
Extending and Embedding Python

リリース 3.7.17

Guido van Rossum
and the Python development team

6 月 28, 2023

目次

第 1 章	おすすめのサードパーティツール	3
第 2 章	サードパーティツールなしで拡張を作る	5
2.1	C や C++ による Python の拡張	5
2.2	拡張の型の定義: チュートリアル	30
2.3	Defining Extension Types: Assorted Topics	61
2.4	C および C++ 拡張のビルド	74
2.5	Windows 上での C および C++ 拡張モジュールのビルド	77
第 3 章	大規模なアプリケーションへの Python ランタイムの埋め込み	81
3.1	他のアプリケーションへの Python の埋め込み	81
付録 A 章	用語集	89
付録 B 章	このドキュメントについて	107
B.1	Python ドキュメント 貢献者	107
付録 C 章	歴史とライセンス	109
C.1	Python の歴史	109
C.2	Terms and conditions for accessing or otherwise using Python	110
C.3	Licenses and Acknowledgements for Incorporated Software	114
付録 D 章	Copyright	129
索引		131
索引		131

このドキュメントでは、Python インタプリタを拡張する新しいモジュールを C または C++ で書く方法を解説しています。このようなモジュールでは新しい関数を定義するだけでなく、新しい型や、そのメソッドを定義することができます。ドキュメントでは他のアプリケーションで Python を拡張言語として使用するために、Python インタプリタをアプリケーションに埋め込む方法についても解説します。最後に、下層のオペレーティングシステムが動的 (実行時) ロードをサポートしていれば、拡張モジュールが動的にライブラリにロードされるように、モジュールをコンパイルしリンクする方法について解説します。

このドキュメントでは、読者は Python について基礎的な知識を持ち合わせているものと仮定しています。形式ばらない Python 言語の入門には、[tutorial-index](#) を読んでください。[reference-index](#) を読めば、Python 言語についてより形式的な定義を得られます。また、[library-index](#) では、Python に広い適用範囲をもたらしている既存のオブジェクト型、関数、および (組み込み、および Python で書かれたものの両方の) モジュールについて解説しています。

Python/C API 全体の詳しい説明は、別のドキュメントである、[c-api-index](#) を参照してください。

おすすめのサードパーティツール

このガイドがカバーするのは、このバージョンの CPython の一部として提供されている、拡張を作成するための基本的なツールだけです。Cython, cffi, SWIG, Numba のようなサードパーティのツールは、Python の C および C++ 拡張を作成するための、より簡潔かつより洗練された手法を提供します。

参考:

[Python Packaging User Guide: Binary Extensions](#) Python Packaging User Guide はバイナリ拡張の作成が簡単になる便利なツールをカバーしているだけでなく、まず始めになぜ拡張モジュールを作ることが望ましいかの様々な理由について議論しています。

サードパーティツールなしで拡張を作る

ガイドのこの節ではサードパーティツールの補助無しに C および C++ 拡張を作成する方法を説明します。これは自分自身の C 拡張を作成するおすすめの方法というよりも、主にそれらのツールを作成する人向けのものです。

2.1 C や C++ による Python の拡張

C プログラムの書き方を知っているなら、Python に新たな組み込みモジュールを追加するのはきわめて簡単です。この新たなモジュール、拡張モジュール (*extension module*) を使うと、Python が直接行えない二つのこと：新しい組み込みオブジェクトの実装、そして全ての C ライブラリ関数とシステムコールに対する呼び出し、ができるようになります。

拡張モジュールをサポートするため、Python API (Application Programmer's Interface) では一連の関数、マクロおよび変数を提供していて、Python ランタイムシステムのほとんどの側面へのアクセス手段を提供しています。Python API は、ヘッダ "Python.h" をインクルードして C ソースに取り込みます。

拡張モジュールのコンパイル方法は、モジュールの用途やシステムの設定方法に依存します。詳細は後の章で説明します。

注釈： C 拡張のインターフェイスは CPython に固有のものであり、これによる拡張モジュールはほかの Python 実装では動作しません。多くの場合、C 拡張を書くことを避けてほかの Python 実装のために移植性を確保することは可能です。たとえば、あなたがしたいことが C ライブラリの関数やシステムコールを呼び出すことである場合、`ctypes` あるいは `cffi` ライブラリの利用を検討すべきです。これらのモジュールは C コードとインターフェイスし、C 拡張を書いてコンパイルするのに較べて Python 実装間のより高い移植性をもった Python コードを書かせてくれます。

2.1.1 簡単な例

spam (Monty Python ファンの好物ですね) という名の拡張モジュールを作成することにして、C ライブラリ関数 `system()` に対する Python インターフェイスを作成したいとします。^{*1} この関数は `null` で終端されたキャラクター文字列を引数にとり、整数を返します。この関数を以下のようにして Python から呼び出せるようにしたいとします。

```
>>> import spam
>>> status = spam.system("ls -l")
```

まずは `spammodule.c` を作成するところから始めます。(伝統として、`spam` という名前のモジュールを作成する場合、モジュールの実装が入った C ファイルを `spammodule.c` と呼ぶことになっています; `spammify` のように長すぎるモジュール名の場合には、単に `spammify.c` にもできます。)

このファイルの最初の 2 行は以下のようにします:

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

これで、Python API を取り込みます (必要なら、モジュールの用途に関する説明や、著作権表示を追加します)。

注釈: Python は、システムによっては標準ヘッダの定義に影響するようなプリプロセッサ定義を行っているので、`Python.h` をいずれの標準ヘッダよりも前にインクルード **せねばなりません**。

`Python.h` をインクルードする前に、常に `PY_SSIZE_T_CLEAN` を定義することが推奨されます。このマクロの解説については [拡張モジュール関数でのパラメタ展開](#) を参照してください。

`Python.h` で定義されているユーザから可視のシンボルは、全て接頭辞 `Py` または `PY` が付いています。ただし、標準ヘッダファイル内の定義は除きます。簡単のためと、Python 内で広範に使うことになるという理由から、`"Python.h"` はいくつかの標準ヘッダファイル: `<stdio.h>`、`<string.h>`、`<errno.h>`、および `<stdlib.h>` をインクルードしています。後者のヘッダファイルがシステム上になれば、`"Python.h"` が関数 `malloc()`、`free()` および `realloc()` を直接定義します。

次にファイルに追加する内容は、Python 式 `spam.system(string)` を評価する際に呼び出されることになる C 関数です (この関数を最終的にどのように呼び出すかは、後ですぐわかります):

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
```

(次のページに続く)

^{*1} この関数へのインタフェースはすでに標準モジュール `os` にあります --- この関数を選んだのは、単純で直接的な例を示したいからです。

(前のページからの続き)

```

int sts;

