
Python/C API リファレンスマニュアル

リリース 2.3.4

Guido van Rossum

Fred L. Drake, Jr., editor

日本語訳: Python ドキュメント翻訳プロジェクト

平成 16 年 9 月 20 日

PythonLabs

Email: docs@python.org

Copyright © 2001, 2002, 2003 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.

Translation Copyright © 2003 Python Document Japanese Translation Project. All rights reserved.

ライセンスおよび許諾に関する完全な情報は、このドキュメントの末尾を参照してください。

概要

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 および C++ プログラマが利用できる API について述べています。 *Extending and Embedding the Python Interpreter* 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。

警告: このドキュメントの現在のバージョンはまだ不完全です。とはいえ、役に立つだろうと思います。引き続きドキュメントの整備を継続して、Python ソースコードのリリースとは別に、その時々で新たなバージョンをリリースするつもりです。

目次

第 1 章	はじめに	1
1.1	インクルードファイル	1
1.2	オブジェクト、型および参照カウント	2
1.3	例外	6
1.4	Python の埋め込み	8
第 2 章	超高レベルレイヤ	11
第 3 章	参照カウント	13
第 4 章	例外処理	15
4.1	標準例外	19
4.2	文字列例外の廃止	20
第 5 章	ユーティリティ関数	21
5.1	オペレーティングシステム関連のユーティリティ	21
5.2	プロセス制御	22
5.3	モジュールの import	22
5.4	データ整形化 (data marshalling) のサポート	25
5.5	引数の解釈と値の構築	25
第 6 章	抽象オブジェクトレイヤ (abstract objects layer)	33
6.1	オブジェクトプロトコル (object protocol)	33
6.2	数値型プロトコル (number protocol)	37
6.3	配列プロトコル (sequence protocol)	41
6.4	マップ型プロトコル (mapping protocol)	43
6.5	イテレータプロトコル (iterator protocol)	43
6.6	バッファプロトコル (buffer protocol)	44
第 7 章	具象オブジェクト (concrete object) レイヤ	47
7.1	基本オブジェクト (fundamental object)	47
7.2	数値型オブジェクト (numeric object)	48
7.3	配列オブジェクト (sequence object)	53
7.4	マップ型オブジェクト (mapping object)	68
7.5	その他のオブジェクト	71
第 8 章	初期化 (initialization)、終了処理 (finalization)、スレッド	79

8.1	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)	83
8.2	プロファイルとトレース (profiling and tracing)	88
8.3	高度なデバッガサポート (advanced debugger support)	89
第 9 章	メモリ管理	91
9.1	概要	91
9.2	メモリインタフェース	92
9.3	例	93
第 10 章	オブジェクト実装サポート (object implementation support)	95
10.1	オブジェクトをヒープ上にメモリ確保する	95
10.2	共通のオブジェクト構造体 (common object structure)	97
10.3	型オブジェクト	99
10.4	マップ型オブジェクト構造体 (mapping object structure)	114
10.5	数値オブジェクト構造体 (number object structure)	114
10.6	配列オブジェクト構造体 (sequence object structure)	114
10.7	バッファオブジェクト構造体 (buffer object structure)	114
10.8	イテレータプロトコルをサポートする	116
10.9	循環参照ガベージコレクションをサポートする	116
付録 A	バグ報告	119
付録 B	歴史とライセンス	121
B.1	History of the software	121
B.2	Terms and conditions for accessing or otherwise using Python	122
付録 C	日本語訳について	125
C.1	このドキュメントについて	125
C.2	翻訳者一覧 (敬称略)	125
	索引	127

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し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>, and <stdlib.h> を (あれば) インクルードします。

警告: 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ため、'Python.h' は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です、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は書いてはなりません。後からコードを読む人を混乱させたり、将来の Python のバージョンで同じ名前が定義されて、ユーザの書いたコードの可搬性を危うくする可能性があります。

ヘッダファイル群は通常 Python と共にインストールされます。UNIX では 'prefix/include/pythonversion/' および 'exec_prefix/include/pythonversion/' に置かれます。prefix と exec_prefix は Python をビルドする際の

`configure` スクリプトに与えたパラメタに対応し、`version` は `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 <python2.3/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、Python リファレンスマニュアル 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 リスト型の時 (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、`incr` 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 (deallocater) 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ーフローすることはありません; というのも、仮想メモリ空間には、(sizeof(long) >= sizeof(char*)) 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見える操作が、上記の動作を引き起こす何らかの Python コードを呼び出しかねないということです; `Py_DECREF()` からユーザへ制御を戻せるようなコードパスが存在するため、ほとんど全ての操作が潜在的に危険をはらむことになります。

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

参照カウントの詳細

Python/C API の各関数における参照カウントの振る舞いは、説明するには、参照の所有権 (*ownership of references*) という言葉でうまく説明できます。所有権は参照に対するもので、オブジェクトに対するものではありません (オブジェクトは誰にも所有されず、常に共有されています)。ある参照の "所有" は、その参照が必要なくなった時点で `Py_DECREF()` を呼び出す役割を担うことを意味します。所有権は委譲でき、あるコードが委譲によって所有権を得ると、今度はそのコードが参照が必要なくなった際に最終的に `Py_DECREF()` や `Py_XDECREF()` を呼び出して `decref` 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、これらは配列に入れる要素に対する参照を盗み取ります (しかし、要素の入る先のタプルやリストの参照は盗み取りません!)。これらの関数は、リストやタプルの中に新たに作成されたオブジェクトを入れていく際の常套的な書き方をしやすくするために、参照を盗み取るように設計されています; 例えば、`(1, 2, "three")` というタプルを生成するコードは以下ようになります (とりあえず例外処理のことは忘れておきます; もっとよい書き方を後で示します):

```

PyObject *t;

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

```

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。このコードは以下のように `PySequence_SetItem()` を使っても書けます; これを見れば、二つの違い (`Py_DECREF()` を別に呼び出している) が分かります:

```

PyObject *l, *x;

l = PyList_New(3);
x = PyInt_FromLong(1L);
PySequence_SetItem(l, 0, x); Py_DECREF(x);
x = PyInt_FromLong(2L);
PySequence_SetItem(l, 1, x); Py_DECREF(x);
x = PyString_FromString("three");
PySequence_SetItem(l, 2, x); Py_DECREF(x);

```

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

```

PyObject *t, *l;

t = Py_BuildValue("(iis)", 1, 2, "three");
l = 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++) {
        if (PyObject_SetItem(target, i, item) < 0)
            return -1;
    }
    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; /* リストではない */
    for (i = 0; i < n; i++) {
        item = PyList_GetItem(list, i); /* 失敗しないはず */
        if (!PyInt_Check(item)) continue; /* 整数でなければ読み飛ばす */
        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; /* 長さの概念がない */
    for (i = 0; i < n; i++) {
        item = PySequence_GetItem(sequence, i);
        if (item == NULL)
            return -1; /* 配列でないか、その他の失敗 */
        if (PyInt_Check(item))
            total += PyInt_AsLong(item);
        Py_DECREF(item); /* GetItem で得た所有権を放棄する */
    }
    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 プログラムは、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 プログラムの場合、エラーチェックは常に明示的に行わねばなりません。Python/C API の全ての関数は、関数のドキュメントで明確に説明がない限り例外を発行する可能性があります。一般的な話として、ある関数が何らかのエラーに遭遇すると、関数は例外を送出して、関数内における参照の所有権を全て放棄し、エラー指標 (error indicator) — 通常は NULL または -1 を返します。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。

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。余計に例外を設定する必要はありません — そんなことをすれば、たった今送されたばかりの例外を上書きしてしまい、エラーの原因そのものに関する重要な情報を失うこととなります。

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)
{
    /* Py_XDECREF 用に全てのオブジェクトを NULL で初期化 */
    PyObject *item = NULL, *const_one = NULL, *incremented_item = NULL;
    int rv = -1; /* 戻り値の初期値を -1 (失敗) に設定しておく */

    item = PyObject_GetItem(dict, key);
    if (item == NULL) {
        /* KeyError だけを処理: */
        if (!PyErr_ExceptionMatches(PyExc_KeyError))
            goto error;

        /* エラーを無かったことに (clear) してゼロを使う: */
        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; /* うまくいった場合 */
    /* 後始末コードに続く */

error:
    /* 成功しても失敗しても使われる後始末コード */

    /* NULL を参照している場合は無視するために Py_XDECREF() を使う */
    Py_XDECREF(item);
    Py_XDECREF(const_one);
    Py_XDECREF(incremented_item);

    return rv; /* エラーなら -1 、 成功なら 0 */
}

```

なんとこの例はCでgoto文を使うお勤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`, および `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()` に続いて `PySys_SetArgv(argc, argv)` 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/python2.3’ 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/python2.3’ 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 が確保した全ての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。

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。開始期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には、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が関係しています。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して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 `sys.exit()` 関数へ渡される整数でしょう。例外が原因でインタプリタが終了した場合は 1、あるいは、引数リストが有効な Python コマンドラインになっていない場合は 2 です。

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

`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(char *command)`

`__main__` モジュールの中で `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。

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

`PyRun_SimpleString()` 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。

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

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実行します。`filename` が `NULL` ならば、"???" が代わりに使われます。`sys.ps1` と `sys.ps2` を使って、ユーザにプロンプトを提示します。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 を返すか、もしくは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, char *filename)

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。*filename* が NULL ならば、"???" が代わりに使われます。sys.ps1 と sys.ps2 を使って、ユーザにプロンプトを提示します。EOF に達すると 0 を返します。

struct _node* **PyParser_SimpleParseString**(char *str, int start)

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, char *filename, int start)

`PyParser_SimpleParseString()` 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**(char *str, int start, PyObject *globals, PyObject *locals)

戻り値: *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, char *filename, int start, PyObject *globals, PyObject *locals)

戻り値: *New reference*.

`PyRun_String()` 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。

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

戻り値: *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。

int **Py_eval_input**

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

int **Py_file_input**

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

int **Py_single_input**

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

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` であってもよいが、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。

以下の関数やマクロ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: `_Py_Dealloc()`, `_Py_ForgetReference()`, `_Py_NewReference()`。また、グローバル変数 `_Py_RefTotal` も同様です。

例外処理

この章で説明する関数を使うと、Python の例外の処理や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っくらか理解することが大切です。例外は UNIX `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関数がこれをクリアしませんが、失敗したときにはエラーの原因を示すために設定します。ほとんどの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) それを持つどんなり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_Print()
    sys.stderr へ標準トレースバックをプリントし、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!)
```

```
PyObject* PyErr_Occurred()
    戻り値: 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。注意: 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、これを呼び出だすべきです。例外が発生していないならば、メモリアクセス違反が起きるでしょう。
```

```
int PyErr_GivenExceptionMatches(PyObject *given, PyObject *exc)
    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。*given* が NULL ならば、メモリアクセス違反が起きるでしょう。

```
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。注意: 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。) 注意: 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, 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, ...)
```

戻り値: *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 (インスタンスではなくクラス) であるべきです。*format* は文字列であるべきであり、`printf()` に似た書式化コードを含んでいます。書式化コードの前の幅・精度 (*width.precision*) は解析されますが、幅の部分は無視されます。

文字	意味
'c'	文字、int 引数として
'd'	10 進数、int 引数として
'x'	16 進数、int 引数として
's'	文字列、char *引数として
'p'	16 進法のポインタ、void *引数として

認識できない書式化文字があると書式化文字列の残りすべてがそのまま結果の文字列へコピーされ、余分の引数はどれも捨てられます。

```
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()
```

戻り値: *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)
```

戻り値: *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_SetFromErrnoWithFilename(PyObject *type, char *filename)
```

戻り値: *Always NULL.*

PyErr_SetFromErrno() に似ていて、filename が NULL でない場合に、それが type 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。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_SetFromWindowsErr(int ierr)
```

戻り値: *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 を返しません。利用可能範囲: Windows。

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

PyErr_SetFromWindowsErr() に似ていて、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。 2.3 で追加された仕様です。

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

戻り値: *Always NULL.*

`PyErr_SetFromWindowsErr()` に似ていて、`filename` が `NULL` でない場合には `WindowsError` のコンストラクタへ第三引数として渡されるという振る舞いが追加されています。利用可能範囲: Windows。

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

`PyErr_SetFromWindowsErrWithFilename()` に似ていて、発生させ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。 2.3 で追加された仕様です。

```
void PyErr_BadInternalCall()
```

`'PyErr_SetString(PyExc_TypeError, 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_Warn(PyObject *category, char *message)
```

警告メッセージを出します。`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。

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 `Warning` のサブクラスでなければならない。デフォルト警告カテゴリは `RuntimeWarning` です。標準 Python 警告カテゴリは `'PyExc_'` に Python 例外名が続く名前のグローバル変数を用いて変更できます。これらは型 `PyObject*` を持ち、すべてクラスオブジェクトです。それらの名前は `PyExc_Warning`, `PyExc_UserWarning`, `PyExc_DeprecationWarning`, `PyExc_SyntaxWarning`, `PyExc_RuntimeWarning` および `PyExc_FutureWarning` です。`PyExc_Warning` は `PyExc_Exception` のサブクラスです。その他の警告カテゴリは `PyExc_Warning` のサブクラスです。

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_WarnExplicit(PyObject *category, char *message, char *filename, int lineno, 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` に設定できます。

```
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わかりません。

`void PyErr_SetInterrupt()`

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せます。

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

戻り値: *New reference.*

このユーティリティ関数は新しい例外オブジェクトを作成して返します。 *name* 引数は新しい例外の名前、`module.class` 形式の C 文字列でなければなりません。 *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) から導出され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* 引数は代替りのベースクラスを指定するために使えます。 *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。

`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生じたコンテキストを特定するための一つの引数 *obj* とともに関数が呼び出されます。 *obj* の `repr` が警告メッセージにプリントされるでしょう。

4.1 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	注記
PyExc_Exception	Exception	(1)
PyExc_StandardError	StandardError	(1)
PyExc_ArithmeticError	ArithmeticError	(1)
PyExc_LookupError	LookupError	(1)
PyExc_AssertionError	AssertionError	
PyExc_AttributeError	AttributeError	
PyExc_EOFError	EOFError	
PyExc_EnvironmentError	EnvironmentError	(1)
PyExc_FloatingPointError	FloatingPointError	
PyExc_IOError	IOError	
PyExc_ImportError	ImportError	
PyExc_IndexError	IndexError	
PyExc_KeyError	KeyError	
PyExc_KeyboardInterrupt	KeyboardInterrupt	
PyExc_MemoryError	MemoryError	
PyExc_NameError	NameError	
PyExc_NotImplementedError	NotImplementedError	
PyExc_OSError	OSError	
PyExc_OverflowError	OverflowError	
PyExc_ReferenceError	ReferenceError	(2)
PyExc_RuntimeError	RuntimeError	
PyExc_SyntaxError	SyntaxError	
PyExc_SystemError	SystemError	
PyExc_SystemExit	SystemExit	
PyExc_TypeError	TypeError	
PyExc_ValueError	ValueError	
PyExc_WindowsError	WindowsError	(3)
PyExc_ZeroDivisionError	ZeroDivisionError	

注記:

- (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。

4.2 文字列例外の廃止

Python へ組み込まれるすべての例外あるいは標準ライブラリに提供されている例外は、`Exception` から導出されています。

インタプリタで既存のコードが変更なしで動作するように、文字列例外は今でもサポートされています。しかし、これも将来のリリースで変更されるでしょう。

ユーティリティ関数

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, 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。

`long PyOS_GetLastModificationTime(char *filename)`

ファイル `filename` の最終更新時刻を返します。結果は標準 C ライブラリ関数 `time()` が返すタイムスタンプと同じ様式で符号化されています。

`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 版と Macintosh 版) にしか利用できません。 `USE_CHECKSTACK` は自動的に定義されます; 自前のコードでこの定義を変更してはなりません。

`PyOS_sighandler_t PyOS_getsig(int i)`

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

`PyOS_sighandler_t PyOS_setsig(int i, PyOS_sighandler_t h)`

シグナル `i` に対する現在のシグナルハンドラを `h` に設定します; 以前のシグナルハンドラを返します。この関数は `sigaction()` または `signal()` のいずれかに対する薄いラップです。 `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。

5.2 プロセス制御

`void Py_FatalError(const char *message)`

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

`void Py_Exit(int status)`

現在のプロセスを終了 (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 も呼び出しはなりません。

5.3 モジュールの import

`PyObject* PyImport_ImportModule(char *name)`

戻り値: *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 (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 — `sys.modules` を使って調べてください)。

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

戻り値: *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` されたモジュールかトップレベルパッケージへの新たな参照になります。失敗した場合には例外をセットし、NULL を返します (その場合でも、モジュールは生成されている場合があります) `__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。

`PyObject* PyImport_Import(PyObject *name)`

戻り値: *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。

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

戻り値: *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かトップレベルパッケージへの新たな参照になります。失敗した場合には例外をセットし、`NULL` を返します (その場合でも、モジュールは生成されている場合があります)

`PyObject*` `PyImport_AddModule(char *name)`

戻り値: *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。注意: 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)`

戻り値: *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 (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)。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。 `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。

`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()`

戻り値: *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。

`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 コ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タの構造体定義です。(Python ソース配布物の `'Tools/freeze/'` を参照してください) この構造体の定義は `'Include/import.h'` にあり、以下のようになっています:

```
struct _frozen {
    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ことができます。

int PyImport_AppendInittab(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ねばなりません。

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ねばなりません。

5.4 データ整列化 (data marshalling) のサポート

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

数値は最小桁が先にくるように記録されます。

`void PyMarshal_WriteLongToFile(long value, FILE *file)`

`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。

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

Python オブジェクト `value` を `file` へ整列化します。

`PyObject* PyMarshal_WriteObjectToString(PyObject *value)`

戻り値: *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。

以下の関数を使うと、整列化された値を読み戻せます。

`long PyMarshal_ReadLongFromFile(FILE *file)`

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

`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)`

戻り値: *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)`

戻り値: *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づつ読み出す代わりにメモリ上のデータを操作できるようにします。対象のファイルから他に何も読み出さないと分かっている場合にのみ、この関数を使ってください。エラーが生じた場合、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, int len)`

