン 2.2 で追加。

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。

バージョン 1.6 で追加.

バージョン 2.5 で変更: この関数は以前は *buffer_len* の型に `int *` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

第 7 章

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。

7.1 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。

7.1.1 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` や `types.TypeType` 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 (subtype) 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。

バージョン 2.2 で追加.

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。

バージョン 2.6 で追加.

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。

バージョン 2.6 で追加.

int **PyType_HasFeature** (*PyObject**o, intfeature)

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** (*PyObject**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。

バージョン 2.0 で追加.

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。

バージョン 2.2 で追加.

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* に対して呼ばれません。 `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_tnitems)

Return value: New reference. バージョン 2.2 で追加.

バージョン 2.5 で変更: この関数は以前は *nitems* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

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

Return value: New reference. バージョン 2.2 で追加.

int **PyType_Ready** (*PyTypeObject**type)

型オブジェクトの後始末処理 (finalize) 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。

バージョン 2.2 で追加.

7.1.2 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 *Py_None* を戻す操作を適切に行うためのマクロです。

バージョン 2.4 で追加.

7.2 数値型オブジェクト (numeric object)

7.2.1 (通常) 整数型オブジェクト (plain integer object)

PyIntObject

この *PyObject* のサブタイプは整数型を表現します。

PyTypeObject **PyInt_Type**

この *PyTypeObject* のインスタンスは Python の (長整数でない) 整数型を表現します。これは Python レイヤにおける `int` や `types.IntType` と同じオブジェクトです。

int PyInt_Check (*PyObject**o)

o が *PyInt_Type* 型か *PyInt_Type* 型のサブタイプであるときに真を返します。

バージョン 2.2 で変更: サブタイプを引数にとれるようになりました。

int PyInt_CheckExact (*PyObject**o)

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

バージョン 2.2 で追加.

*PyObject** **PyInt_FromString** (char*str, char**pend, intbase)

Return value: *New reference.* str の文字列値に基づいて、新たな *PyIntObject* または *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 でなければ、*pend は str 中で数が表現されている部分以後の先頭の文字のアドレスを指しています。base が 0 ならば、str の先頭の文字列に基づいて基数を決定します: もし str が '0x' または '0X' で始まっていると、基数に 16 を使います; str が '0' で始まっていると、基数に 8 を使います; その他の場合には基数に 10 を使います。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。先頭に空白がある場合は無視されます。数字が全くない場合、`ValueError` が送出されます。使用しているマシンの long int 型で表現しきれないくらい大きな数が文字列に入っており、オーバ

フロー警告が抑制されていれば、`PyLongObject` を返します。オーバフロー警告が抑制されていない場合は、`NULL` を返します。

*PyObject** **PyInt_FromLong** (longival)

Return value: New reference. *ival* の値を使って新たな整数オブジェクトを生成します。

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** **PyInt_FromSsize_t** (Py_ssize_tival)

Return value: New reference. *ival* の値を使って新たな整数オブジェクトを生成します。値が `LONG_MAX` より大きいか `LONG_MIN` より小さい場合、長整数オブジェクトを返します。

バージョン 2.5 で追加.

*PyObject** **PyInt_FromSize_t** (size_tival)

ival の値を使って新たな整数オブジェクトを生成します。値が `LONG_MAX` を超えている場合、長整数オブジェクトを返します。

バージョン 2.5 で追加.

long **PyInt_AsLong** (*PyObject**io)

オブジェクトがまだ *PyIntObject* でなければまず型キャストを試み、次にその値を返します。エラーが発生した場合、 -1 が返されます。その時呼び出し側は、`PyErr_Occurred()` を使って、エラーが発生したのか、単に値が -1 だったのかを判断すべきです。

long **PyInt_AS_LONG** (*PyObject**io)

オブジェクト *io* の値を返します。エラーチェックを行いません。

unsigned long **PyInt_AsUnsignedLongMask** (*PyObject**io)

オブジェクトがまだ *PyIntObject* または *PyLongObject* でなければまず型キャストを試み、次にその値を unsigned long 型で返します。この関数はオーバフローをチェックしません。

バージョン 2.3 で追加.

unsigned PY_LONG_LONG **PyInt_AsUnsignedLongLongMask** (*PyObject**io)

オブジェクトがまだ *PyIntObject* または *PyLongObject* でなければまず型キャストを試み、次にその値を unsigned long long 型で返します。オーバフローをチェックしません。

バージョン 2.3 で追加.

Py_ssize_t **PyInt_AsSsize_t** (*PyObject**io)

オブジェクトがまだ *PyIntObject* または *PyLongObject* でなければまず型キャストを試み、次にその値を `Py_ssize_t` 型で返します。

バージョン 2.5 で追加.

long **PyInt_GetMax** ()

システムの知識に基づく、扱える最大の整数値 (システムのヘッダファイルに定義されている

`LONG_MAX`) を返します。

`int PyInt_ClearFreeList()`

整数の free list をクリアします。解放できなかった要素の数を返します。

バージョン 2.6 で追加.

7.2.2 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。

バージョン 2.3 で追加.

*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。

バージョン 2.4 で追加.

`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。

バージョン 2.4 で追加.

*PyObject** `PyBool_FromLong(longv)`

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。

バージョン 2.3 で追加.

7.2.3 長整数型オブジェクト (long integer object)

`PyLongObject`

この *PyObject* のサブタイプは長整数型を表現します。

PyTypeObject `PyLong_Type`

この *PyTypeObject* のインスタンスは Python 長整数型を表現します。これは Python レイヤーにおける `long` や `types.LongType` 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。

バージョン 2.2 で変更: サブタイプを引数にとれるようになりました。

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。

バージョン 2.2 で追加。

*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。

*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*tv*)

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。

バージョン 2.6 で追加。

*PyObject** **PyLong_FromSize_t** (size_t*tv*)

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。

バージョン 2.6 で追加。

*PyObject** **PyLong_FromLongLong** (PY_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 PY_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** (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* でなければ、 **pend* は *str* 中で数が表現されている部分以後の先頭の文字のアドレスを指しています。 *base* が 0 ならば、 *str* の先頭の文字列に基づいて基数を決定します: もし *str* が '0x' または '0X' で始まっていれば、基数に 16 を使います; *str* が '0' で始まっていれば、基数に 8 を使います; その他の場合には基数に 10 を使います。 *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。先頭に空白がある場合は無視されます。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.* Unicode の数字配列を Python の長整数型に変換します。最初のパラメータ *u* は、Unicode 文字列の最初の文字を指し、*length* には文字数を指定し、*base* には変換時の基数を指定します。基数は範囲 [2, 36] になければなりません; 範囲外の基数を指定すると、`ValueError` を送出します。

バージョン 1.6 で追加。

バージョン 2.5 で変更: この関数は以前は *length* に `int *` 型を使用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

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

Return value: *New reference.* ポインタ *p* から Python 整数値または長整数型を生成します。ポインタの値は `PyLong_AsVoidPtr()` を適用した結果から取得できます。

バージョン 1.5.2 で追加。

バージョン 2.5 で変更: 整数値が `LONG_MAX` より大きい場合に、正の長整数を返すようになりました。

long **PyLong_AsLong** (*PyObject***pylong*)

pylong の指す長整数値を、C の `long` 型表現で返します。 *pylong* が `LONG_MAX` よりも大きい場合、`OverflowError` を送出し、`-1` を返します。

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

pylong の値を C の `long` 型表現で返します。 *pylong* が `LONG_MAX` より大きかったり `LONG_MIN` より小さい場合、**overflow* に 1 か `-1` を設定して `-1` を返します。それ以外の場合は **overflow* に 0 を設定します。なにか例外が発生した場合は (`TypeError` や `MemoryError` など)、**overflow* は 0 で戻り値が `-1` になります。

バージョン 2.7 で追加。

`PY_LONG_LONG` **PyLong_AsLongLongAndOverflow** (*PyObject***pylong*, *int***overflow*)

pylong の値を C の `long long` 型表現で返します。 *pylong* が `PY_LLONG_MAX` より大きかったり `PY_LLONG_MIN` より小さい場合、**overflow* に 1 か `-1` を設定して `-1` を返します。それ以外の場合は **overflow* に 0 を設定します。なにか例外が発生した場合は (`TypeError` や `MemoryError` など)、**overflow* は 0 で戻り値が `-1` になります。

バージョン 2.7 で追加。

Py_ssize_t **PyLong_AsSsize_t** (*PyObject***pylong*)

pylong の指す長整数値を、C の `Py_ssize_t` 型表現で返します。 *pylong* が `PY_SSIZE_T_MAX` よりも大きい場合、`OverflowError` を送出し、`-1` を返します。

バージョン 2.6 で追加。

unsigned long **PyLong_AsUnsignedLong** (*PyObject***pylong*)

pylong の指す長整数値を、C の `unsigned long` 型表現で返します。 *pylong* が `ULONG_MAX` よりも大きい場合、`OverflowError` を送出します。

`PY_LONG_LONG` **PyLong_AsLongLong** (*PyObject***pylong*)

pylong の指す長整数値を、C の `long long` 型表現で返します。 *pylong* が `long long` で表せない

場合、`OverflowError` を送出し `-1` を返します。

バージョン 2.2 で追加。

`unsigned PY_LONG_LONG PyLong_AsUnsignedLongLong (PyObject*pylong)`

pylong の指す長整数値を、C の `unsigned long long` 型表現で返します。*pylong* が `unsigned long long` で表せない場合、`OverflowError` を発生させて、`(unsigned long long)-1` を返します。

バージョン 2.2 で追加。

バージョン 2.7 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*io)`

Python 長整数値を、オーバーフローチェックを行わずに C の `unsigned long` 型表現で返します。

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 2.3 で追加。

`unsigned PY_LONG_LONG PyLong_AsUnsignedLongLongMask (PyObject*io)`

Python 長整数値を、オーバーフローチェックを行わずに C の `unsigned long long` 型表現で返します。

Returns `(unsigned PY_LONG_LONG)-1` on error. Use `PyErr_Occurred()` to disambiguate.

バージョン 2.3 で追加。

`double PyLong_AsDouble (PyObject*pylong)`

pylong の指す値を、C の `double` 型表現で返します。*pylong* が `double` を使って近似表現できない場合、`OverflowError` 例外を送出して `-1.0` を返します。

`void* PyLong_AsVoidPtr (PyObject*pylong)`

Python の整数型か長整数型を指す *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。

バージョン 1.5.2 で追加。

バージョン 2.5 で変更: 値が `0..LONG_MAX` の範囲外だった場合でも、符号付き整数と符号無し整数の両方とも利用可能になりました。

7.2.4 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 浮動小数点型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 `float` や `types.FloatType` 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。

バージョン 2.2 で変更: サブタイプを引数にとれるようになりました。

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。

バージョン 2.2 で追加。

PyObject* PyFloat_FromString (PyObject*str, charpend)**

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。引数 *pend* は無視されます。この引数は後方互換性のためだけに残されています。

PyObject* PyFloat_FromDouble (doublev)

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)

pyfloat の指す値を、C の `double` 型表現で返します。 *pyfloat* が Python 浮動小数点オブジェクトではなく、かつ `__float__()` メソッドを持っていれば、 *pyfloat* を浮動小数点に変換するためにこのメソッドが最初に呼ばれます。失敗した場合 `-1.0` を返します。そのため呼び出し元は *PyErr_Occurred()* を呼んでエラーをチェックすべきです。

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)

`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。

バージョン 2.6 で追加。

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。

バージョン 2.6 で追加。

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。

バージョン 2.6 で追加。

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。

バージョン 2.6 で追加。

void **PyFloat_AsString** (char*buf, *PyFloatObject**v)

`str()` と同じルールで *v* を文字列に変換します。 *buf* の長さは 100 以上でなければなりません。

この関数は長さを知らないバッファに書きこむので安全ではありません。

バージョン 2.7 で非推奨: 代わりに `PyObject_Str()` か `PyOS_double_to_string()` を利用してください。

void **PyFloat_AsReprString** (char*buf, *PyFloatObject**v)

`PyFloat_AsString()` とほとんど同じですが、`repr()` とおなじルールを使います。 *buf* の長さは 100 以上でなければなりません。

この関数は長さを知らないバッファに書きこむので安全ではありません。

バージョン 2.7 で非推奨: 代わりに `PyObject_Repr()` か `PyOS_double_to_string()` を利用してください。

7.2.5 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、この型の構造体を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_complexdividend*, *Py_complexdivisor*)

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_complexnum*, *Py_complexexp*)

指数 *exp* の *num* 乗を C の *Py_complex* 型で返します。

num が null で *exp* が正の実数でない場合は、このメソッドはゼロを返し、*errno* に EDOM をセットします。

Python オブジェクトとしての複素数型

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 の `complex` や `types.ComplexType` 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。

バージョン 2.2 で変更: サブタイプを引数にとれるようになりました。

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。

バージョン 2.2 で追加。

*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。失敗時は実数としての -1.0 を返します。

バージョン 2.6 で変更: *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。

7.3 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。

7.3.1 bytearray オブジェクト

バージョン 2.6 で追加。

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*)

バッファプロトコルを実装している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*)

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*)

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。

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。

7.3.2 文字列とバイト列オブジェクト

下記の関数は、文字列オブジェクトを期待している引数に文字列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。

注釈: これらの関数群は Python 3.x では `PyBytes_*` ヘリネームされます。移植を容易にするために、特に注釈されていない限り、3.x で利用できる `PyBytes` 関数群は同等の `PyString_*` 関数へのエイリアスにされています。

PyStringObject

この *PyObject* のサブタイプは Python の文字列オブジェクトを表現します。

PyTypeObject PyString_Type

この *PyTypeObject* のインスタンスは Python の文字列型を表現します; このオブジェクトは Python レイヤにおける `str` や `types.StringType` と同じです。 .

int PyString_Check (PyObject*o)

o が文字列型か文字列型のサブタイプであるときに真を返します。

バージョン 2.2 で変更: サブタイプを引数にとれるようになりました。

int PyString_CheckExact (PyObject*o)

o が文字列型で、かつ文字列型のサブタイプでないときに真を返します。

バージョン 2.2 で追加。

PyObject* PyString_FromString (const char*v)

Return value: New reference. *v* を値に持つ文字列オブジェクトを返します。失敗すると `NULL` を返します。パラメタ *v* は `NULL` であってはなりません; `NULL` かどうかはチェックしません。

PyObject* PyString_FromStringAndSize (const char*v, Py_ssize_tlen)

Return value: New reference. 値が *v* で長さが *len* の新たな文字列オブジェクトを返します。失敗すると `NULL` を返します。 *v* が `NULL` の場合、文字列の中身は未初期化の状態になります。

バージョン 2.5 で変更: この関数は以前は *len* の型に `int` を利用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

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

Return value: New reference. C 関数 `printf()` 形式の *format* 文字列と可変個の引数を取り、書式化済みの文字列長を計算した上で、書式化を行った結果を値とする Python 文字列にして返します。可変個の引数部は C のデータ型でなくてはならず、かつ *format* 文字列内の書式指定文字 (format character) に一致する型でなくてはなりません。利用できる書式化文字は以下の通りです:

書式指定文字	型	コメント
%%	<i>n/a</i>	文字 % のリテラル。
%c	int	C の整数型で表現される単一の文字。
%d	int	C の <code>printf("%d")</code> と全く同じ。
%u	unsigned int	C の <code>printf("%u")</code> と全く同じ。
%ld	long	C の <code>printf("%ld")</code> と全く同じ。
%lu	unsigned long	C の <code>printf("%lu")</code> と全く同じ。
%lld	long long	C の <code>printf("%lld")</code> と全く同じ。
%llu	unsigned long long	C の <code>printf("%llu")</code> と全く同じ。
%zd	Py_ssize_t	C の <code>printf("%zd")</code> と全く同じ。
%zu	size_t	C の <code>printf("%zu")</code> と全く同じ。
%i	int	C の <code>printf("%i")</code> と全く同じ。
%x	int	C の <code>printf("%x")</code> と全く同じ。
%s	char*	null で終端された C の文字列。
%p	void*	C ポインタの 16 進表記。 <code>printf("%p")</code> とほとんど同じだが、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 <code>0x</code> 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: "%lld" と "%llu" 書式指定文字は HAVE_LONG_LONG が定義されている時だけ利用できます。

バージョン 2.7 で変更: "%lld" と "%llu" のサポートが追加されました。

*PyObject** **PyString_FromFormatV** (const char*format, va_listvargs)

Return value: New reference. *PyString_FromFormat()* と同じです。ただし、こちらの関数は二つしか引数を取りません。

Py_ssize_t **PyString_Size** (*PyObject**string)

文字列オブジェクト *string* 内の文字列値の長さを返します。

バージョン 2.5 で変更: この関数は以前は int を返していました。この変更により、64 bit システムを

正しくサポートするには修正が必要になります。

`Py_ssize_t PyString_GET_SIZE (PyObject*string)`

`PyString_Size()` をマクロで実装したもので、エラーチェックを行いません。

バージョン 2.5 で変更: このマクロは以前は `int` を返していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

`char* PyString_AsString (PyObject*string)`

`string` の中身を NUL 文字終端された表現で返します。ポインタは `string` オブジェクトの内部バッファを指し、バッファのコピーを指すわけではありません。 `PyString_FromStringAndSize(NULL, size)` を使って生成した文字列でない限り、バッファ内のデータはいかなる変更もしてはなりません。この文字列をデアロケートしてはなりません。 `string` が Unicode オブジェクトの場合、この関数は `string` のデフォルトエンコーディング版を計算し、デフォルトエンコーディング版に対して操作を行います。 `string` が文字列オブジェクトですらない場合、 `PyString_AsString()` は `NULL` を返して `TypeError` を送出します。

`char* PyString_AS_STRING (PyObject*string)`

`PyString_AsString()` をマクロで実装したもので、エラーチェックを行いません。文字列オブジェクトだけをサポートします; Unicode オブジェクトを渡してはなりません。

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

`obj` の中身を NUL 文字終端された表現にして、出力用の変数 `buffer` と `length` を使って返します。

この関数は文字列オブジェクトと Unicode オブジェクトのどちらも入力として受理します。Unicode オブジェクトの場合、オブジェクトをデフォルトエンコーディングでエンコードしたバージョン (default encoded version) を返します。 `length` が `NULL` の場合、値を返させるバッファには NUL 文字を入れてはなりません; NUL 文字が入っている場合、関数は `-1` を返し、 `TypeError` を送出します。

`buffer` は `obj` の内部文字列バッファを参照し、バッファのコピーを参照するわけではありません。 `PyString_FromStringAndSize(NULL, size)` を使って生成した文字列でない限り、バッファ内のデータはいかなる変更もしてはなりません。この文字列をデアロケートしてはなりません。 `string` が Unicode オブジェクトの場合、この関数は `string` のデフォルトエンコーディング版を計算し、デフォルトエンコーディング版に対して操作を行います。 `string` が文字列オブジェクトですらない場合、 `PyString_AsStringAndSize()` は `-1` を返して `TypeError` を送出します。

バージョン 2.5 で変更: この関数は以前は `length` に `int *` 型を使用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

`void PyString_Concat (PyObject**string, PyObject*newpart)`

新しい文字列オブジェクトを `*string` に作成し、 `newpart` の内容を `string` に追加します; 呼び出し側は新たな参照を所有することになります。 `string` の以前の値に対する参照は盗み取られます。新たな文字列を生成できなければ、 `string` に対する古い参照は無視され、 `*string` の値は `NULL` に設定されます; その際、適切な例外情報が設定されます。

`void PyString_ConcatAndDel (PyObject**string, PyObject*newpart)`

新しい文字列オブジェクトを `*string` に作成し、 `newpart` の内容を `string` に追加します。こちらのバージョンの関数は `newpart` への参照をデクリメントします。

`int PyString_Resize (PyObject**string, Py_ssize_tnewsize)`

”変更不能”である文字列オブジェクトをサイズ変更する手段です。新たな文字列オブジェクトを作成するときにのみ使用してください; 文字列がすでにコードの他の部分で使われているかもしれない場合には、この関数を使ってはなりません。入力する文字列オブジェクトの参照カウントが 1 でない場合、この関数を呼び出すとエラーになります。左側値には、既存の文字列オブジェクトのアドレスを渡し (このアドレスには書き込み操作が起きるかもしれませんが)、新たなサイズを指定します。成功した場合、**string* はサイズ変更された文字列オブジェクトを保持し、0 が返されます; **string* の値は、入力したときの値と異なっているかもしれません。文字列の再アロケーションに失敗した場合、**string* に入っていた元の文字列オブジェクトを解放し、**string* を *NULL* にセットし、メモリ例外をセットし、-1 を返します。

バージョン 2.5 で変更: この関数は以前は *newsize* の型に *int* を利用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyString.Format** (*PyObject**format, *PyObject**args)

Return value: New reference. 新たな文字列オブジェクトを *format* と *args* から生成します。 *format* % *args* と似た働きです。引数 *args* はタプルまたは辞書でなければなりません。

`void PyString_InternInPlace (PyObject**string)`

引数 **string* をインプレースで隔離 (intern) します。引数は Python 文字列オブジェクトを指すポインタへのアドレスでなくてはなりません。 **string* と等しい、すでに隔離済みの文字列が存在する場合、そのオブジェクトを **string* に設定します (かつ、元の文字列オブジェクトの参照カウントをデクリメントし、すでに隔離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)。 (補足: 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; この関数では、関数の呼び出し後にオブジェクトに対して参照の所有権を持てるのは、関数を呼び出す前にすでに所有権を持っていた場合に限りです。)

注釈: この関数は 3.x では利用できず、PyBytes エイリアス也没有ありません。

*PyObject** **PyString_InternFromString** (const char*v)

Return value: New reference. *PyString_FromString()* と *PyString_InternInPlace()* を組み合わせたもので、隔離済みの新たな文字列オブジェクトを返すか、同じ値を持つすでに隔離済みの文字列オブジェクトに対する新たな (”所有権を得た”) 参照を返します。

注釈: この関数は 3.x では利用できず、PyBytes エイリアス也没有ありません。

*PyObject** **PyString.Decode** (const char*s, Py_ssize_tsize, const char*encoding, const char*errors)

Return value: New reference. *size* からなるエンコード済みのバッファ *s* を *encoding* の名前で登録されている codec に渡してデコードし、オブジェクトを生成します。 *encoding* および *errors* は組み込み関数 *unicode()* に与える同名のパラメ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。

注釈: この関数は 3.x では利用できず、PyBytes エイリアス也没有ありません。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyString_AsDecodedObject** (*PyObject***str*, const char**encoding*, const char**errors*)

Return value: *New reference.* 文字列オブジェクトを *encoding* の名前で登録されている codec に渡してデコードし、Python オブジェクトを返します。 *encoding* および *errors* は文字列型の `encode()` メソッドに与える同名のパラメ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。

注釈: この関数は 3.x では利用できず、PyBytes エイリアス也没有ありません。

*PyObject** **PyString_Encode** (const char*s, Py_ssize_tsize, const char**encoding*, const char**errors*)

Return value: *New reference.* *size* で指定されたサイズの `char` バッファを *encoding* の名前で登録されている codec に渡してエンコードし、Python オブジェクトを返します。 *encoding* および *errors* は文字列型の `encode()` メソッドに与える同名のパラメ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。

注釈: この関数は 3.x では利用できず、PyBytes エイリアス也没有ありません。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyString_AsEncodedObject** (*PyObject***str*, const char**encoding*, const char**errors*)

Return value: *New reference.* エンコード名 *encoding* で登録された codec を使って文字列オブジェクトをエンコードし、その結果を Python オブジェクトとして返します。 *encoding* および *errors* は文字列型の `encode()` メソッドに与える同名のパラメ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。

注釈: この関数は 3.x では利用できず、PyBytes エイリアス也没有ありません。

7.3.3 Unicode オブジェクトと codec

Unicode オブジェクト

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_UNICODE

この型は Unicode 序数 (Unicode ordinal) を保持するための基礎単位として、Python が内部的に使用します。Python のデフォルトのビルドでは、`Py_UNICODE` として 16-bit 型を利用し、Unicode の値を内部では UCS-2 で保持します。UCS4 版の Python をビルドすることもできます。(最近の多くの Linux ディストリビューションでは UCS4 版の Python がついてきます) UCS4 版ビルドでは `Py_UNICODE` に 32-bit 型を利用し、内部では Unicode データを UCS4 で保持します。wchar_t が利用できて、Python の Unicode に関するビルドオプションと一致するときは、`Py_UNICODE` は wchar_t を typedef でエイリアスされ、ネイティブプラットフォームに対する互換性を高めます。それ以外のすべてのプラットフォームでは、`Py_UNICODE` は unsigned short (UCS2) か unsigned long (UCS4) の typedef によるエイリアスになります。

UCS2 と UCS4 の Python ビルドの間にはバイナリ互換性がないことに注意してください。拡張やインタフェースを書くときには、このことを覚えておいてください。

PyUnicodeObject

この `PyObject` のサブタイプは Unicode オブジェクトを表します。

PyTypeObject PyUnicode_Type

この `PyTypeObject` のインスタンスは Python の Unicode 型を表します。Python レイヤにおける `unicode` や `types.UnicodeType` と同じオブジェクトです。

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出し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。

バージョン 2.2 で変更: サブタイプを引数にとれるようになりました。

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。

バージョン 2.2 で追加。

Py_ssize_t PyUnicode_GET_SIZE (`PyObject*`*o*)

オブジェクトのサイズを返します。*o* は `PyUnicodeObject` でなければなりません (チェックはしません)。

バージョン 2.5 で変更: この関数は以前は int を返していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

Py_ssize_t PyUnicode_GET_DATA_SIZE (`PyObject*`*o*)

オブジェクトの内部バッファのサイズをバイト数で返します。*o* は `PyUnicodeObject` でなければなりません (チェックはしません)。

バージョン 2.5 で変更: この関数は以前は `int` を返していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

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

オブジェクト内部の *Py_UNICODE* バッファへのポインタを返します。 *o* は *PyUnicodeObject* でなければなりません (チェックはしません)。

`const char*` **PyUnicode_AS_DATA** (*PyObject***o*)

オブジェクト内部バッファへのポインタを返します。 *o* は *PyUnicodeObject* でなければなりません (チェックはしません)。

`int` **PyUnicode_ClearFreeList** ()

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。

バージョン 2.6 で追加.

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_UNICODEch*)

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_UNICODEch*)

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_UNICODEch*)

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_UNICODEch*)

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_UNICODEch*)

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_UNICODEch*)

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_UNICODEch*)

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_UNICODEch*)

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_UNICODEch*)

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_UNICODEch)`

ch が英数文字かどうかに応じて 1 または 0 を返します。

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_UNICODEch)`

ch を小文字に変換したものを返します。

`Py_UNICODE Py_UNICODE_TOUPPER(Py_UNICODEch)`

ch を大文字に変換したものを返します。

`Py_UNICODE Py_UNICODE_TOTITLE(Py_UNICODEch)`

ch をタイトルケース文字に変換したものを返します。

`int Py_UNICODE_TODECIMAL(Py_UNICODEch)`

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_UNICODEch)`

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_UNICODEch)`

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。

Plain Py_UNICODE

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_FromUnicode(const Py_UNICODE*u, Py_ssize_tsize)`

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。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`PyObject* PyUnicode_FromStringAndSize(const char*u, Py_ssize_tsize)`

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 エンコードされているものとします。 *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、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。

バージョン 2.6 で追加.

*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。

バージョン 2.6 で追加.

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

Return value: New reference. printf() スタイルの *format* 文字列と可変長引数を受け取り、結果の unicode 文字の長さを計算し、フォーマットされた文字列を含む unicode オブジェクトを返します。可変長引数は C の型を持っていて、*format* 文字列で指定された書式指定文字に完全に従う必要があります。以下の書式指定文字が利用できます:

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文字。
%d	int	C の printf("%d") と全く同じ
%u	unsigned int	C の printf("%u") と全く同じ
%ld	long	C の printf("%ld") と全く同じ
%lu	unsigned long	C の printf("%lu") と全く同じ
%zd	Py_ssize_t	C の printf("%zd") と全く同じ
%zu	size_t	C の printf("%zu") と全く同じ
%i	int	C の printf("%i") と全く同じ
%x	int	C の printf("%x") と全く同じ
%s	char*	null で終端された C の文字列。
%p	void*	C ポインタの 16 進表記。printf("%p") とほとんど同じですが、プラットフォームにおける printf の定義に関わりなく先頭にリテラル 0x が付きます。
%U	PyObject*	unicode オブジェクト。
%V	PyObject*, char *	unicode オブジェクト (NULL でも良い) と、2 つめの引数として NUL 終端の C 文字列 (2 つめの引数は 1 つめの引数が NULL だった時にのみ利用されます)。
%S	PyObject*	PyObject_Unicode() の戻り値
%R	PyObject*	PyObject_Repr() の戻り値

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。

バージョン 2.6 で追加.