if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;
sts = system(command);
return PyLong_FromLong(sts);
}

```

ここでは、Python の引数リスト (例えば、単一の式 `"ls -l"`) から C 関数に渡す引数にそのまま変換しています。C 関数は常に二つの引数を持ち、便宜的に *self* および *args* と呼ばれます。

self 引数には、モジュールレベルの関数であればモジュールが、メソッドにはオブジェクトインスタンスが渡されます。

args 引数は、引数の入った Python タプルオブジェクトへのポインタになります。タプル内の各要素は、呼び出しの際の引数リストにおける各引数に対応します。引数は Python オブジェクトです --- C 関数で引数を使って何かを行うには、オブジェクトから C の値に変換せねばなりません。Python API の関数 `PyArg_ParseTuple()` は引数の型をチェックし、C の値に変換します。`PyArg_ParseTuple()` はテンプレート文字列を使って、引数オブジェクトの型と、変換された値を入れる C 変数の型を判別します。これについては後で詳しく説明します。

`PyArg_ParseTuple()` は、全ての引数が正しい型を持っていて、アドレス渡しされた各変数に各引数要素を保存したときに真 (非ゼロ) を返します。この関数は不正な引数リストを渡すと偽 (ゼロ) を返します。後者の場合、関数は適切な例外を送出するので、呼び出し側は (例にもあるように) すぐに `NULL` を返すようにしてください。

2.1.2 幕間小話: エラーと例外

Python インタプリタ全体を通して、一つの重要な取り決めがあります: それは、関数が処理に失敗した場合、例外状態をセットして、エラーを示す値 (通常は `NULL` ポインタ) を返さねばならない、ということです。例外はインタプリタ内の静的なグローバル変数に保存されます; この値が `NULL` の場合、例外は何も起きていないことになります。第二のグローバル変数には、例外の " 付属値 (associated value) " (`raise` 文の第二引数) が入ります。第三の値には、エラーの発生源が Python コード内だった場合にスタックトレースバック (stack traceback) が入ります。これらの三つの変数は、`sys.exc_info()` の Python での結果と等価な C の変数です (Python ライブラリリファレンスの `sys` モジュールに関する節を参照してください。) エラーがどのように受け渡されるかを理解するには、これらの変数についてよく知っておくことが重要です。

Python API では、様々な型の例外をセットするための関数をいくつか定義しています。

もっともよく用いられるのは `PyErr_SetString()` です。引数は例外オブジェクトと C 文字列です。例外オブジェクトは通常、`PyExc_ZeroDivisionError` のような定義済みのオブジェクトです。C 文字列はエラーの原因を示し、Python 文字列オブジェクトに変換されて例外の " 付属値 " に保存されます。

もう一つ有用な関数として `PyErr_SetFromErrno()` があります。この関数は引数に例外だけを取り、付属値はグローバル変数 `errno` から構築します。もっとも汎用的な関数は `PyErr_SetObject()` で、二つのオブジェクト、

例外と付属値を引数にとります。これら関数に渡すオブジェクトには `Py_INCREF()` を使う必要はありません。

例外がセットされているかどうかは、`PyErr_Occurred()` を使って非破壊的に調べられます。この関数は現在の例外オブジェクトを返します。例外が発生していない場合には `NULL` を返します。通常は、関数の戻り値からエラーが発生したかを判別できるはずなので、`PyErr_Occurred()` を呼び出す必要はありません。

関数 g を呼び出す f が、前者の関数の呼び出しに失敗したことを検出すると、 f 自体はエラー値 (大抵は `NULL` や `-1`) を返さねばなりません。しかし、`PyErr_*`(\cdot) 関数群のいずれかを呼び出す必要は **ありません** --- なぜなら、 g がすでに呼び出しているからです。次いで f を呼び出したコードもエラーを示す値を **自ら呼び出したコード** に返すことになりますが、同様に `PyErr_*`(\cdot) は **呼び出しません**。以下同様に続きます --- エラーの最も詳しい原因は、最初にエラーを検出した関数がすでに報告しているからです。エラーが Python インタプリタのメインループに到達すると、現在実行中の Python コードは一時停止し、Python プログラマが指定した例外ハンドラを探し出そうとします。

(モジュールが `PyErr_*`(\cdot) 関数をもう一度呼び出して、より詳細なエラーメッセージを提供するような状況があります。このような状況ではそうすべきです。とはいえ、一般的な規則としては、`PyErr_*`(\cdot) を何度も呼び出す必要はなく、ともすればエラーの原因に関する情報を失う結果になりがちです: これにより、ほとんどの操作が様々な理由から失敗するかもしれません)

ある関数呼び出しでの処理の失敗によってセットされた例外を無視するには、`PyErr_Clear()` を呼び出して例外状態を明示的に消去しなくてはなりません。エラーをインタプリタには渡したくなく、自前で (何か他の作業を行ったり、何も起こらなかったかのように見せかけるような) エラー処理を完全に行う場合にのみ、`PyErr_Clear()` を呼び出すようにすべきです。

`malloc()` の呼び出し失敗は、常に例外にしなくてはなりません --- `malloc()` (または `realloc()`) を直接呼び出しているコードは、`PyErr_NoMemory()` を呼び出して、失敗を示す値を返さねばなりません。オブジェクトを生成する全ての関数 (例えば `PyLong_FromLong()`) は `PyErr_NoMemory()` の呼び出しを済ませてしまうので、この規則が関係するのは直接 `malloc()` を呼び出すコードだけです。

また、`PyArg_ParseTuple()` という重要な例外を除いて、整数の状態コードを返す関数はたいいてい、Unix のシステムコールと同じく、処理が成功した際にはゼロまたは正の値を返し、失敗した場合には `-1` を返します。

最後に、エラー標示値を返す際に、(エラーが発生するまでに既に生成してしまったオブジェクトに対して `Py_XDECREF()` や `Py_DECREF()` を呼び出して) ごみ処理を注意深く行ってください!

どの例外を返すかの選択は、ユーザに完全にゆだねられます。`PyExc_ZeroDivisionError` のように、全ての組み込みの Python 例外には対応する宣言済みの C オブジェクトがあり、直接利用できます。もちろん、例外の選択は賢く行わねばなりません --- ファイルが開けなかったことを表すのに `PyExc_TypeError` を使ったりはしないでください (この場合はおそらく `PyExc_IOError` の方にすべきでしょう)。引数リストに問題がある場合には、`PyArg_ParseTuple()` はたいいてい `PyExc_TypeError` を送出します。引数の値が特定の範囲を超えていたり、その他の満たすべき条件を満たさなかった場合には、`PyExc_ValueError` が適切です。

モジュール固有の新たな例外も定義できます。定義するには、通常はファイルの先頭部分に静的なオブジェクト変数の宣言を行います:

```
static PyObject *SpamError;
```

そして、モジュールの初期化関数 (PyInit_spam()) の中で、例外オブジェクトを使って初期化します:

```
PyMODINIT_FUNC
PyInit_spam(void)
{
    PyObject *m;

    m = PyModule_Create(&spammodule);
    if (m == NULL)
        return NULL;

    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    Py_XINCREF(SpamError);
    if (PyModule_AddObject(m, "error", SpamError) < 0) {
        Py_XDECREF(SpamError);
        Py_CLEAR(SpamError);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

Python レベルでの例外オブジェクトの名前は `spam.error` になることに注意してください。PyErr_NewException() 関数は、bltin-exceptions で述べられている Exception クラスを基底クラスに持つ例外クラスも作成できます (NULL の代わりに他のクラスを渡した場合は別です)。

SpamError 変数は、新たに生成された例外クラスへの参照を維持することにも注意してください; これは意図的な仕様です! 外部のコードが例外オブジェクトをモジュールから除去できるため、モジュールから新たに作成した例外クラスが見えなくなり、SpamError がぶら下がりポインタ (dangling pointer) になってしまわないようにするために、クラスに対する参照を所有しておかねばなりません。もし SpamError がぶら下がりポインタになってしまうと、C コードが例外を送出しようとしたときにコアダンプや意図しない副作用を引き起こすことがあります。

この例にある、関数の戻り値型に PyMODINIT_FUNC を使う方法については後で議論します。

PyErr_SetString() を次のように呼び出すと、拡張モジュールで例外 `spam.error` を送出することができます:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
```

(次のページに続く)

(前のページからの続き)

```

    sts = system(command);
    if (sts < 0) {
        PyErr_SetString(SpamError, "System command failed");
        return NULL;
    }
    return PyLong_FromLong(sts);
}

```

2.1.3 例に戻る

先ほどの関数の例に戻ると、今度は以下の実行文を理解できるはずです:

```

if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;

```

この実行文は、`PyArg_ParseTuple()` がセットする例外によって、引数リストに何らかのエラーが生じたときに `NULL` (オブジェクトへのポインタを返すタイプの関数におけるエラー標示値) を返します。エラーでなければ、引数として与えた文字列値はローカルな変数 `command` にコピーされています。この操作はポインタ代入であり、ポインタが指している文字列に対して変更が行われるとは想定されていません (従って、標準 C では、変数 `command` は `const char* command` として適切に定義せねばなりません)。

次の文では、`PyArg_ParseTuple()` で得た文字列を渡して Unix 関数 `system()` を呼び出しています:

```

sts = system(command);

```

`spam.system()` は `sts` を Python オブジェクトとして返さねばなりません。これには、`PyLong_FromLong()` を使います。

```

return PyLong_FromLong(sts);

```

上の場合では、`Py_BuildValue()` は整数オブジェクトを返します。(そう、整数ですら、Python においてはヒープ上のオブジェクトなのです!)

何ら有用な値を返さない関数 (`void` を返す関数) に対応する Python の関数は `None` を返さねばなりません。関数に `None` を返させるには、以下のような慣用句を使います (この慣用句は `Py_RETURN_NONE` マクロに実装されています):

```

Py_INCREF(Py_None);
return Py_None;

```

`Py_None` は特殊な Python オブジェクトである `None` に対応する C での名前です。これまで見てきたようにほとんどのコンテキストで "エラー" を意味する `NULL` ポインタとは違い、`None` は純粋な Python のオブジェクトです。

2.1.4 モジュールのメソッドテーブルと初期化関数

さて、前に約束したように、`spam_system()` を Python プログラムからどうやって呼び出すかをこれから示します。まずは、関数名とアドレスを "メソッドテーブル (method table)" に列挙する必要があります:

```
static PyMethodDef SpamMethods[] = {
    ...
    {"system", spam_system, METH_VARARGS,
     "Execute a shell command."},
    ...
    {NULL, NULL, 0, NULL}        /* Sentinel */
};
```

リスト要素の三つ目のエントリ (`METH_VARARGS`) に注意してください。このエントリは、C 関数が使う呼び出し規約をインタプリタに教えるためのフラグです。通常この値は `METH_VARARGS` か `METH_VARARGS | METH_KEYWORDS` のはずですが、0 は旧式の `PyArg_ParseTuple()` の変換形が使われることを意味します。

`METH_VARARGS` だけを使う場合、C 関数は、Python レベルでの引数が `PyArg_ParseTuple()` が受理できるタプルの形式で渡されるものと想定しなければなりません; この関数についての詳細は下で説明します。

関数にキーワード引数が渡されることになっているのなら、第三フィールドに `METH_KEYWORDS` ビットをセットできます。この場合、C 関数は第三引数に `PyObject *` を受理するようにせねばなりません。このオブジェクトは、キーワード引数の辞書になります。こうした関数で引数を解釈するには、`PyArg_ParseTupleAndKeywords()` を使ってください。

メソッドテーブルはモジュール定義の構造体から参照されていなければなりません:

```
static struct PyModuleDef spammodule = {
    PyModuleDef_HEAD_INIT,
    "spam",      /* name of module */
    spam_doc,    /* module documentation, may be NULL */
    -1,          /* size of per-interpreter state of the module,
                  or -1 if the module keeps state in global variables. */
    SpamMethods
};
```

同様に、この構造体は、モジュールの初期化関数内でインタプリタに渡さねばなりません。初期化関数はモジュールの名前を `name` としたときに `PyInit_name()` という名前でなければならず、モジュールファイル内で定義されているもののうち、唯一の非 `static` 要素でなければなりません:

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

`PyMODINIT_FUNC` は関数の戻り値を `PyObject *` になるように宣言し、プラットフォーム毎に必要とされ

る、特有のリンク宣言 (linkage declaration) を定義すること、さらに C++ の場合には関数を `extern "C"` に宣言することに注意してください。

Python プログラムが初めて `spam` モジュールを `import` するときに、`PyInit_spam()` が呼ばれます。(以下にある Python への埋め込みに関するコメントを参照してください。) そこからモジュールオブジェクトを返す `PyModule_Create()` が呼ばれ、モジュール定義にあるテーブル (`PyMethodDef` 構造体の配列) に基いて新たに作られたモジュールに、組み込み関数オブジェクトが挿入されます。`PyModule_Create()` は作成したモジュールオブジェクトへのポインタを返します。あるエラーによって異常終了するかもしれませんし、モジュールが問題無く初期化できなかった場合には `NULL` を返すかもしれません。`init` 関数はモジュールオブジェクトを呼び出し元に返し、それが `sys.modules` に挿入されるようにしなければなりません。

Python へ埋め込むときに、`PyImport_Inittab` テーブルに存在していても `PyInit_spam()` 関数は自動的に呼ばれません。初期化テーブルにモジュールを追加するには、`PyImport_AppendInittab()` を使ってください。その後にオプションでモジュールを `import` します:

```
int
main(int argc, char *argv[])
{
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }

    /* Add a built-in module, before Py_Initialize */
    PyImport_AppendInittab("spam", PyInit_spam);

    /* Pass argv[0] to the Python interpreter */
    Py_SetProgramName(program);

    /* Initialize the Python interpreter.  Required. */
    Py_Initialize();

    /* Optionally import the module; alternatively,
       import can be deferred until the embedded script
       imports it. */
    PyImport_ImportModule("spam");

    ...

    PyMem_RawFree(program);
    return 0;
}
```

注釈: 単一のプロセス内 (または `fork()` 後の `exec()` が介入していない状態) における複数のインタプリタにおいて、`sys.modules` からエントリを除去したり新たなコンパイル済みモジュールを `import` したりすると、拡

張モジュールによっては問題を生じることがあります。拡張モジュールの作者は、内部データ構造を初期化する際にはよくよく用心すべきです。

より実質的なモジュール例は、Python ソース配布物に `Modules/xxmodule.c` という名前が入っています。このファイルはテンプレートとしても利用できますし、単に例としても読めます。

注釈: `xmodule` は `spam` と異なり、**多段階初期化** (*multi-phase initialization* (Python 3.5 の新機能) を使っています。PyInit_spam が PyModuleDef を返し、モジュールの生成は後に `import` 機構が行います。多段階初期化についての詳細は [PEP 489](#) を参照してください。

2.1.5 コンパイルとリンク

新しい拡張モジュールを使えるようになるまで、まだ二つの作業: コンパイルと、Python システムへのリンク、が残っています。動的読み込み (dynamic loading) を使っているのなら、作業の詳細は自分のシステムが使っている動的読み込みの形式によって変わるかもしれませんが; 詳しくは、拡張モジュールのビルドに関する章 ([C および C++ 拡張のビルド](#) 章) や、Windows におけるビルドに関係する追加情報の章 ([Windows 上での C および C++ 拡張モジュールのビルド](#) 章) を参照してください。

動的読み込みを使えなかったり、モジュールを常時 Python インタプリタの一部にしておきたい場合には、インタプリタのビルド設定を変更して再ビルドしなければならないでしょう。Unix では、幸運なことにこの作業はとても単純です: 単に自作のモジュールファイル (例えば `spammodule.c`) を展開したソース配布物の `Modules/` ディレクトリに置き、`Modules/Setup.local` に自分のファイルを説明する以下の一行:

```
spam spammodule.o
```

を追加して、トップレベルのディレクトリで `make` を実行して、インタプリタを再ビルドするだけです。`Modules/` サブディレクトリでも `make` を実行できますが、前もって '`make Makefile`' を実行して `Makefile` を再ビルドしておかなければなりません。(この作業は `Setup` ファイルを変更するたびに必要です。)

モジュールが別のライブラリとリンクされている必要がある場合、ライブラリも設定ファイルに列挙できます。例えば以下のようにします。

```
spam spammodule.o -lX11
```

2.1.6 C から Python 関数を呼び出す

これまでは、Python からの C 関数の呼び出しに重点を置いて述べてきました。ところでこの逆: C からの Python 関数の呼び出しもまた有用です。とりわけ、いわゆる ”コールバック” 関数をサポートするようなライブラリを作成するにはこの機能が便利です。ある C インタフェースがコールバックを利用している場合、同等の機能を提供する Python コードでは、しばしば Python プログラマにコールバック機構を提供する必要があります; このとき実装では、C で書かれたコールバック関数から Python で書かれたコールバック関数を呼び出すようにする必要がありますでしょう。もちろん、他の用途も考えられます。

幸運なことに、Python インタプリタは簡単に再帰呼び出しでき、Python 関数を呼び出すための標準インタフェースもあります。(Python パーザを特定の入力文字を使って呼び出す方法について詳説するつもりはありません --- この方法に興味があるなら、Python ソースコードの Modules/main.c にある、コマンドラインオプション `-c` の実装を見てください)

Python 関数の呼び出しは簡単です。まず、C のコードに対してコールバックを登録しようとする Python プログラムは、何らかの方法で Python の関数オブジェクトを渡さねばなりません。このために、コールバック登録関数 (またはその他のインタフェース) を提供せねばなりません。このコールバック登録関数が呼び出された際に、引き渡された Python 関数オブジェクトへのポインタをグローバル変数に --- あるいは、どこか適切な場所に --- 保存します (関数オブジェクトを `Py_INCREF()` するようよく注意してください!)。例えば、以下のような関数がモジュールの一部になっていることでしょう:

```
static PyObject *my_callback = NULL;

static PyObject *
my_set_callback(PyObject *dummy, PyObject *args)
{
    PyObject *result = NULL;
    PyObject *temp;

    if (PyArg_ParseTuple(args, "O:set_callback", &temp)) {
        if (!PyCallable_Check(temp)) {
            PyErr_SetString(PyExc_TypeError, "parameter must be callable");
            return NULL;
        }
        Py_XINCREF(temp);           /* Add a reference to new callback */
        Py_XDECREF(my_callback);    /* Dispose of previous callback */
        my_callback = temp;         /* Remember new callback */
        /* Boilerplate to return "None" */
        Py_INCREF(Py_None);
        result = Py_None;
    }
    return result;
}
```

この関数は `METH_VARARGS` フラグを使ってインタプリタに登録せねばなりません; `METH_VARARGS` フラグについては、[モジュールのメソッドテーブルと初期化関数](#) で説明しています。 `PyArg_ParseTuple()` 関数とその引数に

については、[拡張モジュール関数でのパラメタ展開](#) に記述しています。

`Py_XINCRREF()` および `Py_XDECREF()` は、オブジェクトに対する参照カウントをインクリメント/デクリメントするためのマクロで、`NULL` ポインタが渡されても安全に操作できる形式です (とはいえ、上の流れでは `temp` が `NULL` になることはありません)。これらのマクロと参照カウントについては、[参照カウント法](#) で説明しています。

その後、コールバック関数を呼び出す時が来たら、C 関数 `PyObject_CallObject()` を呼び出します。この関数には二つの引数: Python 関数と Python 関数の引数リストがあり、いずれも任意の Python オブジェクトを表すポインタ型です。引数リストは常にタプルオブジェクトでなければならず、その長さは引数の数になります。Python 関数を引数なしで呼び出すのなら、`NULL` か空のタプルを渡します; 単一の引数で関数を呼び出すのなら、単要素 (singleton) のタプルを渡します。`Py_BuildValue()` の書式文字列中に、ゼロ個または一個以上の書式化コードが入った丸括弧がある場合、この関数はタプルを返します。以下に例を示します:

```
int arg;
PyObject *arglist;
PyObject *result;
...
arg = 123;
...
/* Time to call the callback */
arglist = Py_BuildValue("(i)", arg);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
```

`PyObject_CallObject()` は Python オブジェクトへのポインタを返します: これは Python 関数からの戻り値になります。`PyObject_CallObject()` は、引数に対して "参照カウント中立 (reference-count- neutral)" です。上の例ではタプルを生成して引数リストとして提供しており、このタプルは `PyObject_CallObject()` の呼び出し直後に `Py_DECREF()` されています。

`PyObject_CallObject()` は戻り値として "新しい" オブジェクト: 新規に作成されたオブジェクトか、既存のオブジェクトの参照カウントをインクリメントしたものを返します。従って、このオブジェクトをグローバル変数に保存したいのでないかぎり、たとえこの戻り値に興味がなくとも (むしろ、そうであればなおさら!) 何がしかの方法で戻り値オブジェクトを `Py_DECREF()` しなければなりません。

とはいえ、戻り値を `Py_DECREF()` する前には、値が `NULL` でないかチェックしておくことが重要です。もし `NULL` なら、呼び出した Python 関数は例外を送出して終了させられています。`PyObject_CallObject()` を呼び出しているコード自体もまた Python から呼び出されているのであれば、今度は C コードが自分を呼び出している Python コードにエラー標示値を返さねばなりません。それにより、インタプリタはスタックトレースを出力したり、例外を処理するための Python コードを呼び出したりできます。例外の送出不可能だったり、したくないのなら、`PyErr_Clear()` を呼んで例外を消去しておかねばなりません。例えば以下のようにします:

```
if (result == NULL)
    return NULL; /* Pass error back */
...use result...
Py_DECREF(result);
```

Python コールバック関数をどんなインタフェースにしたいかによっては、引数リストを `PyObject_CallObject()` に与えなければならない場合もあります。あるケースでは、コールバック関数を指定したのと同じインタフェースを介して、引数リストも渡されているかもしれません。また別のケースでは、新しいタプルを構築して引数リストを渡さねばならないかもしれません。この場合最も簡単なのは `Py_BuildValue()` を呼ぶやり方です。例えば、整数のイベントコードを渡したければ、以下のようなコードを使うことになるでしょう:

```
PyObject *arglist;
...
arglist = Py_BuildValue("(l)", eventcode);
result = PyObject_CallObject(my_callback, arglist);
Py_DECREF(arglist);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

`Py_DECREF(arglist)` が呼び出しの直後、エラーチェックよりも前に置かれていることに注意してください! また、厳密に言えば、このコードは完全ではありません: `Py_BuildValue()` はメモリ不足におちいるかもしれず、チェックしておくべきです。

通常の引数とキーワード引数をサポートする `PyObject_Call()` を使って、キーワード引数を伴う関数呼び出しをすることができます。上の例と同じように、`Py_BuildValue()` を作って辞書を作ります。

```
PyObject *dict;
...
dict = Py_BuildValue("{s:i}", "name", val);
result = PyObject_Call(my_callback, NULL, dict);
Py_DECREF(dict);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

2.1.7 拡張モジュール関数でのパラメタ展開

`PyArg_ParseTuple()` 関数は以下のように宣言されています:

```
int PyArg_ParseTuple(PyObject *arg, const char *format, ...);
```

引数 *arg* は C 関数から Python に渡される引数リストが入ったタプルオブジェクトでなければなりません。*format* 引数は書式文字列で、Python/C API リファレンスマニュアルの *arg-parsing* で解説されている書法に従わねばなりません。残りの引数は、それぞれの変数のアドレスで、書式化文字列から決まる型になっていなければなりません。

`PyArg_ParseTuple()` は Python 側から与えられた引数が必要な型になっているか調べるのに対し、

PyArg_ParseTuple() は呼び出しの際に渡された C 変数のアドレスが有効な値を持つか調べられないことに注意してください: ここで間違いを犯すと、コードがクラッシュするかもしれませんし、少なくともでたらめなビットをメモリに上書きしてしまいます。慎重に!

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

以下にいくつかの呼び出し例を示します:

```
#define PY_SSIZE_T_CLEAN /* Make "s#" use Py_ssize_t rather than int. */
#include <Python.h>
```

```
int ok;
int i, j;
long k, l;
const char *s;
Py_ssize_t size;

ok = PyArg_ParseTuple(args, ""); /* No arguments */
/* Python call: f() */
```

```
ok = PyArg_ParseTuple(args, "s", &s); /* A string */
/* Possible Python call: f('whoops!') */
```

```
ok = PyArg_ParseTuple(args, "lls", &k, &l, &s); /* Two longs and a string */
/* Possible Python call: f(1, 2, 'three') */
```

```
ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size);
/* A pair of ints and a string, whose size is also returned */
/* Possible Python call: f((1, 2), 'three') */
```

```
{
    const char *file;
    const char *mode = "r";
    int bufsize = 0;
    ok = PyArg_ParseTuple(args, "s|si", &file, &mode, &bufsize);
    /* A string, and optionally another string and an integer */
    /* Possible Python calls:
       f('spam')
       f('spam', 'w')
       f('spam', 'wb', 100000) */
}
```

```
{
    int left, top, right, bottom, h, v;
```

(次のページに続く)

(前のページからの続き)

```

    ok = PyArg_ParseTuple(args, "((ii)(ii))(ii)",
                          &left, &top, &right, &bottom, &h, &v);
    /* A rectangle and a point */
    /* Possible Python call:
       f(((0, 0), (400, 300)), (10, 10)) */
}

```

```

{
    Py_complex c;
    ok = PyArg_ParseTuple(args, "D:myfunction", &c);
    /* a complex, also providing a function name for errors */
    /* Possible Python call: myfunction(1+2j) */
}

```

2.1.8 拡張モジュール関数のキーワードパラメタ

PyArg_ParseTupleAndKeywords() は、以下のように宣言されています:

```

int PyArg_ParseTupleAndKeywords(PyObject *arg, PyObject *kwdict,
                                const char *format, char *kwlist[], ...);

```

arg と *format* パラメタは PyArg_ParseTuple() のものと同じです。 *kwdict* パラメタはキーワード引数の入った辞書で、Python ランタイムシステムから第三パラメタとして受け取ります。 *kwlist* パラメタは各パラメタを識別するための文字列からなる、NULL 終端されたリストです; 各パラメタ名は *format* 中の型情報に対して左から右の順に照合されます。成功すると PyArg_ParseTupleAndKeywords() は真を返し、それ以外の場合には適切な例外を送出して偽を返します。

注釈: キーワード引数を使っている場合、タプルは入れ子にして使えません! *kwlist* 内に存在しないキーワードパラメタが渡された場合、TypeError の送出を引き起こします。

以下にキーワードを使ったモジュール例を示します。これは Geoff Philbrick (philbrick@hks.com) によるプログラム例をもとにしています:

```

#define PY_SSIZE_T_CLEAN /* Make "s#" use Py_ssize_t rather than int. */
#include <Python.h>

static PyObject *
keywarg_parrot(PyObject *self, PyObject *args, PyObject *keywds)
{
    int voltage;
    const char *state = "a stiff";
}

```

(次のページに続く)

(前のページからの続き)

```

const char *action = "vroom";
const char *type = "Norwegian Blue";

static char *kwlist[] = {"voltage", "state", "action", "type", NULL};

if (!PyArg_ParseTupleAndKeywords(args, keywds, "i|sss", kwlist,
                                &voltage, &state, &action, &type))
    return NULL;

printf("-- This parrot wouldn't %s if you put %i Volts through it.\n",
        action, voltage);
printf("-- Lovely plumage, the %s -- It's %s!\n", type, state);

Py_RETURN_NONE;
}

static PyMethodDef keywdarg_methods[] = {
    /* The cast of the function is necessary since PyCFunction values
     * only take two PyObject* parameters, and keywdarg_parrot() takes
     * three.
     */
    {"parrot", (PyCFunction)keywdarg_parrot, METH_VARARGS | METH_KEYWORDS,
     "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL} /* sentinel */
};

static struct PyModuleDef keywdargmodule = {
    PyModuleDef_HEAD_INIT,
    "keywdarg",
    NULL,
    -1,
    keywdarg_methods
};