戻り値: *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。

5.5 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 *Python* インタプリタの拡張と埋め込み 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。一つの書式化単位は一つの Python オブジェクトを表します; 通常は単一の文字か、書式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。

‘s’ (文字列型または Unicode オブジェクト型) [`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 バイトが途中で埋め込まれてはなりません; 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 オブジェクト型または任意の読み出しバッファ互換型) [`char *`, `int`] 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。

‘z’ (文字列型または None) [`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) [`char *`, `int`] ‘s#’ の ‘s’ を ‘z’ にしたような意味です。

‘u’ (Unicode オブジェクト型) [`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 オブジェクト型) [`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` 型配列へのポインタと解釈して扱います。

‘es’ (文字列型、Unicode オブジェクト型または任意の読み出しバッファ互換型) [`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 `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 オブジェクト型または文字列バッファ互換型) [`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 オブジェクト型または文字列バッファ互換型) [`const char *encoding, char **buffer, int *buffer_` ‘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 `char*` 型か `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 `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 オブジェクト型または文字列バッファ互換型) [`const char *encoding, char **buffer`] ‘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) [`char`] Python の整数型を、C の `char` 型の小さな整数に変換します。

‘B’ (整数型) [`unsigned char`] Python の整数型を、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。2.3 で追加された仕様です。

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

‘H’ (整数型) [`unsigned short int`] Python の整数型を、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。2.3 で追加された仕様です。

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

- ‘I’ (整数型) [unsigned int] Python の整数型を、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。2.3 で追加された仕様です。
- ‘l’ (整数型) [long int] Python の整数型を、C の long int 型に変換します。
- ‘k’ (整数型) [unsigned long] Python の整数型を、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。2.3 で追加された仕様です。
- ‘L’ (整数型) [PY_LONG_LONG] Python の整数型を、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。Convert a Python integer to a C long long. This format is only available on platforms that support long long (or _int64 on Windows).
- ‘K’ (整数型) [unsigned PY_LONG_LONG] Python の整数型を、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。2.3 で追加された仕様です。
- ‘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’ (オブジェクト) [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!’ (オブジェクト) [typeobject, PyObject*] Python オブジェクトを C の Python オブジェクト型ポインタに保存します。‘O’ に似ていますが、二つの C の引数をとります: 一つ目の引数は Python の型オブジェクトへのアドレスで、二つ目の引数はオブジェクトへのポインタが保存されている (PyObject* の) C の変数へのアドレスです。Python オブジェクトが指定した型ではない場合、TypeError を送ります。
- ‘O&’ (オブジェクト) [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 関数は例外を送り出さ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 を送ります。C 変数は PyObject* で宣言しておいてもかまいません。
- ‘U’ (Unicode 文字列型) [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 を送ります。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 *, int] ‘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、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。

‘(items)’ (タプル) [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 の配列型でなくてはなりません。各 C 引数は *items* 内の個々の書式化単位に対応づけできねばなりません。配列の書式化単位は入れ子構造にできます。

注意: 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。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

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。言うまでもなく、‘:’ と ‘;’ は相互に排他の文字です。

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せねばなりません。

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せねばなりません。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。

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

固定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_ParseTupleAndKeywords(PyObject *args, PyObject *kw, char *format, char *keywords[], ...)
```

固定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_Parse(PyObject *args, char *format, ...)
```

“旧形式”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 — 旧形式の関数は、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, char *name, int min, int max, ...)
```

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` として宣言しなくてはなりません。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くてはなりません。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くてはならず、各補助引数は `PyObject*` 変数へのポインタでなくてはなりません; これらの補助引数には、*args* の値が入ります; 値の参照は借りた参照です。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ねばなりません。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:

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

と全く等価です。

2.2 で追加された仕様です。

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

戻り値: *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。サイズがゼロや 1 のタプルを返すように強制するには、丸括弧で囲われた書式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ー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、コロンの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' (string or None) [char *]'s' と同じです。

'z#' (string or None) [char *, int]'s#' と同じです。

'u' (Unicode string) [Py_UNICODE *]null 終端された Unicode (UCS-2) データのバッファから Python 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 string) [Py_UNICODE *, int]null 終端された Unicode (UCS-2) データのバッファとその長さから Python 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]'i' と同じです。

'h' (整数型) [short int]'i' と同じです。

'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。

'c' (string of length 1) [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 の複素数に変換します。

'o' (オブジェクト) [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()` 自体は例外を送りません。例外をまだ送出していなければ `SystemError` をセットします。

‘s’ (オブジェクト) [`PyObject *`]‘O’ と同じです。

‘u’ (オブジェクト) [`PyObject *`]‘O’ と同じです。

‘N’ (オブジェクト) [`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&’ (オブジェクト) [`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。

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、Python オブジェクトとのやりとりを型や (数値型全て、配列型全てといった) 大まかなオブジェクト型の種類に関係なく行います。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となります。

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_HasAttrString(PyObject *o, 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_GetAttrString(PyObject *o, char *attr_name)`

戻り値: *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。

`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。

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

戻り値: *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。

`int PyObject_SetAttrString(PyObject *o, 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 `-1` を返します。この関数は Python の式 `'o.attr_name = v'` と同じです。

`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 `-1` を返します。この関数は Python の式 `'o.attr_name = v'` と同じです。

`int PyObject_DelAttrString(PyObject *o, 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。

`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。

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

戻り値: *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。

`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。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。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)`

戻り値: *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)`

戻り値: *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_Unicode(PyObject *o)`

戻り値: *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タリについてチェックを行います。少なくとも一つのエンタ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 — この場合、`__class__` 属性の値と、*cls* の値の間のクラス関係を、関数の戻り値を決定するのに使います。2.1 で追加された仕様です。2.2 で変更された仕様: 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正攻法で行いますが、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定めるには、より汎用の機構を使います。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 — オブジェクトに `__bases__` があるだけで、この決定法を適用する条件を満たしているとみなされます。

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

クラス *derived* が *cls* と同じクラスか、*cls* の導出クラスの場合に 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。2.1 で追加された仕様です。2.3 で変更された仕様: 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)`

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。2.2 で追加された仕様です。

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

戻り値: *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, ...)`

戻り値: *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。

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

戻り値: *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 `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。

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

戻り値: *New reference.*

呼び出し可能な Python オブジェクト *callable_object* 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。2.2 で追加された仕様です。

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

戻り値: *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。2.2 で追加された仕様です。

`int` `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。

`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)`

戻り値: *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。2.2 で追加された仕様です。

`int` `PyObject_Length(PyObject *o)`

`int` `PyObject_Size(PyObject *o)`

o の長さを返します。オブジェクト *o* が配列プロトコルとマップ型プロトコルのいずれかを提供している場合、配列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。

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

戻り値: *New reference.*

成功するとオブジェクト *key* に対応する *o* の要素を返し、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 `-1` 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* に対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)`

戻り値: *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)`

戻り値: *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)`

戻り値: *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)`

戻り値: *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)`

戻り値: *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)`

戻り値: *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)`

戻り値: *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。2.2 で追加された仕様です。

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

戻り値: *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。2.2 で追加された仕様です。

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

戻り値: *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)

戻り値: *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)

戻り値: *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)

戻り値: *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)

戻り値: *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)

戻り値: *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)

戻り値: *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 ' $\sim o$ ' と同じです。

PyObject* **PyNumber_Lshift**(PyObject *o1, PyObject *o2)

戻り値: *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 ' $o1 \ll o2$ ' と同じです。

PyObject* **PyNumber_Rshift**(PyObject *o1, PyObject *o2)

戻り値: *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 ' $o1 \gg o2$ ' と同じです。

PyObject* **PyNumber_And**(PyObject *o1, PyObject *o2)

戻り値: *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)

戻り値: *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 ' $o1 \ \wedge \ o2$ ' と同じです。

PyObject* **PyNumber_Or**(PyObject *o1, PyObject *o2)

戻り値: *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)

戻り値: *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)

戻り値: *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)

戻り値: *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)

戻り値: *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)

戻り値: *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。2.2 で追加された仕様です。

PyObject* **PyNumber_InPlaceTrueDivide**(PyObject *o1, PyObject *o2)

戻り値: *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。2.2 で追加された仕様です。

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

戻り値: *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)

戻り値: *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`)

戻り値: *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`)

戻り値: *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`)

戻り値: *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`)

戻り値: *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`)

戻り値: *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。

`PyObject*` **PyNumber_Int** (`PyObject *o`)

戻り値: *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`)

戻り値: *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`)

戻り値: *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。

6.3 配列プロトコル (sequence protocol)

`int PySequence_Check(PyObject *o)`

オブジェクトが配列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。

`int PySequence_Size(PyObject *o)`

成功すると配列 *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。配列プロトコルをサポートし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。

`int PySequence_Length(PyObject *o)`

`PySequence_Size()` の別名です。

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

戻り値: *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, int count)`

戻り値: *New reference.*

成功するとオブジェクト *o* の *count* 回繰り返しを返し、失敗すると NULL を返します。Python の式 `'o * count'` と同じです。

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

戻り値: *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, int count)`

戻り値: *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。*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 `'o *= count'` と同じです。

`PyObject* PySequence_GetItem(PyObject *o, int i)`

戻り値: *New reference.*

成功すると *o* の *i* 番目の要素を返し、失敗すると NULL を返します。Python の式 `'o[i]'` と同じです。

`PyObject* PySequence_GetSlice(PyObject *o, int i1, int i2)`

戻り値: *New reference.*

成功すると *o* の *i1* から *i2* までの間のスライスを返し、失敗すると NULL を返します。Python の式 `'o[i1:i2]'` と同じです。

`int PySequence_SetItem(PyObject *o, int i, PyObject *v)`

o の *i* 番目の要素に *v* を代入します。失敗すると -1 を返します。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。

`int PySequence_DelItem(PyObject *o, int i)`

o の *i* 番目の要素を削除します。失敗すると -1 を返します。Python の文 `'del o[i]'` と同じです。

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

o の *i1* から *i2* までの間のスライスに *v* を代入します。Python の文 `'o[i1:i2] = v'` と同じです。

`int PySequence_Delslice(PyObject *o, int i1, int i2)`

配列オブジェクト *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。

PyObject* **PySequence_Tuple**(PyObject *o)
 戻り値: *New reference*.
 成功すると *o* をタプルで返します。失敗すると NULL を返します。Python の式 'tuple(*o*)' と同じです。

int **PySequence_Count**(PyObject *o, PyObject *value)
o における *value* の出現回数、すなわち *o[key] == value* となる *key* の個数を返します。失敗すると -1 を返します。Python の式 '*o*.count(*value*)' と同じです。

int **PySequence_Contains**(PyObject *o, PyObject *value)
o に *value* が入っているか判定します。*o* のある要素が *value* と等価 (equal) ならば 1 を返し、それ以外の場合には 0 を返します。エラーが発生すると -1 を返します。Python の式 '*value* in *o*' と同じです。

int **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。

PyObject* **PySequence_List**(PyObject *o)
 戻り値: *New reference*.
 任意の配列 *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)
 戻り値: *New reference*.
 任意の配列 *o* と同じ内容を持つタプルオブジェクトを返します。*o* がタプルの場合、新たな参照を返します。それ以外の場合、適切な内容が入ったタプルを構築して返します。

PyObject* **PySequence_Fast**(PyObject *o, const char *m)
 戻り値: *New reference*.
 配列 *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。オブジェクトが配列でなければ、*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, int i)
 戻り値: *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。

PyObject* **PySequence_ITEM**(PyObject *o, int i)
 戻り値: *New reference*.
 成功すると the *i*th element of *o* を返し、失敗すると NULL を返します。PySequence_GetItem() ですが、PySequence_Check(*o*) 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。2.3 で追加された仕様です。

int **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。

`int 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。

`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* に対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)`

成功すると、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。Python の式 `'o.has_key(key)'` と同じです。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。Python の式 `'o.has_key(key)'` と同じです。この関数呼び出しは常に成功します。

`PyObject* PyMapping_Keys(PyObject *o)`

戻り値: *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)`

戻り値: *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)`

戻り値: *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)`

戻り値: *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)

2.2 で追加された仕様です。

イテレータを扱うための固有の関数は二つしかありません。

```
int PyIter_Check(PyObject *o)
```

o がイテレータプロトコルをサポートする場合に真を返します。

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

戻り値: *New reference*.

反復処理 *o* における次の値を返します。オブジェクトがイテレータの場合、この関数は反復処理における次の値を取り出します。要素が何も残っていない場合には例外がセットされていない状態で NULL を返します。オブジェクトがイテレータでない場合には `TypeError` を送出します。要素を取り出す際にエラーが生じると 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 のコードは以下のようなになります:

```
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;

if (iterator == NULL) {
    /* エラーの伝播処理をここに書く */
}

while (item = PyIter_Next(iterator)) {
    /* 取り出した要素で何らかの処理を行う */
    ...
    /* 終わったら参照を放棄する */
    Py_DECREF(item);
}

Py_DECREF(iterator);

if (PyErr_Occurred()) {
    /* エラーの伝播処理をここに書く */
}
else {
    /* 別の処理を続ける */
}
```

6.6 バッファプロトコル (buffer protocol)

```
int PyObject_AsCharBuffer(PyObject *obj, const char **buffer, int *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。 1.6 で追加された仕様です。

```
int PyObject_AsReadBuffer(PyObject *obj, const char **buffer, int *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。 1.6 で追加された仕様です。

```
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。 2.2 で追加された仕様です。

```
int PyObject_AsWriteBuffer(PyObject *obj, char **buffer, int *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。1.6 で追加された仕様です。

具象オブジェクト (concrete object) レイヤ

この章では、特定の Python オブジェクト型固有の関数について述べています。これらの関数に間違っただけ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

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単量子 (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 (type object)

`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 `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導出されたサブタイプ (subtype) のインスタンスも含まれます。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。2.2 で追加された仕様です。

`int` `PyType_HasFeature(PyObject *o, int feature)`

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

`int` `PyType_IS_GC(PyObject *o)`

型オブジェクト `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。2.0 で追加された仕様です。

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

a が *b* のサブタイプの場合に真を返します。2.2 で追加された仕様です。

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

戻り値: *New reference*.

2.2 で追加された仕様です。

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

戻り値: *New reference*.

2.2 で追加された仕様です。

`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。2.2 で追加された仕様です。

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、オブジェクトのアイデンティティテスト (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。リファレンス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。

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 Python の整数型オブジェクトを表現します。

`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 `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。2.2 で変更された仕様: 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。2.2 で追加された仕様です。

`PyObject* PyInt_FromString(char *str, char **pend, int base)`

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' で始まっているならば、基数に 16 を使います; *str* が '0' で始まっているならば、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。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、オーバーフロー警告が抑制されていれば、`PyLongObject` を返します。オーバーフロー警告が抑制されていなければ、`NULL` を返します。

`PyObject*` **PyInt_FromLong**(*long ival*)

戻り値: *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、-1 から 100 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っています。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 :-)

`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。

`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。この関数はオーバーフローをチェックしません。2.3 で追加された仕様です。

`unsigned 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。オーバーフローをチェックしません。2.3 で追加された仕様です。

`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。

7.2.2 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 `types.LongType` と同じオブジェクトです。

`int` **PyLong_Check**(*PyObject *p*)

引数が `PyLongObject` か `PyLongObject` のサブタイプの際に真を返します。2.2 で変更された仕様: 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。2.2 で追加された仕様です。

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

戻り値: *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*)

戻り値: *New reference.*

C の unsigned 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_FromLongLong**(long long v)

戻り値: *New reference*.

C の long long 型から新たな PyLongObject オブジェクトを生成して返します。失敗のときには NULL を返します。

PyObject* **PyLong_FromUnsignedLongLong**(unsigned long long v)

戻り値: *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)

戻り値: *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)

戻り値: *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, int length, int base)

戻り値: *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。1.6 で追加された仕様です。

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

戻り値: *New reference*.

Python 整数型または長整数型をポインタ p から生成します。ポインタに入れる値は PyLong_AsVoidPtr() を使って得られるような値です。1.5.2 で追加された仕様です。

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 を送出します。

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。

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 を送出します。2.2 で追加された仕様です。

unsigned long long **PyLong_AsUnsignedLongLong**(PyObject *pylong)

pylong の指す値を、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、正の値なら OverflowError を、負の値なら TypeError を送出します。2.2 で追加された仕様です。

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。2.3 で追加された仕様です。

unsigned 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。2.3 で追加された仕様です。

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。1.5.2 で追加された仕様です。

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

PyFloatObject

この PyObject のサブタイプは Python 浮動小数点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 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。2.2 で変更された仕様: 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。2.2 で追加された仕様です。

PyObject* **PyFloat_FromString**(PyObject *str, char **pend)

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**(double v)

戻り値: *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。

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。

7.2.4 浮動小数点オブジェクト (complex number object)

Python の複素数オブジェクトは、C API 側から見ると二つの別個の型として実装されています: 一方は Python プログラムに対して公開されている Python のオブジェクトで、他方は実際の複素数値を表現する C の構造体です。API では、これら双方を扱う関数を提供しています。

C 構造体としての複素数

複素数の C 構造体を引数として受理したり、戻り値として返したりする関数は、ポインタ渡しを行うのではなく 値渡し を行うので注意してください。これは API 全体を通して一貫しています。

Py_complex

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