*PyObject** **PyUnicode_FromFormatV** (const char*format, va_listvargs)

Return value: New reference. 2 引数であることを除いて *PyUnicode_FromFormat()* と同じです。

バージョン 2.6 で追加.

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

Unicode オブジェクトの内部バッファ *Py_UNICODE* に対する読み出し専用のポインタを返します。*unicode* が Unicode オブジェクトでなければ *NULL* を返します。結果の *Py_UNICODE** 文字列には文字列の途中にも nul 文字が含まれうることに注意してください。これはほとんどの C 関数で文字列を切り捨ててしまう結果になります。

Py_ssize_t **PyUnicode_GetSize** (*PyObject**unicode)

Unicode オブジェクトの長さを返します。

バージョン 2.5 で変更: この関数は以前は *int* を返していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode_FromEncodedObject** (*PyObject**obj, const char*encoding, const char*errors)

Return value: New reference. あるエンコード方式でエンコードされたオブジェクト *obj* を Unicode オブジェクトに型変換して、参照カウンタをインクリメントして返します。

文字列やその他の char バッファ互換オブジェクトの場合、オブジェクトは *encoding* に従ってデコードされます。このとき *error* で定義されたエラー処理を用います。これら二つの引数は *NULL* にでき、その場合デフォルト値が使われます (詳細は次の節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。

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

Return value: New reference. *PyUnicode_FromEncodedObject*(obj, *NULL*, "strict") を行うショートカットで、インタプリタは Unicode への型強制が必要な際に常にこの関数を使います。

プラットフォームで *wchar_t* がサポートされていて、かつ *wchar.h* が提供されている場合、Python は以下の関数を使って *wchar_t* に対して直接アクセスすることができます。このアクセスは、Python 自体の *Py_UNICODE* 型がシステムの *wchar_t* と同一の場合に最適化されます。

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。失敗すると *NULL* を返します。

バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

Py_ssize_t **PyUnicode_AsWideChar** (*PyUnicodeObject**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 0-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 0-終端されている場合も、されていない場合もありま

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 0-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 `nul` 文字が含まれることにも注意してください。これはほとんどの C 関数で文字列を切り捨ててしまう結果になります。

バージョン 2.5 で変更: この関数は以前は `int` を返し、`size` の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

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 Unicode オブジェクトコンストラクタである `unicode()` 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 `Py_FileSystemDefaultEncoding` を使わねばなりません。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` を読んだときなどに) 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_tsize, const char*encoding, const char*errors)

Return value: New reference. 何らかのエンコード方式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 オブジェクトを生成します。`encoding` と `errors` は、組み込み関数 `unicode()` 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。

バージョン 2.5 で変更: この関数は以前は `size` の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode_Encode** (const *Py_UNICODE**s, Py_ssize_tsize, 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 文字列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。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*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 文字列オブジェクトとして返します。 *encoding* および *errors* は Unicode 型の `encode()` メソッドに与える同名のパラメータと同じ意味を持ちます。使用する codec の検索は、Python の codec レジストリを使って行います。codec が例外を送出した場合には `NULL` を返します。

UTF-8 Codecs

以下は UTF-8 codec の API です:

*PyObject** **PyUnicode_DecodeUTF8** (const char*s, Py_ssize_tsize, 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。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode_DecodeUTF8Stateful** (const char*s, Py_ssize_tsize, 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。

バージョン 2.4 で追加。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode_EncodeUTF8** (const *Py_UNICODE**s, Py_ssize_tsize, 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 *s* を UTF-8 でエンコードし、Python 文字列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。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*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 文字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。

UTF-32 Codecs

以下は UTF-32 codec API です:

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

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 バイトがバイトオーダーマーク (BOM) だった場合、デコーダは BOM のバイトオーダーに切り替え、BOM は結果の unicode 文字列には含まれません。

**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。

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。

narrow build の場合、BMP 外のコードポイントはサロゲートペアとしてデコードされます。

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。

バージョン 2.6 で追加.

*PyObject** **PyUnicode_DecodeUTF32Stateful** (const char*s, Py_ssize_tsize, const char*errors, int*byteorder, Py_ssize_t*consumed)

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。

バージョン 2.6 で追加.

*PyObject** **PyUnicode_EncodeUTF32** (const *Py_UNICODE**s, Py_ssize_tsize, const char*errors, intbyteorder)

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。

Py_UNICODE_WIDE が定義されていない場合は、サロゲートペアを 1 つのコードポイントとして出力します。

codec が例外を発生させたときは *NULL* を返します。

バージョン 2.6 で追加.

*PyObject** **PyUnicode_AsUTF32String** (*PyObject**unicode)

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 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。

バージョン 2.6 で追加。

UTF-16 Codecs

以下は UTF-16 codec の API です:

*PyObject** **PyUnicode_DecodeUTF16** (const char*s, Py_ssize_tsize, 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。

バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode_DecodeUTF16Stateful** (const char*s, Py_ssize_tsize, 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。

バージョン 2.4 で追加。

バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用し、*consumed* の型に *int ** を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

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

Return value: New reference. *s* 中の Unicode データを UTF-16 でエンコードした結果が入っている Python 文字列オブジェクトを返します。出力は以下のバイトオーダーに従って書き出されます:

```
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。

Py_UNICODE_WIDE が定義されている場合、単一の *Py_UNICODE* 値はサロゲートペアとして表現されることがあります。 *Py_UNICODE_WIDE* が定義されていないければ、各 *Py_UNICODE* 値は UCS-2 文字として表現されます。

codec が例外を発生させたときは *NULL* を返します。

バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode_AsUTF16String** (*PyObject***unicode*)

Return value: New reference. ネイティブバイトオーダの UTF-16 でエンコードされた Python 文字列を返します。文字列は常に BOM マークから始まります。エラー処理は "strict" です。 codec が例外を出した場合には *NULL* を返します。

UTF-7 Codecs

以下は UTF-7 codec の API です:

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

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*)

consumed が *NULL* のとき、 *PyUnicode_DecodeUTF7()* と同じように動作します。 *consumed* が *NULL* でないとき、末尾の不完全な UTF-7 base-64 部分をエラーとしません。不完全な部分のバイト列はデコードせずに、デコードしたバイト数を *consumed* に格納します。

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

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。

Unicode-Escape Codecs

以下は "Unicode Escape" codec の API です:

*PyObject** **PyUnicode_DecodeUnicodeEscape** (const char*s, Py_ssize_tsize, 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。

バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode_EncodeUnicodeEscape** (const *Py_UNICODE**s, Py_ssize_tsize)

Return value: New reference. *size* で指定された長さを持つ *Py_UNICODE* 型バッファを Unicode-Escape でエンコードし、Python 文字列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。

バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode_AsUnicodeEscapeString** (*PyObject**unicode)

Return value: New reference. Unicode-Escape で Unicode オブジェクトをエンコードし、結果を Python 文字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。

Raw-Unicode-Escape Codecs

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

*PyObject** **PyUnicode_DecodeRawUnicodeEscape** (const char*s, Py_ssize_tsize, 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。

バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode_EncodeRawUnicodeEscape** (const *Py_UNICODE**s, Py_ssize_tsize, 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 Raw-Unicode-Escape でエンコードし、Python 文字列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。

バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode_AsRawUnicodeEscapeString** (*PyObject**unicode)

Return value: New reference. Raw-Unicode-Escape で Unicode オブジェクトをエンコードし、結果を Python 文字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。

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_tsize, 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。

バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode_EncodeLatin1** (const *Py_UNICODE**s, Py_ssize_tsize, 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 文字列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。

バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*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 文字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。

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_tsize, 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。

バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode_EncodeASCII** (const *Py_UNICODE**s, Py_ssize_tsize, 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 文字列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。

バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*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 文字列オ

プロジェクトとして返します。エラー処理は "strict" です。codec が例外を送出した場合には *NULL* を返します。

Character Map Codecs

この codec は、多くの様々な codec を実装する際に使われるという点で特殊な codec です (実際、`encodings` パッケージに入っている標準 codecs のほとんどは、この codec を使っています)。この codec は、文字のエンコードやデコードに対応表を使います。

デコード用の対応表は、string 型の文字を Unicode 型の文字、整数 (Unicode 序数として解釈されます)、*None* ("未定義の対応付け" を意味し、エラーを引き起こします) のいずれかに対応付けなければなりません。

エンコード用の対応表は、Unicode 型の文字を string 型の文字、整数 (Latin-1 序数として解釈されます)、*None* ("未定義の対応付け" を意味し、エラーを引き起こします) のいずれかに対応付けなければなりません。

対応表のオブジェクトは、`__getitem__()` マップ型インタフェースをサポートしなければなりません。

ある文字の検索が `LookupError` によって失敗すると、その文字はそのままコピーされます。すなわち、その文字の序数値がそれぞれ Unicode または Latin-1 として解釈されます。このため、codec を実現するマップ型に入れる必要がある対応付け関係は、ある文字を別のコードポイントに対応付けるものだけです。

以下は mapping codec の API です:

*PyObject** **PyUnicode_DecodeCharmap** (const char*s, Py_ssize_tsize, *PyObject**mapping, const char*errors)

Return value: New reference. エンコードされた *size* バイトの文字列 *s* から *mapping* に指定されたオブジェクトを使って 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。もし、*mapping* が *NULL* だった場合、latin-1 でデコードされます。それ以外の場合では、*mapping* は byte に対する辞書マップ (訳注: *s* に含まれる文字の unsigned な値を int 型でキーとして、値として変換対象の Unicode 文字を表す Unicode 文字列になっているような辞書) か、ルックアップテーブルとして扱われる Unicode 文字列です。文字列 (訳注: *mapping* が Unicode 文字列として渡された場合) の長さより大きい byte 値や、(訳注: *mapping* にしたがって変換した結果が) U+FFFE "characters" になる Byte 値は、"定義されていない対応付け (undefined mapping)" として扱われます。

バージョン 2.4 で変更: *mapping* 引数として unicode が使えるようになりました。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode_EncodeCharmap** (const *Py_UNICODE**s, Py_ssize_tsize, *PyObject**mapping, 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 *mapping* に指定されたオブジェクトを使ってエンコードし、Python 文字列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。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*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 Python 文字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。

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.TranslateCharmap** (const *Py_UNICODE**s, Py_ssize_tsize, *PyObject**table,

const char*errors)

Return value: New reference. size で指定された長さを持つ *Py_UNICODE* バッファを、文字変換マップ 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。

mapping テーブルは、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。

対応付けテーブル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。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

Windows 用の MBCS codec

以下は MBCS codec の API です。この codec は現在のところ、Windows 上だけで利用でき、変換の実装には Win32 MBCS 変換機構 (Win32 MBCS converter) を使っています。MBCS (または DBCS) はエンコード方式の種類 (class) を表す言葉で、単一のエンコード方式を表すわけでないの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_tsize, 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。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode.DecodeMBCSStateful** (const char*s, intsize, const char*errors,

int*consumed)

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。

バージョン 2.5 で追加。

*PyObject** **PyUnicode.EncodeMBCS** (const *Py_UNICODE**s, Py_ssize_tsize, 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 文字列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。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*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 文字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。

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 オブジェクトが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_tmaxsplit)

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。

バージョン 2.5 で変更: この関数は以前は maxsplit の型に int を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyUnicode_Splitlines** (*PyObject**s, intkeepend)

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

Return value: New reference. 文字列に文字変換マップ 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。

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。

対応付けテーブル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_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_tstart, Py_ssize_tend, intdirection)`
`substr` が `str[start:end]` の末端 (`direction == -1` は先頭一致、`direction == 1` は末尾一致) だ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。

バージョン 2.5 で変更: この関数は以前は `start, end` の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`Py_ssize_t PyUnicode_Find (PyObject*str, PyObject*substr, Py_ssize_tstart, Py_ssize_tend, intdirection)`
`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。

バージョン 2.5 で変更: この関数は以前は `start, end` の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`Py_ssize_t PyUnicode_Count (PyObject*str, PyObject*substr, Py_ssize_tstart, Py_ssize_tend)`
`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。

バージョン 2.5 で変更: この関数は以前は `start, end` と戻り値の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`PyObject* PyUnicode_Replace (PyObject*str, PyObject*substr, PyObject*replstr, Py_ssize_tmaxcount)`
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。

バージョン 2.5 で変更: この関数は以前は `maxcount` の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`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。

`int PyUnicode_RichCompare (PyObject*left, PyObject*right, intop)`
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)

`Py_EQ` と `Py_NE` の比較は、引数から Unicode への変換が `UnicodeDecodeError` で失敗した時に、`UnicodeWarning` を発生する可能性があることに注意してください。

`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。

7.3.4 buffer オブジェクトと memoryview オブジェクト

C で実装された Python オブジェクトは、“バッファインタフェース (buffer interface)” と呼ばれる一連の関数を公開していることがあります。これらの関数は、あるオブジェクトのデータを生 (raw) のバイト列形式で公開するために使います。このオブジェクトの使い手は、バッファインタフェースを使うことで、オブジェクトをあらかじめコピーしておく必要なしに、オブジェクトのデータに直接アクセスできます。

バッファインタフェースをサポートするオブジェクトの例として、文字列型とアレイ (array) 型の二つがあります。文字列オブジェクトは、その内容をバッファインタフェースのバイト指向の形式で公開しています。アレイはその内容を旧スタイルバッファインターフェイス経由でしか公開できません。この制限は `memoryview` オブジェクトをアレイから構築出来る Python 3 には適用されません。アレイの要素は複数バイトの値になります。

バッファインタフェースの使い手の一例として、ファイルオブジェクトの `write()` メソッドがあります。バッファインタフェースを介してバイト列を公開しているオブジェクトは全て、ファイルへの書き出しができます。オブジェクトのバッファインタフェースを操作し、対象となるオブジェクトからデータを返させる `PyArg_ParseTuple()` には数多くのデータ書式化コードがあります。

バージョン 1.6 から、Python は Python レベルのバッファオブジェクトと、C 言語レベルのバッファ API を提供しており、任意のビルトイン型やユーザー定義型はその文字列表現を公開することができます。しかし、両方共、幾つかの欠点のために廃止予定扱いされていて、Python 3 では公式に削除され、新しい C 言語レベルのバッファ API と新しい Python レベルの `memoryview` という名前のオブジェクトに置き換えられています。

新しいバッファ API は Python 2.6 に逆移植されており、`memoryviews` オブジェクトは Python 2.7 に逆移植されています。古いバージョンとの互換性がなければ、古い API の代わりにこれらを使うことをおすすめします。

新スタイル `Py_buffer` 構造体

`Py_buffer`

void ***buf**

オブジェクトのメモリの開始位置へのポインタ

Py_ssize_t len

メモリのトータルサイズ [byte]

int readonly

バッファが読み込み専用かどうかを示す

const char *format

バッファを通してアクセスできる要素の形式を指定する、`struct` モジュールスタイル文法の、`NULL` 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。

int ndim

メモリが多次元配列を表している時の次元数。0 の場合、`strides` と `suboffsets` は `NULL` でなければなりません。

Py_ssize_t *shape

メモリが多次元配列を表しているとき、その形を示す長さ `ndim` の `Py_ssize_t` の配列。
`(((*shape)[0] * ... * (*shape)[ndims-1])*itemsizesize)` は `len` と等しくなければならないことに気をつけてください。

Py_ssize_t *strides

各次元で次の要素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。

Py_ssize_t *suboffsets

長さ `ndim` の `Py_ssize_t` の配列。`suboffset` の各数値が 0 以上であるとき、その次元に格納されているのはポインタで、`suboffset` の値はそのポインタの参照を解決するときに何バイトのオフセットを足すかを示しています。`suboffset` に負の数が格納されているときは、参照解決が不要であること (連続したメモリブロック内に直接配置されていること) を意味しています。

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。

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;
}
```

Py_ssize_t itemsizesize

これは共有メモリ上の各要素の `byte` 単位のサイズを格納する変数です。これは `PyBuffer_SizeFromFormat()` を使って計算できる値なので技術的には不要なのですが、

バッファを提供する側はフォーマット文字列を解析しなくてもこの情報を知っているでしょうし、バッファを受け取る側にとっては正しく解釈するのに必要な情報です。なので、要素サイズを格納するほうが便利ですし高速です。

`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解放するとき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ますでしょう。バッファを受け取る側は、この値を変更してはなりません。

バッファ関連関数

`int PyObject_CheckBuffer (PyObject*obj)`

obj が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、それ以外の場合に 0 を返します。

`int PyObject_GetBuffer (PyObject*obj, Py_buffer*view, intflags)`

obj を *Py_buffer* view へエクスポートします。これらの引数は `NULL` であってはなりません。 *flag* 引数は呼び出し側がどんなバッファを扱おうとしているのか、バッファ提供側がどんなバッファを返すことが許されているのかを示す、ビットフィールドです。バッファインタフェースは複雑なメモリ共有を可能にしていますが、呼び出し元はすべての複雑なバッファを扱えるとは限らず、バッファ提供側がシンプルなビューを提供できるならそれを利用したいとかもしれません。

バッファ提供側はすべての方法でメモリを共有できるとは限らず、呼び出し側に何かが不可能であることを伝えるためにエラーを発生させる必要があるかもしれません。その場合のエラーは、もしその問題を実際に引き起こしているのが別のエラーだったとしても、`BufferError` でなければなりません。バッファ提供側は *flag* の情報を使って *Py_buffer* 構造体のどのフィールドへの非デフォルト値の設定を省略したり、要求されたシンプルな view を提供できない場合はエラーを発生させたりすることができます。

成功したら 0 が、エラー時には -1 が返されます。

次のテーブルは、*flags* 引数を取りうる値です。

Flag	説明
<code>PyBUF_SIMPLE</code>	これはデフォルトの flag の状態です。結果のバッファは書き込み可能かもしれませんが、不可能かもしれません。データのフォーマットは unsigned byte とします。これは "スタンドアロン" のフラグ定数です。他の定数と ']' を取る必要はありません。提供側はこのような連続したバイト列のバッファを提供できない場合に、エラーを発生させるかもしれません。
<code>PyBUF_WRITEABLE</code>	結果のバッファは書き込み可能でなければなりません。書き込み不可能な場合はエラーを発生させます。
<code>PyBUF_STRIDES</code>	この値は <code>PyBUF_IND</code> を含みます。バッファは strides 情報を提供しなければなりません。(言い換えると、strides は NULL であってははいけません。) このフラグは、呼び出し元が、要素間に隙間のある不連続な配列を扱えるときに使われます。strides を扱うことは、自動的に shape も扱えることを要求されます。提供側は stride 形式のバッファを提供できないとき (例えば、suboffset が必要な場合) はエラーを発生させます。
<code>PyBUF_IND</code>	バッファは shape 情報を提供しなければなりません。メモリは C スタイルの並び (最後の次元が一番高速) だと仮定されます。提供側はこの種類の連続バッファを提供できない場合はエラーを発生させます。このフラグが指定されていない場合は shape は <code>NULL</code> になります。
<code>PyBUF_C_CONTIGUOUS</code> <code>PyBUF_F_CONTIGUOUS</code> <code>PyBUF_ANY_CONTIGUOUS</code>	これらのフラグは、返されるバッファの並びを指定します。それぞれ、C 並び (最後の次元が一番高速)、Fortran 並び (最初の次元が一番高速)、そのどちらでも、を意味します。これらのフラグは <code>PyBUF_STRIDES</code> を含んでおり、strides 情報が正しく格納されていることを保証します。
<code>PyBUF_INDIRECT</code>	このフラグは、返されるバッファが suboffsets 情報を含んでいることを示します。(suboffsets が不要無いたときは NULL でもかまいません。) このフラグは、バッファ利用側が suboffsets を使って参照されている間接配列を扱えるときに利用されます。このフラグは <code>PyBUF_STRIDES</code> を含みます。
<code>PyBUF_FORMAT</code>	返されるバッファは正しい format 情報を持っていないければなりません。このフラグは、バッファ利用側が実際に格納されているデータの '種類' をチェックするときに利用します。バッファ提供側は、要求された場合は常にこの情報を提供できるべきです。format が明示的に要求されていない場合は format は <code>NULL</code> ('B', unsigned byte を意味する) であるべきです。
<code>PyBUF_STRIDED</code>	(<code>PyBUF_STRIDES</code> <code>PyBUF_WRITEABLE</code>) と同じ
<code>PyBUF_STRIDED_RO</code>	(<code>PyBUF_STRIDES</code>) と同じ
<code>PyBUF_RECORDS</code>	(<code>PyBUF_STRIDES</code> <code>PyBUF_FORMAT</code> <code>PyBUF_WRITEABLE</code>) と同じ
<code>PyBUF_RECORDS_RO</code>	(<code>PyBUF_STRIDES</code> <code>PyBUF_FORMAT</code>) と同じ
<code>PyBUF_FULL</code>	(<code>PyBUF_INDIRECT</code> <code>PyBUF_FORMAT</code> <code>PyBUF_WRITEABLE</code>) と同じ
<code>PyBUF_FULL_RO</code>	(<code>PyBUF_INDIRECT</code> <code>PyBUF_FORMAT</code>) と同じ
<code>PyBUF_CONTIG</code>	(<code>PyBUF_IND</code> <code>PyBUF_WRITEABLE</code>) と同じ
<code>PyBUF_CONTIG_RO</code>	(<code>PyBUF_IND</code>) と同じ

`void PyBuffer_Release (Py_buffer*view)`

`view` バッファを解放します。バッファが利用されなくなったときに、そのメモリを解放できるようにこ

の関数を呼び出すべきです。

`Py_ssize_t PyBuffer_SizeFromFormat (const char*)`

`itemsize` の値を `format` から計算して返します。

`int PyBuffer_IsContiguous (Py_buffer*view, charfortran)`

`view` で定義されているメモリが、C スタイル (`fortran == 'C'`) のときか、Fortran スタイル (`fortran == 'F'`) のときか、そのいずれか (`fortran == '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。

`void PyBuffer_FillContiguousStrides (intndims, Py_ssize_t*shape, Py_ssize_t*strides,`

`intitemsize, charfortran)`

`strides` 配列を、`itemsize` の大きさの要素がバイト単位で連続した、`shape` の形をした (`fortran` 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*obj, void*buf, Py_ssize_tlen, intreadonly,`

`intinfoflags)`

バッファ提供側が与えられた長さの “unsigned bytes” の連続した 1 つのメモリブロックしか提供できないものとして、`view` バッファ情報構造体を正しく埋める。成功したら 0 を、エラー時には (例外を発生させつつ) -1 を返す。

memoryview オブジェクト

バージョン 2.7 で追加。

`memoryview` オブジェクトは、新しい、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へのインタフェースです。

`PyObject *PyMemoryView_FromObject (PyObject*obj)`

新しいバッファインタフェースを定義しているオブジェクトから `memoryview` オブジェクトを作ります。

`PyObject *PyMemoryView_FromBuffer (Py_buffer*view)`

buffer-info 構造体 `view` をラップする `memoryview` オブジェクトを作ります。作られた `memoryview` オブジェクトはバッファを所有することになるので、`view` を解放してはいけません。このバッファは `memoryview` オブジェクトが削除されるときに解放されます。

`PyObject *PyMemoryView_GetContiguous (PyObject*obj, intbuffertype, charorder)`

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*obj)`

与えられたオブジェクトにラップされた buffer-info 構造体へのポインタを返します。オブジェクトは

memoryview インスタンスでなければなりません。このマクロはオブジェクトの型をチェックしないので、呼び出し側で保証しなければクラッシュする可能性があります。

旧スタイルバッファオブジェクト

古いバッファインタフェースに関するより詳しい情報は、“バッファオブジェクト構造体” 節 ([バッファオブジェクト構造体 \(buffer object structure\)](#) 節) の、[PyBufferProcs](#) の説明のところにあります。

”バッファオブジェクト” はヘッダファイル `bufferobject.h` の中で定義されています (このファイルは `Python.h` がインクルードしています)。バッファオブジェクトは、Python プログラミングのレベルからは文字列オブジェクトと非常によく似ているように見えます: スライス、インデックス指定、結合、その他標準の文字列操作をサポートしています。しかし、バッファオブジェクトのデータは二つのデータソース: 何らかのメモリブロックか、バッファインタフェースを公開している別のオブジェクト、のいずれかに由来しています。

バッファオブジェクトは、他のオブジェクトのバッファインタフェースから Python プログラマにデータを公開する方法として便利です。バッファオブジェクトはゼロコピーなスライス機構 (zero-copy slicing mechanism) としても使われます。ブロックメモリを参照するというバッファオブジェクトの機能を使うことで、任意のデータをきわめて簡単に Python プログラマに公開できます。メモリブロックは巨大でもかまいませんし、C 拡張モジュール内の定数配列でもかまいません。また、オペレーティングシステムライブラリ側に渡す前の、操作用の生のブロックメモリでもかまいませんし、構造化されたデータをネイティブのメモリ配置形式でやりとりするためにも使えます。

`PyBufferObject`

この `PyObject` のサブタイプはバッファオブジェクトを表現します。

`PyTypeObject PyBuffer_Type`

Python バッファ型 (buffer type) を表現する `PyTypeObject` です; Python レイヤにおける `buffer` や `types.BufferType` と同じオブジェクトです。

`int Py_END_OF_BUFFER`

この定数は、`PyBuffer_FromObject()` や `PyBuffer_FromReadWriteObject()` に `size` パラメタとして渡します。このパラメタを渡すと、`PyBufferObject` は指定された `offset` からバッファの終わりまでを `base` オブジェクトとして参照します。このパラメタを使うことで、関数の呼び出し側が `base` オブジェクトのサイズを調べる必要がなくなります。

`int PyBuffer_Check(PyObject*p)`

引数が `PyBuffer_Type` 型のときに真を返します。

`PyObject* PyBuffer_FromObject(PyObject*base, Py_ssize_toffset, Py_ssize_tsize)`

Return value: *New reference.* 新たな読み出し専用バッファオブジェクトを返します。 `base` が読み出し専用バッファに必要なバッファプロトコルをサポートしていない場合や、厳密に一つのバッファセグメントを提供していない場合には `TypeError` を送出し、`offset` がゼロ以下の場合には `ValueError` を送出します。バッファオブジェクトは `base` オブジェクトに対する参照を保持し、バッファオブジェクトの内容は `base` オブジェクトの `offset` から `size` バイトのバッファインタフェースへの参照になります。 `size` が `Py_END_OF_BUFFER` の場合、新たに作成するバッファオブジェクトの内容は `base` から公開されているバッファの末尾までになります。

バージョン 2.5 で変更: この関数は以前は *offset, size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

***PyObject** PyBuffer_FromReadWriteObject (*PyObject**base, Py_ssize_toffset, Py_ssize_tsize)**

Return value: New reference. 新たな書き込み可能バッファオブジェクトを返します。パラメタおよび例外は *PyBuffer_FromObject()* と同じです。base オブジェクトが書き込み可能バッファに必要なバッファプロトコルを公開していない場合、`TypeError` を送出します。

バージョン 2.5 で変更: この関数は以前は *offset, size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

***PyObject** PyBuffer_FromMemory (void*ptr, Py_ssize_tsize)**

Return value: New reference. メモリ上の指定された場所から指定されたサイズのデータを読み出せる、新たな読み出し専用バッファオブジェクトを返します。この関数が返すバッファオブジェクトが存続する間、*ptr* で与えられたメモリバッファがデアロケートされないようにするのは呼び出し側の責任です。*size* がゼロ以下の場合には `ValueError` を送出します。*size* には `Py_END_OF_BUFFER` を指定してはいけません; 指定すると、`ValueError` を送出します。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

***PyObject** PyBuffer_FromReadWriteMemory (void*ptr, Py_ssize_tsize)**

Return value: New reference. *PyBuffer_FromMemory()* に似ていますが、書き込み可能なバッファを返します。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

***PyObject** PyBuffer_New (Py_ssize_tsize)**

Return value: New reference. *size* バイトのメモリバッファを独自に維持する新たな書き込み可能バッファオブジェクトを返します。*size* がゼロまたは正の値でない場合、`ValueError` を送出します。(*PyObject_AsWriteBuffer()* が返すような) メモリバッファは特に整列されていないので注意して下さい。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

7.3.5 タプルオブジェクト (tuple object)

PyTupleObject

この *PyObject* のサブタイプは Python のタプルオブジェクトを表現します。

***PyTypeObject* PyTuple_Type**

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` や `types.TupleType` と同じオブジェクトです。

int PyTuple_Check (*PyObjectp)**

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。

バージョン 2.2 で変更: サブタイプを引数にとれるようになりました。

`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。

バージョン 2.2 で追加。

`PyObject* PyTuple_New (Py_ssize_tlen)`

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。

バージョン 2.5 で変更: この関数は以前は *len* の型に `int` を利用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

`PyObject* PyTuple_Pack (Py_ssize_tn, ...)`

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。

バージョン 2.4 で追加。

バージョン 2.5 で変更: この関数は以前は *n* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`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。

バージョン 2.5 で変更: この関数は以前は `int` を返していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

`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。

バージョン 2.5 で変更: この関数は以前は `int` を返していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

`PyObject* PyTuple_GetItem (PyObject*p, Py_ssize_tpos)`

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.