PyMODINIT_FUNC
PyInit_keywdarg(void)
{
    return PyModule_Create(&keywdargmodule);
}

```

2.1.9 任意の値を構築する

`Py_BuildValue()` は `PyArg_ParseTuple()` の対極に位置するものです。この関数は以下のように定義されています:

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

`Py_BuildValue()` は、`PyArg_ParseTuple()` の認識する一連の書式単位に似た書式単位を認識します。ただし (関数への出力ではなく、入力に使われる) 引数はポインタではなく、ただの値でなければなりません。Python から呼び出された C 関数が返す値として適切な、新たな Python オブジェクトを返します。

`PyArg_ParseTuple()` とは一つ違う点があります: `PyArg_ParseTuple()` は第一引数をタプルにする必要があります (Python の引数リストは内部的には常にタプルとして表現されるからです) が、`Py_BuildValue()` はタプルを生成するとは限りません。`Py_BuildValue()` は書式文字列中に書式単位が二つかそれ以上入っている場合のみタプルを構築します。書式文字列が空なら、`None` を返します。きっかり一つの書式単位なら、その書式単位が記述している何らかのオブジェクトになります。サイズが 0 や 1 のタプル返させたいのなら、書式文字列を丸括弧で囲みます。

以下に例を示します (左に呼び出し例を、右に構築される Python 値を示します):

<code>Py_BuildValue("")</code>	<code>None</code>
<code>Py_BuildValue("i", 123)</code>	<code>123</code>
<code>Py_BuildValue("iii", 123, 456, 789)</code>	<code>(123, 456, 789)</code>
<code>Py_BuildValue("s", "hello")</code>	<code>'hello'</code>
<code>Py_BuildValue("y", "hello")</code>	<code>b'hello'</code>
<code>Py_BuildValue("ss", "hello", "world")</code>	<code>('hello', 'world')</code>
<code>Py_BuildValue("s#", "hello", 4)</code>	<code>'hell'</code>
<code>Py_BuildValue("y#", "hello", 4)</code>	<code>b'hell'</code>
<code>Py_BuildValue("()")</code>	<code>()</code>
<code>Py_BuildValue("(i)", 123)</code>	<code>(123,)</code>
<code>Py_BuildValue("(ii)", 123, 456)</code>	<code>(123, 456)</code>
<code>Py_BuildValue("(i,i)", 123, 456)</code>	<code>(123, 456)</code>
<code>Py_BuildValue("[i,i]", 123, 456)</code>	<code>[123, 456]</code>
<code>Py_BuildValue("{s:i,s:i}",</code>	
<code> "abc", 123, "def", 456)</code>	<code>{'abc': 123, 'def': 456}</code>
<code>Py_BuildValue("((ii)(ii)) (ii)",</code>	
<code> 1, 2, 3, 4, 5, 6)</code>	<code>(((1, 2), (3, 4)), (5, 6))</code>

2.1.10 参照カウント法

C や C++ のような言語では、プログラマはヒープ上のメモリを動的に確保したり解放したりする責任があります。こうした作業は C では関数 `malloc()` や `free()` で行います。C++ では本質的に同じ意味で演算子 `new` や `delete` が使われます。そこで、以下の議論は C の場合に限定して行います。

`malloc()` が確保する全てのメモリブロックは、最終的には `free()` を厳密に一度だけ呼び出して利用可能メモリのプールに戻さねばなりません。そこで、適切な時に `free()` を呼び出すことが重要になります。あるメモリブロックに対して、`free()` を呼ばなかったにもかかわらずそのアドレスを忘却してしまうと、ブロックが占有しているメモリはプログラムが終了するまで再利用できなくなります。これはメモリリーク (*memory leak*) と呼ばれています。逆に、プログラムがあるメモリブロックに対して `free()` を呼んでおきながら、そのブロックを使い続けようとすると、別の `malloc()` 呼び出しによって行われるブロックの再利用と衝突を起こします。これは解放済みメモリの使用 (*using freed memory*) と呼ばれます。これは初期化されていないデータに対する参照と同様のよくない結果 --- コアダンプ、誤った参照、不可解なクラッシュ --- を引き起こします。

よくあるメモリリークの原因はコード中の普通でない処理経路です。例えば、ある関数があるメモリブロックを確保し、何らかの計算を行って、再度ブロックを解放するとします。さて、関数の要求仕様を変更して、計算に対するテストを追加すると、エラー条件を検出し、関数の途中で処理を戻すようになるかもしれません。この途中での終了が起きるとき、確保されたメモリブロックは解放し忘れやすいのです。コードが後で追加された場合には特にそうです。このようなメモリリークが一旦紛れ込んでしまうと、長い間検出されないままになることがよくあります: エラーによる関数の終了は、全ての関数呼び出しのに対してほんのわずかな割合しか起きず、その一方でほとんどの近代的な計算機は相当量の仮想記憶を持っているため、メモリリークが明らかになるのは、長い間動作していたプロセスがリークを起こす関数を何度も使った場合に限られるからです。従って、この種のエラーを最小限にとどめるようなコーディング規約や戦略を設けて、不慮のメモリリークを避けることが重要なのです。

Python は `malloc()` や `free()` を非常によく利用するため、メモリリークの防止に加え、解放されたメモリの使用を防止する戦略が必要です。このために選ばれたのが参照カウント法 (*reference counting*) と呼ばれる手法です。参照カウント法の原理は簡単です: 全てのオブジェクトにはカウンタがあり、オブジェクトに対する参照がどこかに保存されたらカウンタをインクリメントし、オブジェクトに対する参照が削除されたらデクリメントします。カウンタがゼロになったら、オブジェクトへの最後の参照が削除されたことになり、オブジェクトは解放されます。

もう一つの戦略は自動ガベージコレクション (*automatic garbage collection*) と呼ばれています。(参照カウント法はガベージコレクション戦略の一つとして挙げられることもあるので、二つを区別するために筆者は "自動 (automatic)" を使っています。) 自動ガベージコレクションの大きな利点は、ユーザが `free()` を明示的によばなくてよいことにあります。(速度やメモリの有効利用性も利点として主張されています --- が、これは確たる事実ではありません。) C における自動ガベージコレクションの欠点は、真に可搬性のあるガベージコレクタが存在しないということです。それに対し、参照カウント法は可搬性のある実装ができます (`malloc()` や `free()` を利用できるのが前提です --- C 標準はこれを保証しています)。いつの日か、十分可搬性のあるガベージコレクタが C で使えるようになるかもしれませんが、それまでは参照カウント法でやっていく以外にはないのです。

Python では、伝統的な参照カウント法の実装を行っている一方で、参照の循環を検出するために働く循環参照検出機構 (*cycle detector*) も提供しています。循環参照検出機構のおかげで、直接、間接にかかわらず循環参照の生

成を気にせずにアプリケーションを構築できます; というのも、参照カウント法だけを使ったガベージコレクション実装にとって循環参照は弱点だからです。循環参照は、(間接参照の場合も含めて) 相互への参照が入ったオブジェクトから形成されるため、循環内のオブジェクトは各々非ゼロの参照カウントを持ちます。典型的な参照カウント法の実装では、たとえ循環参照を形成するオブジェクトに対して他に全く参照がないとしても、循環参照内のどのオブジェクトに属するメモリも再利用できません。