```
typedef struct {
    double real;
    double imag;
} Py_complex;
```

Py_complex **_Py_c_sum**(Py_complex left, Py_complex right)

二つの複素数の和を C の Py_complex 型で返します。

Py_complex **_Py_c_diff**(Py_complex left, Py_complex right)

二つの複素数の差を C の Py_complex 型で返します。

Py_complex **_Py_c_neg**(Py_complex complex)

複素数 *complex* の符号反転 C の Py_complex 型で返します。

Py_complex **_Py_c_prod**(Py_complex left, Py_complex right)

二つの複素数の積を C の Py_complex 型で返します。

Py_complex **_Py_c_quot**(Py_complex dividend, Py_complex divisor)

二つの複素数の商を C の Py_complex 型で返します。

Py_complex **_Py_c_pow**(Py_complex num, Py_complex exp)

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

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 の複素数オブジェクトを表現します。

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。

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。2.2 で変更された仕様: 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。2.2 で追加された仕様です。

PyObject* **PyComplex_FromCComplex**(Py_complex v)

戻り値: *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)

戻り値: *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。

7.3 配列オブジェクト (sequence object)

配列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配列オブジェクトについて扱います。

7.3.1 文字列オブジェクト (string object)

以下の関数では、文字列が渡されるはずのパラメタに非文字列が渡された場合に `TypeError` を送出します。

```
PyObject* 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 `types.TypeType` 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。 2.2 で変更された仕様: 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。 2.2 で追加された仕様です。

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

戻り値: *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, int len)
```

戻り値: *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。

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

戻り値: *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 printf("%d") と全く同じ。
%ld	long	C の printf("%ld") 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 の文字配列。
%p	void*	C ポインタの 16 進表記。 printf("%p") とほとんど同じだが、プラットフォームにおけ

PyObject* **PyString_FromFormatV**(const char *format, va_list vargs)

戻り値: *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。

int **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。

int **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。

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, int *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 NULL を返して TypeError を送出します。

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, int newsize)`

“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。

`PyObject* PyString_Format (PyObject *format, PyObject *args)`

戻り値: *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。)

`PyObject* PyString_InternFromString (const char *v)`

戻り値: *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。

`PyObject* PyString_Decode (const char *s, int size, const char *encoding, const char *errors)`

戻り値: *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。

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

戻り値: *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。

`PyObject* PyString_Encode (const char *s, int size, const char *encoding, const char *errors)`

戻り値: *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。

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

戻り値: *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。

7.3.2 Unicode オブジェクト (Unicode object)

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 16 ビットの符号なしストレージ型を表し、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。 `wchar_t` が利用でき、かつ 16 ビットである場合、 `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。その他のプラットフォームでは全て、 `Py_UNICODE` は `unsigned short` の typedef によるエイリアスになります。

PyUnicodeObject

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

`PyTypeObject PyUnicode_Type`

この `PyTypeObject` のインスタンスは Python の Unicode 型を表現します。

以下の API は実際には C マクロで、Unicode オブジェクト内部の読み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。 2.2 で変更された仕様: 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。 2.2 で追加された仕様です。

```
int 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)。

```
int 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)。

```
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)。

Unicode は数多くの異なる文字プロパティ (character property) を提供しています。よく使われる文字プロパティは、以下のマクロで利用できます。これらのマクロは Python の設定に応じて、各々 C の関数に対応

付けられています。

```
int Py_UNICODE_ISSPACE(Py_UNICODE ch)
    ch が空白文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISLOWER(Py_UNICODE ch)
    ch が小文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISUPPER(Py_UNICODE ch)
    ch が大文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISTITLE(Py_UNICODE ch)
    ch がタイトルケース文字 (titlecase character) かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISLINEBREAK(Py_UNICODE ch)
    ch が改行文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISDECIMAL(Py_UNICODE ch)
    ch が 10 進の数字文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISDIGIT(Py_UNICODE ch)
    ch が 2 進の数字文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISNUMERIC(Py_UNICODE ch)
    ch が数字文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISALPHA(Py_UNICODE ch)
    ch がアルファベット文字かどうかに応じて 1 または 0 を返します。

int Py_UNICODE_ISALNUM(Py_UNICODE ch)
    ch が英数文字かどうかに応じて 1 または 0 を返します。
```

以下の API は、高速に直接文字変換を行うために使われます:

```
Py_UNICODE Py_UNICODE_TOLOWER(Py_UNICODE ch)
    ch を小文字に変換したものを返します。

Py_UNICODE Py_UNICODE_Toupper(Py_UNICODE ch)
    ch を大文字に変換したものを返します。

Py_UNICODE Py_UNICODE_TOTITLE(Py_UNICODE ch)
    ch をタイトルケース文字に変換したものを返します。

int Py_UNICODE_TODECIMAL(Py_UNICODE ch)
    ch を 10 進の正の整数に変換したものを返します。不可能ならば -1 を返します。例外を送出しませぬ。

int Py_UNICODE_TODIGIT(Py_UNICODE ch)
    ch を一桁の 2 進整数に変換したものを返します。不可能ならば -1 を返します。例外を送出しませぬ。

double Py_UNICODE_TONUMERIC(Py_UNICODE ch)
    ch を (正の) 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。例外を送出しませぬ。
```

Unicode オブジェクトを生成したり、Unicode の配列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, int size)
```

戻り値: *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。

`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。

`int` `PyUnicode_GetSize(PyObject *unicode)`

Unicode オブジェクトの長さを返します。

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

戻り値: *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 オブジェクトに型強制して、参照カウントをインクリメントして返します。

型強制は以下のようにして行われます:

1. *obj* が Unicode オブジェクトの場合、オブジェクトそのままを、参照カウントをインクリメントして返します。注意: Unicode オブジェクトはデコードできません; *encoding* に NULL でない値を渡すと、`TypeError` になります。
2. 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)
3. その他のオブジェクトは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を `decref` する責任があります。

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

戻り値: *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` に対するインタフェースを確立することがあります。このサポートは、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をもたらします。

`PyObject*` `PyUnicode_FromWideChar(const wchar_t *w, int size)`

戻り値: *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。

`int` `PyUnicode_AsWideChar(PyUnicodeObject *unicode, wchar_t *w, int 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` 文字をコピーします。コピーされた `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。

組み込み codec (built-in codec)

Python では、処理速度を高めるために C で書かれた一そろいの codec を提供しています。これらの codec は全て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タと同じセマンティクスになっています。

`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、例えば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。

以下は汎用 `codec` の API です:

```
PyObject* PyUnicode_Decode(const char *s, int size, const char *encoding, const char *errors)
```

戻り値: *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()` の同名のパラメタと同じ意味を持ちます。使用する `codec` の検索は、Python の `codec` レジストリを使って行います。`codec` が例外を送出した場合には `NULL` を返します。

```
PyObject* PyUnicode_Encode(const Py_UNICODE *s, int size, const char *encoding, const char *errors)
```

戻り値: *New reference.*

`size` で指定されたサイズの `Py_UNICODE` バッファをエンコードした 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タと同じ意味を持ちます。使用する `codec` の検索は、Python の `codec` レジストリを使って行います。`codec` が例外を送出した場合には `NULL` を返します。

```
PyObject* PyUnicode_AsEncodedString(PyObject *unicode, const char *encoding, const char *errors)
```

戻り値: *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。

以下は `UTF-8` `codec` の API です:

```
PyObject* PyUnicode_DecodeUTF8(const char *s, int size, const char *errors)
```

戻り値: *New reference.*

`UTF-8` でエンコードされた `size` バイトの文字列 `s` から `Unicode` オブジェクトを生成します。`codec` が例外を送出した場合には `NULL` を返します。

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

戻り値: *New reference.*

`size` で指定された長さを持つ `Py_UNICODE` 型バッファを `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。

```
PyObject* PyUnicode_AsUTF8String(PyObject *unicode)
```

戻り値: *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。

以下は `UTF-16` `codec` の API です:

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

戻り値: *New reference*.

UTF-16 でエンコードされたバッファ *s* から *size* バイトデコードして、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 でない場合、デコード機構は以下のように指定されたバイト整列 (byte order) に従ってデコードを開始します:

```
*byteorder == -1: リトルエンディアン
*byteorder == 0:  ネイティブ
*byteorder == 1:  ビッグエンディアン
```

その後、入力データ中に見つかった全てのバイト整列マーカ (byte order mark, BOM) に従って、バイト整列を切り替えます。BOM はデコード結果の Unicode 文字列中にはコピーされません。デコードを完了した後、**byteorder* は入力データの終点現在におけるバイト整列に設定されます。

byteorder が NULL の場合、codec はネイティブバイト整列のモードで開始します。

codec が例外を送出した場合には NULL を返します。

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

戻り値: *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。*byteorder* が 0 でない場合、出力は以下のバイト整列指定に従って書き出されます:

```
byteorder == -1: リトルエンディアン
byteorder == 0:  ネイティブ (BOM マーカを書き出します)
byteorder == 1:  ビッグエンディアン
```

バイトオーダが 0 の場合、出力結果となる文字列は常に Unicode BOM マーカ (U+FEFF) で始まりません。それ以外のモードでは、BOM マーカを頭につけません。

Py_UNICODE データは UCS-2 に縮小された UTF-16 として解釈されることに注意してください。このトリックによって、API を傷つけることなく、完全な UTF-16 機能を後で追加できます。

codec が例外を送出した場合には NULL を返します。

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

戻り値: *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 マーカから始まります。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。

以下は “Unicode Escape” codec の API です:

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

戻り値: *New reference*.

Unicode-Escape でエンコードされた *size* バイトの文字列 *s* から Unicode オブジェクトを生成します。codec が例外を送出した場合には NULL を返します。

PyObject* **PyUnicode_EncodeUnicodeEscape**(const Py_UNICODE *s, int size, const char *errors)

戻り値: *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。

`PyObject* PyUnicode_AsUnicodeEscapeString(PyObject *unicode)`

戻り値: *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。

以下は “Raw Unicode Escape” codec の API です:

`PyObject* PyUnicode_DecodeRawUnicodeEscape(const char *s, int size, const char *errors)`

戻り値: *New reference.*

Raw-Unicode-Escape でエンコードされた *size* バイトの文字列 *s* から Unicode オブジェクトを生成します。codec が例外を送出した場合には NULL を返します。

`PyObject* PyUnicode_EncodeRawUnicodeEscape(const Py_UNICODE *s, int size, const char *errors)`

戻り値: *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。

`PyObject* PyUnicode_AsRawUnicodeEscapeString(PyObject *unicode)`

戻り値: *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。

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, int size, const char *errors)`

戻り値: *New reference.*

Latin-1 でエンコードされた *size* バイトの文字列 *s* から Unicode オブジェクトを生成します。codec が例外を送出した場合には NULL を返します。

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

戻り値: *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。

`PyObject* PyUnicode_AsLatin1String(PyObject *unicode)`

戻り値: *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。

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, int size, const char *errors)`

戻り値: *New reference.*

ASCII でエンコードされた *size* バイトの文字列 *s* から Unicode オブジェクトを生成します。codec が例外を送出した場合には NULL を返します。

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

戻り値: *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。

`PyObject* PyUnicode_AsASCIIString(PyObject *unicode)`

戻り値: *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。

以下は mapping codec の API です:

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マップ型 (mapping) を使います。

デコード用のマップ型は、文字列型の字列一組みを、Unicode 型の字列一組、整数 (Unicode 序数として解釈されます) または None (“定義されていない対応付け (undefined mapping)”) 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。

デコード用のマップ型は、Unicode 型の字列一組みを、string 型の字列一組、整数 (Latin-1 序数として解釈されます) または None (“定義されていない対応付け (undefined mapping)”) 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。

マップ型オブジェクトは、`__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点に対応付けるものだけです。

```
PyObject* PyUnicode_DecodeCharmap(const char *s, int size, PyObject *mapping, const char *errors)
```

戻り値: *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。

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

戻り値: *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。

```
PyObject* PyUnicode_AsCharmapString(PyObject *unicode, PyObject *mapping)
```

戻り値: *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, int size, PyObject *table, const char *errors)
```

戻り値: *New reference.*

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。

対応付けを行う `table` は、Unicode 序数を表す整数を Unicode 序数を表す整数または None に対応付けます。(None の場合にはその文字を削除します)

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、辞書や配列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。

以下は MBCS codec の API です。この codec は現在のところ、Windows 上だけで利用でき、変換の実装には Win32 MBCS 変換機構 (Win32 MBCS converter) を使っています。MBCS (または DBCS) はエンコー

ド方式の種類 (class) を表す言葉で、単一のエンコード方式を表すわけでないので注意してください。利用されるエンコード方式 (target encoding) は、codec を動作させているマシン上のユーザ設定で定義されています。

PyObject* **PyUnicode_DecodeMBCS**(const char *s, int size, const char *errors)

戻り値: *New reference*.

MBCS でエンコードされた *size* バイトの文字列 *s* から Unicode オブジェクトを生成します。codec が例外を送出した場合には NULL を返します。

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

戻り値: *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。

PyObject* **PyUnicode_AsMBCSString**(PyObject *unicode)

戻り値: *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。

メソッドおよびスロット関数 (slot function)

以下の API は Unicode オブジェクトおよび文字列を入力に取り (説明では、どちらも文字列と表記しています)、場合に応じて Unicode オブジェクトか整数を返す機能を持っています。

これらの関数は全て、例外が発生した場合には NULL または -1 を返します。

PyObject* **PyUnicode_Concat**(PyObject *left, PyObject *right)

戻り値: *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, int maxsplit)

戻り値: *New reference*.

Unicode 文字列のリストを分割して、Unicode 文字列からなるリストを返します。sep が NULL の場合、全ての空白文字を使って分割を行います。それ以外の場合、指定された文字を使って分割を行います。最大で *maxsplit* 個までの分割を行います。*maxsplit* が負ならば分割数に制限を設けません。分割結果のリスト内には分割文字は含みません。

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

戻り値: *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)

戻り値: *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。

対応付けを行う *table* は、Unicode 序数を表す整数を Unicode 序数を表す整数または None に対応付けます。(None の場合にはその文字を削除します)

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、辞書や配列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。

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)

戻り値: *New reference*.

指定した *separator* で文字列からなる配列を連結 (join) し、連結結果を Unicode 文字列で返します。

PyObject* **PyUnicode_Tailmatch**(PyObject *str, PyObject *substr, int start, int end, int direction)

戻り値: *New reference*.

substr が指定された末尾条件 (*direction* == -1 は前方一致、*direction* == 1 は後方一致) で *str[start:end]* 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。

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

str[start:end] 中に *substr* が最初に出現する場所を返します。このとき指定された検索方向 *direction* (*direction* == 1 は順方向検索、*direction* == -1 は逆方向検索) で検索します。戻り値は最初にマッチが見つかった場所のインデクスです; 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。

int **PyUnicode_Count**(PyObject *str, PyObject *substr, int start, int end)

str[start:end] に *substr* が重複することなく出現する回数を返します。エラーが発生した場合には -1 を返します。

PyObject* **PyUnicode_Replace**(PyObject *str, PyObject *substr, PyObject *replstr, int maxcount)

戻り値: *New reference*.

str 中に出現する *substr* を最大で *maxcount* 個 *replstr* に置換し、置換結果を Unicode オブジェクトにして返します。 *maxcount* == -1 にすると、全ての *substr* を置換します。

int **PyUnicode_Compare**(PyObject *left, PyObject *right)

二つの文字列を比較して、左引数が右引数より小さい場合、左右引数が等価の場合、左引数が右引数より大きい場合、について、それぞれ -1, 0, 1 を返します。

PyObject* **PyUnicode_Format**(PyObject *format, PyObject *args)

戻り値: *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。引数 *args* はタプルでなくてはなりません。

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。

7.3.3 Buffer Objects

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単位形式で公開しています。アレイもその内容を公開していますが、注意する必要があるのはアレイの要素は複数バイトの値になりうる、ということです。

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 (10.7 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、操作前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 `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()` または `PyBuffer_FromReadWriteObject()` *size* パラメータに渡します。このパラメータを渡すと、`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。このパラメータを使うことで、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, int offset, int size)

戻り値: *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。

PyObject* PyBuffer_FromReadWriteObject(PyObject *base, int offset, int size)

戻り値: *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。パラメータおよび例外は `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。

PyObject* PyBuffer_FromMemory(void *ptr, int size)

戻り値: *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 を指定してはなりません; 指定すると、ValueError を送出します。

PyObject* **PyBuffer_FromReadWriteMemory**(void *ptr, int size)

戻り値: *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。

PyObject* **PyBuffer_New**(int size)

戻り値: *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。

7.3.4 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 types.TupleType と同じオブジェクトです。

int **PyTuple_Check**(PyObject *p)

p がタプルオブジェクトか、タプル型のサブタイプのインスタンスである場合に真を返します。2.2 で変更された仕様: 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。2.2 で追加された仕様です。

PyObject* **PyTuple_New**(int len)

戻り値: *New reference*.

サイズが len 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。

int **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。

int **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; エラーチェックを行いません。

PyObject* **PyTuple_GetItem**(PyObject *p, int pos)

戻り値: *Borrowed reference*.

p の指すタプルオブジェクト内の、位置 pos にあるオブジェクトを返します。pos が範囲を超えている場合、NULL を返して IndexError 例外をセットします。

PyObject* **PyTuple_GET_ITEM**(PyObject *p, int pos)

戻り値: *Borrowed reference*.