バージョン 2.5 で変更: この関数は以前は *pos* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`PyObject* PyTuple_GET_ITEM (PyObject*p, Py_ssize_tpos)`

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。

バージョン 2.5 で変更: この関数は以前は *pos* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*PyObject** **PyTuple_GetSlice** (*PyObject***p*, *Py_ssize_t**low*, *Py_ssize_t**high*)

Return value: *New reference.* Return the slice of the tuple pointed to by *p* between *low* and *high*, or *NULL* on failure. This is the equivalent of the Python expression `p[low:high]`. Indexing from the end of the list is not supported.

バージョン 2.5 で変更: この関数は以前は *low*, *high* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

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

Insert a reference to object *o* at position *pos* of the tuple pointed to by *p*. Return 0 on success. If *pos* is out of bounds, return -1 and set an `IndexError` exception.

注釈: This function "steals" a reference to *o* and discards a reference to an item already in the tuple at the affected position.

バージョン 2.5 で変更: この関数は以前は *pos* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`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.

バージョン 2.5 で変更: この関数は以前は *pos* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`int` **PyTuple_Resize** (*PyObject****p*, *Py_ssize_t**newsize*)

タプルをリサイズする際に使えます。 *newsize* はタプルの新たな長さです。タプルは変更不能なオブジェクトということになっているので、この関数はこのオブジェクトに対してただ一つしか参照がない時以外には使ってはなりません。タプルがコード中の他の部分ですでに参照されている場合には、この関数を使ってはなりません。タプルは常に指定サイズの末尾まで伸縮します。成功した場合には 0 を返します。クライアントコードは、 **p* の値が呼び出し前と同じになると期待してはなりません。 **p* が置き換えられた場合、オリジナルの **p* は破壊されます。失敗すると -1 を返し、 **p* を *NULL* に設定して、 `MemoryError` または `SystemError` を送出します。

バージョン 2.2 で変更: 使われていなかった三つ目のパラメタ、 *last_is_sticky* を削除しました。

バージョン 2.5 で変更: この関数は以前は *newsize* の型に `int` を利用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

`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。

バージョン 2.6 で追加。

7.3.6 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。

バージョン 2.2 で変更: サブタイプを引数にとれるようになりました。

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。

バージョン 2.2 で追加。

*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。

バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

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。

バージョン 2.5 で変更: この関数は以前は *int* を返していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

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。

バージョン 2.5 で変更: このマクロは以前は *int* を返していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

*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。

バージョン 2.5 で変更: この関数は以前は *index* の型に `int` を利用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

*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。

バージョン 2.5 で変更: このマクロは以前は *i* の型に `int` を利用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

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

Set the item at index *index* in list to *item*. Return 0 on success. If *index* is out of bounds, return -1 and set an `IndexError` exception.

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。

バージョン 2.5 で変更: この関数は以前は *index* の型に `int` を利用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

`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。

バージョン 2.5 で変更: このマクロは以前は *i* の型に `int` を利用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

`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。

バージョン 2.5 で変更: この関数は以前は *index* の型に `int` を利用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

`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. Return a list of the objects in *list* containing the objects between *low* and *high*. Return `NULL` and set an exception if unsuccessful. Analogous to `list[low:high]`. Indexing from the end of the list is not supported.

バージョン 2.5 で変更: この関数は以前は *low* と *high* に `int *` 型を使用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

`int PyList_SetSlice (PyObject*list, Py_ssize_tlow, Py_ssize_thigh, PyObject*itemlist)`

Set the slice of *list* between *low* and *high* to the contents of *itemlist*. Analogous to `list[low:high] = itemlist`. The *itemlist* may be *NULL*, indicating the assignment of an empty list (slice deletion). Return 0 on success, -1 on failure. Indexing from the end of the list is not supported.

バージョン 2.5 で変更: この関数は以前は *low* と *high* に `int *` 型を使用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

`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。

7.4 マップ型オブジェクト (mapping object)

7.4.1 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 プログラムには `dict` および `types.DictType` として公開されています。

`int PyDict_Check (PyObject*p)`

引数が `PyDictObject` のときに真を返します。

バージョン 2.2 で変更: サブタイプを引数にとれるようになりました。

`int PyDict_CheckExact (PyObject*p)`

p が辞書型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。

バージョン 2.4 で追加。

`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*dict)`

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プロキ

シオブジェクト (proxy object) 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) のクラス辞書を変更させないためにプロキシを作成するために使われます。

バージョン 2.2 で追加。

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。

バージョン 2.4 で追加。

*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。

バージョン 1.6 で追加。

int **PyDict_SetItem** (*PyObject**p, *PyObject**key, *PyObject**val)

辞書 *p* に、*key* をキーとして値 *value* 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。

int **PyDict_SetItemString** (*PyObject**p, const char*key, *PyObject**val)

辞書 *p* に、*key* をキーとして値 *value* を挿入します。*key* は `char*` 型でなければなりません。キーオブジェクトは `PyString_FromString(key)` で生成されます。成功した場合には 0 を、失敗した場合には -1 を返します。

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

辞書 *p* から *key* をキーとするエントリを除去します。*key* はハッシュ可能でなければなりません; ハッシュ可能でない場合、`TypeError` を送出します。成功した場合には 0 を、失敗した場合には -1 を返します。

int **PyDict_DelItemString** (*PyObject**p, 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。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。

*PyObject** **PyDict_GetItemString** (*PyObject**p, const char*key)

Return value: Borrowed reference. `PyDict_GetItem()` と同じですが、*key* は *PyObject** ではなく `char*` で指定します。

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

Return value: New reference. 辞書オブジェクトのメソッド `dict.items()` のように、辞書 *p* 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.* 辞書オブジェクトのメソッド `dict.keys()` のように、辞書 *p* 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.* 辞書オブジェクトのメソッド `dict.values()` のように、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。

バージョン 2.5 で変更: この関数は以前は `int` を返していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

int **PyDict_Next** (*PyObject** *p*, *Py_ssize_t** *ppos*, *PyObject*** *pkey*, *PyObject*** *pvalue*)

辞書 *p* 内の全てのキー/値のペアにわたる反復処理を行います。 *ppos* が参照している *Py_ssize_t* 型は、この関数で反復処理を開始する際に、最初に関数を呼び出すよりも前に 0 に初期化しておかなければなりません; この関数は辞書内の各ペアを取り上げるごとに真を返し、全てのペアを取り上げたことが分かると偽を返します。パラメタ *pkey* および *pvalue* には、それぞれ辞書の各々のキーと値を指すポインタか、または *NULL* が入ります。この関数から返される参照はすべて借りた参照になります。反復処理中に *ppos* を変更してはなりません。この値は内部的な辞書構造体のオフセットを表現しており、構造体はスパースなので、オフセットの値に一貫性がないためです。

例えば:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}
```

反復処理中に辞書 *p* を変更してはなりません。(Python 2.1 からは) 辞書を反復処理する際に、キーに対応する値を変更しても大丈夫になりましたが、キーの集合を変更しないことが前提です。以下に例を示します:

```
PyObject *key, *value;
Py_ssize_t pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    int i = PyInt_AS_LONG(value) + 1;
    PyObject *o = PyInt_FromLong(i);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
}
```

(次のページに続く)

(前のページからの続き)

```
Py_DECREF(o);
}
```

バージョン 2.5 で変更: この関数は以前は *ppos* の型に `int *` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

`int PyDict_Merge(PyObject*a, PyObject*b, intoverride)`

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* が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。

バージョン 2.2 で追加.

`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。

バージョン 2.2 で追加.

`int PyDict_MergeFromSeq2(PyObject*a, PyObject*seq2, intoverride)`

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
```

バージョン 2.2 で追加.

7.5 その他のオブジェクト

7.5.1 クラスとインスタンスオブジェクト

ここで説明されているクラスオブジェクトは旧スタイルクラスのもので、Python 3 では削除されることに注意してください。新たに型を拡張モジュールで作成する場合は、 `type` オブジェクトを扱うべきです。(型オブジェクトを参照)

PyClassObject

ビルトインクラスを表すためのオブジェクトの構造体

***PyObject** PyClass_Type**

クラスオブジェクトのための type オブジェクト。Python レイヤーの、`types.ClassType` と同じオブジェクト。

int PyClass_Check (*PyObjecto)**

標準のクラスオブジェクトから派生した type のインスタンスだった場合を含めて、オブジェクト *o* がクラスオブジェクトだった場合に真を返します。それ以外の場合は偽を返します。

int PyClass_IsSubclass (*PyObjectclass, *PyObject**base)**

class が *base* のサブクラスだった場合に真を返します。それ以外の場合は偽を返します。

インスタンスオブジェクト固有の関数はきわめてわずかです。

***PyTypeObject* PyInstance_Type**

クラスインスタンスの型オブジェクトです。

int PyInstance_Check (*PyObjectobj)**

obj がインスタンスの場合に真を返します。

***PyObject** PyInstance_New (*PyObject**class, *PyObject**arg, *PyObject**kw)**

Return value: *New reference.* 特定のクラスの新たなインスタンスを生成します。パラメタ *arg* および *kw* はそれぞれオブジェクトのコンストラクタに渡す位置依存パラメータおよびキーワードパラメータとして使われます。

***PyObject** PyInstance_NewRaw (*PyObject**class, *PyObject**dict)**

Return value: *New reference.* 特定のクラスの新たなインスタンスを、コンストラクタを呼ばずに生成します。 *class* は新たに作成するオブジェクトのクラスです。 *dict* パラメータはオブジェクトの `__dict__` に使われます; *dict* が `NULL` なら、インスタンス用に新たな辞書が作成されます。

7.5.2 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 (*PyObjecto)**

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` であってはいけません。

***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、名前および `__module__` はコードオブジェクトから取得されます。引数のデフォルト値やクロージャは `NULL` にセットされます。

*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。

7.5.3 メソッドオブジェクト

C API にはメソッドオブジェクトを扱うのに便利な関数があります。

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` できません。

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

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時に呼び出されるオブジェクトです。このメソッドをインスタンスに束縛 (bind) したい場合、*self* をインスタンス自体にして、*class* を *self* のクラスにしなければなりません。それ以外の場合は *self* を *NULL* に、*class* を非束縛メソッドを提供しているクラスにしなければなりません。

*PyObject** **PyMethod_Class** (*PyObject**meth)

Return value: Borrowed reference. メソッドオブジェクト *meth* を生成したクラスオブジェクトを返します; インスタンスがメソッドオブジェクトを生成した場合、戻り値はインスタンスのクラスになります。

*PyObject** **PyMethod_GET_CLASS** (*PyObject**meth)

Return value: Borrowed reference. *PyMethod_Class()* をマクロで実装したバージョンで、エラーチェックを行いません。

*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. *meth* が束縛メソッドの場合には、メソッドに関連付けられているインスタンスを返します。それ以外の場合には *NULL* 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。

バージョン 2.6 で追加。

7.5.4 ファイルオブジェクト

Python の組み込みファイルオブジェクトは、全て標準 C ライブラリの FILE* サポートの上に実装されています。以下の詳細説明は一実装に関するもので、将来の Python のリリースで変更されるかもしれません。

PyFileObject

この *PyObject* のサブタイプは Python のファイル型オブジェクトを表現します。

PyTypeObject **PyFile_Type**

この *PyTypeObject* のインスタンスは Python のファイル型を表現します。このオブジェクトは *file* および *types.FileType* として Python プログラムで公開されています。

int **PyFile_Check** (*PyObject**p)

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

バージョン 2.2 で変更: サブタイプを引数にとれるようになりました。

int **PyFile_CheckExact** (*PyObject**p)

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

バージョン 2.2 で追加。

*PyObject** **PyFile_FromString** (char*filename, char*mode)

Return value: New reference. 成功すると、filename に指定した名前のファイルを mode に指定したファイルモードで開いて得た新たなファイルオブジェクトを返します。mode のセマンティクスは標準 C ルーチン fopen() と同じです。失敗すると NULL を返します。

*PyObject** **PyFile_FromFile** (FILE*fp, char*name, char*mode, int (*close)(FILE*))

Return value: New reference. すでに開かれている標準 C ファイルポインタ fp から新たな *PyFileObject* を生成します。この関数で生成したファイルオブジェクトは、閉じる際に close に指定した関数を呼び出します。失敗すると close を使ってファイルをクローズし、NULL を返します。close の指定は任意で、NULL にもできます。(—訳注: 失敗時に close を呼び出すのは [issue #7732](#) に関係した仕様です。今ここで説明している「失敗時」はコードを読む限り PyString_FromString の失敗時だけです、今のところ。—)

FILE* **PyFile_AsFile** (*PyObject**p)

p に関連付けられたファイルオブジェクトを FILE* で返します。

呼び出し側が *GIL* を解放している間もこの関数が返した FILE* オブジェクトを使うのであれば、以下に解説されている *PyFile_IncUseCount*() と *PyFile_DecUseCount*() 関数を適切に呼び出さなければなりません。

void **PyFile_IncUseCount** (*PyFileObject**p)

PyFileObject 内部の、FILE* が使用中であることを示す使用数カウントをインクリメントします。これは、別のスレッドで使用中の FILE* に対して Python が fclose() を呼び出すことを防ぎます。この関数の呼び出し側は、FILE* を使い終わったときに必ず *PyFile_DecUseCount*() を呼び出さなければなりません。そうしなければ、Python はそのファイルオブジェクトを永遠に閉じません。

この関数を呼び出すときは、*GIL* を取得していなければなりません。

例えば、*PyFile_AsFile*() を呼び出した後、*GIL* を解放する前にこの関数を呼び出します。

```
FILE *fp = PyFile_AsFile(p);
PyFile_IncUseCount(p);
/* ... */
Py_BEGIN_ALLOW_THREADS
do_something(fp);
Py_END_ALLOW_THREADS
/* ... */
PyFile_DecUseCount(p);
```

バージョン 2.6 で追加。

void **PyFile_DecUseCount** (*PyFileObject**p)

PyFileObject 内部の、FILE* が使用中であることを示す unlocked_count メンバーをデクリメントして、呼び出し元が FILE* を使い終わったことを示します。これは、先に行った *PyFile_IncUseCount*() の呼び出しを取り消すためだけに呼び出されるでしょう。

この関数を呼び出すときは、*GIL* を取得していなければなりません。(上の例を参照してください)

バージョン 2.6 で追加.

*PyObject** **PyFile_GetLine** (*PyObject***p*, *intn*)

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。

*PyObject** **PyFile_Name** (*PyObject***p*)

Return value: *Borrowed reference.* *p* に指定したファイルの名前を文字列オブジェクトで返します。

`void` **PyFile_SetBufSize** (*PyFileObject***p*, *intn*)

`setvbuf()` があるシステムでのみ利用できます。この関数を呼び出してよいのはファイルオブジェクトの生成直後のみです。

`int` **PyFile_SetEncoding** (*PyFileObject***p*, *const char***enc*)

Unicode オブジェクトをファイルに出力するときのエンコーディングを *enc* にします。成功すると 1 を、失敗すると 0 を返します。

バージョン 2.3 で追加.

`int` **PyFile_SetEncodingAndErrors** (*PyFileObject***p*, *const char***enc*, **errors*)

Unicode オブジェクトをファイルに出力するときのエンコーディングを *enc* に設定し、そのエラーモードを *err* に設定します。成功すると 1 を、失敗すると 0 を返します。

バージョン 2.6 で追加.

`int` **PyFile_SoftSpace** (*PyObject***p*, *intnewflag*)

この関数はインタプリタの内部的な利用のために存在します。この関数は *p* の `softspace` 属性を *newflag* に設定し、以前の設定値を返します。この関数を正しく動作させるために、*p* がファイルオブジェクトである必然性はありません; 任意のオブジェクトをサポートします (`softspace` 属性が設定されているかどうかのみが問題だと思ってください)。この関数は全てのエラーを解消し、属性値が存在しない場合や属性値を取得する際にエラーが生じると、0 を以前の値として返します。この関数からはエラーを検出できませんが、そもそもそういう必要はありません。

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

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。

7.5.5 モジュールオブジェクト (module object)

モジュールオブジェクト固有の関数は数個しかありません。

PyObject 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。

バージョン 2.2 で変更: サブタイプを引数にとれるようになりました。

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*。

バージョン 2.2 で追加。

*PyObject** PyModule_New (const char*name)

Return value: *New reference.* `__name__` 属性が *name* に設定された新たなモジュールオブジェクトを返します。モジュールの `__doc__` および `__name__` 属性だけに値が入っています; `__file__` 属性に値を入れるのは呼び出し側の責任です。

*PyObject** PyModule_GetDict (*PyObject**module)

Return value: *Borrowed reference.* *module* の名前空間を実現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__` と同じです。この関数が失敗することはありません。拡張モジュールでは、この関数で得たモジュールの `__dict__` を直接いじるより、他の `PyModule_*` () および `PyObject_*` () 関数を使うよう勧めます。

char* PyModule_GetName (*PyObject**module)

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。

char* PyModule_GetFilename (*PyObject**module)

module をロードするために使ったファイルの名前を、*module* の `__file__` 属性から調べて返します。`__file__` が定義され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。

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宜関数です。これは *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。

バージョン 2.0 で追加。

int PyModule_AddIntConstant (*PyObject**module, const char*name, longvalue)

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。

バージョン 2.0 で追加.

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。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。

バージョン 2.0 で追加.

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。

バージョン 2.6 で追加.

int **PyModule_AddStringMacro** (*PyObject**module, macro)

文字列定数を *module* に追加します。

バージョン 2.6 で追加.

7.5.6 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。

バージョン 2.2 で追加.

int **PySeqIter_Check** (op)

PySeqIter_Type の型が *op* のときに真を返します。

バージョン 2.2 で追加.

*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。

バージョン 2.2 で追加.

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。 `iter()` built-in function.

バージョン 2.2 で追加.

int **PyCallIter_Check** (op)

op の型が *PyCallIter_Type* のときに真を返します。

バージョン 2.2 で追加.

*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。

バージョン 2.2 で追加.

7.5.7 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。

バージョン 2.2 で追加.

*PyObject** **PyDescr_NewGetSet** (*PyTypeObject**type, struct *PyGetSetDef**getset)

Return value: *New reference.* バージョン 2.2 で追加.

*PyObject** **PyDescr_NewMember** (*PyTypeObject**type, struct *PyMemberDef**meth)

Return value: *New reference.* バージョン 2.2 で追加.

*PyObject** **PyDescr_NewMethod** (*PyTypeObject**type, struct *PyMethodDef**meth)

Return value: *New reference.* バージョン 2.2 で追加.

*PyObject** **PyDescr_NewWrapper** (*PyTypeObject**type, struct wrapperbase*wrapper, void*wrapped)

Return value: *New reference.* バージョン 2.2 で追加.

*PyObject** **PyDescr_NewClassMethod** (*PyTypeObject**type, *PyMethodDef**method)

Return value: *New reference.* バージョン 2.3 で追加.

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。

バージョン 2.2 で追加.

*PyObject** **PyWrapper_New** (*PyObject**, *PyObject**)

Return value: *New reference.* バージョン 2.2 で追加.

7.5.8 スライスオブジェクト (slice object)

PyObject **PySlice_Type**

スライスオブジェクトの型オブジェクトです。 `slice` や `types.SliceType` と同じです。

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 (*PySliceObject** *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一つが `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)。

おそらくこの関数を使う気にはならないでしょう。バージョン 2.3 以前の Python でスライスオブジェクトを使いたいのなら、 `PySlice_GetIndicesEx()` のソースを適切に名前変更して自分の拡張モジュールのソースコード内に組み込むとよいでしょう。

バージョン 2.5 で変更: この関数は以前は *length* の型に `int` を、 *start*, *stop*, *step* の型に `int *` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

int PySlice_GetIndicesEx (*PySliceObject** *slice*, *Py_ssize_t* *length*, *Py_ssize_t** *start*, *Py_ssize_t** *stop*, *Py_ssize_t** *step*, *Py_ssize_t** *slicelength*)

`PySlice_GetIndices()` の便利な代替です。 *slice* における、 *start*, *stop* および *step* のインデクス値を取得をします。シーケンスの長さを *length*、スライスの長さを *slicelength* に格納します。境界外のインデクスは通常のスライスと一貫した方法でクリップされます。

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。

バージョン 2.3 で追加。

バージョン 2.5 で変更: この関数は以前は *length* の型に `int` を、 *start*, *stop*, *step*, *slicelength* の型に `int *` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

7.5.9 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

ントの点では、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。

7.5.10 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。

バージョン 2.2 で追加.

int **PyWeakref_CheckRef** (ob)

ob が参照オブジェクトの場合に真を返します。

バージョン 2.2 で追加.

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。

バージョン 2.2 で追加.

*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。

バージョン 2.2 で追加.

*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。

バージョン 2.2 で追加.

*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。

バージョン 2.2 で追加。

警告: 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。

バージョン 2.2 で追加。

7.5.11 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 2.7 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` であってはなりません。

失敗した場合、例外を設定して `NULL` を返します。

`name` 文字列は `NULL` か、有効な C 文字列へのポインタです。 `NULL` で無い場合、この文字列は少なくともカプセルより長く生存する必要があります。 (`destructor` の中で解放することは許可されています)

destructor が *NULL* で無い場合、カプセルが削除されるときにそのカプセルを引数として呼び出されます。

このカプセルがモジュールの属性として保存される場合、*name* は `module.attribute` 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。失敗した場合は、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。失敗した場合は、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。失敗した場合は、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, intno_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。失敗した場合は、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_Destructordestructor)`

`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。

7.5.12 C オブジェクト (CObject)

警告: CObject API は Python 2.7 から非推奨になりました。新しい [カプセル API](#) へ移行してください。

PyCObject

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。

`int PyCObject_Check(PyObject*p)`

引数が `PyCObject` の場合に真を返します。

*PyObject** **PyCObject_FromVoidPtr** (void*cobj, void (*destr)(void *))

Return value: New reference. void* cobj から *PyCObject* を生成します。関数 *destr* が *NULL* でない場合、オブジェクトが回収される際に呼び出されます。

*PyObject** **PyCObject_FromVoidPtrAndDesc** (void*cobj, void*desc, void (*destr)(void *, void *))

Return value: New reference. void* cobj から *PyCObject* を生成します。関数 *destr* が *NULL* でない場合、オブジェクトが回収される際に呼び出されます。引数 *desc* を使って、デストラクタ関数に追加のコールバックデータを渡せます。

void* **PyCObject_AsVoidPtr** (*PyObject**self)

PyCObject オブジェクト *self* を生成するのに用いたオブジェクト void * を返します。

void* **PyCObject_GetDesc** (*PyObject**self)

PyCObject オブジェクト *self* を生成するのに用いたコールバックデータ void * を返します。

int **PyCObject_SetVoidPtr** (*PyObject**self, void*cobj)

self 内の void ポインタに *cobj* をセットします。 *PyCObject* にデストラクタが関連づけられていてはなりません。成功すると真値を返し、失敗すると偽値を返します。

7.5.13 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。

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; もし *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。

7.5.14 ジェネレータオブジェクト

ジェネレータ (generator) オブジェクトは、Python がジェネレータ型イテレータを実装するため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()* で明示的に生成されることはなく、値を逐次生成するような関数に対してイテレーションを行うときに生成されます。

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** (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** (ob)

ob の型が *PyGen_Type* の場合に真を返します。*ob* が *NULL* であってはなりません。

*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。パラメタが *NULL* であってはなりません。

7.5.15 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。

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。

バージョン 2.4 で追加.

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。

バージョン 2.4 で追加.

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。

バージョン 2.4 で追加.

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。

バージョン 2.4 で追加.

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。

バージョン 2.4 で追加.

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。

バージョン 2.4 で追加.

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。

バージョン 2.4 で追加.

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。

バージョン 2.4 で追加.

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。

バージョン 2.4 で追加.

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。

バージョン 2.4 で追加.

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** (intyear, intmonth, intday)

Return value: New reference. 指定された年、月、日の `datetime.date` オブジェクトを返します。

バージョン 2.4 で追加.

*PyObject** **PyDateTime_FromDateAndTime** (intyear, intmonth, intday, inthour, intminute, intsecond,
intusecond)

Return value: New reference. 指定された年、月、日、時、分、秒、マイクロ秒の `datetime.datetime` オブジェクトを返します。

バージョン 2.4 で追加.

*PyObject** **PyTime_FromTime** (inthour, intminute, intsecond, intusecond)

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。

バージョン 2.4 で追加.

*PyObject** **PyDelta_FromDSU** (intdays, intseconds, intuseconds)

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。

バージョン 2.4 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。

バージョン 2.4 で追加.

int **PyDateTime_GET_MONTH** (PyDateTime_Date*o)

月を 1 から 12 の間の整数で返します。

バージョン 2.4 で追加.

int **PyDateTime_GET_DAY** (PyDateTime_Date*o)

日を 1 から 31 の間の整数で返します。

バージョン 2.4 で追加.

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。

バージョン 2.4 で追加.

int **PyDateTime_DATE_GET_MINUTE** (PyDateTime_DateTime*o)

分を 0 から 59 の間の整数で返します。

バージョン 2.4 で追加.

int **PyDateTime_DATE_GET_SECOND** (PyDateTime_DateTime*o)

秒を 0 から 59 の間の整数で返します。

バージョン 2.4 で追加.

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。

バージョン 2.4 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。

バージョン 2.4 で追加.

int **PyDateTime_TIME_GET_MINUTE** (PyDateTime_Time*o)

分を 0 から 59 の間の整数で返します。

バージョン 2.4 で追加.

int **PyDateTime_TIME_GET_SECOND** (PyDateTime_Time*o)

秒を 0 から 59 の間の整数で返します。

バージョン 2.4 で追加.

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。

バージョン 2.4 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。

バージョン 2.4 で追加.