循環参照検出器は循環参照を形成しているゴミを見付け回収することができます。gc モジュールは、実行時に検出器を無効にする設定インターフェースだけでなく、検出器を走らせる手段 (`collect()` 関数) も提供します。循環参照検出器はオプションのコンポーネントだと見なされます; デフォルトでは含まれていますが、(Mac OS X を含む) Unix プラットフォームの `configure` スクリプトで `--without-cycle-gc` オプションを使うことで、ビルド時に無効化することができます。循環参照検出器がこの方法で無効化された場合、gc は利用できなくなります。

Python における参照カウント法

Python には、参照カウントのインクリメントやデクリメントを処理する二つのマクロ、`Py_INCREF(x)` と `Py_DECREF(x)` があります。`Py_DECREF()` は、参照カウントがゼロに到達した際に、オブジェクトのメモリ解放も行います。柔軟性を持たせるために、`free()` を直接呼び出しません --- その代わりにオブジェクトの型オブジェクト (*type object*) を介します。このために (他の目的もありますが)、全てのオブジェクトには自身の型オブジェクトに対するポインタが入っています。

さて、まだ重大な疑問が残っています: いつ `Py_INCREF(x)` や `Py_DECREF(x)` を使えばよいのでしょうか? まず、いくつかの用語説明から始めさせてください。まず、オブジェクトは "占有 (own)" されることはありません; しかし、あるオブジェクトに対する参照の所有 *own a reference* はできます。オブジェクトの参照カウントは、そのオブジェクトが参照の所有を受けている回数と定義されています。参照の所有者は、参照がなくなった際に `Py_DECREF()` を呼び出す役割を担います。参照の所有権は委譲 (transfer) できます。所有参照 (owned reference) の放棄には、渡す、保存する、`Py_DECREF()` を呼び出す、という三つの方法があります。所有参照を処理し忘れると、メモリリークを引き起こします。

オブジェクトに対する参照は、借用 (*borrow*) も可能です。^{*2} 参照の借用者は、`Py_DECREF()` を呼んではなりません。借用者は、参照の所有者から借用した期間を超えて参照を保持し続けてはなりません。所有者が参照を放棄した後で借用参照を使うと、解放済みメモリを使用してしまう危険があるので、絶対に避けねばなりません。^{*3}

参照の借用が参照の所有よりも優れている点は、コードがとりうるあらゆる処理経路で参照を廃棄しておくよう注意しなくて済むことです --- 別の言い方をすれば、借用参照の場合には、処理の途中で関数を終了してもメモリリークの危険を冒すことがない、ということです。逆に、所有よりも不利な点は、ごくまともに見えるコードが、実際には参照の借用元で放棄されてしまった後にその参照を使うかもしれないような微妙な状況があるということです。

`Py_INCREF()` を呼び出すと、借用参照を所有参照に変更できます。この操作は参照の借用元の状態には影響しま

^{*2} 参照を "借用する" というメタファは厳密には正しくありません: なぜなら、参照の所有者は依然として参照のコピーを持っているからです。

^{*3} 参照カウントが 1 以上かどうか調べる方法は **うまくいきません** --- 参照カウント自体も解放されたメモリ上にあるため、その領域が他のオブジェクトに使われている可能性があります!

せん --- `Py_INCREF()` は新たな所有参照を生成し、参照の所有者が担うべき全ての責任を課します (つまり、新たな参照の所有者は、以前の所有者と同様、参照の放棄を適切に行わねばなりません)。

所有権にまつわる規則

オブジェクトへの参照を関数の内外に渡す場合には、オブジェクトの所有権が参照と共に渡されるか否かが常に関数インタフェース仕様の一部となります。

オブジェクトへの参照を返すほとんどの関数は、参照とともに所有権も渡します。特に、`PyLong_FromLong()` や `Py_BuildValue()` のように、新しいオブジェクトを生成する関数は全て所有権を相手に渡します。オブジェクトが実際には新たなオブジェクトでなくても、そのオブジェクトに対する新たな参照の所有権を得ます。例えば、`PyLong_FromLong()` はよく使う値をキャッシュしており、キャッシュされた値への参照を返すことがあります。

`PyObject_GetAttrString()` のように、あるオブジェクトから別のオブジェクトを抽出するような関数もまた、参照とともに所有権を委譲します。こちらの方はやや理解しにくいかもしれませんが。というのはよく使われるルーチンのいくつかが例外となっているからです: `PyTuple_GetItem()`、`PyList_GetItem()`、`PyDict_GetItem()`、および `PyDict_GetItemString()` は全て、タプル、リスト、または辞書から借用参照を返します。

`PyImport_AddModule()` は、実際にはオブジェクトを生成して返すことがあるにもかかわらず、借用参照を返します: これが可能なのは、生成されたオブジェクトに対する所有参照は `sys.modules` に保持されるからです。

オブジェクトへの参照を別の関数に渡す場合、一般的には、関数側は呼び出し手から参照を借用します --- 参照を保存する必要があるなら、関数側は `Py_INCREF()` を呼び出して独立した所有者になります。とはいえ、この規則には二つの重要な例外: `PyTuple_SetItem()` と `PyList_SetItem()` があります。これらの関数は、渡された引数要素に対して所有権を乗っ取り (take over) ます --- たとえ失敗してもです! (`PyDict_SetItem()` とその仲間は所有権を乗っ取りません --- これらはいわば ” 普通の ” 関数です。)

Python から C 関数が呼び出される際には、C 関数は呼び出し側から引数への参照を借用します。C 関数の呼び出し側はオブジェクトへの参照を所有しているので、借用参照の生存期間が保証されるのは関数が処理を返すまでです。このようにして借用参照を保存したり他に渡したりしたい場合にのみ、`Py_INCREF()` を使って所有参照にする必要があります。

Python から呼び出された C 関数が返す参照は所有参照でなければなりません --- 所有権は関数から呼び出し側へと委譲されます。

薄氷

数少ない状況において、一見無害に見える借用参照の利用が問題をひきおこすことがあります。この問題はすべて、インタプリタが非明示的に呼び出され、インタプリタが参照の所有者に参照を放棄させてしまう状況と関係しています。

知っておくべきケースのうち最初の、そして最も重要なものは、リスト要素に対する参照を借りている際に起きる、関係ないオブジェクトに対する `Py_DECREF()` の使用です。例えば:

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

上の関数はまず、`list[0]` への参照を借用し、次に `list[1]` を値 0 で置き換え、最後にさきほど借用した参照を出力しています。何も問題ないように見えますね？ でもそうではないのです！

`PyList_SetItem()` の処理の流れを追跡してみましょう。リストは全ての要素に対して参照を所有しているので、要素 1 を置き換えると、以前の要素 1 を放棄します。ここで、以前の要素 1 がユーザ定義クラスのインスタンスであり、さらにこのクラスが `__del__()` メソッドを定義していると仮定しましょう。このクラスインスタンスの参照カウントが 1 だった場合、リストが参照を放棄すると、インスタンスの `__del__()` メソッドが呼び出されます。

クラスは Python で書かれているので、`__del__()` は任意の Python コードを実行できます。この `__del__()` が `bug()` における `item` に何か不正なことをしているのでしょうか？ その通り！ `bug()` に渡したリストが `__del__()` メソッドから操作できるとすると、`del list[0]` の効果を持つような文を実行できてしまいます。もしこの操作で `list[0]` に対する最後の参照が放棄されてしまうと、`list[0]` に関連付けられていたメモリは解放され、結果的に `item` は無効な値になってしまいます。

問題の原因が分かれば、解決は簡単です。一時的に参照回数を増やせばよいのです。正しく動作するバージョンは以下ようになります：