PyTuple_GetItem() に似ていますが、引数に対するエラーチェックを行いません。

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

戻り値: *New reference*.

p の指すタプルオブジェクト内の、位置 low から high までのスライスを取り出して、タプルオブジェクトとして返します。

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

p の指すタプルオブジェクト内の位置 pos に、オブジェクト o への参照を挿入します。成功した場合

には 0 を返します。注意: この関数は *o* への参照を“盗み取り”ます。

```
void PyTuple_SET_ITEM(PyObject *p, int 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。注意: この関数は *o* への参照を“盗み取り”ます。

```
int _PyTuple_Resize(PyObject **p, int 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、この関数は対象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気体してはなりません。**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。2.2 で変更された仕様: 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

7.3.5 List Objects

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 のタプル型を表現します。これは Python レイヤにおける `types.ListType` 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。2.2 で変更された仕様: 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 がリストオブジェクトで、かつリスト型のサブタイプのインスタンスでない場合に真を返します。2.2 で追加された仕様です。

```
PyObject* PyList_New(int len)
```

戻り値: *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。

```
int 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。

```
int PyList_GET_SIZE(PyObject *list)
```

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

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

戻り値: *Borrowed reference*.

p の指すリストオブジェクト内の、位置 *pos* にあるオブジェクトを返します。*pos* が範囲を超えている場合、NULL を返して `IndexError` 例外をセットします。

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

戻り値: *Borrowed reference*.

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

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

リストオブジェクト内の位置 *index* に、オブジェクト *item* を挿入します。成功した場合には 0 を返し、失敗すると -1 を返します。注意: この関数は *item* への参照を“盗み取り”ます。また、変更先のインデクスにすでに別の要素が入っている場合、その要素に対する参照を放棄します。

```
void PyList_SET_ITEM(PyObject *list, int 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。この関数は、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。注意: この関数は *item* への参照を“盗み取り”ます。また、PyList_SetItem() と違って、要素の置き換えが生じても置き換えられるオブジェクトへの参照を放棄しません; その結果、*list* 中の位置 *i* で参照されていたオブジェクトがメモリリークを引き起こします。

```
int PyList_Insert(PyObject *list, int index, PyObject *item)
```

要素 *item* をインデクス *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、例外を送出します。*list.insert(index, item)* に類似した機能です。

```
int PyList_Append(PyObject *list, PyObject *item)
```

オブジェクト *item* を *list* の末尾に追加します。成功すると 0 を返します; 失敗すると -1 を返し、例外をセットします。*list.append(item)* に類似した機能です。

```
PyObject* PyList_GetSlice(PyObject *list, int low, int high)
```

戻り値: *New reference*.
list 内の、*low* から *high* の間のオブジェクトからなるリストを返します。失敗すると NULL を返し、例外をセットします。*list[low:high]* に類似した機能です。

```
int PyList_SetSlice(PyObject *list, int low, int high, PyObject *itemlist)
```

list 内の、*low* から *high* の間のオブジェクトを、*itemlist* の内容にします。*list[low:high] = itemlist* と類似の機能です。*itemlist* は NULL でもよく、空リストの代入 (指定スライスの削除) になります。成功した場合には 0 を、失敗した場合には -1 を返します。

```
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。success, -1 on failure. '*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)
```

戻り値: *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 `types.DictType` および `types.DictionaryType` として公開されています。

int PyDict_Check(PyObject *p)

p が辞書型オブジェクトか、辞書型のサブタイプのインスタンスである場合に真を返します。2.2 で変更された仕様: サブタイプを引数にとれるようになりました

PyObject* PyDict_New()

戻り値: *New reference*.

新たな空の辞書を返します。失敗すると NULL を返します。

PyObject* PyDictProxy_New(PyObject *dict)

戻り値: *New reference*.

ある辞書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。2.2 で追加された仕様です。

void PyDict_Clear(PyObject *p)

現在辞書に入っている全てのキーと値のペアを除去して空にします。

PyObject* PyDict_Copy(PyObject *p)

戻り値: *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。1.6 で追加された仕様です。

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, 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)

戻り値: *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, char *key)

戻り値: *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)

戻り値: *New reference*.

辞書オブジェクトのメソッド `item()` のように、辞書内の全ての要素対が入った `PyListObject` を返します。(`items()` については *Python* ライブラリリファレンス を参照してください。)

PyObject* PyDict_Keys(PyObject *p)

戻り値: *New reference*.

辞書オブジェクトのメソッド `keys()` のように、辞書内の全てのキーが入った `PyListObject` を返します。(`keys()` については *Python* ライブラリリファレンス を参照してください。)

PyObject* PyDict_Values(PyObject *p)

戻り値: *New reference*.

辞書オブジェクトのメソッド `values()` のように、辞書内の全ての値が入った `PyListObject` を返します。(`values()` については *Python* ライブラリリファレンス を参照してください。)

`int PyDict_Size(PyObject *p)`

辞書内の要素の数を返します。辞書に対して `'len(p)'` を実行するのと同じです。

`int PyDict_Next(PyObject *p, int *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 `int` 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。

以下に例を示します:

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

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* 取り出した値で何らかの処理を行う... */
    ...
}
```

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

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

マップ型オブジェクト b の全ての要素にわたって、反復的にキー/値のペアを辞書 a に追加します。 b は辞書か、 `PyMapping_Keys()` または `PyObject_GetItem()` をサポートする何らかのオブジェクトにできます。 $override$ が真ならば、 a のキーと一致するキーが b にある際に、既存のペアを置き換えます。それ以外の場合は、 b のキーに一致するキーが a にないときのみ追加を行います。成功した場合には 0 を返し、例外が送出された場合には -1 を返します。 2.2 で追加された仕様です。

`int PyDict_Update(PyObject *a, PyObject *b)`

C で表せば `PyDict_Merge(a, b, 1)` と同じ、Python で表せば `a.update(b)` と同じです。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。 2.2 で追加された仕様です。

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

$seq2$ 内のキー/値ペアを使って、辞書 a の内容を更新したり統合したりします。 $seq2$ は、キー/値のペアとみなせる長さ 2 の反復可能オブジェクト (iterable object) を生成する反復可能オブジェクトでなければなりません。重複するキーが存在する場合、 $override$ が真ならば先に出現したキーを使い、そうでない場合は後に出現したキーを使います。成功した場合には 0 を返し、例外が送出された場合には -1 を返します。

(戻り値以外は) 等価な Python コードを書くと、以下のようになります:

```
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value
```

2.2 で追加された仕様です。

7.5 その他のオブジェクト

7.5.1 ファイルオブジェクト (file object)

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 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 が PyFileObject のサブタイプのときに真を返します。2.2 で変更された仕様: 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。2.2 で追加された仕様です。

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

戻り値: *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*))

戻り値: *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 NULL を返します。

FILE* PyFile_AsFile(PyFileObject *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。

PyObject* PyFile_GetLine(PyObject *p, int n)

戻り値: *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)`

戻り値: *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, int n)`

`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_Encoding(PyFileObject *p, 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ド方式を `enc` にします。成功すると 1 を、失敗すると 0 を返します。2.3 で追加された仕様です。

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

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, PyFileObject *p, int flags)`

オブジェクト `obj` をファイルオブジェクト `p` に書き込みます。 `flag` 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, PyFileObject *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。

7.5.2 インスタンスオブジェクト (instance object)

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(PyObject *obj)`

`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)`

戻り値: *New reference*.

特定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実引数およびキーワードパラメタとして使われます。

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

戻り値: *New reference*.

特定クラスの新たなインスタンスを生成します。 `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。

7.5.3 メソッドオブジェクト (method object)

メソッドオブジェクトを操作する上で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 のメソッドオブジェクト型を表現します。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。パラメタは NULL にできません。

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

戻り値: *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)

戻り値: *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)

戻り値: *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)

戻り値: *Borrowed reference.*

meth に関連付けられている関数オブジェクトを返します。

PyObject* **PyMethod_GET_FUNCTION**(PyObject *meth)

戻り値: *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)

戻り値: *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)

戻り値: *Borrowed reference.*

PyMethod_Self() のマクロ版で、エラーチェックを行いません。

7.5.4 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。

PyTypeObject **PyModule_Type**

この PyTypeObject のインスタンスは Python のモジュールオブジェクト型を表現します。このオブジェクトは、Python プログラムには types.ModuleType として公開されています。

int **PyModule_Check**(PyObject *p)

o 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。2.2

で変更された仕様: 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. 2.2 で追加された仕様です。

`PyObject* PyModule_New(char *name)`

戻り値: *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)`

戻り値: *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, 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。エラーのときには `-1` を、成功したときには `0` を返します。2.0 で追加された仕様です。

`int PyModule_AddIntConstant(PyObject *module, char *name, int value)`

`module` に整数定数を `name` として追加します。この便宜関数はモジュールの初期化関数から利用されています。エラーのときには `-1` を、成功したときには `0` を返します。2.0 で追加された仕様です。

`int PyModule_AddStringConstant(PyObject *module, char *name, 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。2.0 で追加された仕様です。

7.5.5 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。一つ目は配列イテレータで、`__getitem__()` メソッドをサポートする任意の配列を取り扱います。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、配列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、組み込み配列型に対して 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。2.2 で追加された仕様です。

`int PySeqIter_Check(op)`

`PySeqIter_Type` の型が `op` のときに真を返します。2.2 で追加された仕様です。

PyObject* **PySeqIter_New**(PyObject *seq)

戻り値: *New reference*.

一般的な配列オブジェクト *seq* を扱うイテレータを返します。反復処理は、配列が添字指定操作の際に `IndexError` を返したときに終了します。2.2 で追加された仕様です。

PyTypeObject **PyCallIter_Type**

`PyCallIter_New()` や、組み込み関数 `iter()` の 2 引数形式が返すイテレータオブジェクトの型オブジェクトです。 `iter()` built-in function. 2.2 で追加された仕様です。

int **PyCallIter_Check**(op)

`PyCallIter_Type` の型が *op* のときに真を返します。2.2 で追加された仕様です。

PyObject* **PyCallIter_New**(PyObject *callable, PyObject *sentinel)

戻り値: *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。2.2 で追加された仕様です。

7.5.6 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。2.2 で追加された仕様です。

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

戻り値: *New reference*.

2.2 で追加された仕様です。

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

戻り値: *New reference*.

2.2 で追加された仕様です。

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

戻り値: *New reference*.

2.2 で追加された仕様です。

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

戻り値: *New reference*.

2.2 で追加された仕様です。

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

2.3 で追加された仕様です。

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。2.2 で追加された仕様です。

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

戻り値: *New reference*.

2.2 で追加された仕様です。

7.5.7 スライスオブジェクト (slice object)

PyTypeObject **PySlice_Type**

スライスオブジェクトの型オブジェクトです。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)

戻り値: *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, int length, int *start, int *stop, int *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。このとき配列の長さを *length* と仮定します。length よりも大きなインデクスになるとエラーとして扱います。

成功のときには 0 を、エラーのときには例外をセットせずに -1 を返します (ただし、指定インデクスのいずれかが None ではなく、かつ整数に変換できなかった場合を除きます。この場合、-1 を返して例外をセットします)。

おそらく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。

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

PySlice_GetIndices の置き換えとして使える関数です。

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。このとき配列の長さを *length* と仮定します。スライスの長さを *slicelength* に記憶します。境界をはみだしたインデクスは、通常のスライスを扱うのと同じ一貫したやり方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。

2.3 で追加された仕様です。

7.5.8 弱参照オブジェクト (weak reference object)

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。2.2 で追加された仕様です。

int **PyWeakref_CheckRef**(ob)

ob が参照オブジェクトの場合に真を返します。2.2 で追加された仕様です。

int **PyWeakref_CheckProxy**(ob)

ob がプロキシオブジェクトの場合に真を返します。2.2 で追加された仕様です。

PyObject* **PyWeakref_NewRef**(PyObject *ob, PyObject *callback)

戻り値: *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* がガベー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。 2.2 で追加された仕様です。

`PyObject*` `PyWeakref_NewProxy(PyObject*ob, PyObject*callback)`

戻り値: *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* がガベー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。 2.2 で追加された仕様です。

`PyObject*` `PyWeakref_GetObject(PyObject*ref)`

戻り値: *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、 None を返します。 2.2 で追加された仕様です。

`PyObject*` `PyWeakref_GET_OBJECT(PyObject*ref)`

戻り値: *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。 2.2 で追加された仕様です。

7.5.9 C オブジェクト (CObject)

このオブジェクトの使用法に関する情報は、*Python* インタプリタの拡張と埋め込み 1.12 節、“Providing a C API for an Extension Module,” を参照してください。

`PyCObject`

この `PyObject` のサブタイプは不透明型値 (opaque value) を表現します。C 拡張モジュールが Python コードから不透明型値を (`void*` ポインタで) 他の C コードに渡す必要があるときに便利です。正規の `import` 機構を使って動的にロードされるモジュール内で定義されている C API にアクセスするために、あるモジュール内で定義されている C 関数ポインタを別のモジュールでも利用できるようにするためによく使われます。

`int` `PyCObject_Check(PyObject*p)`

引数が `PyCObject` の場合に真を返します。

`PyObject*` `PyCObject_FromVoidPtr(void*obj, void(*destr)(void*))`

戻り値: *New reference.*

`void*obj` から `PyCObject` を生成します。関数 *destr* が NULL でない場合、オブジェクトを再利用する際に呼び出します。

`PyObject*` `PyCObject_FromVoidPtrAndDesc(void*obj, void*desc, void(*destr)(void*, void*))`

戻り値: *New reference.*

`void*obj` から `PyCObject` を生成します。関数 *destr* が NULL でない場合、オブジェクトを再利用する際に呼び出します。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)`

PyObject 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* PyObject_GetDesc(PyObject* self)
```

PyObject 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。

7.5.10 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。

PyObject

セルオブジェクトに使われる C 構造体です。

PyTypeObject PyObject 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)
```

戻り値: *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)
```

戻り値: *New reference*.

cell の内容を返します。

```
PyObject* PyCell_GET(PyObject *cell)
```

戻り値: *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。

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

`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()`、`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__` および `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_SetArgv()` 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。

`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まれます) 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。

`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 `__builtin__`、`__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` および `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 ライブラリの同じ 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グローバルインタプリタロック (global interpreter lock) が保持されていなければならず、関数が処理を戻した際にも保持されたままになります; しかし、他の Python/C API 関数とは違い、関数から戻ったときの現在のスレッド状態が関数に入るときと同じとは限らないので注意してください)。

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 を行うと、拡張モジュールの `initmodule` は再度呼び出されます。

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なので、インタプリタ間の絶縁性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せてしまいます。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ために、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进行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 (XXX この問題は修正が難しいバグで、将来のリリースで解決される予定です)

`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グローバルインタプリタロックを保持しておかねばならず、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。

`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ねばなりません。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ず、その内容はプログラムの実行中に変更してはなりません。Python インタプリタ内のコードで、この記憶領域の内容を変更するものは一切ありません。

`char* Py_GetProgramName()`

`Py_SetProgramName()` で設定されたプログラム名が、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。UNIX でも有用です。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 のみで有用です。

背景: プラットフォーム依存のファイル (実行形式や共有ライブラリ) が、別個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。

概して、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ることがあります。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はずで

`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。UNIX のみで有用です。

`char* Py_GetPath()`

デフォルト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デリミタ文字で分割された一連のディレクトリ名からなります。デリミタ文字は UNIX では `':'`、Windows では `','`、Macintosh では `'\n'` (ASCII 改行文字) 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 コードからはリスト `sys.path` として利用できます。このリストは、値を修正して将来モジュールをロードする際に使う検索パスを変更できます。

`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、

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

ような形式の文字列です。

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、メジャーバージョンとマイナーバージョン、そしてそれを分割しているピリオドです。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。Macintosh では `'mac'` 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、例えば `'Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam'` 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。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、各括弧で囲った文字列を返します。例えば:

```
"[GCC 2.7.2.2]"
```

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 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"
```

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 Python コードからは `sys.version` の一部として取り出せます。

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

`argc` および `argv` に基づいて `sys.argv` を設定します。このパラメタはプログラムの `main()` に渡したパラメタに似ていますが、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、致命的エラー条件を `Py_FatalError()` でシグナルします。

8.1 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スレッド安全 (thread safe) 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、グローバルインタプリタロック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。マルチスレッドの Python プログラムをサポートするため、インタプリタは定期的に — デフォルトの設定ではバイトコード 100 命令ごとに (この値は `sys.setcheckinterval()` で変更できます) — ロックを解放したり獲得したりします。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何らかの予約情報を持っておかねばなりません — このため、Python は `PyThreadState` と呼ばれるデータ構造を用います。これは Python 1.5 からの仕様です; 以前のバージョンでは、スレッド状態はグローバル変数内に保存されていて、スレッドを切り替えると問題を起こすことがありました。特に、アプリケーションが `sys.exc_info()` を使って、現在のスレッドで送出された最新の例外にアクセスする際の例外処理は、いまやスレッド安全になりました。とはいえ、グローバル変数はまだ一つだけ残っています: それは現在の `PyThreadState` 構造体を指すポインタです。ほとんどのスレッドパッケージが“スレッドごとのグローバルデータ”を保存する手段を持っている一方で、Python の内部的なプラットフォーム非依存のスレッド抽象層はこれをサポートしていません。従って、現在のスレッド状態を明示的に操作するようにはしなければなりません。

ほとんどのケースで、このような操作は十分簡単にできます。グローバルインタプリタロックを操作数ほとんどのコードは、以下のような単純な構造を持ちます:

スレッド状態をローカル変数に保存する。
インタプリタロックを解放する。
... ブロックが起きるような何らかの I/O 操作...
インタプリタロックを獲得する。
ローカル変数からスレッド状態を回復する。

このやりかたは非常に一般的なもので、作業を単純にするために二つのマクロが用意されています:

```
Py_BEGIN_ALLOW_THREADS  
... ブロックが起きるような何らかの I/O 操作...  
Py_END_ALLOW_THREADS
```

`Py_BEGIN_ALLOW_THREADS` マクロは新たなブロック文を開始し、隠しローカル変数を宣言します;
`Py_END_ALLOW_THREADS` はブロック文を終了します。これらの二つのマクロを使うもうひとつの利点
は、Python をスレッドサポートなしでコンパイルしたとき、マクロの内容、すなわちスレッド状態の回避
とロック操作が空になるという点です。

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();  
... ブロックが起きるような何らかの I/O 操作...  
PyEval_RestoreThread(_save);
```

より低水準のプリミティブを使うと、以下のようにしてほぼ同じ効果を得られます:

```
PyThreadState *_save;  
  
_save = PyThreadState_Swap(NULL);  
PyEval_ReleaseLock();  
... ブロックが起きるような何らかの I/O 操作...  
PyEval_AcquireLock();  
PyThreadState_Swap(_save);
```