*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。

バージョン 2.4 で追加。

7.5.16 Set オブジェクト

バージョン 2.5 で追加。

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ト可能な Python オブジェクトに対して動作します。

`int` **PySet_Check** (*PyObject**p)

p が `set` かそのサブタイプのオブジェクトであるときに真を返します。

バージョン 2.6 で追加。

`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真を返します。

バージョン 2.6 で追加。

`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* がイテレート可能で無い場合は、 `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無い場合は、 `TypeError` を送出します。

バージョン 2.6 で変更: 完全に新しい `frozenset` オブジェクトを返すことが保証されるようになりました。以前は、大きさがゼロの `frozenset` はシングルトンでした。これによりまっさらな `frozenset` から `PySet_Add()` を使って組み立てることが出来ます。

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` 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。

バージョン 2.5 で変更: この関数は以前は `int` を返していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

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` のインスタンスに使わないで下さい。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なら、 `TypeError` を送出します。 `set` を大きくする余裕が無い場合は、 `MemoryError` 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無い場合は、 `SystemError` を送出します。

バージョン 2.6 で変更: `frozenset` やそのサブタイプのインスタンスに対して利用できるようになりました。 `PyTuple_SetItem()` のように、新しい `frozenset` を他のコードに渡すまえに内容を追加するためのに使うことができます。

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* の中の要素のどれかに対する新しい参照を返し、そのオブジェクトを *set* から削除します。失敗したら `NULL` を返します。*set* が空の場合には `KeyError` を送出します。*set* が `set` とそのサブタイプのインスタンスでない場合は、`SystemError` を送出します。

`int PySet_Clear (PyObject*set)`

set を空にします。

7.5.17 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 (PyObject*co)`

co 内の自由変数 (free variables) の数を返します。

*PyCodeObject** **PyCode_New** (intargcount, intnlocals, intstacksize, intflags, *PyObject**code, *PyObject**consts, *PyObject**names, *PyObject**varnames, *PyObject**freevars, *PyObject**cellvars, *PyObject**filename, *PyObject**name, intfirstlineno, *PyObject**lnotab)

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ことがあります。

`int PyCode_NewEmpty` (const char*filename, const char*funcname, intfirstlineno)

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。

第 8 章

初期化 (initialization)、終了処理 (finalization)、スレッド

8.1 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ねばなりません。ただし、`Py_SetProgramName()`、`Py_SetPythonHome()`、`PyEval_InitThreads()`、`PyEval_ReleaseLock()`、および `PyEval_AcquireLock()` は例外です。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、`__builtin__`、`__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_Finalize()` 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。

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。

バージョン 2.4 で追加。

int Py_IsInitialized()

Python インタプリタがすでに初期化済みの場合に真 (非ゼロ) を返し、そうでない場合には偽 (ゼロ) を返します。`Py_Finalize()` を呼び出すと、次に `Py_Initialize()` を呼び出すまでこの関数は偽を返します。

void Py_Finalize()

`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。戻り値はありません。終了処理中のエラーは無視され

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_Finalize()` を 2 度以上呼び出すと起こり得ます。

8.2 プロセスワイドのパラメータ

`void Py_SetProgramName (char*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。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。

`char* 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。

`char* 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。

`char* 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 のみで有効です。

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。

`char* 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。

`char* 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、その後に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、たいいていそうされます。

`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

```
"1.5 (#67, Dec 31 1997, 22:34:28) [GCC 2.7.2.2]"
```

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 (intargc, char**argv, intupdatepath)`

`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さしていない場合)、 `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。

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。

2.6.6 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 2.6.6 で追加.

void **PySys_SetArgv** (int *argc*, char***argv*)

`PySys_SetArgvEx()` の `updatepath` に 1 を設定したのと同じように動作します。

void **Py_SetPythonHome** (char**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。

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。

8.3 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 (*global interpreter lock*, *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.setcheckinterval() 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。

8.3.1 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。Python をスレッドサポートなしでコンパイルしたときでもこの 2 つのマクロは利用可能です (単に、空行に展開されます)。

スレッドサポートが有効になっている場合、上記のブロックは以下のような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。

8.3.2 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 を混ぜて使うことはサポートされていません。

注意しないといけないもう一つの重要な点は、C の `fork()` を呼び出した時の動作です。ほとんどの `fork()` を持っているシステムでは、`fork` されたプロセスには `fork` を実行したスレッドしか存在しません。これは、別のスレッドに取得されたロックがずっと解放されないことを意味します。Python は `fork` する前にロックを取得し、その後に `fork` を解放することでこの問題を解決しています。加えて、子プロセスではすべての lock-objects をリセットします。Python を拡張したり埋め込んだりしている場合、Python に `fork` 前に取得したり `fork` 後に解放しなければならない追加の (Python 以外の) ロックを Python に教える手段がありません。Python と同じことを実現するには、`pthread_atfork()` のような OS の機能を使う必要があります。加えて、Python を拡張したり埋め込んだりしているときに、`os.fork()` を通してではなく直接 `fork()` を呼び出すと、`fork` 後に失われるスレッドに取得されていた Python の内部ロックのためにデッドロックが発生するかもしれません。`PyOS_AfterFork()` は必要なロックのリセットを試みますが、いつでもそれが可能とは限りません。

8.3.3 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

単一のスレッドの状態を表現する表現するデータ構造体です。スレッドのインタプリタ状態を指すポインタ `PyInterpreterState *interp` だけが公開されているデータメンバです。

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_ReleaseLock()` 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。この関数を `Py_Initialize()` の前に呼び出して安全です。

注釈: メインスレッドだけが存在しているとき、GIL 操作は必要ありません。(ほとんどの Python プログラムはスレッドを使わないので) これはよくある状況であり、ロック操作はインタプリタの動作を少しだけ遅くします。なので、ロックは最初には作成されません。これはロックを獲得しているのと同じ状態になります: スレッドが 1 つしかないとき、オブジェクトへの全てのアクセスは安全です。またこれらのことから、この関数がグローバルインタプリタロックを初期化するときは、その取得も行います。Python の `_thread` モジュールは新しいスレッドを作成する前に、ロックが既にあるのかロックはまだ作成されていないのか知るために、`PyEval_InitThreads()` を呼び出します。呼び出しが返ってきたとき、ロックが作成されており、`PyEval_InitThreads()` を呼び出したスレッドがそれを獲得したことが保証されます。

(どんなスレッドであれ) どのスレッドが現在グローバルインタプリタロックを持っているか分からないときに、この関数を呼び出すのは安全ではありません。

この関数は、コンパイル時にスレッドサポートが無効にされているときは利用できません。

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。この関数はコンパイル時にスレッドサポートを無効化すると利用できません。

バージョン 2.4 で追加。

***PyThreadState** PyEval_SaveThread()**

(GIL が生成されていて、スレッドサポートが有効の場合) GIL を解放して、スレッド状態を *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。(この関数はコンパイル時にスレッドサポートが無効にされていても利用できます。)

void PyEval_RestoreThread(*PyThreadState tstate)**

(GIL が生成されていて、スレッドサポートが有効の場合) 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。(この関数はコンパイル時にスレッドサポートが無効にされていても利用できます。)

***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。

void PyEval_ReInitThreads()

この関数は *PyOS_AfterFork()* から呼び出され、新しい子プロセスが、子プロセス内で実行していないスレッドが持っているロックを持たないようにします。

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はありません:

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。

バージョン 2.3 で追加。

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、必ず同スレッド内で対応する `PyGILState.Release()` を呼び出してください。

バージョン 2.3 で追加。

`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。

バージョン 2.3 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。コンパイル時にスレッドサポートが無効化されていると何も行いません。

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。コンパイル時にスレッドサポートが無効化されていると何も行いません。

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。コンパイル時にスレッドサポートが無効化されていると何も行いません。

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。コンパイル時にスレッドサポートが無効化されていると何も行いません。

8.3.4 低レベル API

以下の全ての関数はコンパイル時にスレッドサポートが有効になっている時だけ利用でき、呼び出すのは GIL がすでに作成されている場合だけにしなくてはなりません。

`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。

`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。

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。

*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。

バージョン 2.3 で変更: 以前は、現在のスレッドがアクティブなときのみ呼び出せるようになっており、*NULL* は例外が送出されたことを意味していました。

int **PyThreadState_SetAsyncExc** (longid, *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。素朴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。

バージョン 2.3 で追加。

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。

PyEval_RestoreThread() はいつでも (スレッドサポートが有効でないときやスレッドが初期化されたいないときでさえ) 利用可能な高レベル関数です。

void **PyEval_ReleaseThread** (*PyThreadState**tstate)

PyEval_RestoreThread() はより高レベルな関数で常に (スレッドサポートが無効な場合やスレッドが初期化されていない場合でも) 利用できます。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が、その値が現在のスレッド状態を表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 (スレッドサポートが無効な場合やスレッドが初期化されていない場合でも) 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。

警告: 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。

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。

警告: 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。

8.4 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。サブインタプリタはこれを可能にします。 *PyThreadState_Swap()* 関数を使ってサブインタプリタを切り替えることができます。以下の関数を使ってサブインタプリタの作成と削除を行えます:

*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 C ライブラリと同じ FILE 構造体を参照しています)。

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ない、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。)

拡張モジュールは以下のような形で (サブ) インタプリタ間で共有されます: ある特定の拡張モジュールを最初に import すると、モジュールを通常通りに初期化し、そのモジュールの辞書の (浅い) コピーをしまい込んでおきます。他の (サブ) インタプリタが同じ拡張モジュールを import すると、新たなモジュールを初期化し、先ほどのコピーの内容で辞書の値を埋めます; 拡張モジュールの `init` 関数は呼び出されません。この挙動は、`Py_Finalize()` および `Py_Initialize()` を呼び出してインタプリタを完全に再初期化した後に拡張モジュールを import した際の挙動とは異なるので注意してください; 再初期化後に import を行くと、拡張モジュールの `initmodule` は再度 呼び出されます。

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_Finalize()` 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。

8.4.1 バグと注意事項

サブインタプリタ (とメインインタプリタ) は同じプロセスの一部分なので、インタプリタ間の隔離は完璧ではありません — 例えば、`os.close()` のような低レベルのファイル操作を使うと、(偶然なり故意なりに) 互いのインタプリタ下にある開かれたファイルに影響を及ぼしてしまいます。拡張モジュールを (サブ) インタプリタ間で共有する方法のせいで、拡張モジュールによっては正しく動作しないかもしれません; 拡張モジュールが (静的な) グローバル変数を利用している場合や、拡張モジュールが初期化後に自身のモジュール辞書を操作する場合には特にそうです。一つのサブインタプリタで生成されたオブジェクトは他のサブインタプリタの名前空間への挿入が可能です; ユーザ定義関数、メソッド、インスタンスおよびクラスをサブインタプリタをサブインタプリタ間で共有しないように十分注意してください。というのは、これらの共有オブジェクトが実行した `import` 文は間違った (サブ) インタプリタのロード済みモジュール辞書に影響を及ぼす場合があるからです。

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。

8.5 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 2.7 で追加。

8.6 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。

Python 2.2 になってから、この機能の実装は実質的に作り直され、C から呼び出すためのインタフェースが追加されました。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, intwhat, 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。最初のパラメー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_EXCEPTION`, あるいは `PyTrace_C_RETURN` 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。

`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と共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) トレース関数の `what` パラメータとして渡す値です。

`int PyTrace_RETURN`

呼び出しが返るときに `Py_tracefunc` 関数に `what` パラメータとして渡す値です。

`int PyTrace_C_CALL`

C 関数を呼び出す直前に `Py_tracefunc` 関数の `what` パラメータとして渡す値です。

`int PyTrace_C_EXCEPTION`

C 関数が例外を送出したときに `Py_tracefunc` 関数の `what` パラメータとして渡す値です。

`int PyTrace_C_RETURN`

C 関数から戻るときに `Py_tracefunc` 関数の `what` パラメータとして渡す値です。

void **PyEval_SetProfile** (*Py_tracefunc*func, *PyObject**obj)

Set the profiler function to *func*. The *obj* parameter is passed to the function as its first parameter, and may be any Python object, or *NULL*. If the profile function needs to maintain state, using a different value for *obj* for each thread provides a convenient and thread-safe place to store it. The profile function is called for all monitored events except `PyTrace_LINE` and `PyTrace_EXCEPTION`.

void **PyEval_SetTrace** (*Py_tracefunc*func, *PyObject**obj)

Set the tracing function to *func*. This is similar to `PyEval_SetProfile()`, except the tracing function does receive line-number events and does not receive any event related to C function objects being called. Any trace function registered using `PyEval_SetTrace()` will not receive `PyTrace_C_CALL`, `PyTrace_C_EXCEPTION` or `PyTrace_C_RETURN` as a value for the *what* parameter.

*PyObject** **PyEval_GetCallStats** (*PyObject**self)

Return a tuple of function call counts. There are constants defined for the positions within the tuple:

名前	値
<code>PCALL_ALL</code>	0
<code>PCALL_FUNCTION</code>	1
<code>PCALL_FAST_FUNCTION</code>	2
<code>PCALL_FASTER_FUNCTION</code>	3
<code>PCALL_METHOD</code>	4
<code>PCALL_BOUND_METHOD</code>	5
<code>PCALL_CFUNCTION</code>	6
<code>PCALL_TYPE</code>	7
<code>PCALL_GENERATOR</code>	8
<code>PCALL_OTHER</code>	9
<code>PCALL_POP</code>	10

`PCALL_FAST_FUNCTION` means no argument tuple needs to be created. `PCALL_FASTER_FUNCTION` means that the fast-path frame setup code is used.

If there is a method call where the call can be optimized by changing the argument tuple and calling the function directly, it gets recorded twice.

This function is only present if Python is compiled with `CALL_PROFILE` defined.

8.7 高度なデバッガサポート (advanced debugger support)

以下の関数は高度なデバッグツールでの使用のためだけのものです。

*PyInterpreterState** **PyInterpreterState_Head** ()

インタプリタ状態オブジェクトからなるリストのうち、先頭にあるものを返します。

バージョン 2.2 で追加.

*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。

バージョン 2.2 で追加.

*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。

バージョン 2.2 で追加.

*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。

バージョン 2.2 で追加.

第 9 章

メモリ管理

9.1 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) が確実にを行います。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()` で操作しようとしてはなりません。こうした関数を使うと、C のメモリ操作関数と Python メモリマネージャとの間で関数呼び出しが交錯します。C のメモリ操作関数と Python メモリマネージャは異なるアルゴリズムで実装されていて、異なるヒープを操作するため、呼び出しの交錯は致命的な結果を招きます。とはいえ、個別の目的のためなら、C ライブラリのメモリ操作関数を使って安全にメモリを確保したり解放したりできます。例えば、以下がそのような例です：

```
PyObject *res;  
char *buf = (char *) malloc(BUFSIZ); /* for I/O */
```

(次のページに続く)

(前のページからの続き)

```

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyString_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 メモリマネージャは戻り値として返される文字列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 (memory footprint) 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。

9.2 メモリインタフェース

Python ヒープに対してメモリを確保したり解放したりするために、以下の関数セットが利用できます。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 バイトの領域を要求した際の動作についても定義しています:

void* PyMem_Malloc (size_t n)

n バイトをメモリ確保し、確保されたメモリを指す `void*` 型のポインタを返します。確保要求に失敗した場合には `NULL` を返します。0 バイトをリクエストすると、可能ならば独立した非 `NULL` のポインタを返します。このポインタは `PyMem_Malloc(1)` を代わりに呼んだときのようなメモリ領域を指しています。確保されたメモリ領域はいかなる初期化も行われていません。

void* PyMem_Realloc (void*p, size_t n)

p が指しているメモリブロックを n バイトにサイズ変更します。メモリの内容のうち、新旧のサイズのうち小さい方までの領域は変更されません。 p が `NULL` ならば、この関数は `PyMem_Malloc(n)` と等価になります; p が `NULL` ではなく n がゼロに等しい場合は、メモリブロックはサイズ変更されますが、解放はされず、非 `NULL` のポインタを返します。 p の値が `NULL` でないならば、それは以前呼び出した `PyMem_Malloc()` や `PyMem_Realloc()` の返した値でなければなりません。メモリ要求が失敗した場合は、`PyMem_Realloc()` は `NULL` を返し、 p は以前のメモリ領域への正しいポインタのままとなります。

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()` や

`PyMem_Realloc()` 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` なら、何も行いません。

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_tn)`

`PyMem_Malloc()` と同じですが、`(n * sizeof(TYPE))` バイトのメモリを確保します。`TYPE*` に型キャストされたポインタを返します。メモリには何の初期化も行われていません。

`TYPE* PyMem_Resize (void*p, TYPE, size_tn)`

`PyMem_Realloc()` と同じですが、`(n * sizeof(TYPE))` バイトにサイズ変更されたメモリを確保します。`TYPE*` に型キャストされたポインタを返します。関数が終わったとき、`p` は新しいメモリ領域のポインタか、失敗した場合は `NULL` になります。これは C プリプロセッサのマクロで、`p` は常に上書きされます。エラーを処理するときにメモリを失う事を避けるには、`p` の元の値を保存しておいてください。

`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()`、`PyMem_REALLOC()`、`PyMem_FREE()`。

`PyMem_NEW()`、`PyMem_RESIZE()`、`PyMem_DEL()`。

9.3 Object allocators

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：

デフォルトではこれらの関数は `pymalloc` メモリアロケータを利用します。

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_tn)`

`n` バイトを割り当て、そのメモリを指す `void*` 型のポインタを返します。要求が失敗した場合 `NULL` を返します。

Requesting zero bytes returns a distinct non-`NULL` pointer if possible, as if `PyObject_Malloc(1)` had been called instead. The memory will not have been initialized in any way.

`void* PyObject_Realloc (void*p, size_tn)`

`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。

If *p* is *NULL*, the call is equivalent to `PyObject_Malloc(n)`; else if *n* is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-*NULL*.

Unless *p* is *NULL*, it must have been returned by a previous call to `PyObject_Malloc()`, `PyObject_Realloc()` or `PyObject_Calloc()`.

If the request fails, `PyObject_Realloc()` returns *NULL* and *p* remains a valid pointer to the previous memory area.

void **PyObject_Free** (void*p)

Frees the memory block pointed to by *p*, which must have been returned by a previous call to `PyObject_Malloc()`, `PyObject_Realloc()` or `PyObject_Calloc()`. Otherwise, or if `PyObject_Free(p)` has been called before, undefined behavior occurs.

p が *NULL* の場合何もしません。

In addition, the following macro sets are provided:

- `PyObject_MALLOC()`: alias to `PyObject_Malloc()`
- `PyObject_REALLOC()`: alias to `PyObject_Realloc()`
- `PyObject_FREE()`: alias to `PyObject_Free()`
- `PyObject_Del()`: alias to `PyObject_Free()`
- `PyObject_DEL()`: alias to `PyObject_FREE()` (so finally an alias to `PyObject_Free()`)

9.4 pymalloc アロケータ

Python has a *pymalloc* allocator optimized for small objects (smaller or equal to 512 bytes) with a short lifetime. It uses memory mappings called "arenas" with a fixed size of 256 KiB. It falls back to `malloc()` and `realloc()` for allocations larger than 512 bytes.

pymalloc is the default allocator of `PyObject_Malloc()`.

アリーナアロケータは、次の関数を使います:

- 利用できる場合、`mmap()` と `munmap()`、
- それ以外の場合は `malloc()` と `free()`。

バージョン 2.7.7 で変更: The threshold changed from 256 to 512 bytes. The arena allocator now uses `mmap()` if available.

9.5 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 = PyString_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 = PyString_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 ファミリー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。

第 10 章

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型 (new object type) 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。

10.1 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. バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

void **PyObject_Del** (*PyObject**op)

*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。

バージョン 2.5 で変更: この関数は以前は *size* の型に *int* を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*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_tsize)

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、例えばタプ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。

バージョン 2.5 で変更: この関数は以前は *size* の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

void PyObject_Del (*PyObject**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_InitModule (char*name, *PyMethodDef**methods)

Return value: Borrowed reference. *name* と関数のテーブルに基づいて新たなモジュールオブジェクトを生成し、生成されたモジュールオブジェクトを返します。

バージョン 2.3 で変更: 以前のバージョンの Python では、*methods* 引数の値として `NULL` をサポートしていませんでした。

PyObject* Py_InitModule3 (char*name, *PyMethodDef**methods, char*doc)

Return value: Borrowed reference. *name* と関数のテーブルに基づいて新たなモジュールオブジェクトを生成し、生成されたモジュールオブジェクトを返します。*doc* が `NULL` でない場合、モジュールの docstring として使われます。

バージョン 2.3 で変更: 以前のバージョンの Python では、*methods* 引数の値として `NULL` をサポートしていませんでした。

PyObject* Py_InitModule4 (char*name, *PyMethodDef**methods, char*doc, *PyObject**self, intapiver)

Return value: Borrowed reference. *name* と関数のテーブルに基づいて新たなモジュールオブジェクトを生成し、生成されたモジュールオブジェクトを返します。*doc* が `NULL` でない場合、モジュールの docstring として使われます。*self* が `NULL` でない場合、モジュール内の各関数の第一引数として渡されます (`NULL` の時には第一引数も `NULL` になります)。(この関数は実験的な機能のために追加されたもので、現在の Python のバージョンで使われてはいないはずです。) *apiver* に渡してよい値は、`PYTHON_API_VERSION` で定義されている定数だけです。

注釈: この関数のほとんどの用途は、代わりに *Py_InitModule3()* を使えるはずですが、本当に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 2.3 で変更: 以前のバージョンの Python では、*methods* 引数の値として `NULL` をサポートしていませんでした。

PyObject *Py_NoneStruct*

Python から `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` マクロを使わねばなりません。

10.2 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* および *PyVarObject* 型で表現されます。 *PyObject* 型や *PyVarObject* 型もまた、他の全ての Python オブジェクトを定義する上で直接的・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、あるオブジェクトに対するオブジェクトとしてのポインタを Python から扱う必要がある際に必要な情報が入っています。通常に ”リリースされている” 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。 `PyObject_HEAD` マクロ展開で定義されているフィールドに対応します。

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。 `PyObject_VAR_HEAD` マクロ展開で定義されているフィールドに対応します。

これらのマクロは *PyObject* と *PyVarObject* の定義で使われています:

PyObject_HEAD

PyObject 型のフィールド宣言に展開されるマクロです; 可変でない長さを持つオブジェクトを表現する新たな型を宣言する場合に使います。展開によってどのフィールドが宣言されるかは、 `Py_TRACE_REFS` の定義に依存します。デフォルトでは、 `Py_TRACE_REFS` は定義されておらず、 *PyObject_HEAD* は以下のコードに展開されます:

```
Py_ssize_t ob_refcnt;
PyTypeObject *ob_type;
```

`Py_TRACE_REFS` が定義されている場合、以下のように展開されます:

```
PyObject *_ob_next, *_ob_prev;
Py_ssize_t ob_refcnt;
PyTypeObject *ob_type;
```

PyObject_VAR_HEAD

マクロです。 *PyVarObject* 型のフィールド宣言に展開されるマクロです; インスタンスによって可変の長さを持つオブジェクトを表現する新たな型を宣言する場合に使います。マクロは常に以下のように展開されます:

```
PyObject_HEAD  
Py_ssize_t ob_size;
```

マクロ展開結果の一部に `PyObject_HEAD` が含まれており、`PyObject_HEAD` の展開結果は `Py_TRACE_REFS` の定義に依存します。

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)
```

バージョン 2.6 で追加.

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)
```

バージョン 2.6 で追加.

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)
```

バージョン 2.6 で追加.

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

ほとんどの Python の呼び出し可能オブジェクトを C で実装する際に用いられている関数の型です。この型の関数は二つの `PyObject*` 型パラメータをとり、`PyObject*` 型の値を返します。戻り値を `NULL` にする場合、例外をセットしておかなければなりません。 `NULL` でない値を返す場合、戻り値は

Python に関数の戻り値として公開される値として解釈されます。この型の関数は新たな参照を返さなければなりません。

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	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	char *	docstring の内容を指すポインタ

ml_meth は C の関数ポインタです。関数は別の型で定義されていてもかまいませんが、常に *PyObject** を返します。関数が *PyFunction* でない場合、メソッドテーブル内でキャストを行うようコンパイラが要求することになるでしょう。*PyCFunction* では最初のパラメタが *PyObject** 型であると定義していますが、固有の C 型を *self* オブジェクトに使う実装はよく行われています。

ml_flags フィールドはビットフィールドで、以下のフラグが入ります。個々のフラグは呼び出し規約 (calling convention) や束縛規約 (binding convention) を表します。呼び出し規約フラグでは、*METH_VARARGS* および *METH_KEYWORDS* だけが組み合わせられます。呼び出し規約フラグは束縛規約フラグと組み合わせられます。

METH_VARARGS

PyCFunction 型のメソッドで典型的に使われる呼び出し規約です。関数は *PyObject** 型の引数値を二つ要求します。最初の引数はメソッドの *self* オブジェクトです; モジュール関数の場合、これはモジュールオブジェクトです。第二のパラメタ (よく *args* と呼ばれます) は、全ての引数を表現するタプルオブジェクトです。パラメタは通常、*PyArg_ParseTuple()* や *PyArg_UnpackTuple()* で処理されます。

METH_KEYWORDS

このフラグを持つメソッドは *PyCFunctionWithKeywords* 型でなければなりません。*PyCFunctionWithKeywords* は三つのパラメタ: *self*、*args*、およびキーワード引数全てからなる辞書、を要求します。このフラグは通常 *METH_VARARGS* と組み合わせられ、パラメタは *PyArg_ParseTupleAndKeywords()* で処理されます。

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

単一のオブジェクト引数だけをとるメソッドは、*PyArg_ParseTuple()* を引数 "O" にして呼び出す代わりに、*METH_O* フラグつきで指定できます。メソッドは *PyCFunction* 型で、*self* パラメタと単一の引数を表現する *PyObject** パラメタを伴います。

METH_OLDARGS

この呼び出し規約は撤廃されました。メソッドは *PyCFunction* 型でなければなりません。第二引数は、引数がない場合には *NULL*、単一の引数の場合にはその引数オブジェクト、複数の引数の場合には引数オブジェクトからなるタプルです。この呼び出し規約を使うと、複数の引数の場合と、単一のタプルが唯一引数の場合を区別できなくなってしまいます。

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。

バージョン 2.3 で追加.

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。

バージョン 2.3 で追加.

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。

バージョン 2.4 で追加.

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
<i>name</i>	<i>char *</i>	メンバ名
<i>type</i>	<i>int</i>	C 構造体の中のメンバの型
<i>offset</i>	<i>Py_ssize_t</i>	そのメンバの <i>type object</i> 構造体中の場所の <i>offset</i> バイト数
<i>flags</i>	<i>int</i>	フィールドが読み込み専用か書き込み可能なのかを示すビットフラグ
<i>doc</i>	<i>char *</i>	<i>docstring</i> 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 type
T_SHORT	short
T_INT	int
T_LONG	long 型
T_FLOAT	float
T_DOUBLE	double
T_STRING	char *
T_OBJECT	PyObject *
T_OBJECT_EX	PyObject *
T_CHAR	char
T_BYTE	char
T_UBYTE	unsigned char
T_UINT	unsigned int
T_USHORT	unsigned short
T_ULONG	unsigned long
T_BOOL	char
T_LONGLONG	long long
T_ULONGLONG	unsigned long long
T_PYSSIZET	Py_ssize_t

T_OBJECT と T_OBJECT_EX については、T_OBJECT がメンバが *NULL* だったときに *None* を返すのに対し、T_OBJECT_EX は *AttributeError* を発生させる点が異なります。T_OBJECT_EX は属性に対する *del* 文をより正しくあつかうので、できれば T_OBJECT よりも T_OBJECT_EX を使ってください。

flags には読み書きアクセス可能なら 0 で、読み込み専用なら *READONLY* を設定します。type に T_STRING を使うと、*READONLY* 扱いになります。T_OBJECT メンバと T_OBJECT_EX メンバだけが削除できます (*NULL* が代入されます)。

PyGetSetDef

Structure to define property-like access for a type. See also description of the *PyTypeObject*. *tp_getset* slot.

フィールド	C の型	意味
名前	char *	attribute name
get	getter	C Function to get the attribute
集合	setter	optional C function to set or delete the attribute, if omitted the attribute is read-only
doc	char *	optional docstring
closure	void *	optional function pointer, providing additional data for getter and setter

The `get` function takes one *PyObject** parameter (the instance) and a function pointer (the associated closure):

```
typedef PyObject *(*getter) (PyObject *, void *);
```

It should return a new reference on success or *NULL* with a set exception on failure.

set functions take two *PyObject** parameters (the instance and the value to be set) and a function pointer (the associated closure):

```
typedef int (*setter) (PyObject *, PyObject *, void *);
```

In case the attribute should be deleted the second parameter is *NULL*. Should return 0 on success or -1 with a set exception on failure.

*PyObject** **Py_FindMethod** (*PyMethodDef*[], *PyObject**ob, char*name)

Return value: New reference. C で実装された拡張型の束縛メソッドオブジェクトを返します。
PyObject_GenericGetAttr() 関数を使わない *tp_getattro* や *tp_getattr* ハンドラを実装する際に便利です。

10.3 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決める基盤ですから、インタプリタ自体や新たな型を定義する拡張モジュールでは非常に重要な存在です。

型オブジェクトは標準の型 (standard type) に比べるとかなり大きな構造体です。各型オブジェクトは多くの値を保持しており、そのほとんどは C 関数へのポインタで、それぞれの関数はその型の機能の小さい部分を実装しています。この節では、型オブジェクトの各フィールドについて詳細を説明します。各フィールドは、構造体内で出現する順番に説明されています。

Typedefs: *unaryfunc*, *binaryfunc*, *ternaryfunc*, *inquiry*, *coercion*, *intargfunc*, *intintargfunc*, *intobjargproc*, *intintoobjargproc*, *objobjargproc*, *destructor*, *freefunc*, *printfunc*, *getattrfunc*, *getattrofunc*, *setattrfunc*, *setattrofunc*, *cmpfunc*, *reprfunc*, *hashfunc*

PyTypeObject の構造体定義は *Include/object.h* で見つけれられるはずです。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
    char *tp_name; /* For printing, in format "<module>.<name>" */
    int tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

```

(次のページに続く)

(前のページからの続き)

```
destructor tp_dealloc;
printfunc tp_print;
getattrfunc tp_getattr;
setattrfunc tp_setattr;
cmpfunc tp_compare;
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;
getattrfunc tp_getattro;
setattrfunc tp_setattro;

/* Functions to access object as input/output buffer */
PyBufferProcs *tp_as_buffer;

/* Flags to define presence of optional/expanded features */
long tp_flags;

char *tp_doc; /* Documentation string */

/* Assigned meaning in release 2.0 */
/* call function for all accessible objects */
traverseproc tp_traverse;

/* delete references to contained objects */
inquiry tp_clear;

/* Assigned meaning in release 2.1 */
/* rich comparisons */
richcmpfunc tp_richcompare;

/* weak reference enabler */
long tp_weaklistoffset;

/* Added in release 2.2 */
/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;

/* Attribute descriptor and subclassing stuff */
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
```

(次のページに続く)

(前のページからの続き)

```

struct PyGetSetDef *tp_getset;
struct _typeobject *tp_base;
PyObject *tp_dict;
descrgetfunc tp_descr_get;
descrsetfunc tp_descr_set;
long 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;

} PyTypeObject;

```

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 `PYTHONDUMPREFS` 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。

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。

サブタイプはこのフィールドを継承しません。

バージョン 2.5 で変更: このフィールドは以前は `int` でした。この変更により、64bit システムを正しくサポートするには修正が必要になります。

*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ねばなりません。`PyType_Ready()` は `ob_type` が `NULL` かどうか調べ、`NULL` の場合には初期化します: Python 2.2 では、`&PyType_Type` にセットします; in Python 2.2.1 およびそれ以降では基底クラスの `ob_type` フィールドに初期化します。`ob_type` が非ゼロの場合、`PyType_Ready()` はこのフィールドを変更しません。

Python 2.2 では、サブタイプはこのフィールドを継承しません。2.2.1 と 2.3 以降では、サブタイプはこのフィールドを継承します。

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。

サブタイプはこのフィールドを継承しません。

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す。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。

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` フィールドがなくはならず、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 `N` かける `tp_itemsize` の加算になります。`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、長整数では負の値を `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` の正しい初期化パラメタを得るには、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 (これは Python 2.2 からの新しい仕様です; 2.1 や 2.0 では、GC ヘッダサイズは `tp_basicsize` に入っていました)。

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。

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。

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。

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、最後に (デストラクタ関数の最後の操作として) 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 (deallocator) 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。

サブタイプはこのフィールドを継承します。

`printfunc PyObject.tp_print`

オプションのフィールドで、インスタンスの出力 (print) を行う関数を指すポインタです。

出力関数は、インスタンスが実体のある (*real*) ファイルに出力される場合にのみ呼び出されます; (StringIO インスタンスのような) 擬似ファイルに出力される場合には、インスタンスの `tp_repr` や `tp_str` が指す関数が呼び出され、文字列への変換を行います。また、`tp_print` が `NULL` の場合にもこれらの関数が呼び出されます。`tp_repr` や `tp_str` と異なる出力を生成するような `tp_print` は、決して型に実装してはなりません。

出力関数は `PyObject_Print()` と同じシグネチャ: `int tp_print(PyObject *self, FILE *file, int flags)` で呼び出されます。`self` 引数は出力されるインスタンスを指します。`file` 引数は出力先となる標準入出力 (stdio) ファイルです。`flags` 引数はフラグビットを組み合わせた値です。現在定義されているフラグビットは `Py_PRINT_RAW` のみです。`Py_PRINT_RAW` フラグビットがセットされていれば、インスタンスは `tp_str` と同じ書式で出力されます; `Py_PRINT_RAW` フラグビットがクリアされていれば、インスタンスは `tp_repr` と同じ書式で出力されます。操作中にエラーが生じたときは、この関数は `-1` を返して例外状態をセットしなければなりません。

`tp_print` フィールドは撤廃されるかもしれません。いずれにせよ、`tp_print` は定義せず、代わりに `tp_repr` や `tp_str` に頼って出力を行うようにしてください。

サブタイプはこのフィールドを継承します。

`getattrfunc PyObject.tp_getattr`

オプションのフィールドです。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。シグネチャは次の通りです:

```
PyObject * tp_getattr(PyObject *o, char *attr_name);
```

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 PyObject.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。シグネチャは次の通りです:

```
PyObject * tp_setattr(PyObject *o, char *attr_name, PyObject *v);
```

引数 `v` に `NULL` を設定すると属性を削除します。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` が両方とも `NULL` のとき、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。

cmpfunc `PyObject.tp_compare`

オプションのフィールドです。ポインタで、三値比較 (three-way comparison) を行う関数を指します。

シグネチャは `PyObject_Compare()` と同じです。この関数は `self` が `other` よりも大きければ 1, `self` と `other` の値が等しければ 0, `self` が `other` より小さければ -1 を返します。この関数は、比較操作中にエラーが生じた場合、例外状態をセットして -1 を返さねばなりません。

このフィールドは `tp_richcompare` および `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_compare`, `tp_richcompare` および `tp_hash` が共に `NULL` の場合、サブタイプは基底タイプから `tp_compare`, `tp_richcompare`, `tp_hash` の三つを一緒に継承します。

reprfunc `PyObject.tp_repr`

オプションのフィールドです。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。

シグネチャは `PyObject_Repr()` と同じです。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。

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。

サブタイプはこのフィールドを継承します。

***PyNumberMethods** `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。

`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** `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。

`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** `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 [マップ型オブジェクト構造体](#) で説明されています。

`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 `PyObject.tp_hash`

オプションのフィールドです。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。

シグネチャは `PyObject_Hash()` と同じです。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。

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。

このフィールドが設定されていない場合、二つの可能性があります: `tp_compare` および `tp_richcompare` フィールドの両方が `NULL` の場合、オブジェクトのアドレスに基づいたデフォルトのハッシュ値が返されます; それ以外の場合、`TypeError` が送出されます。

このフィールドは `tp_compare` および `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_compare`, `tp_richcompare` および `tp_hash` が共に `NULL` の場合、サブタイプは基底タイプから `tp_compare`, `tp_richcompare`, `tp_hash` の三つをいずれも継承します。

ternaryfunc `PyTypeObject.tp_call`

オプションのフィールドです。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。

サブタイプはこのフィールドを継承します。

reprfunc `PyTypeObject.tp_str`

オプションのフィールドです。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。))

シグネチャは `PyObject_Str()` と同じです; 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ねばなりません。また、この関数はオブジェクトを "分かりやすく (friendly)" 表現した文字列を返さねばなりません。というのは、この文字列は `print` 文で使われることになる表記だからです。

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。

サブタイプはこのフィールドを継承します。

getattrofunc `PyTypeObject.tp_getattro`

オプションのフィールドです。ポインタで、`get-attribute` を実装している関数を指します。

シグネチャは `PyObject_GetAttr()` と同じです。対する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うまくして便利です。

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。

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。

シグネチャは `PyObject_SetAttr()` と同じですが、`v` 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ことについては別の場合は便利です。

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。

`PyBufferProcs*` `PyObject.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。

`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。

`long` `PyObject.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。

このフィールドの継承は複雑です。ほとんどのフラグビットは個別に継承されます。つまり、基底タイプであるフラグビットがセットされている場合、サブタイプはそのフラグビットを継承します。機能拡張のための構造体に関するフラグビットは、その機能拡張構造体が継承されるときに限定して継承されます。すなわち、基底タイプのフラグビットの値は、機能拡張構造体へのポインタと一緒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` および `tp_clear` フィールドと合わせてコピーされます。すなわち、サブタイプの `Py_TPFLAGS_HAVE_GC` フラグビットがクリアで、かつ (`Py_TPFLAGS_HAVE_RICHCOMPARE` フラグビットの指定によって) `tp_traverse` および `tp_clear` フィールドがサブタイプ内に存在しており、かつ値が `NULL` の場合に基底タイプから値を継承します。

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。 `PyObject_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_HAVE_GETCHARBUFFER`

このビットがセットされていれば、 `tp_as_buffer` が参照する `PyBufferProcs` 構造体には `bf_getcharbuffer` フィールドがあります。

`Py_TPFLAGS_HAVE_SEQUENCE_IN`

このビットがセットされていれば、 `tp_as_sequence` が参照する `PySequenceMethods` 構造体には `sq_contains` フィールドがあります。

`Py_TPFLAGS_GC`

このビットは旧式のもので、このシンボルが指し示していたビットはもはや使われていません。

シンボルの現在の定義はゼロになっています。

Py_TPFLAGS_HAVE_INPLACEOPS

このビットがセットされていれば、`tp_as_sequence` が参照する *PySequenceMethods* 構造体、および `tp_as_number` が参照する *PyNumberMethods* 構造体には in-place 演算に関するフィールドが入っています。具体的に言うと、*PyNumberMethods* 構造体はフィールド `nb_inplace_add`, `nb_inplace_subtract`, `nb_inplace_multiply`, `nb_inplace_divide`, `nb_inplace_remainder`, `nb_inplace_power`, `nb_inplace_lshift`, `nb_inplace_rshift`, `nb_inplace_and`, `nb_inplace_xor`, および `nb_inplace_or` を持つことになります; また、*PySequenceMethods* 構造体はフィールド `sq_inplace_concat` および `sq_inplace_repeat` を持つことになります。

Py_TPFLAGS_CHECKTYPES

このビットがセットされていれば、`tp_as_number` が参照する *PyNumberMethods* 構造体内で定義されている二項演算子および三項演算子は任意のオブジェクト型を非演算子にとるようになり、必要に応じて引数の型変換を行います。このビットがクリアなら、演算子は全ての引数が現在のオブジェクト型と同じであるよう要求し、演算の呼び出し側は演算に先立って型変換を行うものと想定します。対象となる演算子は `nb_add`, `nb_subtract`, `nb_multiply`, `nb_divide`, `nb_remainder`, `nb_divmod`, `nb_power`, `nb_lshift`, `nb_rshift`, `nb_and`, `nb_xor`, および `nb_or` です。

Py_TPFLAGS_HAVE_RICHCOMPARE

このビットがセットされていれば、型オブジェクトには *tp_richcompare* フィールド、そして *tp_traverse* および *tp_clear* フィールドがあります。

Py_TPFLAGS_HAVE_WEAKREFS

このビットがセットされていれば、構造体には *tp_weaklistoffset* フィールドが定義されています。 *tp_weaklistoffset* フィールドの値がゼロより大きければ、この型のインスタンスは弱参照で参照できます。

Py_TPFLAGS_HAVE_ITER

このビットがセットされていれば、型オブジェクトには *tp_iter* および *tp_iternext* フィールドがあります。

Py_TPFLAGS_HAVE_CLASS

このビットがセットされていれば、型オブジェクトは Python 2.2 以降で定義されている新たなフィールド: *tp_methods*, *tp_members*, *tp_getset*, *tp_base*, *tp_dict*, *tp_descr_get*, *tp_descr_set*, *tp_dictoffset*, *tp_init*, *tp_alloc*, *tp_new*, *tp_free*, *tp_is_gc*, *tp_bases*, *tp_mro*, *tp_cache*, *tp_subclasses*, および *tp_weaklist* があります。

Py_TPFLAGS_HEAPTYPE

型オブジェクト自体がヒープにメモリ確保される場合にセットされるビットです。型オブジェクト自体がヒープにメモリ確保される場合、インスタンスの `ob_type` フィールドは型オブジェクトへの参照とみなされます。この場合、新たなインスタンスを生成する度に型オブジェクトを INCREMENT、インスタンスを解放するたびに DECREMENT します (サブタイプのインスタンスには適当されま

せん; インスタンスが `ob.type` で参照している型だけが INCREMENT および DECREMENT されます)。

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)。

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。

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。

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()` を使って破壊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。このビットはまた、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を示します; しかし、これらのフィールドは `Py_TPFLAGS_HAVE_GC` がクリアでも `Py_TPFLAGS_HAVE_RICHCOMPARE` がセットされている場合には存在します。

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_GETCHARBUFFER`, `Py_TPFLAGS_HAVE_SEQUENCE_IN`, `Py_TPFLAGS_HAVE_INPLACEOPS`, `Py_TPFLAGS_HAVE_RICHCOMPARE`, `Py_TPFLAGS_HAVE_WEAKREFS`, `Py_TPFLAGS_HAVE_ITER`, および `Py_TPFLAGS_HAVE_CLASS` が入っています。

char* PyTypeObject.tp_doc

オプションのフィールドです。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。

サブタイプはこのフィールドを継承しません。

以下の三つのフィールドは、`Py_TPFLAGS_HAVE_RICHCOMPARE` フラグビットがセットされている場合にのみ存在します。

***traverseproc* PyTypeObject.tp_traverse**

オプションのフィールドです。ポインタで、ガベージコレクタのためのトラバーサル関数 (traversal 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。Python のガベージコレクションの枠組みに関する詳細は [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。

`tp.traverse` ポインタは、ガベージコレクタが循環参照を見つけるために使われます。`tp.traverse` 関数の典型的な実装は、インスタンスの各メンバのうち Python オブジェクトに対して `Py_VISIT()` を呼び出します。例えば、次のコードは `thread` 拡張モジュールの `local_traverse()` 関数になります:

```
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。

`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。

このフィールドは `tp_clear` および `Py_TPFLAGS_HAVE_GC` フラグビットと一緒に継承されます: フラグビット、`tp_traverse`, および `tp_clear` の値がサブタイプで全てゼロになっており、かつサブタイプで `Py_TPFLAGS_HAVE_RICHCOMPARE` フラグビットがセットされている場合に、基底タイプから値を継承します。

inquiry `PyObject.tp_clear`

オプションのフィールドです。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。

`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` 関数を提供するべきです。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);
}
```

(次のページに続く)

(前のページからの続き)

```

    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。

このフィールドは `tp_traverse` および `Py_TPFLAGS_HAVE_GC` フラグビットと一緒に継承されます: フラグビット、`tp_traverse`, および `tp_clear` の値がサブタイプで全てゼロになっており、かつサブタイプで `Py_TPFLAGS_HAVE_RICHCOMPARE` フラグビットがセットされている場合に、基底タイプから値を継承します。

richcmpfunc `PyObject.tp_richcompare`

オプションのフィールドで、拡張比較関数 (rich comparison function) を指すポインタです。拡張比較関数のシグネチャは `PyObject *tp_richcompare(PyObject *a, PyObject *b, int op)` です。

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。

注釈: 限られた種類の比較だけが可能 (例えば、`==` と `!=` が可能で `<` などが不可能) な型を実装したい場合、拡張比較関数で直接 `TypeError` を返します。

このフィールドは `tp_compare` および `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_compare`, `tp_richcompare` および `tp_hash` が共に `NULL` の場合、サブタイプは基底タイプから `tp_compare`, `tp_richcompare`, `tp_hash` の三つと一緒に継承します。

`tp_richcompare` および `PyObject_RichCompare()` 関数の第三引数に使うための定数としては以下が定義されています:

定数	比較
Py_LT	<
Py_LE	<=
Py_EQ	==
Py_NE	!=
Py_GT	>
Py_GE	>=

次のフィールドは、`Py_TPFLAGS_HAVE_WEAKREFS` フラグビットがセットされている場合にのみ存在します。

long `PyTypeObject.tp_weaklistoffset`

型のインスタンスが弱参照可能な場合、このフィールドはゼロよりも大きな数になり、インスタンス構造体における弱参照リストの先頭を示すオフセットが入ります (GC ヘッドがある場合には無視します); このオフセット値は `PyObject_ClearWeakRefs()` および `PyWeakref_*()` 関数が利用します。インスタンス構造体には、`NULL` に初期化された `PyObject*` 型のフィールドが入っていないければなりません。

このフィールドを `tp_weaklist` と混同しないようにしてください; これは型オブジェクト自身への弱参照からなるリストの先頭です。

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。

次の二つのフィールドは、`Py_TPFLAGS_HAVE_ITER` フラグビットがセットされている場合にのみ存在します。

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し、旧スタイルクラスのインスタンスは `__iter__()` メソッドを定義していなくてもこの関数を持っています)

この関数は `PyObject_GetIter()` と同じシグネチャを持っています。

サブタイプはこのフィールドを継承します。

iternextfunc `PyTypeObject.tp_iternext`

オプションのフィールドで、イテレータにおいて次の要素を返す関数へのポインタです。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、通常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 (ただし、旧スタイルのインスタンスでは、たとえば `next()` メソッドが定義されていなくても常にこの関数を持っています)。

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。

サブタイプはこのフィールドを継承します。

次の `tp_weaklist` までのフィールドは、 `Py_TPFLAGS_HAVE_CLASS` フラグビットがセットされている場合にのみ存在します。

struct `PyMethodDef* PyTypeObject.tp_methods`

オプションのフィールドです。ポインタで、この型の正規 (regular) のメソッドを宣言している `PyMethodDef` 構造体からなる、 `NULL` で終端された静的な配列を指します。

配列の各要素ごとに、メソッドデスクリプタの入った、要素が型の辞書 (下記の `tp_dict` 参照) に追加されます。

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`

オプションのフィールドです。ポインタで、型の正規 (regular) のデータメンバ (フィールドおよびスロット) を宣言している `PyMemberDef` 構造体からなる、 `NULL` で終端された静的な配列を指します。

配列の各要素ごとに、メンバデスクリプタの入った、要素が型の辞書 (下記の `tp_dict` 参照) に追加されます。

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`

オプションのフィールドです。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。

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`

オプションのフィールドです。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へのポインタです。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。

(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。しかし、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 `PyTypeReady()` 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、`PyTypeReady()` 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。`PyTypeReady()` 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__())` のようなオーバーロード用演算でないときだけです。

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)。

`descrgetfunc` `PyTypeObject.tp_descr_get`

オプションのフィールドです。ポインタで、”デスクリプタ `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);
```

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);
```

`value` 引数に `NULL` を設定して値を消します。このフィールドはサブタイプに継承されます。

`long` `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。長整数は符号を記憶するのに `ob_size` の符号を使うため、`ob_size` は絶対値を使います。(この計算を自分で行う必要はまったくありません; `PyObject_GetDictPtr()` がやってくれます。)

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。)

initproc `PyTypeObject.tp_init`

オプションのフィールドです。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__()` を呼び出す際の固定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、型の呼び出しで普通にインスタンスを生成する際に、型の `tp_new` がインスタンスを返した後に `tp_init` が呼び出されます。 `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。(VERSION NOTE: ここに書かれている内容は、Python 2.2.1 以降での実装に関するものです。Python 2.2 では、 `tp_init` は `NULL` でない限り `tp_new` が返す全てのオブジェクトに対して常に呼び出されます。) not `NULL`.)

サブタイプはこのフィールドを継承します。

allocfunc `PyTypeObject.tp_alloc`

オプションのフィールドです。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)
```

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が、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` になるはずですが。

この関数をインスタンス初期化の他のどの処理にも、追加でメモリ確保をする場合でさえ使ってはなりません; そうした処理は `tp_new` で行わねばなりません。

静的なサブタイプはこのフィールドを継承しますが、動的なサブタイプ (class 文で生成するサブタイプ) の場合は継承しません; 後者の場合、このフィールドは常に `PyType_GenericAlloc()` にセットされ、標準のヒープ上メモリ確保戦略が強制されます。静的に定義する型の場合でも、`PyType_GenericAlloc()` を推奨します。

newfunc **PyTypeObject.tp_new**

オプションのフィールドです。ポインタで、インスタンス生成関数を指します。

このフィールドが `NULL` を指している型では、型を呼び出して新たなインスタンスを生成できません; こうした型では、おそらくファクトリ関数のように、インスタンスを生成する他の方法があるはずですが。

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、変更不能な型では初期化は全て `tp_new` で行い、一方、変更可能な型ではほとんどの初期化を `tp_init` に回すべきです。

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。後者が例外になっているのは、旧式の拡張型が Python 2.2 でリンクされたときに呼び出し可能オブジェクトにならないようにするための予防措置です。

destructor **PyTypeObject.tp_free**

オプションのフィールドです。ポインタで、インスタンスのメモリ解放関数を指します。

この関数のシグネチャは少し変更されています; Python 2.2 および 2.2.1 では、シグネチャは `destructor`

```
void tp_free(PyObject *)
```

でしたが、Python 2.3 以降では、シグネチャは `freefunc` になっています:

```
void tp_free(void *)
```

両方のバージョンと互換性のある初期値は `_PyObject_Del` です。 `_PyObject_Del` の定義は Python 2.3 で適切に対応できるよう変更されました。

静的なサブタイプはこのフィールドを継承しますが、動的なサブタイプ (`class` 文で生成するサブタイプ) の場合は継承しません; 後者の場合、このフィールドには `PyType_GenericAlloc()` と `Py_TPFLAGS_HAVE_GC` フラグビットの値に対応させるのにふさわしいメモリ解放関数がセットされます。

inquiry `PyTypeObject.tp_is_gc`

オプションのフィールドです。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。)

サブタイプはこのフィールドを継承します。(VERSION NOTE: Python 2.2 では、このフィールドは継承されませんでした。2.2.1 以降のバージョンから継承されるようになりました。)

*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。

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。

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**

使用されていません。継承されません。内部で使用するためだけのものです。

*PyObject** **PyTypeObject.tp_subclasses**

サブクラスへの弱参照からなるリストです。継承されません。内部で使用するためだけのものです。

*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。

The remaining fields are only defined if the feature test macro `COUNT_ALLOCS` is defined, and are for internal use only. They are documented here for completeness. None of these fields are inherited by subtypes. See the `PYTHONSHOWALLOCCOUNT` environment variable.

`Py_ssize_t` **PyTypeObject.tp_allocs**

メモリ確保の回数です。

`Py_ssize_t` **PyTypeObject.tp_frees**

メモリ解放の回数です。

`Py_ssize_t` **PyTypeObject.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_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。

10.4 数値オブジェクト構造体

PyNumberMethods

拡張型で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するために使われる構造体です。以下のほとんどすべての関数は [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_divide;
```

(次のページに続く)

(前のページからの続き)

```

    binaryfunc nb_remainder;
    binaryfunc nb_divmod;
    ternaryfunc nb_power;
    unaryfunc nb_negative;
    unaryfunc nb_positive;
    unaryfunc nb_absolute;
    inquiry nb_nonzero;          /* Used by PyObject_IsTrue */
    unaryfunc nb_invert;
    binaryfunc nb_lshift;
    binaryfunc nb_rshift;
    binaryfunc nb_and;
    binaryfunc nb_xor;
    binaryfunc nb_or;
    coercion nb_coerce;          /* Used by the coerce() function */
    unaryfunc nb_int;
    unaryfunc nb_long;
    unaryfunc nb_float;
    unaryfunc nb_oct;
    unaryfunc nb_hex;

    /* Added in release 2.0 */
    binaryfunc nb_inplace_add;
    binaryfunc nb_inplace_subtract;
    binaryfunc nb_inplace_multiply;
    binaryfunc nb_inplace_divide;
    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;

    /* Added in release 2.2 */
    binaryfunc nb_floor_divide;
    binaryfunc nb_true_divide;
    binaryfunc nb_inplace_floor_divide;
    binaryfunc nb_inplace_true_divide;

    /* Added in release 2.5 */
    unaryfunc nb_index;
} PyNumberMethods;

```

2 引数および 3 引数の関数は、`Py_TPFLAGS_CHECKTYPES` フラグによっては、異なる種類の引数を受け取るかもしれません。

- `Py_TPFLAGS_CHECKTYPES` がセットされていない場合、関数の引数はオブジェクトの型であることが保証されます。呼び出し側は `nb_coerce` メンバで指定されている型強制メソッドを呼び出して引数を変換する責任があります。

coercion **`PyNumberMethods.nb_coerce`**

この関数は `PyNumber_CoerceEx()` から利用され、同じシグネチャを持ちます。最初の引数は定義された型のオブジェクトを指すポインタでなければなりません。共通の "大きな" 型への変換が可能であれば、この関数はポインタを変換後のオブジェクトへの新しい参照へ置き換えて、0 を返します。変換ができないなら、この関数は 1 を返します。エラーが設定荒れた場合は、-1 を返します。

- `Py_TPFLAGS_CHECKTYPES` フラグがセットされている場合、2 引数および 3 引数関数はすべてのオペランドの型をチェックし、必要な変換を行わなければなりません。(少なくとも、オペランドのうち 1 つは定義している型のもので) これは推奨された方式です。Python 3 では型強制は完全に取り除かれています。

与えられたオペランドに対して操作が定義されていない場合は、2 引数および 3 引数関数は `Py_NotImplemented` を返さなければなりません。その他のエラーが発生した場合は、例外を設定して `NULL` を返さなければなりません。

10.5 マップ型オブジェクト構造体

PyMappingMethods

拡張型で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するために使われる構造体です。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_Length()` 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()` 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 `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()` および `PyObject_DelItem()` 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。

10.6 シーケンスオブジェクト構造体

PySequenceMethods

拡張型で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するために使われる構造体です。

lenfunc `PySequenceMethods.sq_length`

この関数は `PySequence_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。

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**

この関数は *PySequence_GetItem()* 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 *PySequence_Check()* が 1 を返すためには埋めなければならない、それ以外の場合は *NULL* の可能性があります。

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、それを呼び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**

この関数は *PySequence_SetItem()* から利用され、同じシグネチャを持っています。オブジェクトが要素の代入と削除をサポートしていない場合は、このスロットは *NULL* かもしれません。

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**

この関数は *PySequence_InPlaceConcat()* から利用され、同じシグネチャを持っています。この関数は最初の実数値を修正してそれを返すべきです。

ssizeargfunc **PySequenceMethods.sq_inplace_repeat**

この関数は *PySequence_InPlaceRepeat()* から利用され、同じシグネチャを持っています。この関数は最初の実数値を修正してそれを返すべきです。

10.7 バッファオブジェクト構造体 (buffer object structure)

バッファインタフェースは、あるオブジェクトの内部データを一連のデータチャンク (chunk) として見せるモデルを外部から利用できるようにします。各チャンクはポインタ/データ長からなるペアで指定します。チャンクはセグメント (*segment*) と呼ばれ、メモリ内に不連続的に配置されるものと想定されています。

バッファインタフェースを利用できるようにしたくないオブジェクトでは、*PyObject* 構造体の *tp_as_buffer* メンバを *NULL* にしなくてはなりません。利用できるようにする場合、*tp_as_buffer* は *PyBufferProcs* 構造体を指さねばなりません。

注釈: *PyObject* 構造体の *tp_flags* メンバの値を 0 でなく *Py_TPFLAGS_DEFAULT* にしておくことがとても重要です。この設定は、*PyBufferProcs* 構造体に *bf_getcharbuffer* スロットが入っている

ることを Python ランタイムに教えます。Python の古いバージョンには `bf_getcharbuffer` メンバが存在しないので、古い拡張モジュールを使おうとしている新しいバージョンの Python インタプリタは、このメンバがあるかどうかテストしてから使えるようにする必要があります。

PyBufferProcs

バッファプロトコルの実装を定義している関数群へのポインタを保持するのに使われる構造体です。

The first slot is `bf_getreadbuffer`, of type `readbufferproc`. If this slot is `NULL`, then the object does not support reading from the internal data. This is non-sensical, so implementors should fill this in, but callers should test that the slot contains a non-`NULL` value.

The next slot is `bf_getwritebuffer` having type `writebufferproc`. This slot may be `NULL` if the object does not allow writing into its returned buffers.

The third slot is `bf_getsegcount`, with type `segcountproc`. This slot must not be `NULL` and is used to inform the caller how many segments the object contains. Simple objects such as `PyString_Type` and `PyBuffer_Type` objects contain a single segment.

The last slot is `bf_getcharbuffer`, of type `charbufferproc`. This slot will only be present if the `Py_TPFLAGS_HAVE_GETCHARBUFFER` flag is present in the `tp_flags` field of the object's `PyTypeObject`. Before using this slot, the caller should test whether it is present by using the `PyType_HasFeature()` function. If the flag is present, `bf_getcharbuffer` may be `NULL`, indicating that the object's contents cannot be used as 8-bit characters. The slot function may also raise an error if the object's contents cannot be interpreted as 8-bit characters. For example, if the object is an array which is configured to hold floating point values, an exception may be raised if a caller attempts to use `bf_getcharbuffer` to fetch a sequence of 8-bit characters. This notion of exporting the internal buffers as "text" is used to distinguish between objects that are binary in nature, and those which have character-based content.