上の二つには微妙な違いがあります; とりわけ、`PyEval_RestoreThread()` はグローバル変数 `errno`
の値を保存しておいて元に戻す点が異なります。というのは、ロック操作が `errno` に何もしないという
保証がないからです。また、スレッドサポートが無効化されている場合、`PyEval_SaveThread()` およ
び `PyEval_RestoreThread()` はロックを操作しません; この場合、`PyEval_ReleaseLock()` およ
び `PyEval_AcquireLock()` は利用できません。この仕様は、スレッドサポートを無効化してコンパイルさ
れているインタプリタが、スレッドサポートが有効化された状態でコンパイルされている動的ロード拡張
モジュールをロードできるようにするためのものです。

グローバルインタプリタロックは、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。

なぜここまで詳しく説明しようとするかわかりでしょうか? それは、C でスレッドを生成した場合、そ
のスレッドにはグローバルインタプリタロックがなく、スレッド状態データ構造体もないからです。この
ようなスレッドが Python/C API を利用するには、まずスレッド状態データ構造体を生成し、次にロックを

獲得し、そしてスレッド状態ポインタを保存するといったように、自分自身をブートストラップして生成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をメモリ解放しなければなりません。

スレッドデータ構造体を生成するには、インタプリタ状態データ構造体を指定する必要があります。インタプリタ状態データ構造体は、インタプリタ内の全てのスレッド間で共有されているグローバルなデータ、例えばモジュール管理データ (codesys.modules) を保持しています。必要に応じて、新たなインタプリタ状態データ構造体を作成するなり、Python メインスレッドが使っているインタプリタ状態データ構造体を共有するなりできます (後者のデータにアクセスするためには、スレッド状態データ構造体を獲得して、その `interp` メンバにアクセスしなければなりません; この処理は、Python が作成したスレッドから行うか、Python を初期化した後で主スレッドから行わねばなりません)。

インタプリタオブジェクトにアクセスできるという仮定の下では、C のスレッドから Python を呼び出す際の典型的な常套句は以下ようになります。

```
PyThreadState *tstate;
PyObject *result;

/* interp is your reference to an interpreter object. */
tstate = PyThreadState_New(interp);
PyEval_AcquireThread(tstate);

/* Perform Python actions here. */
result = CallSomeFunction();
/* evaluate result */

/* Release the thread. No Python API allowed beyond this point. */
PyEval_ReleaseThread(tstate);

/* You can either delete the thread state, or save it
   until you need it the next time. */
PyThreadState_Delete(tstate);
```

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 (state) 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き、初期状態では何も共有されていません。グローバルインタプリタロックもまた、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。データメンバ `PyInterpreterState *interp` だけが公開されていて、スレッドのインタプリタ状態を指すポインタになっています。

void PyEval_InitThreads()

グローバルインタプリタロックを初期化し、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。

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。主スレッドしか存在しないのであれば、ロック操作は必要ありません。これはよくある状況です (ほとんどの Python プログラムはスレッドを使いません)、ロック操作はインタプリタをごくわずかに低速化します。従って、初期状態ではロックは生成されません。ロックを使わない状況は、すでに

ロックを獲得している状況と同じです: 単一のスレッドしかなければ、オブジェクトへのアクセスは全て安全です。従って、この関数がロックを初期化すると、同時にロックを獲得するようになっていきます。Python の `thread` モジュールは、新たなスレッドを作成する前に、ロックが存在するか、あるいはまだ作成されていないかを調べ、`PyEval_InitThreads()` を呼び出します。この関数から処理が戻った場合、ロックの作成と獲得が保証されています。

どのスレッドが現在グローバルインタプリタロックを (存在する場合) 持っているか分からない時にこの関数を使う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を無効化すると利用できません。

`void PyEval_AcquireLock()`

グローバルインタプリタロックを獲得します。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。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。

`void PyEval_ReleaseLock()`

グローバルインタプリタロックを解放します。ロックは前もって作成されていなければなりません。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。

`void PyEval_AcquireThread(PyThreadState *tstate)`

グローバルインタプリタロックを獲得し、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。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。

`void PyEval_ReleaseThread(PyThreadState *tstate)`

現在のスレッド状態をリセットして `NULL` にし、グローバルインタプリタロックを解放します。ロックはあらかじめ作成されていなければならず、かつ現在のスレッドが保持していなければなりません。`tstate` は `NULL` であってはなりません、その値が現在のスレッド状態を表現しているかどうかを調べるためにだけ使われます — もしそうでなければ、致命的エラーが報告されます。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。

`PyThreadState* PyEval_SaveThread()`

(インタプリタロックが生成されていて、スレッドサポートが有効の場合) インタプリタロックを解放して、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。

`void PyEval_RestoreThread(PyThreadState *tstate)`

(インタプリタロックが生成されていて、スレッドサポートが有効の場合) インタプリタロックを獲得して、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。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を無効化すると利用できません。)

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きません。

`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きません。

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きません。

Py_UNBLOCK_THREADS

このマクロを展開すると `'_save = PyEval_SaveThread();'` になります: 閉じ波括弧のない `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きません。

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、呼び出すのはインタプリタロック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。インタプリタロック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。インタプリタロック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。インタプリタロック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。インタプリタロック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。インタプリタロック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。インタプリタロックを保持していなければなりません。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。

PyThreadState* PyThreadState_Get()

現在のスレッド状態を返します。インタプリタロック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` であってはなりません。インタプリタロックを保持していなければなりません。

PyObject* PyThreadState_GetDict()

戻り値: *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ねばなりません。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、例外はまったく送らさ

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ねばなりません。2.3 で変更された仕様: 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(long id, PyObject *exc)`

スレッド内で非同期的に例外を送出します。`id` 引数はターゲットとなるスレッドのスレッド id です; `exc` は送出する例外オブジェクトです。この関数は `exc` に対する参照を一切盗み取りません。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ねばなりません。グローバルインタプリタロックを保持した状態で呼び出さなければなりません。変更を受けたスレッド状態の数を返します; 1 よりも大きな数を返した場合、何らかのトラブルに巻き込まれていることになり、`exc` を NULL にして再度呼び出すことで効果を打ち消さねばなりません。この関数自体は例外を送出しません。2.3 で追加された仕様です。

8.2 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とレース関数をスレッドごとにインストールでき、トレース関数に報告される基本イベント (basic event) は以前のバージョンにおいて Python レベルのトレース関数で報告されていたものと同じです。

`int (*Py_tracefunc)(PyObject *obj, PyFrameObject *frame, int what, PyObject *arg)`

`PyEval_SetProfile()` および `PyEval_SetTrace()` を使って登録できるトレース関数の形式です。最初のパラメタはオブジェクトで、登録関数に `obj` として渡されます。`frame` はイベントが属している実行フレームオブジェクトで、`what` は定数 `PyTrace_CALL`, `PyTrace_EXCEPT`, `PyTrace_LINE` あるいは `PyTrace_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 NULL です。
<code>PyTrace_EXCEPT</code>	<code>sys.exc_info()</code> の返す例外情報です。
<code>PyTrace_LINE</code>	常に NULL です。
<code>PyTrace_RETURN</code>	呼び出し側に返される予定の値です。

`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注意してください。

`int PyTrace_EXCEPT`

例外が送出された際の `Py_tracefunc` の `what` の値です。現在実行されているフレームで例外がセットされ、何らかのバイトコードが処理された後に、`what` にこの値がセットされた状態でコールバック関数が呼び出されます。

この結果、例外の伝播によって Python が呼び出しスタックを逆戻りする際に、各フレームから処理が戻るごとにコールバック関数が呼び出されます。トレース関数だけがこれらのイベントを受け取ります; プロファイラはこの種のイベントを必要としません。

`int PyTrace_LINE`

行番号イベントを報告するときに(プロファイル関数ではなく)トレース関数の *what* パラメタとして渡す値です。

`int PyTrace_RETURN`

関数呼び出しが例外の伝播なしに返るときに `Py_tracefunc` 関数 *what* パラメタとして渡す値です。

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

プロファイル関数を *func* に設定します。*obj* パラメタは関数の第一パラメタとして渡され、何らかの Python オブジェクトかまたは `NULL` になります。プロファイル関数がスレッド状態を維持する必要があるなら、各々のスレッドに異なる *obj* を使うことで、状態を記憶しておく便利でスレッドセーフな場所を提供できます。プロファイル関数は、モニタされているイベントのうち、行番号イベントを除く全てのイベントに対して呼び出されます。

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

トレース関数を *func* にセットします。`PyEval_SetProfile()` に似ていますが、トレース関数は行番号イベントを受け取る点が違います。

8.3 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。2.2 で追加された仕様です。

`PyInterpreterState* PyInterpreterState_Next(PyInterpreterState *interp)`

インタプリタ状態オブジェクトからなるリストのうち、*interp* の次にあるものを返します。2.2 で追加された仕様です。

`PyThreadState * PyInterpreterState_ThreadHead(PyInterpreterState *interp)`

インタプリタ *interp* に関連付けられているスレッドからなるリストのうち、先頭にある `PyThreadState` オブジェクトを返します。2.2 で追加された仕様です。

`PyThreadState* PyThreadState_Next(PyThreadState *tstate)`

tstate と同じ `PyInterpreterState` オブジェクトに属しているスレッド状態オブジェクトのうち、*tstate* の次にあるものを返します。2.2 で追加された仕様です。

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*) が確実に行います。Python メモリマネージャには、共有 (sharing)、セグメント分割 (segmentation)、事前割り当て (preallocation)、キャッシュ化 (caching) といった、様々な動的記憶管理の側面を扱うために、個別のコンポーネントがあります。

最低水準層では、素のメモリ操作関数 (raw memory allocator) がオペレーティングシステムのメモリ管理機構とやりとりして、プライベートヒープ内に Python 関連の全てのデータを記憶するのに十分な空きがあるかどうか確認します。素のメモリ操作関数の上には、いくつかのオブジェクト固有のメモリ操作関数があります。これらは同じヒープを操作し、各オブジェクト型固有の事情に合ったメモリ管理ポリシーを実装しています。例えば、整数オブジェクトは文字列やタプル、辞書とは違ったやり方でヒープ内で管理されます。というのも、整数には値を記憶する上で特別な要件があり、速度/容量のトレードオフが存在するからです。このように、Python メモリマネージャは作業のいくつかをオブジェクト固有のメモリ操作関数に委譲しますが、これらの関数がプライベートヒープからはみ出してメモリ管理を行わないようにしています。

重要なのは、たとえユーザがいつもヒープ内のメモリブロックを指すようなオブジェクトポインタを操作しているとしても、Python 用ヒープの管理はインタプリタ自体が行うもので、ユーザがそれを制御する余地はないと理解することです。Python オブジェクトや内部使用されるバッファを入れるためのヒープ空間のメモリ確保は、必要に応じて、Python メモリマネージャがこのドキュメント内で列挙している Python/C API 関数群を介して行います。

メモリ管理の崩壊を避けるため、拡張モジュールの作者は決して Python オブジェクトを C ライブラリが公開している関数: `malloc()`、`calloc()`、`realloc()` および `free()` で操作しようとはなりません。こうした関数を使うと、C のメモリ操作関数と Python メモリマネージャとの間で関数呼び出しが交錯します。C のメモリ操作関数と Python メモリマネージャは異なるアルゴリズムで実装されていて、異なるヒープを操作するため、呼び出しの交錯は致命的な結果を招きます。とはいえ、個別の目的のためなら、C ライブラリのメモリ操作関数を使って安全にメモリを確保したり解放したりできます。例えば、以下がそのような例です:

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

if (buf == NULL)
    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; それ以外の場合で、 n がゼロに等しければ、メモリブロックはサイズ変更されませんが、解放されず、非 `NULL` のポインタを返します。 p の値を `NULL` にしないのなら、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。

```
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、すでに `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_t n)
```

`PyMem_Malloc()` と同じですが、 $(n * \text{sizeof}(TYPE))$ 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_t n)
```

`PyMem_Realloc()` と同じですが、 $(n * \text{sizeof}(TYPE))$ 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。

```
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 例

最初に述べた関数セットを使って、9.1 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。またこれらの関数にはマクロ `PyObject_NEW()`、`PyObject_NEW_VAR()`、および `PyObject_DEL()` が対応します。

これらの関数については、次章のCによる新しいオブジェクト型の定義や実装に関する記述の中で説明します。

オブジェクト実装サポート (object implementation support)

この章では、新しいオブジェクト型 (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)`

戻り値: *New reference.*

`PyObject*` `_PyObject_NewVar(PyTypeObject *type, int size)`

戻り値: *New reference.*

`void` `_PyObject_Del(PyObject *op)`

`PyObject*` `PyObject_Init(PyObject *op, PyTypeObject *type)`

戻り値: *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, int size)`

戻り値: *Borrowed reference.*

`PyObject_Init()` の全ての処理を行い、可変サイズオブジェクトの場合には長さ情報も初期化します。

`TYPE*` `PyObject_New(TYPE, PyTypeObject *type)`

C 構造体型 `TYPE` と Python 型オブジェクト `type` を使って新たな Python オブジェクトをメモリ確保します。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タは 1 になります。メモリ確保のサイズは型オブジェクトの `tp_basicsize` で決定します。

`TYPE*` `PyObject_NewVar(TYPE, PyTypeObject *type, int size)`

C 構造体型 `TYPE` と Python 型オブジェクト `type` を使って新たな Python オブジェクトをメモリ確保します。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、`vartype` の `tp_itemsize` フィールドで指定されているサイズ中の `size` フィールドを収容できます。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に対するアロケーション操作を一つに

して埋め込むと、アロケーション回数が減り、メモリ管理の処理効率が向上します。

`void PyObject_Del(PyObject *op)`

`PyObject_New()` や `PyObject_NewVar()` で確保されたメモリを解放します。この関数は、通常オブジェクトの型に指定されている `tp_dealloc` ハンドラ内で呼び出します。この関数を呼び出した後では、オブジェクトのメモリ領域はもはや有効な Python オブジェクトを表現してはいないので、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。

`TYPE* PyObject_NEW(TYPE, PyTypeObject *type)`

安全性と引き換えにパフォーマンスを高めるための `PyObject_New()` のマクロ版です。 `type` が `NULL` かどうかチェックしません。

`TYPE* PyObject_NEW_VAR(TYPE, PyTypeObject *type, int size)`

安全性と引き換えにパフォーマンスを高めるための `PyObject_NewVar()` のマクロ版です。 `type` が `NULL` かどうかチェックしません。

`void PyObject_DEL(PyObject *op)`

`PyObject_Del()` のマクロ版です。

`PyObject* Py_InitModule(char *name, PyMethodDef *methods)`

戻り値: *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。2.3 で変更された仕様: 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)`

戻り値: *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。2.3 で変更された仕様: 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, int apiver)`

戻り値: *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が。) `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。

注意: 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; 本当にこの関数を使いたいときにだけ利用してください。2.3 で変更された仕様: 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

`DL_IMPORT`

`PyObject _Py_NoneStruct`

Python からは `None` に見えるオブジェクトです。この値へのアクセスは、このオブジェクトへのポインタを評価する `Py_None` マクロを使わねばなりません。

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。

`PyObject` および `PyVarObject` の定義には以下のマクロが使われています:

`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:

```
int 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;
int 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
int 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。

`PyObject_HEAD_INIT`

`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 データ型	意味
<code>ml_name</code>	<code>char *</code>	メソッド名
<code>ml_meth</code>	<code>PyCFunction</code>	C 実装へのポインタ
<code>ml_flags</code>	<code>int</code>	呼び出しをどのように行うかを示すフラグビット
<code>ml_doc</code>	<code>char *</code>	<code>docstring</code> 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` では最初のパラメータが `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` を組み合わせられます (ただし、`METH_KEYWORDS` 単体の指定を行っても `METH_VARARGS | METH_KEYWORDS` と同じなので注意してください)。呼び出し規約フラグは束縛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、`Py_InitModule4()` に与えることになる値が入ります (`NULL` にすると `Py_InitModule()` が使われます)。第二のパラメータ (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。パラメータは通常、`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` は三つのパラメータ:`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、パラメータは `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。オブジェクトのメソッドに使った場合、第一のパラメータは `self` になり、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、第二のパラメータは `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` パラメータと単一の引数を表現する `PyObject*` パラメータを伴います。

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、複数個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、複数個の引数の場合と、

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う。

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。2.3 で追加された仕様です。

METH_STATIC

メソッドの最初の引数は、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。2.3 で追加された仕様です。

`PyObject* Py_FindMethod(PyMethodDef table[], PyObject *ob, char *name)`

戻り値: *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

新形式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一つでしょう。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。とはいえ、型オブジェクトはオブジェクトがどのように振舞うかを定める基盤ですから、インタプリタ自体や新たな型を定義する拡張モジュールでは非常に重要な存在です。

型オブジェクトは標準の型 (*standard type*) に比べるとかなり大きな構造体です。その理由は、型オブジェクトがある型の様々な機能を実現する小さな機能単位を実装した C 関数へのポインタが大部分を占めるような多数の値を保持しているからです。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`, `intintobjargproc`, `objobjargproc`, `destructor`, `freefunc`, `printfunc`, `getattrfunc`, `getattrofunc`, `setattrfunc`, `setattrofunc`, `cmpfunc`, `reprfunc`, `hashfunc`

`PyTypeObject` の構造体定義は `'Include/object.h'` で見つけられるはずですが、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 */

    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;
getattrofunc tp_getattro;
setattrofunc tp_setattro;

/* Functions to access object as input/output buffer */
PyBufferProcs *tp_as_buffer;

/* Flags to define presence of optional/expanded features */
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;
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) 動的な型に使います。 `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 フィールドがなければなりません。

PyObject* _ob_next

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。

int ob_refcnt

かた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。

PyTypeObject* 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。

int 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* 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入っているはずですが。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、こうした型オブジェクトは pickle 化できないことになります。

サブタイプはこのフィールドを継承しません。