注釈: 現在のポリシでは、文字 (character) はマルチバイト文字でもかまわないと決めているように思われます。従って、サイズ N のバッファが N 個のキャラクタからなるとはかぎらないことになります。

Py_TPFLAGS_HAVE_GETCHARBUFFER

型構造体中のフラグビットで、`bf_getcharbuffer` スロットが既知の値になっていることを示します。このフラグビットがセットされていたとしても、オブジェクトがバッファインタフェースをサポートしていることや、`bf_getcharbuffer` スロットが `NULL` でないことを示すわけではありません。

`Py_ssize_t (*readbufferproc) (PyObject* self, Py_ssize_t segment, void** ptrptr)`

`*ptrptr` 中の読み出し可能なバッファセグメントへのポインタを返します。この関数は例外を送出してもよく、送出的場合には `-1` を返さねばなりません。 `segment` に渡す値はゼロまたは正の値で、`bf_getsegcount` スロット関数が返すセグメント数よりも必ず小さな値でなければなりません。成功すると、セグメントのサイズを返し、`*ptrptr` をそのセグメントを指すポインタ値にセットします。

`Py_ssize_t (*writebufferproc) (PyObject* self, Py_ssize_t segment, void** ptrptr)`

読み出し可能なバッファセグメントへのポインタを `*ptrptr` に返し、セグメントの長さを関数の戻り

値として返します。エラーによる例外の場合には `-1` を `-1` を返さねばなりません。オブジェクトが呼び出し専用バッファしかサポートしていない場合には `TypeError` を、`segment` が存在しないセグメントを指している場合には `SystemError` を送出しなければなりません。

`Py_ssize_t (*segcountproc) (PyObject*self, Py_ssize_t*lenp)`

バッファを構成するメモリセグメントの数を返します。 `lenp` が `NULL` でない場合、この関数の実装は全てのセグメントのサイズ(バイト単位)の合計値を `*lenp` を介して報告しなければなりません。この関数呼び出しは失敗させられません。

`Py_ssize_t (*charbufferproc) (PyObject*self, Py_ssize_t segment, char**ptrptr)`

セグメント `segment` のメモリバッファを `ptrptr` に入れ、そのサイズを返します。エラーのときに `-1` を返します。

10.8 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_tsize)`

`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。

バージョン 2.5 で変更: この関数は以前は `size` の型に `int` を利用していました。この変更により、64bit システムを正しくサポートするには修正が必要になります。

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いるではありません。

バージョン 2.5 で変更: この関数は以前は *newsize* の型に *int* を利用していました。この変更により、64 bit システムを正しくサポートするには修正が必要になります。

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。

void **PyObject_GC_TRACK** (*PyObject***op*)

PyObject_GC_Track() のマクロ版です。拡張モジュールに使ってはなりません。

同様に、オブジェクトのメモリ解放関数も以下の二つの規則に適合しなければなりません:

1. 他のコンテナを参照しているフィールドを無効化する前に、*PyObject_GC_UnTrack()* を呼び出さなければなりません。
2. オブジェクトのメモリは *PyObject_GC_Del()* で解放しなければなりません。

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。

void **PyObject_GC_UNTRACK** (*PyObject***op*)

PyObject_GC_UnTrack() のマクロ版です。拡張モジュールに使ってはなりません。

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ければなりません:

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;
}
```

バージョン 2.4 で追加.

`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。

付録 A

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。

... 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。

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) `abstract-base-classes` 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になる場合に インタフェースを定義する方法を提供します。Python は沢山のビルトイン ABCs を、(`collections` モジュールで) データ構造、(`numbers` モジュールで) 数値型、(`io` モジュールで) ストリーム型で提供しています。`abc` モジュールを利用して独自の ABC を作成することもできます。

argument 関数を呼び出す際に、*function* (または *method*) に渡す値です。引数には 2 種類あります。

- キーワード引数: 関数呼び出しの際に引数の前に識別子がついたもの (例: `name=`) や、`**` に続けた辞書の中の値として渡された引数。例えば、次の `complex()` の呼び出しでは、3 と 5 がキーワード引数です:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- 位置引数: キーワード引数以外の引数。位置引数は引数リストの先頭に書くことができ、また `*` に続けた *iterable* の要素として渡すことができます。例えば、次の例では 3 と 5 は両方共位置引数です:

```
complex(3, 5)
complex(*(3, 5))
```

実引数は関数の実体において名前付きのローカル変数に割り当てられます。割り当てを行う規則については `calls` を参照してください。シンタックスにおいて実引数を表すためにあらゆる式を使うことが出来ます。評価された値はローカル変数に割り当てられます。

仮引数、FAQ の 実引数と仮引数の違いは何ですか? を参照してください。

`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。

`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。

`bytes-like object` (バイト様オブジェクト) `str`、`bytearray` や `memoryview` といった *buffer protocol* をサポートするオブジェクトです。バイト様オブジェクトは圧縮、バイナリデータの保存、ソケット経由の送信等の様々な操作に用いることが出来ます。操作によってはバイナリデータはミューテブルでなくてはなりませんが、その場合全てのバイト様オブジェクトを用いることが出来るわけではありません。

`bytecode` (バイトコード) Python のソースコードはバイトコードへとコンパイルされます。バイトコードは Python プログラムの CPython インタプリタ内部での表現です。バイトコードはまた、`.pyc` や `.pyo` ファイルにキャッシュされ、同じファイルの実行が二度目に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)。このバイトコードは、各々のバイトコードに対応するサブルーチン呼び出すような "仮想機械 (*virtual machine*)" で動作する "中間言語 (*intermediate language*)" 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は期待されていませんし、Python リリース間で安定でもありません。

バイトコードの命令一覧は `dis` モジュールにあります。

`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。

`classic class` (旧スタイルクラス) `object` を継承していないクラス全てを指します。新スタイルクラス (*new-style class*) も参照してください。旧スタイルクラスは Python 3 で削除されます。

`coercion` (型強制) 同じ型の 2 つの引数を要する演算の最中に、ある型のインスタンスを別の型に暗黙のうちに変換することです。例えば、`int(3.15)` は浮動小数点数を整数の 3 にします。しかし、`3+4.5` の場合、各引数は型が異なっていて (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しなければいけません。そうでないと、`TypeError` 例外が投げられます。2 つの被演算子間の型強制は組み込み関数の `coerce` を使って行えます。従って、`3+4.5` は `operator.add(*coerce(3, 4.5))` を呼び出すことに等しく、`operator.add(3.0, 4.5)` という結果になります。型強制を行わない場合、たとえ互換性のある型であっても、すべての引数はプログラマーが、単に `3+4.5` とするのではなく、`float(3)+4.5` というように、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) `with` 文で扱われる、`__enter__()` と `__exit__()` メソッドを定義することで環境を制御するオブジェクトです。PEP 343 を参照してください。

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)

@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__()` が定義されている新スタイル (*new-style*) のオブジェクトです。あるクラス属性がデスクリプタである場合、その属性を参照するときに、そのデスクリプタに束縛されている特別な動作を呼び出します。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` がデスクリプタの場合にはデスクリプタで定義されたメソッドを呼び出します。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 view (ビュー) `dict.viewkeys()`, `dict.viewvalues()`, `dict.viewitems()` が返すオブジェクトを辞書ビュー (dictionary view) と呼びます。これらは辞書のエントリに対する動的なビューで、つまり辞書の変更されるとビューはそれら変更を反映します。辞書ビューを完全なリストにするには `list(dictview)` としてください。`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

で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) 何かの値に評価される、一つづきの構文 (a piece of syntax). 言い換えると、リテラル、名前、属性アクセス、演算子や関数呼び出しといった、値を返す式の要素の組み合わせ。他の多くの言語と違い、Python 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。 `print` や `if` のように、式にはならない、文 (*statement*) 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。

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 3 つの種類があります: 生のバイナリーファイル、バッファされたバイナリーファイル、そしてテキストファイルです。これらのインターフェースは `io` モジュール内で定義されています。ファイルオブジェクトを作成する標準的な方法は `open()` 関数を利用することです。

file-like object *file object* と同義です。

finder モジュールの *loader* を探すオブジェクト。 `findmodule()` という名前のメソッドを実装していなければなりません。詳細については **PEP 302** 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。

関数 (関数) 呼び出し側に値を返す一連の文のことです。関数には 0 以上の **実引数** を渡すことが出来ます。実体の実行時に引数を使用することが出来ます。 **仮引数**、**メソッド**、`function` を参照してください。

__future__ 互換性のない新たな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。例えば、式 `11/4` は現状では 2 になります。この式を実行しているモジュールで

```
from __future__ import division
```

を行って 真の除算操作 (*true division*) を有効にすると、式 `11/4` は `2.75` になります。実際に `__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がいつで、いつデフォルトの機能になる予定かわかります。


```
>>> 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) それ以上使われなくなったメモリを解放する処理です。Python は、参照カウントと、循環参照を見つけて破壊する循環ガベージコレクタと、を使ってガベージコレクションを行います。

generator A function which returns an iterator. It looks like a normal function except that it contains `yield` statements for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function. Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

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` 式、範囲、そして省略可能な `if` 式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
```

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 (*bytecode*) 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) などの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に対して、楽にインタプリタをマルチスレッド化できます。

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 を解放するように設計されています。また、I/O 処理をするときも GIL は常に解放されます。

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。

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__()` か `__cmp__()` 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ハッシュ可能ですが、ミュータブルなコンテ

ナ (例えばリストや辞書) はハッシュ不可能です。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。それらは全て非等価を比較し (自身を除いて)、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。

integer division (整数除算) 剰余を考慮しない数学的除算です。例えば、式 $11/4$ は現状では 2.75 ではなく 2 になります。これは切り捨て除算 (*floor division*) とも呼ばれます。二つの整数間で除算を行うと、結果は (端数切捨て関数が適用されて) 常に整数になります。しかし、被演算子の一方が (`float` のような) 別の数値型の場合、演算の結果は共通の型に型強制されます (型強制 (*coercion*) 参照)。例えば、浮動小数点数で整数を除算すると結果は浮動小数点になり、場合によっては端数部分を伴います。// 演算子を / の代わりに使うと、整数除算を強制できます。 `__future__` も参照してください。

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 (*interactive*) も参照してください。

iterable (反復可能オブジェクト) 要素を一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 `file` といった幾つかの非シーケンス型、あるいは `__iter__()` か `__getitem__()` メソッドを実装した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。反復可能オブジェクトを使う際には、通常 `iter()` を呼んだり、イテレータオブジェクトを自分で扱う必要はありません。 `for` 文ではこの操作を自動的に扱い、無名の変数を作成してループの間イテレータを記憶します。イテレータ (*iterator*) シーケンス (*sequence*), およびジェネレータ (*generator*) 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て呼び出すと、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` を送じます。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 には、キー関数を受け付けて要素がどのように順序付けられたりグループ化されたりするかを制御するような、いくつかのツールがあります。例えば、`min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, `itertools.groupby()` です。

キー関数を作る方法はいくつかあります。例えば、`str.lower()` メソッドをキー関数として使って大文字小文字を区別しないソートができます。他には、`lambda` 式を使って `lambda r: (r[0], r[2])` のようなアドホックな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、使い方に関する例は、`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 (*expression*) 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 (*sequence*) 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、その結果からなるリストを返す、コンパクトな書き方です。`result = ["0x%02x" % 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。詳細は [PEP 302](#) を参照してください。

magic method [special method](#) のくだけた同義語です。

mapping (マップ、マッピング) 任意のキーに対する検索をサポートしていて、`Mapping` が `MutableMapping` の抽象基底クラスを実装しているコンテナオブジェクト。例えば、`dict`, `collections.defaultdict`, `collections.OrderedDict`, `collections.Counter` はマップ型です。

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 (*argument*) 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。

モジュール (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。

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 (*immutable*) も参照してください。

named tuple (名前付きタプル) タプルに似ていて、インデックス指定できる要素に名前付き属性でもアクセス出来るクラスです (例えば、`time.localtime()` はタプルに似たオブジェクトを返し、その `year` には `t[0]` のようなインデックスによるアクセスと、`t.tm_year` のような名前付き要素としてのアクセスが可能です)。

名前付きタプルには、`time.struct_time` のような組み込み型もありますし、通常のクラス定義によって作成することもできます。名前付きタプルを `collections.namedtuple()` ファクトリ関数で作成することもできます。最後の方法で作った名前付きタプルには自動的に、`Employee(name='jones', title='programmer')` のような自己ドキュメント表現 (self-documenting representation) などの機能が付いてきます。

namespace (名前空間) 変数を記憶している場所です。名前空間は辞書を用いて実装されています。名前

空間には、ローカル、グローバル、組み込み名前空間、そして(メソッド内の)オブジェクトのネストされた名前空間があります。例えば、関数 `_builtin_.open()` と `os.open()` は名前空間で区別されます。名前空間はまた、ある関数をどのモジュールが実装しているかをはっきりさせることで、可読性やメンテナンス性に寄与します。例えば、`random.seed()`、`itertools.izip()` と書くことで、これらの関数がそれぞれ `random` モジュールや `itertools` モジュールで実装されていることがはっきりします。

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。具体的に言えば、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からは変数の参照だけができ、外側の変数を変更できないことに注意してください。それは常に最も内側のスコープに対する書き込みになります。対照的に、ローカル変数は最も内側のスコープ内の読み書き両方します。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。

new-style class (新スタイルクラス) `object` から継承したクラス全てを指します。これには `list` や `dict` のような全ての組み込み型が含まれます。`__slots__()`、デスクリプタ、プロパティ、`__getattr__()` といった、Python の新しい機能を使うのは新スタイルクラスだけです。

より詳しい情報は `newstyle` を参照してください。

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 (*new-style class*) 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。

parameter (仮引数) 名前付の実体で `term:関数<function>` (や `term:メソッド<method>`) の定義において関数が受ける **実引数** を明示します。仮引数には 4 種類あります:

- 位置またはキーワード: **位置** または **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。Python に位置専用仮引数を定義する文法はありません。しかし、組み込み関数には位置専用仮引数を持つもの(例: `abs()`)があります。
- 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。

argument、FAQ の 実引数と仮引数の違いは何ですか?、function を参照してください。

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。

位置引数 実引数 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
```

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。

__slots__ 新スタイルクラス (*new-style class*) 内で、インスタンス属性の記憶に必要な領域をあらかじめ定義しておき、それとひきかえにインスタンス辞書を排除してメモリの節約を行うための宣言です。これはよく使われるテクニックですが、正しく動作させるのには少々手際を要するので、例えばメモリが死活問題となるようなアプリケーション内にインスタンスが大量に存在するといった稀なケースを除き、使わないのがベストです。

sequence (シーケンス) 特殊メソッド `__getitem__()` で整数インデックスによる効率的な要素へのアクセスをサポートし、`len()` で長さを返すような反復可能オブジェクト (*iterable*) です。組み込みシーケンス型には、`list`, `str`, `tuple`, `unicode` などがあります。`dict` は `__getitem__()` と `__len__()` もサポートしますが、検索の際に任意の変更不能 (*immutable*) なキーを使うため、シーケンスではなくマップ (mapping) とみなされているので注意してください。

slice (スライス) 多くの場合、シーケンス (*sequence*) の一部を含むオブジェクト。スライスは、添字記号 `[]` で数字の間にコロンを書いたときに作られます。例えば、`variable_name[1:3:5]` です。添字記

号は `slice` オブジェクトを内部で利用しています。(もしくは、古いバージョンの、`__getslice__()` と `__setslice__()` を利用します。)

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 つがついていません。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。

struct sequence A tuple with named elements. Struct sequences expose an interface similiar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

triple-quoted string (三重クオート文字列) 3 つの連続したクオート記号 (") かアポストロフィー (') で囲まれた文字列。通常の (一重) クオ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 つの連続したクオ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の、ドキュメンテーション文字列を書く時に特に便利です。

型 (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。

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 `str.splitlines()` も参照してください。

virtual environment (仮想環境) 協調的に切り離された実行環境です。これにより Python ユーザとアプリケーションは同じシステム上で動いている他の Python アプリケーションの挙動に干渉することなく Python パッケージのインストールと更新を行うことができます。

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 (*bytecode*) を実行します。

Zen of Python (Python の悟り) Python を理解し利用する上での導きとなる、Python の設計原則と哲学をリストにしたものです。対話プロンプトで `"import this"` とするとこのリストを読めます。

付録 B

このドキュメントについて

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セットの開発は、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!

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!

付録 C

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 Research 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。

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen Python-Labs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see <https://www.zope.org/>). In 2001, the Python Software Foundation (PSF, see <https://www.python.org/psf/>) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

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

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 2.7.17

1. This LICENSE AGREEMENT is between the Python Software Foundation,
→ ("PSF"), and
the Individual or Organization ("Licensee") accessing and otherwise,
→ using Python
2.7.17 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 2.7.17 alone or in any derivative
version, provided, however, that PSF's License Agreement and PSF's,
→ notice of
copyright, i.e., "Copyright ^c2^a9 2001-2019 Python Software,
→ Foundation; All Rights
Reserved" are retained in Python 2.7.17 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 2.7.17 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
2.7.17.

4. PSF is making Python 2.7.17 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 2.7.17 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.17
FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A
→RESULT OF
MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.17, 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 2.7.17, 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

(次のページに続く)

(前のページからの続き)

- to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
 4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
 5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
 6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
 7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement,

(次のページに続く)

(前のページからの続き)

Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: <http://hdl.handle.net/1895.22/1013>."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing 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.

(次のページに続く)

(前のページからの続き)

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.3 取り入れられているソフトウェアのライセンスと謝辞

この節では、Python 配布物に取り入れられているサードパーティソフトウェアのライセンスと謝辞の、不完全だが成長し続けるリストを記述します。

C.3.1 メルセンヌツイスター (Mersenne Twister)

`random` モジュールは、<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html> からダウンロードしたコードに基づいたコードベースを含んでいます。オリジナルのコードのコメントをそのまま次に示します:

```
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.
```

(次のページに続く)

(前のページからの続き)

3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

C.3.2 ソケット

socket モジュールは、WIDE プロジェクト <http://www.wide.ad.jp/> の別々のソースファイルにコーディングされていた `getaddrinfo()` 関数と `getnameinfo()` 関数を使っています:

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)

(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.3.3 浮動小数点数例外の制御

fpectl モジュールのソースコードは次の告知を含んでいます:

```
-----  
/                               Copyright (c) 1996.                               \  
|                               The Regents of the University of California.          \  
|                               All rights reserved.                                \  
|                                                                                 \  
| Permission to use, copy, modify, and distribute this software for                \  
| any purpose without fee is hereby granted, provided that this en-               \  
| tire notice is included in all copies of any software which is or               \  
| includes a copy or modification of this software and in all                    \  
| copies of the supporting documentation for such software.                       \  
|                                                                                 \  
| This work was produced at the University of California, Lawrence                 \  
| Livermore National Laboratory under contract no. W-7405-ENG-48                   \  
| between the U.S. Department of Energy and The Regents of the                   \  
| University of California for the operation of UC LLNL.                          \  
|                                                                                 \  
|                               DISCLAIMER                                             \  
|                                                                                 \  
| This software was prepared as an account of work sponsored by an                \  
| agency of the United States Government. Neither the United States               \  
| Government nor the University of California nor any of their em-               \  
| ployees, makes any warranty, express or implied, or assumes any                 \  
| liability or responsibility for the accuracy, completeness, or                  \  
| usefulness of any information, apparatus, product, or process                   \  
| disclosed, or represents that its use would not infringe                       \  
| privately-owned rights. Reference herein to any specific commer-               \  
| cial products, process, or service by trade name, trademark,                   \  
| manufacturer, or otherwise, does not necessarily constitute or                  \  
| imply its endorsement, recommendation, or favoring by the United               \  
| States Government or the University of California. The views and                 \  
| opinions of authors expressed herein do not necessarily state or                \  
| reflect those of the United States Government or the University                 \  
| of California, and shall not be used for advertising or product                 \  
| \ endorsement purposes.                                                         \  
-----
```

C.3.4 MD5 メッセージダイジェストアルゴリズム

md5 モジュールのソースコードは次の告知を含んでいます:

Copyright (C) 1999, 2002 Aladdin Enterprises. All rights reserved.

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.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose text is available at

<http://www.ietf.org/rfc/rfc1321.txt>

The code is derived from the text of the RFC, including the test suite (section A.5) but excluding the rest of Appendix A. It does not include any code or documentation that is identified in the RFC as being copyrighted.

The original and principal author of md5.h is L. Peter Deutsch <ghost@aladdin.com>. Other authors are noted in the change history that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed references to Ghostscript; clarified derivation from RFC 1321; now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5); added conditionalization for C++ compilation from Martin Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.

C.3.5 非同期ソケットサービス

asynchat モジュールと asyncore モジュールは次の告知を含んでいます:

Copyright 1996 by Sam Rushing

(次のページに続く)

(前のページからの続き)

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.3.6 Cookie の管理

Cookie モジュールは次の告知を含んでいます:

Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Timothy O'Malley not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.7 実行の追跡

trace モジュールは次の告知を含んでいます:

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.3.8 UUencode 関数と UUdecode 関数

uu モジュールは次の告知を含んでいます:

Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
Modified by Jack Jansen, CWI, July 1995:

(次のページに続く)

(前のページからの続き)

- 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.3.9 XML リモートプロシージャコール

`xmlrpclib` モジュールは次の告知を含んでいます:

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.3.10 `test_epoll`

`test_epoll` 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

(次のページに続く)

(前のページからの続き)

```

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.3.11 Select kqueue

select は kqueue インターフェースについての次の告知を含んでいます:

```

Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

```

```

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

```

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

```

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.

```

C.3.12 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
 * 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.3.13 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

```

(次のページに続く)

(前のページからの続き)

```

*   distribution.
*
* 3. All advertising materials mentioning features or use of this
*   software must display the following acknowledgment:
*   "This product includes software developed by the OpenSSL Project
*   for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
*
* 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
*   endorse or promote products derived from this software without
*   prior written permission. For written permission, please contact
*   openssl-core@openssl.org.
*
* 5. Products derived from this software may not be called "OpenSSL"
*   nor may "OpenSSL" appear in their names without prior written
*   permission of the OpenSSL Project.
*
* 6. Redistributions of any form whatsoever must retain the following
*   acknowledgment:
*   "This product includes software developed by the OpenSSL Project
*   for use in the OpenSSL Toolkit (http://www.openssl.org/)"
*
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com).  This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

```

Original SSLeay License

```

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*

```

(次のページに続く)

(前のページからの続き)

```
* 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
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
*    must display the following acknowledgement:
*    "This product includes cryptographic software written by
*      Eric Young (eay@cryptsoft.com)"
*    The word 'cryptographic' can be left out if the rouines from the library
*    being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
*    the apps directory (application code) you must include an acknowledgement:
*    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/
```

C.3.14 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.3.15 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.3.16 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) 1995-2010 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
jloup@gzip.org

Mark Adler
madler@alumni.caltech.edu

付録 D

Copyright

Python and this documentation is:

Copyright 2001-2019 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

..., **199**
 __all__ (package variable), 33
 __builtin__
 モジュール, 11, 141
 __dict__ (module attribute), 124
 __doc__ (module attribute), 124
 __file__ (module attribute), 124
 __future__, **202**
 __import__
 組み込み関数, 34
 __main__
 モジュール, 11, 141, 152
 __name__ (module attribute), 124
 __slots__, **208**
 _frozen (C のデータ型), 36
 _inittab (C のデータ型), 37
 _Py_c_diff (C の関数), 80
 _Py_c_neg (C の関数), 80
 _Py_c_pow (C の関数), 81
 _Py_c_prod (C の関数), 80
 _Py_c_quote (C の関数), 81
 _Py_c_sum (C の関数), 80
 _Py_NoneStruct (C の変数), 167
 _PyImport_FindExtension (C の関数), 36
 _PyImport_Fini (C の関数), 36
 _PyImport_FixupExtension (C の関数), 36
 _PyImport_Init (C の関数), 36
 _PyObject_Del (C の関数), 165
 _PyObject_GC_TRACK (C の関数), 197
 _PyObject_GC_UNTRACK (C の関数), 197
 _PyObject_New (C の関数), 165
 _PyObject_NewVar (C の関数), 165
 _PyString_Resize (C の関数), 85
 _PyTuple_Resize (C の関数), 112
 >>>, **199**
 2to3, **199**

 abort (), 33
 abs
 組み込み関数, 60
 abstract base class, **199**
 apply
 組み込み関数, 56
 argument, **199**
 argv (in module sys), 144
 attribute, **200**

 BDFL, **200**
 buffer
 オブジェクト, 104
 buffer interface, 104
 BufferType (in module types), 109
 builtins
 モジュール, 152
 bytearray
 オブジェクト, 82
 bytecode, **200**
 bytes
 組み込み関数, 55
 bytes-like object, **200**

 calloc (), 159

Capsule
 オブジェクト, 129
 charbufferproc (C のデータ型), 196
 class, **200**
 class
 オブジェクト, 118
 classic class, **200**
 classmethod
 組み込み関数, 170
 ClassType (in module types), 119
 cleanup functions, 33
 close () (in module os), 153
 cmp
 組み込み関数, 55
 CO_FUTURE_DIVISION (C の変数), 17
 CObject
 オブジェクト, 131
 code object, 139
 coerce
 組み込み関数, 62
 coercion, **200**
 compile
 組み込み関数, 35
 complex number, **200**
 complex number
 オブジェクト, 80
 context manager, **201**
 copyright (in module sys), 144
 CPython, **201**

 decorator, **201**
 descriptor, **201**
 dictionary, **201**
 dictionary
 オブジェクト, 115
 dictionary view, **201**
 DictionaryType (in module types), 115
 DictType (in module types), 115
 divmod
 組み込み関数, 59
 docstring, **201**
 duck-typing, **201**

 EAFP, **202**
 EOFError (built-in exception), 123
 exc_info () (in module sys), 9
 exc_traceback (in module sys), 9, 21
 exc_type (in module sys), 9, 21
 exc_value (in module sys), 9, 21
 exceptions
 モジュール, 11
 exec.prefix, 4
 executable (in module sys), 143
 exit (), 33
 expression, **202**
 extension module, **202**

 file
 オブジェクト, 121
 file object, **202**
 file-like object, **202**
 FileType (in module types), 121

finder, [202](#)
float
 組み込み関数, [62](#)
floating point
 オブジェクト, [78](#)
FloatType (*in modules types*), [79](#)
floor division, [202](#)
fopen(), [122](#)
free(), [159](#)
freeze utility, [36](#)
frozenset
 オブジェクト, [137](#)
function
 オブジェクト, [119](#)

garbage collection, [203](#)
generator, [203](#)
generator, [203](#)
generator expression, [203](#)
generator expression, [203](#)
GIL, [203](#)
GIL, [145](#)
global interpreter lock, [203](#)
global interpreter lock, [145](#)

hash
 組み込み関数, [57](#), [178](#)
hashable, [203](#)

IDLE, [204](#)
ihooks
 モジュール, [34](#)
immutable, [204](#)
importer, [204](#)
importing, [204](#)
incr_item(), [10](#)
inquiry (C のデータ型), [198](#)
instance
 オブジェクト, [119](#)
int
 組み込み関数, [62](#)
integer
 オブジェクト, [73](#)
integer division, [204](#)
interactive, [204](#)
interpreted, [204](#)
interpreter lock, [145](#)
IntType (*in modules types*), [73](#)
iterable, [204](#)
iterator, [204](#)

key function, [205](#)
KeyboardInterrupt (*built-in exception*), [26](#)
keyword argument, [205](#)

lambda, [205](#)
LBYL, [205](#)
len
 組み込み関数, [58](#), [63](#), [66](#), [113](#), [117](#), [138](#)
list, [205](#)
list
 オブジェクト, [113](#)
list comprehension, [205](#)
loader, [205](#)
lock, interpreter, [145](#)
long
 組み込み関数, [62](#)
long integer
 オブジェクト, [75](#)
LONG_MAX, [74](#), [77](#)
LongType (*in modules types*), [75](#)

magic

 method, [206](#)
magic method, [206](#)
main(), [142](#), [144](#)
malloc(), [159](#)
mapping, [206](#)
mapping
 オブジェクト, [115](#)
metaclass, [206](#)
METH.CLASS (組み込み変数), [170](#)
METH.COEXIST (組み込み変数), [170](#)
METH.KEYWORDS (組み込み変数), [169](#)
METH.NOARGS (組み込み変数), [169](#)
METH.O (組み込み変数), [169](#)
METH.OLDARGS (組み込み変数), [169](#)
METH.STATIC (組み込み変数), [170](#)
METH.VARARGS (組み込み変数), [169](#)
method, [206](#)
method
 magic, [206](#)
 special, [209](#)
 オブジェクト, [120](#)
method resolution order, [206](#)
MethodType (*in module types*), [119](#), [120](#)
module
 search path, [11](#), [141](#), [143](#)
 オブジェクト, [124](#)
modules (*in module sys*), [33](#), [141](#)
ModuleType (*in module types*), [124](#)
MRO, [206](#)
mutable, [206](#)

named tuple, [206](#)
namespace, [206](#)
nested scope, [207](#)
new-style class, [207](#)
None
 オブジェクト, [73](#)
numeric
 オブジェクト, [73](#)

object, [207](#)
object
 code, [139](#)
OverflowError (*built-in exception*), [77](#), [78](#)

package, [207](#)
package variable
 __all__, [33](#)
parameter, [207](#)
PATH, [11](#)
path
 module search, [11](#), [141](#), [143](#)
path (*in module sys*), [11](#), [141](#), [143](#)
PEP, [208](#)
platform (*in module sys*), [144](#)
pow
 組み込み関数, [59](#), [61](#)
prefix, [4](#)
Py_AddPendingCall (C の関数), [154](#)
Py_AddPendingCall(), [154](#)
Py_AtExit (C の関数), [33](#)
Py_BEGIN_ALLOW_THREADS, [146](#)
Py_BEGIN_ALLOW_THREADS (C のマクロ), [150](#)
Py_BLOCK_THREADS (C のマクロ), [150](#)
Py_buffer (C のデータ型), [104](#)
Py_buffer.buf (C のメンバ変数), [104](#)
Py_buffer.internal (C のメンバ変数), [106](#)
Py_buffer.itemsize (C のメンバ変数), [105](#)
Py_buffer.ndim (C のメンバ変数), [105](#)
Py_buffer.readonly (C のメンバ変数), [105](#)
Py_buffer.shape (C のメンバ変数), [105](#)
Py_buffer.strides (C のメンバ変数), [105](#)
Py_buffer.suboffsets (C のメンバ変数), [105](#)