```
int tp_basicsize
```

```
int 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 フィールドがなくてはならず、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)。

バイト整列 (alignment) に関する注釈: 変数の各要素を配置する際に特定のバイト整列が必要となる場合、tp_basicsize の値に気をつけなければなりません。一例: 例えばある型が double の配列を実装しているとします。tp_itemsize は sizeof(double) です。(double のバイト整列条件に従って) tp_basicsize が sizeof(double) 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 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、Py_DECREF() や Py_XDECREF() マクロで、操作後の参照カウントがゼロになった際に呼び出されます。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(かならず最後に行う操作として)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と同じファミリ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_VarNew()` 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 tp_print`

オプションのフィールドです。ポインタで、インスタンスの出力 (`print`) を行う関数を指します。

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` 引数は出力す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ならば、インスタンスは `tp_repr` と同じ書式で出力されます。この関数は、操作中にエラーが生じた場合、`-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 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。

このフィールドは撤廃されています。このフィールドを定義する場合、`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 `PyObject_GetAttrString()` と同じです。

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` を一緒に継承します。

`setattrfunc tp_setattr`

オプションのフィールドです。ポインタで、`set-attribute-string` を行う関数を指します。

このフィールドは撤廃されています。このフィールドを定義する場合、`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 `PyObject_SetAttrString()` と同じです。

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` を一緒に継承します。

`cmpfunc 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 `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ねばなりません。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ねばなりません。

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;
```

```
XXX
```

```
PySequenceMethods *tp_as_sequence;
```

```
XXX
```

```
PyMappingMethods *tp_as_mapping;
```

```
XXX
```

hashfunc `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` の三つを一緒に継承します。

ternaryfunc `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 `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 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こと往々にして便利です。

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` を一緒に継承します。

`setattrofunc tp_setattro`

オプションのフィールドです。ポインタで、`set-attribute` を行う関数を指します。

シグネチャは `PyObject_SetAttr()` と同じです。対する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往々にして便利です。

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` を一緒に継承します。

`PyBufferProcs* 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” (10.7 節) 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、フィールド内に入っているフィールドは個別に継承されます。

`long 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: `extention structure`) 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る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であれば、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。

以下の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 フィールドの値にできます。PyType_HasFeature() マクロは型とフラグ値、tp および f をとり、tp->tp_flags & f が非ゼロかどうか調べます。

Py_TPFLAGS_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す。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 を持つこととなります; また、PySequenceMethods 構造体はフィールド sq_inplace_concat および sq_inplace_repeat を持つこととなります。

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 INCREf し、インスタンスを解放するたびに DECREf 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 INCREf および DECREf 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 XXX 節のガベージコレクションに関する説明中にあります。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* 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 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。ガベージコレクションに関する詳細は 10.9 にあります。

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 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。ガベージコレクションに関する詳細は [10.9](#) 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 tp_richcompare`

オプションのフィールドです。ポインタで、拡張比較関数 (rich comparison function) を指します。

シグネチャは `PyObject_RichCompare()` と同じです。この関数は、要求された比較演算が真を返す場合に 1 を、偽を返す場合に 0 を返さなければなりません。比較中にエラーが発生した場合には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_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` の三つを一緒に継承します。

`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
<code>Py_LT</code>	<
<code>Py_LE</code>	<=
<code>Py_EQ</code>	==
<code>Py_NE</code>	!=
<code>Py_GT</code>	>
<code>Py_GE</code>	>=

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 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 ヘッダがある場合には無視します); このオフセット値は `PyObject_ClearWeakRefs()` および `PyWeakref_*()` 関数が利用します。インスタンス構造体には、NULL に初期化された `PyObject*` 型のフィールドが入っていないなければなりません。

このフィールドを `tp_weaklist` と混同しないようにしてください; `tp_weaklist` は型オブジェクト自体の弱参照リストの先頭です。

サブタイプはこのフィールドを継承しますが、以下の規則があるので読んでください。サブタイプはこのオフセット値をオーバーライドできます; 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

`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__` 宣言中に `__weakref__` という名前のスロットが入っている場合、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__` 宣言に `__weakref__` という名前のスロットが入っていない場合、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_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。

`getiterfunc tp_iter`

An optional pointer to a function that returns an iterator for the object. Its presence normally signals that the instances of this type are iterable (although sequences may be iterable without this function, and classic instances always have this function, even if they don't define an `__iter__()` method).

This function has the same signature as `PyObject_GetIter()`.

サブタイプはこのフィールドを継承します。

`iternextfunc tp_iternext`

オプションのフィールドです。ポインタで、イテレータにおいて次の要素を返すか、イテレータの要素がなくなると `StopIteration` を送出する関数を指します。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、旧形式のインスタンスでは、たとえ `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` 関数も定義していなければならないが、`tp_iter` 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* 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エントリが型辞書 (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* 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エントリが型辞書 (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* 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` で終端された静的配列を指します。

配列の各要素ごとに、`getset` デスクリプタの入ったエントリが型辞書 (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)。

Docs for PyGetSetDef (XXX belong elsewhere):

```
typedef PyObject *(*getter)(PyObject *, void *);
typedef int (*setter)(PyObject *, PyObject *, void *);

typedef struct PyGetSetDef {
    char *name; /* 属性名 */
    getter get; /* 属性の get を行う C 関数 */
    setter set; /* 属性の set を行う C 関数 */
    char *doc; /* オプションの docstring */
    void *closure; /* オプションの get/set 関数用追加データ */
} PyGetSetDef;
```

PyObject* **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* **tp_dict**

型の辞書は PyType_Ready() によってこのフィールドに収められます。

このフィールドは通常、PyType_Ready() を呼び出す前に NULL に初期化しておかねばなりません; あるいは、型の初期属性の入った辞書で初期化しておいてもかまいません。PyType_Ready() が型をひとたび初期化すると、型の新たな属性をこの辞書に追加できるのは、属性が (__add__) のような) オーバロード用演算でないときだけです。

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 **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。

XXX blah, blah.

サブタイプはこのフィールドを継承します。

descrsetfunc **tp_descr_set**

オプションのフィールドです。ポインタで、"デスクリプタ set" 関数を指します。

XXX blah, blah.

サブタイプはこのフィールドを継承します。

long **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 と混同しないでください; tp_dict は型オブジェクト自体の属性のための辞書です。

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の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ので注意してください。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。

`tp_dictoffset` が負の場合、インスタンスにおける実際の辞書のオフセットは以下のようにして計算されます:

```
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。)

サブタイプはこのフィールドを継承しますが、以下の規則があるので読んでください。サブタイプはこのオフセット値をオーバーライドできます; 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。辞書へのオフセット常に `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。

`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。)

`initproc 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

```
int tp_init(PyObject *self, PyObject *args, PyObject *kwargs)
```

です。

`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 `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

```
PyObject *tp_alloc(PyTypeObject *self, int nitems)
```

です。

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を持ち、適切にバイト整列され、ゼロで初期化され、ただし `ob_refcnt` を 1 にセットされ、`ob_type` が型引数 (type argument) にセットされているようなメモリブロック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*)` の倍数で丸め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 `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

```
PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwargs)
```

です。

引数 `subtype` 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固定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等価というわけではないので注意してください; `tp_new` 関数を呼び出すときに使う型 (と無関係ではない) 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、初期化で本当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。経験則からいうと、変更不能な型の場合、初期化は全て `tp_new` で行い、変更可能な型の場合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 `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`:

```
void tp_free(void *)
```

になっています。

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に変更されました。

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 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、オブジェクトがガベージとして収集可能かどうかを知る必要があります。これを知るには、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、静的なメモリ確保と動的なメモリ確保が混じっているインスタンスを持つような型や、静的にメモリ確保されたインスタンスは収集できません。こうした型では、このフィールドに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収集可能の場合には 1 を、収集不能の場合には 0 を返さねばなりません。シグネチャは

```
int tp_is_gc(PyObject *self)
```

です。

(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* 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* tp_mro`

基底クラス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 (Method Resolution Order) の順に並んでいます。

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* 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* 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* 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。

残りのフィールドは、機能テスト用のマクロである `COUNT_ALLOCS` が定義されている場合のみ利用でき、内部で使用するためだけのものです。これらのフィールドについて記述するのは単に完全性のためです。サブタイプはこれらのフィールドを継承しません。

int tp_allocs

メモリ確保の回数です。

int tp_frees

メモリ解放の回数です。

int 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* tp_next

tp_allocs フィールドが非ゼロの、(リンクリストの) 次の型オブジェクトを指すポインタです。

10.4 マップ型オブジェクト構造体 (mapping object structure)

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。

10.5 数値オブジェクト構造体 (number object structure)

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。

10.6 配列オブジェクト構造体 (sequence object structure)

PySequenceMethods

拡張型で配列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。

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、PyTypeObject 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。

注意: PyTypeObject 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。

最初のスロットは `bf_getreadbuffer` で、`getreadbufferproc` 型です。このスロットが `NULL` の場合、オブジェクトは内部データの読み出しをサポートしません。そのような仕様には意味がないので、実装を行う側はこのスロットに値を埋めるはずですが、呼び出し側では非 `NULL` の値かどうかきちんと調べておくべきです。

次のスロットは `bf_getwritebuffer` で、`getwritebufferproc` 型です。オブジェクトが返すバッファに対して書き込みを許可しない場合はこのスロットを `NULL` にできます。

第三のスロットは `bf_getsegcount` で、`getsegcountproc` 型です。このスロットは `NULL` であってはならず、オブジェクトにいくつセグメントが入っているかを呼び出し側に教えるために使われます。 `PyString_Type` や `PyBuffer_Type` オブジェクトのような単純なオブジェクトには単一のセグメントしか入っていません。

最後のスロットは `bf_getcharbuffer` で、`getcharbufferproc` です。オブジェクトの `PyType-Object` 構造体における `tp_flags` フィールドに、`Py_TPFLAGS_HAVE_GETCHARBUFFER` ビットフラグがセットされている場合にのみ、このスロットが存在することになります。このスロットの使用に先立って、呼び出し側は `PyType_HasFeature()` を使ってスロットが存在するか調べねばなりません。スロットが存在する場合、値は `NULL` のときもあり、`NULL` はオブジェクトの内容を 8 ビット文字列として利用できないことを示します。このスロットに入る関数も、オブジェクトの内容を 8 ビット文字列に変換できない場合に例外を送出することがあります。例えば、オブジェクトが浮動小数点数を保持するように設定されたアレイの場合、呼び出し側が `bf_getcharbuffer` を使って 8 ビット文字列としてデータを取り出そうとすると例外を送出することができます。この、内部バッファを“テキスト”として取り出すという概念は、本質的にはバイナリで、文字ベースの内容を持ったオブジェクト間の区別に使われます。

注意: 現在のポリシーでは、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。

```
int (*getreadbufferproc) (PyObject *self, int 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返します。この関数は例外を送出してはよく、送出する場合には `-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、バッファメモリのサイズを返し、 `*ptrptr` をバッファメモリを指すポインタ値にセットします。

```
int (*getwritebufferproc) (PyObject *self, int 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。

```
int (*getsegcountproc) (PyObject *self, int *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。