- Py_BuildValue (C の関数), 45
- Py_CLEAR (C の関数), 19
- Py_CompileString (C の関数), 16
- Py_CompileString(), 16, 17
- Py_CompileStringFlags (C の関数), 16
- Py_complex (C のデータ型), 80
- Py_DECREF (C の関数), 19
- Py_DECREF(), 5
- Py_Ellipsis (C の変数), 127
- Py_END_ALLOW_THREADS, 146
- Py_END_ALLOW_THREADS (C のマクロ), 150
- Py_END_OF_BUFFER (C の変数), 109
- Py_EndInterpreter (C の関数), 153
- Py_EnterRecursiveCall (C の関数), 28
- Py_eval_input (C の変数), 16
- Py_Exit (C の関数), 33
- Py_False (C の変数), 75
- Py_FatalError (C の関数), 33
- Py_FatalError(), 144
- Py_FdIsInteractive (C の関数), 31
- Py_file_input (C の変数), 16
- Py_Finalize (C の関数), 141
- Py_Finalize(), 33, 141, 153
- Py_FindMethod (C の関数), 172
- Py_GetBuildInfo (C の関数), 144
- Py_GetCompiler (C の関数), 144
- Py_GetCopyright (C の関数), 144
- Py_GetExecPrefix (C の関数), 142
- Py_GetExecPrefix(), 11
- Py_GetPath (C の関数), 143
- Py_GetPath(), 11, 142
- Py_GetPlatform (C の関数), 144
- Py_GetPrefix (C の関数), 142
- Py_GetPrefix(), 11
- Py_GetProgramFullPath (C の関数), 143
- Py_GetProgramFullPath(), 11
- Py_GetProgramName (C の関数), 142
- Py_GetPythonHome (C の関数), 145
- Py_GetVersion (C の関数), 143
- Py_INCREF (C の関数), 19
- Py_INCREF(), 5
- Py_Initialize (C の関数), 141
- Py_Initialize(), 11, 142, 148, 153
- Py_InitializeEx (C の関数), 141
- Py_InitModule (C の関数), 166
- Py_InitModule3 (C の関数), 166
- Py_InitModule4 (C の関数), 166
- Py_IsInitialized (C の関数), 141
- Py_IsInitialized(), 11
- Py_LeaveRecursiveCall (C の関数), 28
- Py_Main (C の関数), 13
- Py_NewInterpreter (C の関数), 152
- Py_None (C の変数), 73
- Py_PRINT_RAW, 123
- Py_REFCNT (C のマクロ), 168
- Py_RETURN_FALSE (C のマクロ), 75
- Py_RETURN_NONE (C のマクロ), 73
- Py_RETURN_TRUE (C のマクロ), 75
- Py_SetProgramName (C の関数), 142
- Py_SetProgramName(), 11, 141–143
- Py_SetPythonHome (C の関数), 145
- Py_single_input (C の変数), 17
- Py_SIZE (C のマクロ), 168
- PY_SSIZE_T_MAX, 77
- Py_TPFLAGS_BASETYPE (組み込み変数), 182
- Py_TPFLAGS_CHECKTYPES (組み込み変数), 181
- Py_TPFLAGS_DEFAULT (組み込み変数), 182
- Py_TPFLAGS_GC (組み込み変数), 180
- Py_TPFLAGS_HAVE_CLASS (組み込み変数), 181
- Py_TPFLAGS_HAVE_GC (組み込み変数), 182
- Py_TPFLAGS_HAVE_GETCHARBUFFER (組み込み変数), 180, 195
- Py_TPFLAGS_HAVE_INPLACEOPS (組み込み変数), 181
- Py_TPFLAGS_HAVE_ITER (組み込み変数), 181
- Py_TPFLAGS_HAVE_RICHCOMPARE (組み込み変数), 181
- Py_TPFLAGS_HAVE_SEQUENCE_IN (組み込み変数), 180
- Py_TPFLAGS_HAVE_WEAKREFS (組み込み変数), 181
- Py_TPFLAGS_HEAPTYPE (組み込み変数), 181
- Py_TPFLAGS_READY (組み込み変数), 182
- Py_TPFLAGS_READYING (組み込み変数), 182
- Py_tracefunc (C のデータ型), 154
- Py_True (C の変数), 75
- Py_TYPE (C のマクロ), 168
- Py_UNBLOCK_THREADS (C のマクロ), 150
- Py_UNICODE (C のデータ型), 88
- Py_UNICODE_ISALNUM (C の関数), 89
- Py_UNICODE_ISALPHA (C の関数), 89
- Py_UNICODE_ISDECIMAL (C の関数), 89
- Py_UNICODE_ISDIGIT (C の関数), 89
- Py_UNICODE_ISLINEBREAK (C の関数), 89
- Py_UNICODE_ISLOWER (C の関数), 89
- Py_UNICODE_ISNUMERIC (C の関数), 89
- Py_UNICODE_ISSPACE (C の関数), 89
- Py_UNICODE_ISTITLE (C の関数), 89
- Py_UNICODE_ISSUPPER (C の関数), 89
- Py_UNICODE_TODECIMAL (C の関数), 90
- Py_UNICODE_TODIGIT (C の関数), 90
- Py_UNICODE_TOLOWER (C の関数), 90
- Py_UNICODE_TONUMERIC (C の関数), 90
- Py_UNICODE_TOTITLE (C の関数), 90
- Py_UNICODE_TOUPPER (C の関数), 90
- Py_VaBuildValue (C の関数), 47
- Py_VISIT (C の関数), 198
- Py_XDECREF (C の関数), 19
- Py_XDECREF(), 10
- Py_XINCRF (C の関数), 19
- PyAnySet_Check (C の関数), 137
- PyAnySet_CheckExact (C の関数), 137
- PyArg_Parse (C の関数), 44
- PyArg_ParseTuple (C の関数), 43
- PyArg_ParseTupleAndKeywords (C の関数), 44
- PyArg_UnpackTuple (C の関数), 44
- PyArg_VaParse (C の関数), 44
- PyArg_VaParseTupleAndKeywords (C の関数), 44
- PyBool_Check (C の関数), 75
- PyBool_FromLong (C の関数), 75
- PyBuffer_Check (C の関数), 109
- PyBuffer_FillContiguousStrides (C の関数), 108
- PyBuffer_FillInfo (C の関数), 108
- PyBuffer_FromMemory (C の関数), 110
- PyBuffer_FromObject (C の関数), 109
- PyBuffer_FromReadWriteMemory (C の関数), 110
- PyBuffer_FromReadWriteObject (C の関数), 110
- PyBuffer_IsContiguous (C の関数), 108
- PyBuffer_New (C の関数), 110
- PyBuffer_Release (C の関数), 107
- PyBuffer_SizeFromFormat (C の関数), 108
- PyBuffer_Type (C の変数), 109
- PyBufferObject (C のデータ型), 109
- PyBufferProcs, 109
- PyBufferProcs (C のデータ型), 195
- PyByteArray_AS_STRING (C の関数), 83
- PyByteArray_AsString (C の関数), 82
- PyByteArray_Check (C の関数), 82
- PyByteArray_CheckExact (C の関数), 82
- PyByteArray_Concat (C の関数), 82
- PyByteArray_FromObject (C の関数), 82
- PyByteArray_FromStringAndSize (C の関数), 82
- PyByteArray_GET_SIZE (C の関数), 83
- PyByteArray_Resize (C の関数), 82
- PyByteArray_Size (C の関数), 82
- PyByteArray_Type (C の変数), 82
- PyByteArrayObject (C のデータ型), 82
- PyCallable_Check (C の関数), 56
- PyCallIter_Check (C の関数), 126
- PyCallIter_New (C の関数), 126
- PyCallIter_Type (C の変数), 125

PyCapsule (*C* のデータ型), 129
 PyCapsule_CheckExact (*C* の関数), 129
 PyCapsule_Destructor (*C* のデータ型), 129
 PyCapsule_GetContext (*C* の関数), 130
 PyCapsule_GetDestructor (*C* の関数), 130
 PyCapsule_GetName (*C* の関数), 130
 PyCapsule_GetPointer (*C* の関数), 130
 PyCapsule_Import (*C* の関数), 130
 PyCapsule_IsValid (*C* の関数), 130
 PyCapsule_New (*C* の関数), 129
 PyCapsule_SetContext (*C* の関数), 131
 PyCapsule_SetDestructor (*C* の関数), 131
 PyCapsule_SetName (*C* の関数), 131
 PyCapsule_SetPointer (*C* の関数), 131
 PyCell_Check (*C* の関数), 132
 PyCell_GET (*C* の関数), 132
 PyCell_Get (*C* の関数), 132
 PyCell_New (*C* の関数), 132
 PyCell_SET (*C* の関数), 133
 PyCell_Set (*C* の関数), 132
 PyCell_Type (*C* の変数), 132
 PyCellObject (*C* のデータ型), 132
 PyCFunction (*C* のデータ型), 168
 PyClass_Check (*C* の関数), 119
 PyClass_IsSubclass (*C* の関数), 119
 PyClass_Type (*C* の変数), 118
 PyClassObject (*C* のデータ型), 118
 PyCObject (*C* のデータ型), 131
 PyCObject_AsVoidPtr (*C* の関数), 132
 PyCObject_Check (*C* の関数), 131
 PyCObject_FromVoidPtr (*C* の関数), 131
 PyCObject_FromVoidPtrAndDesc (*C* の関数), 132
 PyCObject_GetDesc (*C* の関数), 132
 PyCObject_SetVoidPtr (*C* の関数), 132
 PyCode_Check (*C* の関数), 139
 PyCode_GetNumFree (*C* の関数), 139
 PyCode_New (*C* の関数), 139
 PyCode_NewEmpty (*C* の関数), 139
 PyCode_Type (*C* の変数), 139
 PyCodec_BackslashReplaceErrors (*C* の関数), 52
 PyCodec_Decode (*C* の関数), 50
 PyCodec_Decoder (*C* の関数), 51
 PyCodec_Encode (*C* の関数), 50
 PyCodec_Encoder (*C* の関数), 51
 PyCodec_IgnoreErrors (*C* の関数), 51
 PyCodec_IncrementalDecoder (*C* の関数), 51
 PyCodec_IncrementalEncoder (*C* の関数), 51
 PyCodec_KnownEncoding (*C* の関数), 50
 PyCodec_LookupError (*C* の関数), 51
 PyCodec_Register (*C* の関数), 50
 PyCodec_RegisterError (*C* の関数), 51
 PyCodec_ReplaceErrors (*C* の関数), 51
 PyCodec_StreamReader (*C* の関数), 51
 PyCodec_StreamWriter (*C* の関数), 51
 PyCodec_StrictErrors (*C* の関数), 51
 PyCodec_XMLCharRefReplaceErrors (*C* の関数), 52
 PyCodeObject (*C* のデータ型), 139
 PyCompilerFlags (*C* のデータ型), 17
 PyComplex_AsCComplex (*C* の関数), 81
 PyComplex_Check (*C* の関数), 81
 PyComplex_CheckExact (*C* の関数), 81
 PyComplex_FromCComplex (*C* の関数), 81
 PyComplex_FromDoubles (*C* の関数), 81
 PyComplex_ImagAsDouble (*C* の関数), 81
 PyComplex_RealAsDouble (*C* の関数), 81
 PyComplex_Type (*C* の変数), 81
 PyComplexObject (*C* のデータ型), 81
 PyDate_Check (*C* の関数), 133
 PyDate_CheckExact (*C* の関数), 133
 PyDate_FromDate (*C* の関数), 135
 PyDate_FromTimestamp (*C* の関数), 136
 PyDateTime_Check (*C* の関数), 134
 PyDateTime_CheckExact (*C* の関数), 134

PyDateTime_DATE_GET_HOUR (*C* の関数), 135
 PyDateTime_DATE_GET_MICROSECOND (*C* の関数), 136
 PyDateTime_DATE_GET_MINUTE (*C* の関数), 136
 PyDateTime_DATE_GET_SECOND (*C* の関数), 136
 PyDateTime_FromDateAndTime (*C* の関数), 135
 PyDateTime_FromTimestamp (*C* の関数), 136
 PyDateTime_GET_DAY (*C* の関数), 135
 PyDateTime_GET_MONTH (*C* の関数), 135
 PyDateTime_GET_YEAR (*C* の関数), 135
 PyDateTime_TIME_GET_HOUR (*C* の関数), 136
 PyDateTime_TIME_GET_MICROSECOND (*C* の関数), 136
 PyDateTime_TIME_GET_MINUTE (*C* の関数), 136
 PyDateTime_TIME_GET_SECOND (*C* の関数), 136
 PyDelta_Check (*C* の関数), 134
 PyDelta_CheckExact (*C* の関数), 134
 PyDelta_FromDSU (*C* の関数), 135
 PyDescr_IsData (*C* の関数), 126
 PyDescr_NewClassMethod (*C* の関数), 126
 PyDescr_NewGetSet (*C* の関数), 126
 PyDescr_NewMember (*C* の関数), 126
 PyDescr_NewMethod (*C* の関数), 126
 PyDescr_NewWrapper (*C* の関数), 126
 PyDict_Check (*C* の関数), 115
 PyDict_CheckExact (*C* の関数), 115
 PyDict_Clear (*C* の関数), 116
 PyDict_Contains (*C* の関数), 116
 PyDict_Copy (*C* の関数), 116
 PyDict_DelItem (*C* の関数), 116
 PyDict_DelItemString (*C* の関数), 116
 PyDict_GetItem (*C* の関数), 116
 PyDict_GetItemString (*C* の関数), 116
 PyDict_Items (*C* の関数), 116
 PyDict_Keys (*C* の関数), 116
 PyDict_Merge (*C* の関数), 118
 PyDict_MergeFromSeq2 (*C* の関数), 118
 PyDict_New (*C* の関数), 115
 PyDict_Next (*C* の関数), 117
 PyDict_SetItem (*C* の関数), 116
 PyDict_SetItemString (*C* の関数), 116
 PyDict_Size (*C* の関数), 117
 PyDict_Type (*C* の変数), 115
 PyDict_Update (*C* の関数), 118
 PyDict_Values (*C* の関数), 117
 PyDictObject (*C* のデータ型), 115
 PyDictProxy_New (*C* の関数), 115
 PyErr_BadArgument (*C* の関数), 23
 PyErr_BadInternalCall (*C* の関数), 24
 PyErr_CheckSignals (*C* の関数), 25
 PyErr_Clear (*C* の関数), 22
 PyErr_Clear (), 9, 10
 PyErr_ExceptionMatches (*C* の関数), 22
 PyErr_ExceptionMatches (), 10
 PyErr_Fetch (*C* の関数), 22
 PyErr_Format (*C* の関数), 23
 PyErr_GivenExceptionMatches (*C* の関数), 22
 PyErr_NewException (*C* の関数), 26
 PyErr_NewExceptionWithDoc (*C* の関数), 26
 PyErr_NoMemory (*C* の関数), 23
 PyErr_NormalizeException (*C* の関数), 22
 PyErr_Occurred (*C* の関数), 21
 PyErr_Occurred (), 8
 PyErr_Print (*C* の関数), 21
 PyErr_PrintEx (*C* の関数), 21
 PyErr_Restore (*C* の関数), 22
 PyErr_SetExcFromWindowsErr (*C* の関数), 24
 PyErr_SetExcFromWindowsErrWithFilename (*C* の関数), 24
 PyErr_SetExcFromWindowsErrWithFilenameObject (*C* の関数), 24
 PyErr_SetFromErrno (*C* の関数), 23
 PyErr_SetFromErrnoWithFilename (*C* の関数), 24
 PyErr_SetFromErrnoWithFilenameObject (*C* の関数), 23

PyErr_SetFromWindowsErr (C の関数), 24
 PyErr_SetFromWindowsErrWithFilename (C の関数), 24
 PyErr_SetFromWindowsErrWithFilenameObject (C の関数), 24
 PyErr_SetInterrupt (C の関数), 26
 PyErr_SetNone (C の関数), 23
 PyErr_SetObject (C の関数), 23
 PyErr_SetString (C の関数), 23
 PyErr_SetString(), 9
 PyErr_Warn (C の関数), 25
 PyErr_WarnEx (C の関数), 25
 PyErr_WarnExplicit (C の関数), 25
 PyErr_WarnPy3k (C の関数), 25
 PyErr_WriteUnraisable (C の関数), 26
 PyEval_AcquireLock (C の関数), 152
 PyEval_AcquireLock(), 141
 PyEval_AcquireThread (C の関数), 151
 PyEval_EvalCode (C の関数), 16
 PyEval_EvalCodeEx (C の関数), 16
 PyEval_EvalFrame (C の関数), 16
 PyEval_EvalFrameEx (C の関数), 16
 PyEval_GetBuiltins (C の関数), 49
 PyEval_GetCallStats (C の関数), 156
 PyEval_GetFrame (C の関数), 50
 PyEval_GetFuncDesc (C の関数), 50
 PyEval_GetFuncName (C の関数), 50
 PyEval_GetGlobals (C の関数), 49
 PyEval_GetLocals (C の関数), 49
 PyEval_GetRestricted (C の関数), 50
 PyEval_InitThreads (C の関数), 148
 PyEval_InitThreads(), 141
 PyEval_MergeCompilerFlags (C の関数), 16
 PyEval_ReInitThreads (C の関数), 149
 PyEval_ReleaseLock (C の関数), 152
 PyEval_ReleaseLock(), 141, 148
 PyEval_ReleaseThread (C の関数), 151
 PyEval_ReleaseThread(), 148
 PyEval_RestoreThread (C の関数), 149
 PyEval_RestoreThread(), 146, 148
 PyEval_SaveThread (C の関数), 149
 PyEval_SaveThread(), 146, 148
 PyEval_SetProfile (C の関数), 155
 PyEval_SetTrace (C の関数), 156
 PyEval_ThreadsInitialized (C の関数), 148
 PyExc_ArithmeticError, 28
 PyExc_AssertionError, 28
 PyExc_AttributeError, 28
 PyExc_BaseException, 28
 PyExc_BufferError, 28
 PyExc_BytesWarning, 30
 PyExc_DeprecationWarning, 30
 PyExc_EnvironmentError, 28
 PyExc_EOFError, 28
 PyExc_Exception, 28
 PyExc_FloatingPointError, 28
 PyExc_FutureWarning, 30
 PyExc_GeneratorExit, 28
 PyExc_ImportError, 28
 PyExc_ImportWarning, 30
 PyExc_IndentationError, 28
 PyExc_IndexError, 28
 PyExc_IOError, 28
 PyExc_KeyboardInterrupt, 28
 PyExc_KeyError, 28
 PyExc_LookupError, 28
 PyExc_MemoryError, 28
 PyExc_NameError, 28
 PyExc_NotImplementedError, 28
 PyExc_OSError, 28
 PyExc_OverflowError, 28
 PyExc_PendingDeprecationWarning, 30
 PyExc_ReferenceError, 28
 PyExc_RuntimeError, 28
 PyExc_RuntimeWarning, 30
 PyExc_StandardError, 28
 PyExc_StopIteration, 28
 PyExc_SyntaxError, 28
 PyExc_SyntaxWarning, 30
 PyExc_SystemError, 28
 PyExc_SystemExit, 28
 PyExc_TabError, 28
 PyExc_TypeError, 28
 PyExc_UnboundLocalError, 28
 PyExc_UnicodeDecodeError, 28
 PyExc_UnicodeEncodeError, 28
 PyExc_UnicodeError, 28
 PyExc_UnicodeTranslateError, 28
 PyExc_UnicodeWarning, 30
 PyExc_UserWarning, 30
 PyExc_ValueError, 28
 PyExc_VMSError, 28
 PyExc_Warning, 30
 PyExc_WindowsError, 28
 PyExc_ZeroDivisionError, 28
 PyFile_AsFile (C の関数), 122
 PyFile_Check (C の関数), 121
 PyFile_CheckExact (C の関数), 121
 PyFile_DecUseCount (C の関数), 122
 PyFile_FromFile (C の関数), 122
 PyFile_FromString (C の関数), 122
 PyFile_GetLine (C の関数), 123
 PyFile_IncUseCount (C の関数), 122
 PyFile_Name (C の関数), 123
 PyFile_SetBufSize (C の関数), 123
 PyFile_SetEncoding (C の関数), 123
 PyFile_SetEncodingAndErrors (C の関数), 123
 PyFile_SoftSpace (C の関数), 123
 PyFile_Type (C の変数), 121
 PyFile_WriteObject (C の関数), 123
 PyFile_WriteString (C の関数), 123
 PyFileObject (C のデータ型), 121
 PyFloat_AS_DOUBLE (C の関数), 79
 PyFloat_AsDouble (C の関数), 79
 PyFloat_AsReprString (C の関数), 80
 PyFloat_AsString (C の関数), 80
 PyFloat_Check (C の関数), 79
 PyFloat_CheckExact (C の関数), 79
 PyFloat_ClearFreeList (C の関数), 79
 PyFloat_FromDouble (C の関数), 79
 PyFloat_FromString (C の関数), 79
 PyFloat_GetInfo (C の関数), 79
 PyFloat_GetMax (C の関数), 79
 PyFloat_GetMin (C の関数), 79
 PyFloat_Type (C の変数), 78
 PyFloatObject (C のデータ型), 78
 PyFrame_GetLineNumber (C の関数), 50
 PyFrozenSet_Check (C の関数), 137
 PyFrozenSet_CheckExact (C の関数), 137
 PyFrozenSet_New (C の関数), 138
 PyFrozenSet_Type (C の変数), 137
 PyFunction_Check (C の関数), 119
 PyFunction_GetClosure (C の関数), 120
 PyFunction_GetCode (C の関数), 120
 PyFunction_GetDefaults (C の関数), 120
 PyFunction_GetGlobals (C の関数), 120
 PyFunction_GetModule (C の関数), 120
 PyFunction_New (C の関数), 119
 PyFunction_SetClosure (C の関数), 120
 PyFunction_SetDefaults (C の関数), 120
 PyFunction_Type (C の変数), 119
 PyFunctionObject (C のデータ型), 119
 PyGen_Check (C の関数), 133
 PyGen_CheckExact (C の関数), 133
 PyGen_New (C の関数), 133
 PyGen_Type (C の変数), 133
 PyGenObject (C のデータ型), 133

- PyGetSetDef (C のデータ型), 171
- PyGILState_Ensure (C の関数), 149
- PyGILState_GetThisThreadState (C の関数), 150
- PyGILState_Release (C の関数), 149
- PyImport_AddModule (C の関数), 34
- PyImport_AppendInittab (C の関数), 36
- PyImport_Cleanup (C の関数), 36
- PyImport_ExecCodeModule (C の関数), 35
- PyImport_ExecCodeModuleEx (C の関数), 35
- PyImport_ExtendInittab (C の関数), 37
- PyImport_FrozenModules (C の変数), 36
- PyImport_GetImporter (C の関数), 35
- PyImport_GetMagicNumber (C の関数), 35
- PyImport_GetModuleDict (C の関数), 35
- PyImport_Import (C の関数), 34
- PyImport_ImportFrozenModule (C の関数), 36
- PyImport_ImportModule (C の関数), 33
- PyImport_ImportModuleEx (C の関数), 34
- PyImport_ImportModuleLevel (C の関数), 34
- PyImport_ImportModuleNoBlock (C の関数), 33
- PyImport_ReloadModule (C の関数), 34
- PyIndex_Check (C の関数), 62
- PyInstance_Check (C の関数), 119
- PyInstance_New (C の関数), 119
- PyInstance_NewRaw (C の関数), 119
- PyInstance_Type (C の変数), 119
- PyInt_AS_LONG (C の関数), 74
- PyInt_AsLong (C の関数), 74
- PyInt_AsSsize_t (C の関数), 74
- PyInt_AsUnsignedLongLongMask (C の関数), 74
- PyInt_AsUnsignedLongMask (C の関数), 74
- PyInt_Check (C の関数), 73
- PyInt_CheckExact (C の関数), 73
- PyInt_ClearFreeList (C の関数), 75
- PyInt_FromLong (C の関数), 74
- PyInt_FromSize_t (C の関数), 74
- PyInt_FromSsize_t (C の関数), 74
- PyInt_FromString (C の関数), 73
- PyInt_GetMax (C の関数), 74
- PyInt_Type (C の変数), 73
- PyInterpreterState (C のデータ型), 148
- PyInterpreterState_Clear (C の関数), 150
- PyInterpreterState_Delete (C の関数), 151
- PyInterpreterState_Head (C の関数), 156
- PyInterpreterState_New (C の関数), 150
- PyInterpreterState_Next (C の関数), 156
- PyInterpreterState_ThreadHead (C の関数), 157
- PyIntObject (C のデータ型), 73
- PyIter_Check (C の関数), 67
- PyIter_Next (C の関数), 67
- PyList_Append (C の関数), 114
- PyList_AsTuple (C の関数), 115
- PyList_Check (C の関数), 113
- PyList_CheckExact (C の関数), 113
- PyList_GET_ITEM (C の関数), 114
- PyList_GET_SIZE (C の関数), 113
- PyList_GetItem (C の関数), 113
- PyList_GetItem(), 7
- PyList_GetSlice (C の関数), 114
- PyList_Insert (C の関数), 114
- PyList_New (C の関数), 113
- PyList_Reverse (C の関数), 115
- PyList_SET_ITEM (C の関数), 114
- PyList_SetItem (C の関数), 114
- PyList_SetItem(), 6
- PyList_SetSlice (C の関数), 115
- PyList_Size (C の関数), 113
- PyList_Sort (C の関数), 115
- PyList_Type (C の変数), 113
- PyListObject (C のデータ型), 113
- PyLong_AsDouble (C の関数), 78
- PyLong_AsLong (C の関数), 77
- PyLong_AsLongAndOverflow (C の関数), 77
- PyLong_AsLongLong (C の関数), 77
- PyLong_AsLongLongAndOverflow (C の関数), 77
- PyLong_AsSsize_t (C の関数), 77
- PyLong_AsUnsignedLong (C の関数), 77
- PyLong_AsUnsignedLongLong (C の関数), 78
- PyLong_AsUnsignedLongLongMask (C の関数), 78
- PyLong_AsUnsignedLongMask (C の関数), 78
- PyLong_AsVoidPtr (C の関数), 78
- PyLong_Check (C の関数), 75
- PyLong_CheckExact (C の関数), 76
- PyLong_FromDouble (C の関数), 76
- PyLong_FromLong (C の関数), 76
- PyLong_FromLongLong (C の関数), 76
- PyLong_FromSize_t (C の関数), 76
- PyLong_FromSsize_t (C の関数), 76
- PyLong_FromString (C の関数), 76
- PyLong_FromUnicode (C の関数), 76
- PyLong_FromUnsignedLong (C の関数), 76
- PyLong_FromUnsignedLongLong (C の関数), 76
- PyLong_FromVoidPtr (C の関数), 77
- PyLong_Type (C の変数), 75
- PyLongObject (C のデータ型), 75
- PyMapping_Check (C の関数), 66
- PyMapping_DelItem (C の関数), 66
- PyMapping_DelItemString (C の関数), 66
- PyMapping_GetItemString (C の関数), 67
- PyMapping_HasKey (C の関数), 66
- PyMapping_HasKeyString (C の関数), 66
- PyMapping_Items (C の関数), 66
- PyMapping_Keys (C の関数), 66
- PyMapping_Length (C の関数), 66
- PyMapping_SetItemString (C の関数), 67
- PyMapping_Size (C の関数), 66
- PyMapping_Values (C の関数), 66
- PyMappingMethods (C のデータ型), 193
- PyMappingMethods.mp_ass_subscript (C のメンバ変数), 193
- PyMappingMethods.mp_length (C のメンバ変数), 193
- PyMappingMethods.mp_subscript (C のメンバ変数), 193
- PyMarshal_ReadLastObjectFromFile (C の関数), 38
- PyMarshal_ReadLongFromFile (C の関数), 38
- PyMarshal_ReadObjectFromFile (C の関数), 38
- PyMarshal_ReadObjectFromString (C の関数), 38
- PyMarshal_ReadShortFromFile (C の関数), 38
- PyMarshal_WriteLongToFile (C の関数), 37
- PyMarshal_WriteObjectToFile (C の関数), 37
- PyMarshal_WriteObjectToString (C の関数), 37
- PyMem_Del (C の関数), 161
- PyMem_Free (C の関数), 160
- PyMem_Malloc (C の関数), 160
- PyMem_New (C の関数), 161
- PyMem_Realloc (C の関数), 160
- PyMem_Resize (C の関数), 161
- PyMemberDef (C のデータ型), 170
- PyMemoryView_Check (C の関数), 108
- PyMemoryView_FromBuffer (C の関数), 108
- PyMemoryView_FromObject (C の関数), 108
- PyMemoryView_GET_BUFFER (C の関数), 108
- PyMemoryView_GetContiguous (C の関数), 108
- PyMethod_Check (C の関数), 120
- PyMethod_Class (C の関数), 121
- PyMethod_ClearFreeList (C の関数), 121
- PyMethod_Function (C の関数), 121
- PyMethod_GET_CLASS (C の関数), 121
- PyMethod_GET_FUNCTION (C の関数), 121
- PyMethod_GET_SELF (C の関数), 121
- PyMethod_New (C の関数), 120
- PyMethod_Self (C の関数), 121
- PyMethod_Type (C の変数), 120
- PyMethodDef (C のデータ型), 169
- PyModule_AddIntConstant (C の関数), 124
- PyModule_AddIntMacro (C の関数), 125
- PyModule_AddObject (C の関数), 124

PyModule_AddStringConstant (C の関数), 125
 PyModule_AddStringMacro (C の関数), 125
 PyModule_Check (C の関数), 124
 PyModule_CheckExact (C の関数), 124
 PyModule_GetDict (C の関数), 124
 PyModule_GetFilename (C の関数), 124
 PyModule_GetName (C の関数), 124
 PyModule_New (C の関数), 124
 PyModule_Type (C の変数), 124
 PyNumber_Absolute (C の関数), 59
 PyNumber_Add (C の関数), 58
 PyNumber_And (C の関数), 60
 PyNumber_AsSsize_t (C の関数), 62
 PyNumber_Check (C の関数), 58
 PyNumber_Coerce (C の関数), 62
 PyNumber_CoerceEx (C の関数), 62
 PyNumber_Divide (C の関数), 59
 PyNumber_Divmod (C の関数), 59
 PyNumber_Float (C の関数), 62
 PyNumber_FloorDivide (C の関数), 59
 PyNumber_Index (C の関数), 62
 PyNumber_InPlaceAdd (C の関数), 60
 PyNumber_InPlaceAnd (C の関数), 61
 PyNumber_InPlaceDivide (C の関数), 60
 PyNumber_InPlaceFloorDivide (C の関数), 60
 PyNumber_InPlaceLshift (C の関数), 61
 PyNumber_InPlaceMultiply (C の関数), 60
 PyNumber_InPlaceOr (C の関数), 61
 PyNumber_InPlacePower (C の関数), 61
 PyNumber_InPlaceRemainder (C の関数), 61
 PyNumber_InPlaceRshift (C の関数), 61
 PyNumber_InPlaceSubtract (C の関数), 60
 PyNumber_InPlaceTrueDivide (C の関数), 61
 PyNumber_InPlaceXor (C の関数), 61
 PyNumber_Int (C の関数), 62
 PyNumber_Invert (C の関数), 60
 PyNumber_Long (C の関数), 62
 PyNumber_Lshift (C の関数), 60
 PyNumber_Multiply (C の関数), 59
 PyNumber_Negative (C の関数), 59
 PyNumber_Or (C の関数), 60
 PyNumber_Positive (C の関数), 59
 PyNumber_Power (C の関数), 59
 PyNumber_Remainder (C の関数), 59
 PyNumber_Rshift (C の関数), 60
 PyNumber_Subtract (C の関数), 59
 PyNumber_ToBase (C の関数), 62
 PyNumber_TrueDivide (C の関数), 59
 PyNumber_Xor (C の関数), 60
 PyNumberMethods (C のデータ型), 191
 PyNumberMethods.nb_coerce (C のメンバ変数), 192
 PyObject (C のデータ型), 167
 PyObject.ob_next (C のメンバ変数), 174
 PyObject.ob_prev (C のメンバ変数), 174
 PyObject.ob_refcnt (C のメンバ変数), 174
 PyObject.ob_type (C のメンバ変数), 174
 PyObject_AsCharBuffer (C の関数), 68
 PyObject_AsFileDescriptor (C の関数), 58
 PyObject_AsReadBuffer (C の関数), 68
 PyObject_AsWriteBuffer (C の関数), 68
 PyObject_Bytes (C の関数), 55
 PyObject_Call (C の関数), 56
 PyObject_CallFunction (C の関数), 56
 PyObject_CallFunctionObjArgs (C の関数), 57
 PyObject_CallMethod (C の関数), 56
 PyObject_CallMethodObjArgs (C の関数), 57
 PyObject_CallObject (C の関数), 56
 PyObject_CheckBuffer (C の関数), 106
 PyObject_CheckReadBuffer (C の関数), 68
 PyObject_Cmp (C の関数), 55
 PyObject_Compare (C の関数), 55
 PyObject_Del (C の関数), 166
 PyObject_DelAttr (C の関数), 54
 PyObject_DelAttrString (C の関数), 54
 PyObject_DelItem (C の関数), 58
 PyObject_Dir (C の関数), 58
 PyObject_Free (C の関数), 162
 PyObject_GC_Del (C の関数), 197
 PyObject_GC_New (C の関数), 196
 PyObject_GC_NewVar (C の関数), 196
 PyObject_GC_Resize (C の関数), 196
 PyObject_GC_Track (C の関数), 197
 PyObject_GC_UnTrack (C の関数), 197
 PyObject_GenericGetAttr (C の関数), 53
 PyObject_GenericSetAttr (C の関数), 54
 PyObject_GetAttr (C の関数), 53
 PyObject_GetAttrString (C の関数), 53
 PyObject_GetBuffer (C の関数), 106
 PyObject_GetItem (C の関数), 58
 PyObject_GetIter (C の関数), 58
 PyObject_HasAttr (C の関数), 53
 PyObject_HasAttrString (C の関数), 53
 PyObject_Hash (C の関数), 57
 PyObject_HashNotImplemented (C の関数), 57
 PyObject_HEAD (C のマクロ), 167
 PyObject_HEAD_INIT (C のマクロ), 168
 PyObject_Init (C の関数), 165
 PyObject_InitVar (C の関数), 165
 PyObject_IsInstance (C の関数), 55
 PyObject_IsSubclass (C の関数), 56
 PyObject_IsTrue (C の関数), 57
 PyObject_Length (C の関数), 57
 PyObject_Malloc (C の関数), 161
 PyObject_New (C の関数), 165
 PyObject_NewVar (C の関数), 166
 PyObject_Not (C の関数), 57
 PyObject_Print (C の関数), 53
 PyObject_Realloc (C の関数), 161
 PyObject_Repr (C の関数), 55
 PyObject_RichCompare (C の関数), 54
 PyObject_RichCompareBool (C の関数), 54
 PyObject_SetAttr (C の関数), 54
 PyObject_SetAttrString (C の関数), 54
 PyObject_SetItem (C の関数), 58
 PyObject_Size (C の関数), 57
 PyObject_Str (C の関数), 55
 PyObject_Type (C の関数), 57
 PyObject_TypeCheck (C の関数), 57
 PyObject_Unicode (C の関数), 55
 PyObject_VAR_HEAD (C のマクロ), 167
 PyOS_AfterFork (C の関数), 31
 PyOS_ascii_atof (C の関数), 49
 PyOS_ascii_formatd (C の関数), 48
 PyOS_ascii_strtod (C の関数), 48
 PyOS_CheckStack (C の関数), 31
 PyOS_double_to_string (C の関数), 48
 PyOS_getsig (C の関数), 31
 PyOS_setsig (C の関数), 31
 PyOS_snprintf (C の関数), 47
 PyOS_strerror (C の関数), 49
 PyOS_string_to_double (C の関数), 48
 PyOS_strncmp (C の関数), 49
 PyOS_vsnprintf (C の関数), 47
 PyParser_SimpleParseFile (C の関数), 15
 PyParser_SimpleParseFileFlags (C の関数), 15
 PyParser_SimpleParseString (C の関数), 15
 PyParser_SimpleParseStringFlags (C の関数), 15
 PyParser_SimpleParseStringFlagsFilename (C の関数), 15
 PyProperty_Type (C の変数), 126
 PyRun_AnyFile (C の関数), 13
 PyRun_AnyFileEx (C の関数), 13
 PyRun_AnyFileExFlags (C の関数), 13
 PyRun_AnyFileFlags (C の関数), 13
 PyRun_File (C の関数), 15
 PyRun_FileEx (C の関数), 15

PyRun_FileExFlags (C の関数), 16
 PyRun_FileFlags (C の関数), 15
 PyRun_InteractiveLoop (C の関数), 14
 PyRun_InteractiveLoopFlags (C の関数), 14
 PyRun_InteractiveOne (C の関数), 14
 PyRun_InteractiveOneFlags (C の関数), 14
 PyRun_SimpleFile (C の関数), 14
 PyRun_SimpleFileEx (C の関数), 14
 PyRun_SimpleFileExFlags (C の関数), 14
 PyRun_SimpleFileFlags (C の関数), 14
 PyRun_SimpleString (C の関数), 14
 PyRun_SimpleStringFlags (C の関数), 14
 PyRun_String (C の関数), 15
 PyRun_StringFlags (C の関数), 15
 PySeqIter_Check (C の関数), 125
 PySeqIter_New (C の関数), 125
 PySeqIter_Type (C の変数), 125
 PySequence_Check (C の関数), 63
 PySequence_Concat (C の関数), 63
 PySequence_Contains (C の関数), 64
 PySequence_Count (C の関数), 64
 PySequence_DelItem (C の関数), 64
 PySequence_DelSlice (C の関数), 64
 PySequence_Fast (C の関数), 65
 PySequence_Fast_GET_ITEM (C の関数), 65
 PySequence_Fast_GET_SIZE (C の関数), 66
 PySequence_Fast_ITEMS (C の関数), 65
 PySequence_GetItem (C の関数), 63
 PySequence_GetItem (), 7
 PySequence_GetSlice (C の関数), 64
 PySequence_Index (C の関数), 65
 PySequence_InPlaceConcat (C の関数), 63
 PySequence_InPlaceRepeat (C の関数), 63
 PySequence_ITEM (C の関数), 65
 PySequence_Length (C の関数), 63
 PySequence_List (C の関数), 65
 PySequence_Repeat (C の関数), 63
 PySequence_SetItem (C の関数), 64
 PySequence_SetSlice (C の関数), 64
 PySequence_Size (C の関数), 63
 PySequence_Tuple (C の関数), 65
 PySequenceMethods (C のデータ型), 193
 PySequenceMethods.sq_ass_item (C のメンバ変数), 194
 PySequenceMethods.sq_concat (C のメンバ変数), 193
 PySequenceMethods.sq_contains (C のメンバ変数), 194
 PySequenceMethods.sq_inplace_concat (C のメンバ変数), 194
 PySequenceMethods.sq_inplace_repeat (C のメンバ変数), 194
 PySequenceMethods.sq_item (C のメンバ変数), 194
 PySequenceMethods.sq_length (C のメンバ変数), 193
 PySequenceMethods.sq_repeat (C のメンバ変数), 194
 PySet_Add (C の関数), 138
 PySet_Check (C の関数), 137
 PySet_Clear (C の関数), 139
 PySet_Contains (C の関数), 138
 PySet_Discard (C の関数), 139
 PySet_GET_SIZE (C の関数), 138
 PySet_New (C の関数), 138
 PySet_Pop (C の関数), 139
 PySet_Size (C の関数), 138
 PySet_Type (C の変数), 137
 PySetObject (C のデータ型), 137
 PySignal_SetWakeupFd (C の関数), 26
 PySlice_Check (C の関数), 127
 PySlice_GetIndices (C の関数), 127
 PySlice_GetIndicesEx (C の関数), 127
 PySlice_New (C の関数), 127
 PySlice_Type (C の変数), 127
 PyString_AS_STRING (C の関数), 85
 PyString_AsDecodedObject (C の関数), 87
 PyString_AsEncodedObject (C の関数), 87
 PyString_AsString (C の関数), 85

PyString_AsStringAndSize (C の関数), 85
 PyString_Check (C の関数), 83
 PyString_CheckExact (C の関数), 83
 PyString_Concat (C の関数), 85
 PyString_ConcatAndDel (C の関数), 85
 PyString_Decode (C の関数), 86
 PyString_Encode (C の関数), 87
 PyString_Format (C の関数), 86
 PyString_FromFormat (C の関数), 83
 PyString_FromFormatV (C の関数), 84
 PyString_FromString (C の関数), 83
 PyString_FromString (), 116
 PyString_FromStringAndSize (C の関数), 83
 PyString_GET_SIZE (C の関数), 85
 PyString_InternFromString (C の関数), 86
 PyString_InternInPlace (C の関数), 86
 PyString_Size (C の関数), 84
 PyString_Type (C の変数), 83
 PyStringObject (C のデータ型), 83
 PySys_AddWarnOption (C の関数), 32
 PySys_GetFile (C の関数), 32
 PySys_GetObject (C の関数), 32
 PySys_ResetWarnOptions (C の関数), 32
 PySys_SetArgv (C の関数), 145
 PySys_SetArgv (), 141
 PySys_SetArgvEx (C の関数), 144
 PySys_SetArgvEx (), 11, 141
 PySys_SetObject (C の関数), 32
 PySys_SetPath (C の関数), 32
 PySys_WriteStderr (C の関数), 32
 PySys_WriteStdout (C の関数), 32
 Python 3000, 208
 Python Enhancement Proposals
 PEP 1, 208
 PEP 238, 17, 202
 PEP 278, 209
 PEP 302, 202, 206
 PEP 3116, 209
 PEP 343, 201
 PYTHONDUMPREFS, 174
 PYTHONHOME, 11, 145
 Pythonic, 208
 PYTHONPATH, 11
 PYTHONSHAWALLOCCOUNT, 191
 PyThreadState, 146
 PyThreadState (C のデータ型), 148
 PyThreadState_Clear (C の関数), 151
 PyThreadState_Delete (C の関数), 151
 PyThreadState_Get (C の関数), 149
 PyThreadState_GetDict (C の関数), 151
 PyThreadState_New (C の関数), 151
 PyThreadState_Next (C の関数), 157
 PyThreadState_SetAsyncExc (C の関数), 151
 PyThreadState_Swap (C の関数), 149
 PyTime_Check (C の関数), 134
 PyTime_CheckExact (C の関数), 134
 PyTime_FromTime (C の関数), 135
 PyTrace_C_CALL (C の変数), 155
 PyTrace_C_EXCEPTION (C の変数), 155
 PyTrace_C_RETURN (C の変数), 155
 PyTrace_CALL (C の変数), 155
 PyTrace_EXCEPTION (C の変数), 155
 PyTrace_LINE (C の変数), 155
 PyTrace_RETURN (C の変数), 155
 PyTuple_Check (C の関数), 110
 PyTuple_CheckExact (C の関数), 111
 PyTuple_ClearFreeList (C の関数), 112
 PyTuple_GET_ITEM (C の関数), 111
 PyTuple_GET_SIZE (C の関数), 111
 PyTuple_GetItem (C の関数), 111
 PyTuple_GetSlice (C の関数), 111
 PyTuple_New (C の関数), 111
 PyTuple_Pack (C の関数), 111

PyTuple_SET_ITEM (C の関数), 112
 PyTuple_SetItem (C の関数), 112
 PyTuple_SetItem(), 6
 PyTuple_Size (C の関数), 111
 PyTuple_Type (C の変数), 110
 PyTupleObject (C のデータ型), 110
 PyType_Check (C の関数), 71
 PyType_CheckExact (C の関数), 71
 PyType_ClearCache (C の関数), 72
 PyType_GenericAlloc (C の関数), 72
 PyType_GenericNew (C の関数), 72
 PyType_HasFeature (C の関数), 72
 PyType_HasFeature(), 195
 PyType_IS_GC (C の関数), 72
 PyType_IsSubtype (C の関数), 72
 PyType_Modified (C の関数), 72
 PyType_Ready (C の関数), 72
 PyType_Type (C の変数), 71
 PyTypeObject (C のデータ型), 71
 PyTypeObject.tp_alloc (C のメンバ変数), 188
 PyTypeObject.tp_allocs (C のメンバ変数), 191
 PyTypeObject.tp_as_buffer (C のメンバ変数), 180
 PyTypeObject.tp_base (C のメンバ変数), 186
 PyTypeObject.tp_bases (C のメンバ変数), 190
 PyTypeObject.tp_basicsize (C のメンバ変数), 175
 PyTypeObject.tp_cache (C のメンバ変数), 190
 PyTypeObject.tp_call (C のメンバ変数), 179
 PyTypeObject.tp_clear (C のメンバ変数), 183
 PyTypeObject.tp_compare (C のメンバ変数), 177
 PyTypeObject.tp_dealloc (C のメンバ変数), 176
 PyTypeObject.tp_descr_get (C のメンバ変数), 187
 PyTypeObject.tp_descr_set (C のメンバ変数), 187
 PyTypeObject.tp_dict (C のメンバ変数), 187
 PyTypeObject.tp_dictoffset (C のメンバ変数), 187
 PyTypeObject.tp_doc (C のメンバ変数), 182
 PyTypeObject.tp_flags (C のメンバ変数), 180
 PyTypeObject.tp_free (C のメンバ変数), 189
 PyTypeObject.tp_frees (C のメンバ変数), 191
 PyTypeObject.tp_getattr (C のメンバ変数), 177
 PyTypeObject.tp_getattro (C のメンバ変数), 179
 PyTypeObject.tp_getset (C のメンバ変数), 186
 PyTypeObject.tp_hash (C のメンバ変数), 178
 PyTypeObject.tp_init (C のメンバ変数), 188
 PyTypeObject.tp_is_gc (C のメンバ変数), 190
 PyTypeObject.tp_itemsize (C のメンバ変数), 175
 PyTypeObject.tp_iter (C のメンバ変数), 185
 PyTypeObject.tp_iternext (C のメンバ変数), 186
 PyTypeObject.tp_maxalloc (C のメンバ変数), 191
 PyTypeObject.tp_members (C のメンバ変数), 186
 PyTypeObject.tp_methods (C のメンバ変数), 186
 PyTypeObject.tp_mro (C のメンバ変数), 190
 PyTypeObject.tp_name (C のメンバ変数), 175
 PyTypeObject.tp_new (C のメンバ変数), 189
 PyTypeObject.tp_next (C のメンバ変数), 191
 PyTypeObject.tp_print (C のメンバ変数), 177
 PyTypeObject.tp_repr (C のメンバ変数), 178
 PyTypeObject.tp_richcompare (C のメンバ変数), 184
 PyTypeObject.tp_setattr (C のメンバ変数), 177
 PyTypeObject.tp_setattro (C のメンバ変数), 179
 PyTypeObject.tp_str (C のメンバ変数), 179
 PyTypeObject.tp_subclasses (C のメンバ変数), 191
 PyTypeObject.tp_traverse (C のメンバ変数), 182
 PyTypeObject.tp_weaklist (C のメンバ変数), 191
 PyTypeObject.tp_weaklistoffset (C のメンバ変数), 185
 PyTZInfo_Check (C の関数), 134
 PyTZInfo_CheckExact (C の関数), 134
 PyUnicode_AS_DATA (C の関数), 89
 PyUnicode_AS_UNICODE (C の関数), 89
 PyUnicode_AsASCIIString (C の関数), 99
 PyUnicode_AsCharmapString (C の関数), 100
 PyUnicode_AsEncodedString (C の関数), 94
 PyUnicode_AsLatin1String (C の関数), 99
 PyUnicode_AsMBCSString (C の関数), 101
 PyUnicode_AsRawUnicodeEscapeString (C の関数), 98
 PyUnicode_AsUnicode (C の関数), 91
 PyUnicode_AsUnicodeEscapeString (C の関数), 98
 PyUnicode_AsUTF16String (C の関数), 97
 PyUnicode_AsUTF32String (C の関数), 95
 PyUnicode_AsUTF8String (C の関数), 94
 PyUnicode_AsWideChar (C の関数), 92
 PyUnicode_Check (C の関数), 88
 PyUnicode_CheckExact (C の関数), 88
 PyUnicode_ClearFreeList (C の関数), 89
 PyUnicode_Compare (C の関数), 103
 PyUnicode_Concat (C の関数), 102
 PyUnicode_Contains (C の関数), 104
 PyUnicode_Count (C の関数), 103
 PyUnicode_Decode (C の関数), 93
 PyUnicode_DecodeASCII (C の関数), 99
 PyUnicode_DecodeCharmap (C の関数), 100
 PyUnicode_DecodeLatin1 (C の関数), 99
 PyUnicode_DecodeMBCS (C の関数), 101
 PyUnicode_DecodeMBCSStateful (C の関数), 101
 PyUnicode_DecodeRawUnicodeEscape (C の関数), 98
 PyUnicode_DecodeUnicodeEscape (C の関数), 98
 PyUnicode_DecodeUTF16 (C の関数), 96
 PyUnicode_DecodeUTF16Stateful (C の関数), 96
 PyUnicode_DecodeUTF32 (C の関数), 94
 PyUnicode_DecodeUTF32Stateful (C の関数), 95
 PyUnicode_DecodeUTF7 (C の関数), 97
 PyUnicode_DecodeUTF7Stateful (C の関数), 97
 PyUnicode_DecodeUTF8 (C の関数), 94
 PyUnicode_DecodeUTF8Stateful (C の関数), 94
 PyUnicode_Encode (C の関数), 93
 PyUnicode_EncodeASCII (C の関数), 99
 PyUnicode_EncodeCharmap (C の関数), 100
 PyUnicode_EncodeLatin1 (C の関数), 99
 PyUnicode_EncodeMBCS (C の関数), 101
 PyUnicode_EncodeRawUnicodeEscape (C の関数), 98
 PyUnicode_EncodeUnicodeEscape (C の関数), 98
 PyUnicode_EncodeUTF16 (C の関数), 96
 PyUnicode_EncodeUTF32 (C の関数), 95
 PyUnicode_EncodeUTF7 (C の関数), 97
 PyUnicode_EncodeUTF8 (C の関数), 94
 PyUnicode_Find (C の関数), 103
 PyUnicode_Format (C の関数), 104
 PyUnicode_FromEncodedObject (C の関数), 92
 PyUnicode_FromFormat (C の関数), 91
 PyUnicode_FromFormatV (C の関数), 91
 PyUnicode_FromObject (C の関数), 92
 PyUnicode_FromString (C の関数), 91
 PyUnicode_FromStringAndSize (C の関数), 90
 PyUnicode_FromUnicode (C の関数), 90
 PyUnicode_FromWideChar (C の関数), 92
 PyUnicode_GET_DATA_SIZE (C の関数), 88
 PyUnicode_GET_SIZE (C の関数), 88
 PyUnicode_GetSize (C の関数), 92
 PyUnicode_Join (C の関数), 102
 PyUnicode_Replace (C の関数), 103
 PyUnicode_RichCompare (C の関数), 103
 PyUnicode_Split (C の関数), 102
 PyUnicode_Splitlines (C の関数), 102
 PyUnicode_Tailmatch (C の関数), 102
 PyUnicode_Translate (C の関数), 102
 PyUnicode_TranslateCharmap (C の関数), 101
 PyUnicode_Type (C の変数), 88
 PyUnicodeDecodeError.Create (C の関数), 27
 PyUnicodeDecodeError.GetEncoding (C の関数), 27
 PyUnicodeDecodeError.GetEnd (C の関数), 27
 PyUnicodeDecodeError.GetObject (C の関数), 27
 PyUnicodeDecodeError.GetReason (C の関数), 28
 PyUnicodeDecodeError.GetStart (C の関数), 27
 PyUnicodeDecodeError.SetEnd (C の関数), 27
 PyUnicodeDecodeError.SetReason (C の関数), 28
 PyUnicodeDecodeError.SetStart (C の関数), 27
 PyUnicodeEncodeError.Create (C の関数), 27

PyUnicodeEncodeError.GetEncoding (C の関数), 27
 PyUnicodeEncodeError.GetEnd (C の関数), 27
 PyUnicodeEncodeError.GetObject (C の関数), 27
 PyUnicodeEncodeError.GetReason (C の関数), 28
 PyUnicodeEncodeError.GetStart (C の関数), 27
 PyUnicodeEncodeError.SetEnd (C の関数), 27
 PyUnicodeEncodeError.SetReason (C の関数), 28
 PyUnicodeEncodeError.SetStart (C の関数), 27
 PyUnicodeObject (C のデータ型), 88
 PyUnicodeTranslateError.Create (C の関数), 27
 PyUnicodeTranslateError.GetEnd (C の関数), 27
 PyUnicodeTranslateError.GetObject (C の関数), 27
 PyUnicodeTranslateError.GetReason (C の関数), 28
 PyUnicodeTranslateError.GetStart (C の関数), 27
 PyUnicodeTranslateError.SetEnd (C の関数), 27
 PyUnicodeTranslateError.SetReason (C の関数), 28
 PyUnicodeTranslateError.SetStart (C の関数), 27
 PyVarObject (C のデータ型), 167
 PyVarObject.ob_size (C のメンバ変数), 175
 PyVarObject.HEAD_INIT (C のマクロ), 168
 PyWeakref.Check (C の関数), 128
 PyWeakref.CheckProxy (C の関数), 128
 PyWeakref.CheckRef (C の関数), 128
 PyWeakref.GET_OBJECT (C の関数), 129
 PyWeakref.GetObject (C の関数), 128
 PyWeakref.NewProxy (C の関数), 128
 PyWeakref.NewRef (C の関数), 128
 PyWrapper.New (C の関数), 126

readbufferproc (C のデータ型), 195
 realloc(), 159
 reference count, 208
 reload
 組み込み関数, 34
 repr
 組み込み関数, 55, 178
 rexec
 モジュール, 34

search
 path, module, 11, 141, 143
 segcountproc (C のデータ型), 196
 sequence, 208
 sequence
 オブジェクト, 82
 set
 オブジェクト, 137
 set_all(), 7
 setcheckinterval() (in module sys), 145
 setvbuf(), 123
 SIGINT, 26
 signal
 モジュール, 26
 slice, 208
 SliceType (in module types), 127
 softspace (file attribute), 123
 special
 method, 209
 special method, 209
 statement, 209
 staticmethod
 組み込み関数, 170
 stderr (in module sys), 152
 stdin (in module sys), 152
 stdout (in module sys), 152
 str
 組み込み関数, 55
 strerror(), 23
 string
 オブジェクト, 83
 StringType (in module types), 83
 struct sequence, 209
 sum_list(), 8

sum_sequence(), 8, 9
 sys
 モジュール, 11, 141, 152
 SystemError (built-in exception), 124

thread
 モジュール, 148
 tp.as_mapping (C のメンバ変数), 178
 tp.as_number (C のメンバ変数), 178
 tp.as_sequence (C のメンバ変数), 178
 traverseproc (C のデータ型), 197
 triple-quoted string, 209
 tuple
 オブジェクト, 110
 組み込み関数, 65, 115
 TupleType (in module types), 110
 type
 オブジェクト, 4, 71
 組み込み関数, 57
 TypeType (in module types), 71

ULONG_MAX, 77
 unicode
 組み込み関数, 55
 universal newlines, 209

version (in module sys), 144
 virtual environment, 209
 virtual machine, 209
 visitproc (C のデータ型), 197

writebufferproc (C のデータ型), 195

Zen of Python, 209

オブジェクト
 buffer, 104
 bytearray, 82
 Capsule, 129
 class, 118
 CObject, 131
 complex number, 80
 dictionary, 115
 file, 121
 floating point, 78
 frozenset, 137
 function, 119
 instance, 119
 integer, 73
 list, 113
 long integer, 75
 mapping, 115
 method, 120
 module, 124
 None, 73
 numeric, 73
 sequence, 82
 set, 137
 string, 83
 tuple, 110
 type, 4, 71

モジュール, 206
 モジュール
 __builtin__, 11, 141
 __main__, 11, 141, 152
 builtins, 152
 exceptions, 11
 ihooks, 34
 rexec, 34
 signal, 26
 sys, 11, 141, 152
 thread, 148

位置引数, 208

環境変数

 exec.prefix, 4

 PATH, 11

 prefix, 4

 PYTHONDUMPREFS, 174

 PYTHONHOME, 11, 145

 PYTHONPATH, 11

 PYTHONSHOWALLOCCOUNT, 191

関数, 202

型, 209

組み込み関数

 __import__, 34

 abs, 60

 apply, 56

 bytes, 55

 classmethod, 170

 cmp, 55

 coerce, 62

 compile, 35

 divmod, 59

 float, 62

 hash, 57, 178

 int, 62

 len, 58, 63, 66, 113, 117, 138

 long, 62

 pow, 59, 61

 reload, 34

 repr, 55, 178

 staticmethod, 170

 str, 55

 tuple, 65, 115

 type, 57

 unicode, 55