```
int (*getcharbufferproc) (PyObject *self, int segment, const char **ptrptr)
セグメント segment のメモリバッファを ptrptr に入れ、メモリバッファのサイズを返します。*ptrptr がメモリバッファを表す値になるようにセットされます。
```

10.8 イテレータプロトコルをサポートする

10.9 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。

ここで説明しているインタフェースの使い方を示した例は、Python の拡張と埋め込みの“循環参照の収集をサポートする”にあります。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* フラグがなくてはならず、*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を変更可能なオブジェクトにするなら、*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_VarNew()* で確保しなければなりません。
2. 一度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、*PyObject_GC_Track()* を呼び出さねばなりません。

*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*, *int size*)

PyObject_NewVar() に似ていますが、*Py_TPFLAGS_HAVE_GC* のセットされたコンテナオブジェクト用です。

*PyVarObject ** **PyObject_GC_Resize**(*PyVarObject *op*, *int*)

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。

void **PyObject_GC_Track**(*PyObject *op*)

ガベージコレクタが追跡しているコンテナオブジェクトの集合にオブジェクト *op* を追加します。ガベージコレクタの動作する回数は予測不能なので、追加対象にするオブジェクトは追跡されている間ずっと有効なオブジェクトでなければなりません。この関数は、通常コンストラクタの最後付近で、*tp_traverse* ハンドラ以降の全てのフィールドが有効な値になった時点で呼び出さねばなりません。

`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()` を呼び出さねばなりません。
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(PyObject *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(PyObject *op)`

ガベージコレクタが追跡しているコンテナオブジェクトの集合からオブジェクト *op* を除去します。`PyObject_GC_Track()` を呼び出して、除去したオブジェクトを再度追跡対象セットに追加できるので注意してください。メモリ解放関数 (`dealloc`, `tp_dealloc` ハンドラ) は、`tp_traverse` ハンドラが使用しているフィールドのいずれかが無効化されるよりも以前にオブジェクトに対して呼び出されていなければなりません。

`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` ハンドラは以下の型を持つ関数を引数の一つとしてとります:

`int (*visitproc)(PyObject *object, void *arg)`

`tp_traverse` ハンドラに渡すビジタ関数 (visitor function) の型です。この関数は追跡すべきオブジェクトを *object* に、`tp_traverse` ハンドラの第三引数を *arg* にして呼び出されます。

`tp_traverse` ハンドラは以下の型で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* 関数を呼び出さねばなりません。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* 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_clear` ハンドラは `inquiry` 型にするか、オブジェクトが変更不能の場合には `NULL` にしなければなりません。NULL if the object is immutable.

`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ぬ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。

バグ報告

Python is a mature programming language which has established a reputation for stability. In order to maintain this reputation, the developers would like to know of any deficiencies you find in Python or its documentation.

Before submitting a report, you will be required to log into SourceForge; this will make it possible for the developers to contact you for additional information if needed. It is not possible to submit a bug report anonymously.

All bug reports should be submitted via the Python Bug Tracker on SourceForge (http://sourceforge.net/bugs/?group_id=5470). The bug tracker offers a Web form which allows pertinent information to be entered and submitted to the developers.

The first step in filing a report is to determine whether the problem has already been reported. The advantage in doing so, aside from saving the developers time, is that you learn what has been done to fix it; it may be that the problem has already been fixed for the next release, or additional information is needed (in which case you are welcome to provide it if you can!). To do this, search the bug database using the search box near the left side of the page.

If the problem you're reporting is not already in the bug tracker, go back to the Python Bug Tracker (http://sourceforge.net/bugs/?group_id=5470). Select the "Submit a Bug" link at the top of the page to open the bug reporting form.

The submission form has a number of fields. The only fields that are required are the "Summary" and "Details" fields. For the summary, enter a *very* short description of the problem; less than ten words is good. In the Details field, describe the problem in detail, including what you expected to happen and what did happen. Be sure to include the version of Python you used, whether any extension modules were involved, and what hardware and software platform you were using (including version information as appropriate).

The only other field that you may want to set is the "Category" field, which allows you to place the bug report into a broad category (such as "Documentation" or "Library").

Each bug report will be assigned to a developer who will determine what needs to be done to correct the problem. You will receive an update each time action is taken on the bug.

参考資料:

How to Report Bugs Effectively

(<http://www-mice.cs.ucl.ac.uk/multimedia/software/documentation/ReportingBugs.html>)

Article which goes into some detail about how to create a useful bug report. This describes what kind of information is useful and why it is useful.

Bug Writing Guidelines

(<http://www.mozilla.org/quality/bug-writing-guidelines.html>)

Information about writing a good bug report. Some of this is specific to the Mozilla project, but describes general good practices.

歴史とライセンス

B.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see <http://www.cwi.nl/>) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see <http://www.cnri.reston.va.us/>) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see <http://www.zope.com/>). In 2001, the Python Software Foundation (PSF, see <http://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.

All Python releases are Open Source (see <http://www.opensource.org/> for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 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
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.3	2004	PSF	yes
2.3.2	2.3.4	2004	PSF	yes

注意: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

B.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.3.4

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.3.4 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 2.3.4 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2003 Python Software Foundation; All Rights Reserved" are retained in Python 2.3.4 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 2.3.4 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 2.3.4.
4. PSF is making Python 2.3.4 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 2.3.4 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.3.4 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.3.4, 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 2.3.4, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an “AS IS” basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
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.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (“CNRI”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI’s License Agreement and CNRI’s notice of copyright, i.e., “Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved” are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI’s License Agreement, Licensee may substitute the following text (omitting the quotes): “Python 1.6.1 is made available subject to the terms and conditions in CNRI’s License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: <http://hdl.handle.net/1895.22/1013>.”
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an “AS IS” basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
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.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

日本語訳について

C.1 このドキュメントについて

この文書は、Python ドキュメント翻訳プロジェクトによる Python C/API Reference Release 2.3 の日本語訳版です。日本語訳に対する質問や提案などがありましたら、Python ドキュメント翻訳プロジェクトのメーリングリスト

<http://www.python.jp/mailman/listinfo/python-doc-jp>

または、プロジェクトのバグ管理ページ

http://sourceforge.jp/tracker/?atid=116\&group_id=11\&func=browse

までご報告ください。

C.2 翻訳者一覧 (敬称略)

2.3.3 和訳 Minami Masanori, Yasushi Masuda (May 16, 2004)

2.3.4 差分 Yasushi Masuda (September 20, 2004)

索引

Symbols

`_PyImport_FindExtension()`, 23
`_PyImport_Fini()`, 23
`_PyImport_FixupExtension()`, 23
`_PyImport_Init()`, 23
`_PyObject_Del()`, 95
`_PyObject_GC_TRACK()`, 117
`_PyObject_GC_UNTRACK()`, 117
`_PyObject_New()`, 95
`_PyObject_NewVar()`, 95
`_PyString_Resize()`, 55
`_PyTuple_Resize()`, 67
`_Py_NoneStruct`, 96
`_Py_c_diff()`, 52
`_Py_c_neg()`, 52
`_Py_c_pow()`, 52
`_Py_c_prod()`, 52
`_Py_c_quot()`, 52
`_Py_c_sum()`, 52
`__all__` (package variable), 22
`__builtin__` (組み込みモジュール), 9, 79, 80
`__dict__` (module attribute), 74
`__doc__` (module attribute), 74
`__file__` (module attribute), 74
`__import__` () (組み込み関数), 22
`__main__` (組み込みモジュール), 9, 79, 80
`__name__` (module attribute), 74
`_ob_next` (PyObject のメンバ), 101
`_ob_prev` (PyObject のメンバ), 101

A

`abort()`, 22
`abs()` (組み込み関数), 38
`apply()` (組み込み関数), 35
`argv` (in module sys), 83

B

buffer
 object, 64
 オブジェクト, 64
buffer interface, 64
BufferType (in module types), 65

C

`calloc()`, 91
`classmethod()` (組み込み関数), 99
cleanup functions, 22
`close()` (in module os), 80
`cmp()` (組み込み関数), 34
CObject
 object, 77
 オブジェクト, 77
`coerce()` (組み込み関数), 40
`compile()` (組み込み関数), 23
complex number
 object, 51
 オブジェクト, 51
`copyright` (in module sys), 82

D

dictionary
 object, 68
 オブジェクト, 68
DictionaryType (in module types), 68
DictType (in module types), 68
`divmod()` (組み込み関数), 38

E

environment variables
 PATH, 9
 PYTHONDUMPREFS, 101
 PYTHONHOME, 9
 PYTHONPATH, 9

- exec_prefix, 1, 2
 - prefix, 1, 2
- EOFError (built-in exception), 72
- errno, 84
- exc_info() (in module sys), 7, 83
- exc_traceback (in module sys), 7, 15
- exc_type (in module sys), 7, 15
- exc_value (in module sys), 7, 15
- Exception (built-in exception), 20
- exceptions (組み込みモジュール), 9
- exec_prefix, 1, 2
- executable (in module sys), 82
- exit(), 22

F

- file
 - object, 71
 - オブジェクト, 71
- FileType (in module types), 71
- float() (組み込み関数), 40
- floating point
 - object, 51
 - オブジェクト, 51
- FloatType (in modules types), 51
- fopen(), 71
- free(), 91
- freeze utility, 24

G

- global interpreter lock, 83

H

- hash() (組み込み関数), 36, 104

I

- ihooks (標準モジュール), 23
- incr_item(), 7, 8
- instance
 - object, 72
 - オブジェクト, 72
- int() (組み込み関数), 40
- getcharbufferproc (C の型), 116
- getreadbufferproc (C の型), 115
- getsegcountproc (C の型), 115
- getwritebufferproc (C の型), 115
- inquiry (C の型), 117
- Py_tracefunc (C の型), 88
- traverseproc (C の型), 117

- visitproc (C の型), 117

- integer
 - object, 48
 - オブジェクト, 48

- interpreter lock, 83

- IntType (in modules types), 48

K

- KeyboardInterrupt (built-in exception), 18, 19

L

- len() (組み込み関数), 36, 41, 43, 67, 70

- list
 - object, 67
 - オブジェクト, 67

- ListType (in module types), 67

- lock, interpreter, 83

- long() (組み込み関数), 40

- long integer
 - object, 49
 - オブジェクト, 49

- LONG_MAX, 49, 50

- LongType (in modules types), 49

M

- main(), 81, 83

- malloc(), 91

- mapping
 - object, 68
 - オブジェクト, 68

- METH_CLASS (のデータ), 99

- METH_KEYWORDS (のデータ), 98

- METH_NOARGS (のデータ), 98

- METH_O (のデータ), 98

- METH_OLDARGS (のデータ), 98

- METH_STATIC (のデータ), 99

- METH_VARARGS (のデータ), 98

- method
 - object, 73
 - オブジェクト, 73

- MethodType (in module types), 73

- module
 - object, 73
 - オブジェクト, 73
 - search path, 9, 79, 82

- modules (in module sys), 22, 79

ModuleType (in module types), 73

N

None

- object, 48
- オブジェクト, 48

numeric

- object, 48
- オブジェクト, 48

O

ob_refcnt (PyObject のメンバ), 101

ob_size (PyVarObject のメンバ), 101

ob_type (PyObject のメンバ), 101

object

- buffer, 64
- CObject, 77
- complex number, 51
- dictionary, 68
- file, 71
- floating point, 51
- instance, 72
- integer, 48
- list, 67
- long integer, 49
- mapping, 68
- method, 73
- module, 73
- None, 48
- numeric, 48
- sequence, 53
- string, 53
- tuple, 66
- type, 2, 47

オブジェクト

- buffer, 64
- CObject, 77
- complex number, 51
- dictionary, 68
- file, 71
- floating point, 51
- instance, 72
- integer, 48
- list, 67
- long integer, 49
- mapping, 68
- method, 73

module, 73

None, 48

numeric, 48

sequence, 53

string, 53

tuple, 66

type, 2, 47

OverflowError (built-in exception), 50

P

package variable

__all__, 22

PATH, 9

path

module search, 9, 79, 82

path (in module sys), 9, 79, 82

platform (in module sys), 82

pow() (組み込み関数), 38, 39

prefix, 1, 2

Py_AtExit(), 22

Py_BEGIN_ALLOW_THREADS, 84

Py_BEGIN_ALLOW_THREADS (マクロ), 86

Py_BLOCK_THREADS (マクロ), 87

Py_BuildValue(), 31

Py_CompileString(), 12

Py_CompileString(), 12

Py_complex (C の型), 52

Py_DECREF(), 13

Py_DECREF(), 2

Py_END_ALLOW_THREADS, 84

Py_END_ALLOW_THREADS (マクロ), 86

Py_END_OF_BUFFER, 65

Py_EndInterpreter(), 80

Py_eval_input, 12

Py_Exit(), 22

Py_FatalError(), 22

Py_FatalError(), 83

Py_FdIsInteractive(), 21

Py_file_input, 12

Py_Finalize(), 79

Py_Finalize(), 22, 79, 80

Py_FindMethod(), 99

Py_GetBuildInfo(), 82

Py_GetCompiler(), 82

Py_GetCopyright(), 82

Py_GetExecPrefix(), 81

Py_GetExecPrefix(), 9

[Py_GetPath\(\)](#), 82
[Py_GetPath\(\)](#), 9, 81
[Py_GetPlatform\(\)](#), 82
[Py_GetPrefix\(\)](#), 81
[Py_GetPrefix\(\)](#), 9
[Py_GetProgramFullPath\(\)](#), 81
[Py_GetProgramFullPath\(\)](#), 9
[Py_GetProgramName\(\)](#), 81
[Py_GetVersion\(\)](#), 82
[Py_INCREF\(\)](#), 13
[Py_INCREF\(\)](#), 2
[Py_Initialize\(\)](#), 79
[Py_Initialize\(\)](#), 9, 80, 81, 85
[Py_InitModule\(\)](#), 96
[Py_InitModule3\(\)](#), 96
[Py_InitModule4\(\)](#), 96
[Py_IsInitialized\(\)](#), 79
[Py_IsInitialized\(\)](#), 9
[Py_Main\(\)](#), 11
[Py_NewInterpreter\(\)](#), 80
[Py_None](#), 48
[Py_PRINT_RAW](#), 72
[Py_SetProgramName\(\)](#), 80
[Py_SetProgramName\(\)](#), 9, 79, 81, 82
[Py_single_input](#), 12
[Py_TPFLAGS_BASETYPE \(のデータ\)](#), 107
[Py_TPFLAGS_CHECKTYPES \(のデータ\)](#), 106
[Py_TPFLAGS_DEFAULT \(のデータ\)](#), 107
[Py_TPFLAGS_GC \(のデータ\)](#), 106
[Py_TPFLAGS_HAVE_CLASS \(のデータ\)](#), 106
[Py_TPFLAGS_HAVE_GC \(のデータ\)](#), 107, 116
[Py_TPFLAGS_HAVE_GETCHARBUFFER \(のデータ\)](#), 106, 115
[Py_TPFLAGS_HAVE_INPLACEOPS \(のデータ\)](#), 106
[Py_TPFLAGS_HAVE_ITER \(のデータ\)](#), 106
[Py_TPFLAGS_HAVE_RICHCOMPARE \(のデータ\)](#), 106
[Py_TPFLAGS_HAVE_SEQUENCE_IN \(のデータ\)](#), 106
[Py_TPFLAGS_HAVE_WEAKREFS \(のデータ\)](#), 106
[Py_TPFLAGS_HEAPTYPE \(のデータ\)](#), 107
[Py_TPFLAGS_READY \(のデータ\)](#), 107
[Py_TPFLAGS_READYING \(のデータ\)](#), 107
[Py_UNBLOCK_THREADS \(マクロ\)](#), 87
[Py_UNICODE \(C の型\)](#), 56
[Py_UNICODE_ISALNUM\(\)](#), 57
[Py_UNICODE_ISALPHA\(\)](#), 57
[Py_UNICODE_ISDECIMAL\(\)](#), 57
[Py_UNICODE_ISDIGIT\(\)](#), 57
[Py_UNICODE_ISLINEBREAK\(\)](#), 57
[Py_UNICODE_ISLOWER\(\)](#), 57
[Py_UNICODE_ISNUMERIC\(\)](#), 57
[Py_UNICODE_ISSPACE\(\)](#), 57
[Py_UNICODE_ISTITLE\(\)](#), 57
[Py_UNICODE_ISUPPER\(\)](#), 57
[Py_UNICODE_TODECIMAL\(\)](#), 57
[Py_UNICODE_TODIGIT\(\)](#), 57
[Py_UNICODE_TOLOWER\(\)](#), 57
[Py_UNICODE_TONUMERIC\(\)](#), 57
[Py_UNICODE_TOTITLE\(\)](#), 57
[Py_UNICODE_TOUPPER\(\)](#), 57
[Py_XDECREF\(\)](#), 13
[Py_XDECREF\(\)](#), 8
[Py_XINCREASE\(\)](#), 13
[PyArg_Parse\(\)](#), 30
[PyArg_ParseTuple\(\)](#), 30
[PyArg_ParseTupleAndKeywords\(\)](#), 30
[PyArg_UnpackTuple\(\)](#), 30
[PyBuffer_Check\(\)](#), 65
[PyBuffer_FromMemory\(\)](#), 65
[PyBuffer_FromObject\(\)](#), 65
[PyBuffer_FromReadWriteMemory\(\)](#), 66
[PyBuffer_FromReadWriteObject\(\)](#), 65
[PyBuffer_New\(\)](#), 66
[PyBuffer_Type](#), 65
[PyBufferObject \(C の型\)](#), 65
[PyBufferProcs](#), 65
[PyBufferProcs \(C の型\)](#), 115
[PyCallable_Check\(\)](#), 35
[PyCallIter_Check\(\)](#), 75
[PyCallIter_New\(\)](#), 75
[PyCallIter_Type](#), 75
[PyCell_Check\(\)](#), 78
[PyCell_GET\(\)](#), 78
[PyCell_Get\(\)](#), 78
[PyCell_New\(\)](#), 78
[PyCell_SET\(\)](#), 78
[PyCell_Set\(\)](#), 78
[PyCell_Type](#), 78
[PyCellObject \(C の型\)](#), 78
[PyCFunction \(C の型\)](#), 97
[PyCObject \(C の型\)](#), 77

PyCObject_AsVoidPtr(), 77
 PyCObject_Check(), 77
 PyCObject_FromVoidPtr(), 77
 PyCObject_FromVoidPtrAndDesc(), 77
 PyCObject_GetDesc(), 78
 PyComplex_AsCComplex(), 53
 PyComplex_Check(), 52
 PyComplex_CheckExact(), 52
 PyComplex_FromCComplex(), 52
 PyComplex_FromDoubles(), 52
 PyComplex_ImagAsDouble(), 53
 PyComplex_RealAsDouble(), 52
 PyComplex_Type, 52
 PyComplexObject (C の型), 52
 PyDescr_IsData(), 75
 PyDescr_NewClassMethod(), 75
 PyDescr_NewGetSet(), 75
 PyDescr_NewMember(), 75
 PyDescr_NewMethod(), 75
 PyDescr_NewWrapper(), 75
 PyDict_Check(), 68
 PyDict_Clear(), 69
 PyDict_Copy(), 69
 PyDict_DelItem(), 69
 PyDict_DelItemString(), 69
 PyDict_GetItem(), 69
 PyDict_GetItemString(), 69
 PyDict_Items(), 69
 PyDict_Keys(), 69
 PyDict_Merge(), 70
 PyDict_MergeFromSeq2(), 70
 PyDict_New(), 68
 PyDict_Next(), 70
 PyDict_SetItem(), 69
 PyDict_SetItemString(), 69
 PyDict_Size(), 69
 PyDict_Type, 68
 PyDict_Update(), 70
 PyDict_Values(), 69
 PyDictObject (C の型), 68
 PyDictProxy_New(), 69
 PyErr_BadArgument(), 17
 PyErr_BadInternalCall(), 18
 PyErr_CheckSignals(), 18
 PyErr_Clear(), 16
 PyErr_Clear(), 6, 8
 PyErr_ExceptionMatches(), 15
 PyErr_ExceptionMatches(), 8
 PyErr_Fetch(), 16
 PyErr_Format(), 16
 PyErr_GivenExceptionMatches(), 15
 PyErr_NewException(), 19
 PyErr_NoMemory(), 17
 PyErr_NormalizeException(), 16
 PyErr_Occurred(), 15
 PyErr_Occurred(), 6
 PyErr_Print(), 15
 PyErr_Restore(), 16
 PyErr_SetExcFromWindowsErr(), 17
 PyErr_SetExcFromWindowsErrWithFilename(), 18
 PyErr_SetFromErrno(), 17
 PyErr_SetFromErrnoWithFilename(), 17
 PyErr_SetFromWindowsErr(), 17
 PyErr_SetFromWindowsErrWithFilename(), 18
 PyErr_SetInterrupt(), 19
 PyErr_SetNone(), 17
 PyErr_SetObject(), 16
 PyErr_SetString(), 16
 PyErr_SetString(), 6
 PyErr_Warn(), 18
 PyErr_WarnExplicit(), 18
 PyErr_WriteUnraisable(), 19
 PyEval_AcquireLock(), 86
 PyEval_AcquireLock(), 79, 84
 PyEval_AcquireThread(), 86
 PyEval_InitThreads(), 85
 PyEval_InitThreads(), 79
 PyEval_ReleaseLock(), 86
 PyEval_ReleaseLock(), 79, 84, 85
 PyEval_ReleaseThread(), 86
 PyEval_ReleaseThread(), 85
 PyEval_RestoreThread(), 86
 PyEval_RestoreThread(), 84
 PyEval_SaveThread(), 86
 PyEval_SaveThread(), 84
 PyEval_SetProfile(), 89
 PyEval_SetTrace(), 89
 PyFile_AsFile(), 71
 PyFile_Check(), 71
 PyFile_CheckExact(), 71
 PyFile_Encoding(), 72
 PyFile_FromFile(), 71

PyFile_FromString(), 71
 PyFile_GetLine(), 71
 PyFile_Name(), 72
 PyFile_SetBufSize(), 72
 PyFile_SoftSpace(), 72
 PyFile_Type, 71
 PyFile_WriteObject(), 72
 PyFile_WriteString(), 72
 PyFileObject (C の型), 71
 PyFloat_AS_DOUBLE(), 51
 PyFloat_AsDouble(), 51
 PyFloat_Check(), 51
 PyFloat_CheckExact(), 51
 PyFloat_FromDouble(), 51
 PyFloat_FromString(), 51
 PyFloat_Type, 51
 PyFloatObject (C の型), 51
 PyImport_AddModule(), 23
 PyImport_AppendInittab(), 24
 PyImport_Cleanup(), 23
 PyImport_ExecCodeModule(), 23
 PyImport_ExtendInittab(), 24
 PyImport_FrozenModules, 24
 PyImport_GetMagicNumber(), 23
 PyImport_GetModuleDict(), 23
 PyImport_Import(), 22
 PyImport_ImportFrozenModule(), 23
 PyImport_ImportModule(), 22
 PyImport_ImportModuleEx(), 22
 PyImport_ReloadModule(), 23
 PyInstance_Check(), 72
 PyInstance_New(), 72
 PyInstance_NewRaw(), 72
 PyInstance_Type, 72
 PyInt_AS_LONG(), 49
 PyInt_AsLong(), 49
 PyInt_AsUnsignedLongLongMask(), 49
 PyInt_AsUnsignedLongMask(), 49
 PyInt_Check(), 48
 PyInt_CheckExact(), 48
 PyInt_FromLong(), 49
 PyInt_FromString(), 48
 PyInt_GetMax(), 49
 PyInt_Type, 48
 PyInterpreterState (C の型), 85
 PyInterpreterState_Clear(), 87
 PyInterpreterState_Delete(), 87
 PyInterpreterState_Head(), 89
 PyInterpreterState_New(), 87
 PyInterpreterState_Next(), 89
 PyInterpreterState_ThreadHead(), 89
 PyIntObject (C の型), 48
 PyIter_Check(), 44
 PyIter_Next(), 44
 PyList_Append(), 68
 PyList_AsTuple(), 68
 PyList_Check(), 67
 PyList_CheckExact(), 67
 PyList_GET_ITEM(), 67
 PyList_GET_SIZE(), 67
 PyList_GetItem(), 67
 PyList_GetItem(), 5
 PyList_GetSlice(), 68
 PyList_Insert(), 68
 PyList_New(), 67
 PyList_Reverse(), 68
 PyList_SET_ITEM(), 68
 PyList_SetItem(), 67
 PyList_SetItem(), 3
 PyList_SetSlice(), 68
 PyList_Size(), 67
 PyList_Sort(), 68
 PyList_Type, 67
 PyListObject (C の型), 67
 PyLong_AsDouble(), 51
 PyLong_AsLong(), 50
 PyLong_AsLongLong(), 50
 PyLong_AsUnsignedLong(), 50
 PyLong_AsUnsignedLongLong(), 50
 PyLong_AsUnsignedLongLongMask(), 51
 PyLong_AsUnsignedLongMask(), 51
 PyLong_AsVoidPtr(), 51
 PyLong_Check(), 49
 PyLong_CheckExact(), 49
 PyLong_FromDouble(), 50
 PyLong_FromLong(), 49
 PyLong_FromLongLong(), 50
 PyLong_FromString(), 50
 PyLong_FromUnicode(), 50
 PyLong_FromUnsignedLong(), 49
 PyLong_FromUnsignedLongLong(), 50
 PyLong_FromVoidPtr(), 50
 PyLong_Type, 49
 PyLongObject (C の型), 49

PyMapping_Check(), 43
 PyMapping_DelItem(), 43
 PyMapping_DelItemString(), 43
 PyMapping_GetItemString(), 43
 PyMapping_HasKey(), 43
 PyMapping_HasKeyString(), 43
 PyMapping_Items(), 43
 PyMapping_Keys(), 43
 PyMapping_Length(), 43
 PyMapping_SetItemString(), 43
 PyMapping_Values(), 43
 PyMappingMethods (C の型), 114
 PyMarshal_ReadLastObjectFromFile(), 25
 PyMarshal_ReadLongFromFile(), 25
 PyMarshal_ReadObjectFromFile(), 25
 PyMarshal_ReadObjectFromString(), 25
 PyMarshal_ReadShortFromFile(), 25
 PyMarshal_WriteLongToFile(), 25
 PyMarshal_WriteObjectToFile(), 25
 PyMarshal_WriteObjectToString(), 25
 PyMem_Del(), 92
 PyMem_Free(), 92
 PyMem_Malloc(), 92
 PyMem_New(), 92
 PyMem_Realloc(), 92
 PyMem_Resize(), 92
 PyMethod_Check(), 73
 PyMethod_Class(), 73
 PyMethod_Function(), 73
 PyMethod_GET_CLASS(), 73
 PyMethod_GET_FUNCTION(), 73
 PyMethod_GET_SELF(), 73
 PyMethod_New(), 73
 PyMethod_Self(), 73
 PyMethod_Type, 73
 PyMethodDef (C の型), 98
 PyModule_AddIntConstant(), 74
 PyModule_AddObject(), 74
 PyModule_AddStringConstant(), 74
 PyModule_Check(), 73
 PyModule_CheckExact(), 74
 PyModule_GetDict(), 74
 PyModule_GetFilename(), 74
 PyModule_GetName(), 74
 PyModule_New(), 74
 PyModule_Type, 73
 PyNumber_Absolute(), 38
 PyNumber_Add(), 37
 PyNumber_And(), 38
 PyNumber_Check(), 37
 PyNumber_Coerce(), 40
 PyNumber_Divide(), 37
 PyNumber_Divmod(), 38
 PyNumber_Float(), 40
 PyNumber_FloorDivide(), 37
 PyNumber_InPlaceAdd(), 39
 PyNumber_InPlaceAnd(), 40
 PyNumber_InPlaceDivide(), 39
 PyNumber_InPlaceFloorDivide(), 39
 PyNumber_InPlaceLshift(), 40
 PyNumber_InPlaceMultiply(), 39
 PyNumber_InPlaceOr(), 40
 PyNumber_InPlacePower(), 39
 PyNumber_InPlaceRemainder(), 39
 PyNumber_InPlaceRshift(), 40
 PyNumber_InPlaceSubtract(), 39
 PyNumber_InPlaceTrueDivide(), 39
 PyNumber_InPlaceXor(), 40
 PyNumber_Int(), 40
 PyNumber_Invert(), 38
 PyNumber_Long(), 40
 PyNumber_Lshift(), 38
 PyNumber_Multiply(), 37
 PyNumber_Negative(), 38
 PyNumber_Or(), 39
 PyNumber_Positive(), 38
 PyNumber_Power(), 38
 PyNumber_Remainder(), 38
 PyNumber_Rshift(), 38
 PyNumber_Subtract(), 37
 PyNumber_TrueDivide(), 38
 PyNumber_Xor(), 39
 PyNumberMethods (C の型), 114
 PyObject (C の型), 97
 PyObject_AsCharBuffer(), 44
 PyObject_AsFileDescriptor(), 37
 PyObject_AsReadBuffer(), 44
 PyObject_AsWriteBuffer(), 44
 PyObject_Call(), 35
 PyObject_CallFunction(), 35
 PyObject_CallFunctionObjArgs(), 36
 PyObject_CallMethod(), 35
 PyObject_CallMethodObjArgs(), 36

PyObject_CallObject(), 35
 PyObject_CheckReadBuffer(), 44
 PyObject_Cmp(), 34
 PyObject_Compare(), 34
 PyObject_DEL(), 96
 PyObject_Del(), 96
 PyObject_DelAttr(), 34
 PyObject_DelAttrString(), 33
 PyObject_DelItem(), 37
 PyObject_Dir(), 37
 PyObject_GC_Del(), 117
 PyObject_GC_New(), 116
 PyObject_GC_NewVar(), 116
 PyObject_GC_Resize(), 116
 PyObject_GC_Track(), 116
 PyObject_GC_UnTrack(), 117
 PyObject_GetAttr(), 33
 PyObject_GetAttrString(), 33
 PyObject_GetItem(), 36
 PyObject_GetIter(), 37
 PyObject_HasAttr(), 33
 PyObject_HasAttrString(), 33
 PyObject_Hash(), 36
 PyObject_HEAD (マクロ), 97
 PyObject_HEAD_INIT (マクロ), 97
 PyObject_Init(), 95
 PyObject_InitVar(), 95
 PyObject_IsInstance(), 34
 PyObject_IsSubclass(), 35
 PyObject_IsTrue(), 36
 PyObject_Length(), 36
 PyObject_NEW(), 96
 PyObject_New(), 95
 PyObject_NEW_VAR(), 96
 PyObject_NewVar(), 95
 PyObject_Not(), 36
 PyObject_Print(), 33
 PyObject_Repr(), 34
 PyObject_RichCompare(), 34
 PyObject_RichCompareBool(), 34
 PyObject_SetAttr(), 33
 PyObject_SetAttrString(), 33
 PyObject_SetItem(), 36
 PyObject_Size(), 36
 PyObject_Str(), 34
 PyObject_Type(), 36
 PyObject_TypeCheck(), 36
 PyObject_Unicode(), 34
 PyObject_VAR_HEAD (マクロ), 97
 PyOS_AfterFork(), 21
 PyOS_CheckStack(), 21
 PyOS_GetLastModificationTime(), 21
 PyOS_getsig(), 21
 PyOS_setsig(), 21
 PyParser_SimpleParseFile(), 12
 PyParser_SimpleParseString(), 12
 PyProperty_Type, 75
 PyRun_AnyFile(), 11
 PyRun_File(), 12
 PyRun_InteractiveLoop(), 11
 PyRun_InteractiveOne(), 11
 PyRun_SimpleFile(), 11
 PyRun_SimpleString(), 11
 PyRun_String(), 12
 PySeqIter_Check(), 74
 PySeqIter_New(), 75
 PySeqIter_Type, 74
 PySequence_Check(), 41
 PySequence_Concat(), 41
 PySequence_Contains(), 42
 PySequence_Count(), 42
 PySequence_DelItem(), 41
 PySequence_DelSlice(), 41
 PySequence_Fast(), 42
 PySequence_Fast_GET_ITEM(), 42
 PySequence_Fast_GET_SIZE(), 42
 PySequence_GetItem(), 41
 PySequence_GetItem(), 5
 PySequence_GetSlice(), 41
 PySequence_Index(), 42
 PySequence_InPlaceConcat(), 41
 PySequence_InPlaceRepeat(), 41
 PySequence_ITEM(), 42
 PySequence_Length(), 41
 PySequence_List(), 42
 PySequence_Repeat(), 41
 PySequence_SetItem(), 41
 PySequence_SetSlice(), 41
 PySequence_Size(), 41
 PySequence_Tuple(), 41, 42
 PySequenceMethods (C の型), 114
 PySlice_Check(), 76
 PySlice_GetIndices(), 76
 PySlice_GetIndicesEx(), 76

PySlice_New(), 76
 PySlice_Type, 76
 PyString_AS_STRING(), 54
 PyString_AsDecodedObject(), 55
 PyString_AsEncodedObject(), 56
 PyString_AsString(), 54
 PyString_AsStringAndSize(), 54
 PyString_Check(), 53
 PyString_CheckExact(), 53
 PyString_Concat(), 54
 PyString_ConcatAndDel(), 55
 PyString_Decode(), 55
 PyString_Encode(), 55
 PyString_Format(), 55
 PyString_FromFormat(), 53
 PyString_FromFormatV(), 54
 PyString_FromString(), 53
 PyString_FromString(), 69
 PyString_FromStringAndSize(), 53
 PyString_GET_SIZE(), 54
 PyString_InternFromString(), 55
 PyString_InternInPlace(), 55
 PyString_Size(), 54
 PyString_Type, 53
 PyStringObject (C の型), 53
 PySys_SetArgv(), 83
 PySys_SetArgv(), 9, 79
 PYTHONDUMPPREFS, 101
 PYTHONHOME, 9
 PYTHONPATH, 9
 PyThreadState, 83
 PyThreadState (C の型), 85
 PyThreadState_Clear(), 87
 PyThreadState_Delete(), 87
 PyThreadState_Get(), 87
 PyThreadState_GetDict(), 87
 PyThreadState_New(), 87
 PyThreadState_Next(), 89
 PyThreadState_SetAsyncExc(), 88
 PyThreadState_Swap(), 87
 PyTrace_CALL, 88
 PyTrace_EXCEPT, 88
 PyTrace_LINE, 89
 PyTrace_RETURN, 89
 PyTuple_Check(), 66
 PyTuple_CheckExact(), 66
 PyTuple_GET_ITEM(), 66
 PyTuple_GET_SIZE(), 66
 PyTuple_GetItem(), 66
 PyTuple_GetSlice(), 66
 PyTuple_New(), 66
 PyTuple_SET_ITEM(), 67
 PyTuple_SetItem(), 66
 PyTuple_SetItem(), 3
 PyTuple_Size(), 66
 PyTuple_Type, 66
 PyTupleObject (C の型), 66
 PyType_Check(), 47
 PyType_CheckExact(), 47
 PyType_GenericAlloc(), 48
 PyType_GenericNew(), 48
 PyType_HasFeature(), 47
 PyType_HasFeature(), 115
 PyType_IS_GC(), 47
 PyType_IsSubtype(), 48
 PyType_Ready(), 48
 PyType_Type, 47
 PyTypeObject (C の型), 47
 PyUnicode_AS_DATA(), 56
 PyUnicode_AS_UNICODE(), 56
 PyUnicode_AsASCIIString(), 61
 PyUnicode_AsCharmapString(), 62
 PyUnicode_AsEncodedString(), 59
 PyUnicode_AsLatin1String(), 61
 PyUnicode_AsMBCSString(), 63
 PyUnicode_AsRawUnicodeEscapeString(),
 61
 PyUnicode_AsUnicode(), 58
 PyUnicode_AsUnicodeEscapeString(),
 61
 PyUnicode_AsUTF16String(), 60
 PyUnicode_AsUTF8String(), 59
 PyUnicode_AsWideChar(), 58
 PyUnicode_Check(), 56
 PyUnicode_CheckExact(), 56
 PyUnicode_Compare(), 64
 PyUnicode_Concat(), 63
 PyUnicode_Contains(), 64
 PyUnicode_Count(), 64
 PyUnicode_Decode(), 59
 PyUnicode_DecodeASCII(), 61
 PyUnicode_DecodeCharmap(), 62
 PyUnicode_DecodeLatin1(), 61
 PyUnicode_DecodeMBCS(), 63

PyUnicode_DecodeRawUnicodeEscape(),
 61
 PyUnicode_DecodeUnicodeEscape(), 60
 PyUnicode_DecodeUTF16(), 59
 PyUnicode_DecodeUTF8(), 59
 PyUnicode_Encode(), 59
 PyUnicode_EncodeASCII(), 61
 PyUnicode_EncodeCharmap(), 62
 PyUnicode_EncodeLatin1(), 61
 PyUnicode_EncodeMBCS(), 63
 PyUnicode_EncodeRawUnicodeEscape(),
 61
 PyUnicode_EncodeUnicodeEscape(), 60
 PyUnicode_EncodeUTF16(), 60
 PyUnicode_EncodeUTF8(), 59
 PyUnicode_Find(), 64
 PyUnicode_Format(), 64
 PyUnicode_FromEncodedObject(), 58
 PyUnicode_FromObject(), 58
 PyUnicode_FromUnicode(), 57
 PyUnicode_FromWideChar(), 58
 PyUnicode_GET_DATA_SIZE(), 56
 PyUnicode_GET_SIZE(), 56
 PyUnicode_GetSize(), 58
 PyUnicode_Join(), 64
 PyUnicode_Replace(), 64
 PyUnicode_Split(), 63
 PyUnicode_Splitlines(), 63
 PyUnicode_Tailmatch(), 64
 PyUnicode_Translate(), 63
 PyUnicode_TranslateCharmap(), 62
 PyUnicode_Type, 56
 PyUnicodeObject (C の型), 56
 PyVarObject (C の型), 97
 PyWeakref_Check(), 76
 PyWeakref_CheckProxy(), 76
 PyWeakref_CheckRef(), 76
 PyWeakref_GET_OBJECT(), 77
 PyWeakref_GetObject(), 77
 PyWeakref_NewProxy(), 77
 PyWeakref_NewRef(), 76
 PyWrapper_New(), 75

R

realloc(), 91
 reload() (組み込み関数), 23
 repr() (組み込み関数), 34, 104

rexec (標準モジュール), 23

S

search
 path, module, 9, 79, 82
 sequence
 object, 53
 オブジェクト, 53
 set_all(), 5
 setcheckinterval() (in module sys), 83
 setvbuf(), 72
 SIGINT, 18, 19
 signal (組み込みモジュール), 18
 SliceType (in module types), 76
 softspace (file attribute), 72
 staticmethod() (組み込み関数), 99
 stderr (in module sys), 80
 stdin (in module sys), 80
 stdout (in module sys), 80
 str() (組み込み関数), 34
 strerror(), 17
 string
 object, 53
 オブジェクト, 53
 StringType (in module types), 53
 _frozen (C の型), 24
 _inittab (C の型), 24
 sum_list(), 5
 sum_sequence(), 6, 7
 sys (組み込みモジュール), 9
 SystemError (built-in exception), 74

T

thread (組み込みモジュール), 86
 tp_alloc (PyTypeObject のメンバ), 112
 tp_allocs (PyTypeObject のメンバ), 114
 tp_as_buffer (PyTypeObject のメンバ), 105
 tp_base (PyTypeObject のメンバ), 110
 tp_bases (PyTypeObject のメンバ), 113
 tp_basicsize (PyTypeObject のメンバ), 102
 tp_cache (PyTypeObject のメンバ), 113
 tp_call (PyTypeObject のメンバ), 104
 tp_clear (PyTypeObject のメンバ), 108
 tp_compare (PyTypeObject のメンバ), 103
 tp_dealloc (PyTypeObject のメンバ), 102
 tp_descr_get (PyTypeObject のメンバ), 110
 tp_descr_set (PyTypeObject のメンバ), 110

tp_dict (PyTypeObject のメンバ), 110
tp_dictoffset (PyTypeObject のメンバ), 110
tp_doc (PyTypeObject のメンバ), 107
tp_flags (PyTypeObject のメンバ), 105
tp_free (PyTypeObject のメンバ), 112
tp_frees (PyTypeObject のメンバ), 114
tp_getattr (PyTypeObject のメンバ), 103
tp_getattro (PyTypeObject のメンバ), 105
tp_getset (PyTypeObject のメンバ), 109
tp_hash (PyTypeObject のメンバ), 104
tp_init (PyTypeObject のメンバ), 111
tp_is_gc (PyTypeObject のメンバ), 113
tp_itemsize (PyTypeObject のメンバ), 102
tp_iter (PyTypeObject のメンバ), 109
tp_iternext (PyTypeObject のメンバ), 109
tp_maxalloc (PyTypeObject のメンバ), 114
tp_members (PyTypeObject のメンバ), 109
tp_methods (PyTypeObject のメンバ), 109
tp_mro (PyTypeObject のメンバ), 113
tp_name (PyTypeObject のメンバ), 101
tp_new (PyTypeObject のメンバ), 112
tp_next (PyTypeObject のメンバ), 114
tp_print (PyTypeObject のメンバ), 103
tp_repr (PyTypeObject のメンバ), 104
tp_richcompare (PyTypeObject のメンバ), 108
tp_setattr (PyTypeObject のメンバ), 103
tp_setattro (PyTypeObject のメンバ), 105
tp_str (PyTypeObject のメンバ), 104
tp_subclasses (PyTypeObject のメンバ), 113
tp_traverse (PyTypeObject のメンバ), 107
tp_weaklist (PyTypeObject のメンバ), 114
tp_weaklistoffset (PyTypeObject のメンバ),
108

tuple

object, 66

オブジェクト, 66

tuple() (組み込み関数), 42, 68

TupleType (in module types), 66

type

object, 2, 47

オブジェクト, 2, 47

type() (組み込み関数), 36

TypeType (in module types), 47

U

ULONG_MAX, 50

unicode() (組み込み関数), 34

V

version (in module sys), 82, 83