```
void
no_bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);

    Py_INCREF(item);
    PyList_SetItem(list, 1, PyLong_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

これは実際にあった話です。以前のバージョンの Python には、このバグの一種が潜っていて、`__del__()` メソッドがどうしてもうまく動かないのかを調べるために C デバッガで相当時間を費やした人がいました...

二つ目は、借用参照がスレッドに関係しているケースです。通常は、Python インタプリタにおける複数のスレッドは、グローバルインタプリタロックがオブジェクト空間全体を保護しているため、互いに邪魔し合うことはありません。とはいえ、ロックは `Py_BEGIN_ALLOW_THREADS` マクロで一時的に解除したり、`Py_END_ALLOW_THREADS` で再獲得したりできます。これらのマクロはブロックの起こる I/O 呼び出しの周囲によく置かれ、I/O が完了す

るまでの間に他のスレッドがプロセッサを利用できるようにします。明らかに、以下の関数は上の例と似た問題をはらんでいます:

```
void
bug(PyObject *list)
{
    PyObject *item = PyList_GetItem(list, 0);
    Py_BEGIN_ALLOW_THREADS
    ...some blocking I/O call...
    Py_END_ALLOW_THREADS
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

NULL ポインタ

一般論として、オブジェクトへの参照を引数にとる関数はユーザが NULL ポインタを渡すとは予想しておらず、渡そうとするとコアダンプになる (か、あとでコアダンプを引き起こす) ことでしょう。一方、オブジェクトへの参照を返すような関数は一般に、例外の発生を示す場合にのみ NULL を返します。引数に対して NULL テストを行わない理由は、関数はしばしば受け取ったオブジェクトを他の関数へと引き渡すからです --- 各々の関数が NULL テストを行えば、冗長なテストが大量に行われ、コードはより低速に動くことになります。

従って、NULL のテストはオブジェクトの ”発生源”、すなわち値が NULL になるかもしれないポインタを受け取ったときだけにしましょう。malloc() や、例外を送出する可能性のある関数がその例です。

マクロ Py_INCREF() および Py_DECREF() は NULL ポインタのチェックを行いません --- しかし、これらのマクロの変化形である Py_XINCREF() および Py_XDECREF() はチェックを行います。

特定のオブジェクト型について調べるマクロ (Pytype_Check()) は NULL ポインタのチェックを行いません --- 繰り返しますが、様々な異なる型を想定してオブジェクトの型を調べる際には、こうしたマクロを続けて呼び出す必要があるので、個別に NULL ポインタのチェックをすると冗長なテストになってしまうのです。型を調べるマクロには、NULL チェックを行う変化形はありません。

Python から C 関数を呼び出す機構は、C 関数に渡される引数リスト (例でいうところの args) が決して NULL にならないよう保証しています --- 実際には、常にタプル型になるよう保証しています。^{*4}

NULL ポインタを Python ユーザレベルに ”逃がし” てしまうと、深刻なエラーを引き起こします。

^{*4} ”旧式の” 呼び出し規約を使っている場合には、この保証は適用されません --- 既存のコードにはいまだに旧式の呼び出し規約が多々あります。

2.1.11 C++ での拡張モジュール作成

C++ でも拡張モジュールは作成できます。ただしいくつか制限があります。メインプログラム (Python インタプリタ) は C コンパイラでコンパイルされリンクされているので、グローバル変数や静的オブジェクトをコンストラクタで作成できません。メインプログラムが C++ コンパイラでリンクされているならこれは問題ではありません。Python インタプリタから呼び出される関数 (特にモジュール初期化関数) は、`extern "C"` を使って宣言しなければなりません。また、Python ヘッダファイルを `extern "C" {...}` に入れる必要はありません--- シンボル `__cplusplus` (最近の C++ コンパイラは全てこのシンボルを定義しています) が定義されているときに `extern "C" {...}` が行われるように、ヘッダファイル内にすでに書かれているからです。

2.1.12 拡張モジュールに C API を提供する

多くの拡張モジュールは単に Python から使える新たな関数や型を提供するだけですが、時に拡張モジュール内のコードが他の拡張モジュールでも便利ことがあります。例えば、あるモジュールでは順序概念のないリストのように動作する "コレクション (collection)" クラスを実装しているかもしれません。ちょうどリストを生成したり操作したりできる C API を備えた標準の Python リスト型のように、この新たなコレクション型も他の拡張モジュールから直接操作できるようにするには一連の C 関数を持つていなければなりません。

一見するとこれは簡単なこと: 単に関数を (もちろん `static` などとは宣言せずに) 書いて、適切なヘッダファイルを提供し、C API を書けばよいだけ、に思えます。そして実際のところ、全ての拡張モジュールが Python インタプリタに常に静的にリンクされている場合にはうまく動作します。ところがモジュールが共有ライブラリの場合には、一つのモジュールで定義されているシンボルが他のモジュールから不可視なことがあります。可視性の詳細はオペレーティングシステムによります; あるシステムは Python インタプリタと全ての拡張モジュール用に単一のグローバルな名前空間を用意しています (例えば Windows)。別のシステムはモジュールのリンク時に取り込まれるシンボルを明示的に指定する必要があります (AIX がその一例です)、また別のシステム (ほとんどの Unix) では、違った戦略を選択肢として提供しています。そして、たとえシンボルがグローバル変数として可視であっても、呼び出したい関数の入ったモジュールがまだロードされていないことだってあります!

従って、可搬性の点からシンボルの可視性には何ら仮定をしてはならないことになります。つまり拡張モジュール中の全てのシンボルは `static` と宣言せねばなりません。例外はモジュールの初期化関数で、これは ([モジュールのメソッドテーブルと初期化関数](#) で述べたように) 他の拡張モジュールとの間で名前が衝突するのを避けるためです。また、他の拡張モジュールからアクセスを **受けるべきではない** シンボルは別のやり方で公開せねばなりません。

Python はある拡張モジュールの C レベルの情報 (ポインタ) を別のモジュールに渡すための特殊な機構: Capsule (カプセル) を提供しています。Capsule はポインタ (`void *`) を記憶する Python のデータ型です。Capsule は C API を介してのみ生成したりアクセスしたりできますが、他の Python オブジェクトと同じように受け渡します。とりわけ、Capsule は拡張モジュールの名前空間内にある名前に代入できます。他の拡張モジュールはこのモジュールを `import` でき、次に名前を取得し、最後に Capsule へのポインタを取得します。

拡張モジュールの C API を公開するために、様々な方法で Capsule が使われます。各関数を 1 つのオブジェクトに入れたり、全ての C API のポインタ配列を Capsule に入れることができます。そして、ポインタに対する

保存や取得といった様々な作業は、コードを提供しているモジュールとクライアントモジュールとの間では異なる方法で分散できます。

どの方法を選ぶにしても、Capsule の name を正しく設定することは重要です。PyCapsule_New() は name 引数 (const char *) を取ります。NULL を name に渡すことも許可されていますが、name を設定することを強く推奨します。正しく名前を付けられた Capsule はある程度の実行時型安全性を持ちます。名前を付けられていない Capsule を他の Capsule と区別する現実的な方法はありません。

特に、C API を公開するための Capsule には次のルールに従った名前を付けるべきです:

```
modulename.attributename
```

PyCapsule_Import() という便利関数は、Capsule の名前がこのルールに一致しているときにのみ、簡単に Capsule 経由で公開されている C API をロードすることができます。この挙動により、C API のユーザーが、確実に正しい C API を格納している Capsule をロードできたことを確かめることができます。

以下の例では、名前を公開するモジュールの作者にほとんどの負荷が掛かりますが、よく使われるライブラリを作る際に適切なアプローチを実演します。このアプローチでは、全ての C API ポインタ (例中では一つだけですが!) を、Capsule の値となる void ポインタの配列に保存します。拡張モジュールに対応するヘッダファイルは、モジュールの import と C API ポインタを取得するよう手配するマクロを提供します; クライアントモジュールは、C API にアクセスする前にこのマクロを呼ぶだけです。

名前を公開する側のモジュールは、[簡単な例](#) 節の spam モジュールを修正したものです。関数 spam.system() は C ライブラリ関数 system() を直接呼び出さず、PySpam_System() を呼び出します。この関数はもちろん、実際には (全てのコマンドに "spam" を付けるといったような) より込み入った処理を行います。この関数 PySpam_System() はまた、他の拡張モジュールにも公開されます。

関数 PySpam_System() は、他の全ての関数と同様に static で宣言された通常の C 関数です:

```
static int
PySpam_System(const char *command)
{
    return system(command);
}
```

spam_system() には取るに足らない変更が施されています:

```
static PyObject *
spam_system(PyObject *self, PyObject *args)
{
    const char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = PySpam_System(command);
```

(次のページに続く)

(前のページからの続き)

```
    return PyLong_FromLong(sts);
}
```

モジュールの先頭にある以下の行

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

の直後に、以下の二行を必ず追加してください:

```
#define SPAM_MODULE
#include "spammodule.h"
```

`#define` は、ファイル `spammodule.h` をインクルードしているのが名前を公開する側のモジュールであって、クライアントモジュールではないことをヘッダファイルに教えるために使われます。最後に、モジュールの初期化関数は C API のポインタ配列を初期化するよう手配しなければなりません:

```
PyMODINIT_FUNC
PyInit_spam(void)
{
    PyObject *m;
    static void *PySpam_API[PySpam_API_pointers];
    PyObject *c_api_object;

    m = PyModule_Create(&spammodule);
    if (m == NULL)
        return NULL;

    /* Initialize the C API pointer array */
    PySpam_API[PySpam_System_NUM] = (void *)PySpam_System;

    /* Create a Capsule containing the API pointer array's address */
    c_api_object = PyCapsule_New((void *)PySpam_API, "spam._C_API", NULL);

    if (PyModule_AddObject(m, "_C_API", c_api_object) < 0) {
        Py_XDECREF(c_api_object);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}
```

`PySpam_API` が `static` と宣言されていることに注意してください; そうしなければ、`PyInit_spam()` が終了したときにポインタアレイは消滅してしまいます!

からくりの大部分はヘッダファイル `spammodule.h` 内にあり、以下のようになっています:


```

#ifndef Py_SPAMMODULE_H
#define Py_SPAMMODULE_H
#ifdef __cplusplus
extern "C" {
#endif

/* Header file for spammodule */

/* C API functions */
#define PySpam_System_NUM 0
#define PySpam_System_RETURN int
#define PySpam_System_PROTO (const char *command)

/* Total number of C API pointers */
#define PySpam_API_pointers 1

#ifdef SPAM_MODULE
/* This section is used when compiling spammodule.c */

static PySpam_System_RETURN PySpam_System PySpam_System_PROTO;

#else
/* This section is used in modules that use spammodule's API */

static void **PySpam_API;

#define PySpam_System \
    (*(PySpam_System_RETURN (*)(PySpam_System_PROTO) PySpam_API[PySpam_System_NUM])

/* Return -1 on error, 0 on success.
 * PyCapsule_Import will set an exception if there's an error.
 */
static int
import_spam(void)
{
    PySpam_API = (void **)PyCapsule_Import("spam._C_API", 0);
    return (PySpam_API != NULL) ? 0 : -1;
}

#endif

#ifdef __cplusplus
}
#endif

#endif /* !defined(Py_SPAMMODULE_H) */

```

PySpam_System() へのアクセス手段を得るためにクライアントモジュール側がしなければならないことは、初期

化関数内での `import_spam()` 関数 (またはマクロ) の呼び出しです:

```
PyMODINIT_FUNC
PyInit_client(void)
{
    PyObject *m;

    m = PyModule_Create(&clientmodule);
    if (m == NULL)
        return NULL;
    if (import_spam() < 0)
        return NULL;
    /* additional initialization can happen here */
    return m;
}
```

このアプローチの主要な欠点は、`spammodule.h` がやや難解になるということです。とはいえ、各関数の基本的な構成は公開されるものと同じなので、書き方を一度だけ学べば済みます。

最後に、Capsule は、自身に保存されているポインタをメモリ確保したり解放したりする際に特に便利な、もう一つの機能を提供しているということに触れておかねばなりません。詳細は Python/C API リファレンスマニュアルの `capsules`、および Capsule の実装部分 (Python ソースコード配布物中のファイル `Include/pycapsule.h` および `Objects/pycapsule.c` に述べられています。

脚注

2.2 拡張の型の定義: チュートリアル

Python では、組み込みの `str` 型や `list` 型のような Python コードから走査できる新しい型を C 拡張モジュールの作者が定義できます。全ての拡張の型のコードはあるパターンに従うのですが、書き始める前に理解しておくべき細かいことがあります。このドキュメントはその話題についてのやさしい入門です。

2.2.1 基本的なこと

CPython ランタイムは Python の全てのオブジェクトを `PyObject*` 型の変数と見なします。`PyObject*` は Python の全てのオブジェクトの "基底型 (base type)" となっています。`PyObject` 構造体自身は [参照カウント](#) と、オブジェクトの "型オブジェクト (type object)" へのポインタのみを持ちます。ここには動作が定義されています; 型オブジェクトは、例えば、ある属性があるオブジェクトから検索されたり、メソッドが呼ばれたり、他のオブジェクトによって操作されたりしたときに、どの (C) 関数がインタプリタから呼ばれるのかを決定します。これらの C 関数は "タイプメソッド (type method)" と呼ばれます。

それなので、新しい拡張の型を定義したいときは、新しい型オブジェクトを作成すればよいわけです。

この手のことは例を見たほうが早いでしょうから、以下に C 拡張モジュール `custom` にある `Custom` という名前の新しい型を定義する、最小限ながら完全なモジュールをあげておきます:

注釈: ここで紹介している例は、**静的な** 拡張の型を定義する伝統的な実装方法です。これはほとんどの場面で十分なもののなのです。C API では、`PyType_FromSpec()` 関数を使い、ヒープ上に配置された拡張の型も定義できますが、これについてはこのチュートリアルでは扱いません。

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

typedef struct {
    PyObject_HEAD
    /* Type-specific fields go here. */
} CustomObject;

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = PyType_GenericNew,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
    }
}
```

(次のページに続く)

(前のページからの続き)

```
Py_DECREF(m);
return NULL;
}

return m;
}
```

一度に把握するにはちょっと量が多いですが、前の章よりはとっつきやすくなっていることと重います。このファイルでは、3つの要素が定義されています:

1. **Custom オブジェクト** が何を含んでいるか: これが `CustomObject` 構造体で、`Custom` インスタンスごとに1回だけメモリ確保が行われます。
2. **Custom 型** がどのように振る舞うか: これが `CustomType` 構造体で、フラグと関数ポインタの集まりを定義しています。特定の操作が要求されたときに、この関数ポインタをインタプリタが見に行きます。
3. **custom モジュール** をどう初期化するか: これが `PyInit_custom` 関数とそれに関する `custommodule` 構造体です。

まず最初はこれです:

```
typedef struct {
    PyObject_HEAD
} CustomObject;
```

これが `Custom` オブジェクトの内容です。`PyObject_HEAD` はそれぞれのオブジェクト構造体の先頭に必須なもので、`PyObject` 型の `ob_base` という名前のフィールドを定義します。`PyObject` 型には (それぞれ `Py_REFCNT` マクロおよび `Py_TYPE` マクロからアクセスできる) 型オブジェクトへのポインタと参照カウントが格納されています。このマクロが用意されている理由は、構造体のレイアウトを抽象化し、デバッグビルドでフィールドを追加できるようにするためです。

注釈: 上の例では `PyObject_HEAD` マクロの後にセミコロンはありません。うっかりセミコロンを追加しないように気を付けてください: これを警告するコンパイラもあります。

もちろん、一般的にはオブジェクトは標準的な `PyObject_HEAD` ボイラープレートの他にもデータを保持しています; 例えば、これは Python 標準の浮動小数点数の定義です:

```
typedef struct {
    PyObject_HEAD
    double ob_fval;
} PyFloatObject;
```

2つ目は型オブジェクトの定義です。

```
static PyObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT,
    .tp_new = PyType_GenericNew,
};
```

注釈: 上にあるように C99 スタイルの指示付き初期化子を使って、PyObject の特に関心の無いフィールドまで全て並べたり、フィールドを宣言する順序に気を使ったりせずに済ませるのをお勧めします。

object.h にある実際の PyObject の定義には上の定義にあるよりもっと多くの フィールド があります。ここに出てきていないフィールドは C コンパイラによってゼロで埋められるので、必要でない限り明示的には値の指定をしないのが一般的な作法になっています。

一度に 1 つずつフィールドを取り上げていきましょう:

```
PyVarObject_HEAD_INIT(NULL, 0)
```

この行は、上で触れた ob_base フィールドの初期化に必須のボイラープレートです。

```
.tp_name = "custom.Custom",
```

実装している型の名前です。これは、オブジェクトのデフォルトの文字列表現やエラーメッセージに現れます。例えば次の通りです:

```
>>> "" + custom.Custom()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "custom.Custom") to str
```

型の名前が、モジュール名とモジュールにおける型の名前の両方をドットでつないだ名前になっていることに注意してください。この場合は、モジュールは custom で型は Custom なので、型の名前を custom.Custom に設定しました。実際のドット付きのインポートパスを使うのは、pydoc モジュールや pickle モジュールと互換性を持たせるために重要なのです。

```
.tp_basicsize = sizeof(CustomObject),
.tp_itemsize = 0,
```

tp_basicsize は、新しい Custom インスタンスを作るとき Python が割り当てるべきメモリがどのくらいなのかを知るためのものです。tp_itemsize は可変サイズのオブジェクトでのみ使うものなので、サイズが可変でな

いオブジェクトでは 0 にすべきです。

注釈: あなたのタイプを Python でサブクラス化可能にしたい場合、そのタイプが基底タイプと同じ `tp_basicsize` をもっていると多重継承のときに問題が生じることがあります。そのタイプを Python のサブクラスにしたとき、その `__bases__` リストにはあなたのタイプが最初にくるようにしなければなりません。さもないとエラーの発生なしにあなたのタイプの `__new__()` メソッドを呼び出すことはできなくなります。この問題を回避するには、つねにあなたのタイプの `tp_basicsize` をその基底タイプよりも大きくしておくことです。ほとんどの場合、あなたのタイプは `object` か、そうでなければ基底タイプにデータ用のメンバを追加したものでしょうから、したがって大きさはつねに増加するためこの条件は満たされています。

`Py_TPFLAGS_DEFAULT` にクラスフラグを設定します。

```
.tp_flags = Py_TPFLAGS_DEFAULT,
```

すべての型はフラグにこの定数を含めておく必要があります。これは最低でも Python 3.3 ままでに定義されているすべてのメンバを許可します。それ以上のメンバが必要なら、対応するフラグの OR をとる必要があります。

この型の docstring は `tp_doc` に入れます。

```
.tp_doc = "Custom objects",
```

オブジェクトが生成できるように、`tp_new` ハンドラを提供する必要があります。これは Python のメソッド `__new__()` と同等のものですが、明示的に与える必要があります。今の場合は、API 関数の `PyType_GenericNew()` として提供されるデフォルトをそのまま使えます。

```
.tp_new = PyType_GenericNew,
```

ファイルの残りの部分はきっと馴染みやすいものだと思いますが、`PyInit_custom()` の一部のコードはそうではないでしょう:

```
if (PyType_Ready(&CustomType) < 0)
    return;
```

これは、NULL に初期化された `ob_type` も含めて、いくつかのメンバーを適切なデフォルト値で埋めて、`Custom` 型を初期化します。

```
Py_INCREF(&CustomType);
if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
    Py_DECREF(&CustomType);
    Py_DECREF(m);
    return NULL;
}
```

これは型をモジュールの辞書に追加します。こうすることで Custom クラスの呼び出しで Custom インスタンスが作成できるようになります:

```
>>> import custom
>>> mycustom = custom.Custom()
```

以上です! 残りの作業はビルドだけです; custom.c という名前のファイルにここまでのコードを書き込み、次の内容を持つ setup.py を用意します:

```
from distutils.core import setup, Extension
setup(name="custom", version="1.0",
      ext_modules=[Extension("custom", ["custom.c"])])
```

そして、シェルから以下のように入力します

```
$ python setup.py build
```

すると custom.so ファイルがサブディレクトリに生成されます。そのディレクトリに移動して、Python を起動します -- これで import custom して、Custom オブジェクトで遊べるようになっているはずです。

そんなにむずかしくありません、よね?

もちろん、現在の Custom 型は面白みに欠けています。何もデータを持っていないし、何もできません。継承してサブクラスを作ることさえできないのです。

注釈: この文書では、標準の distutils モジュールを使って C 拡張をビルドしていますが、現実のユースケースでは、より新しく、保守されている setuptools ライブラリを利用することを推奨します。これを行う方法を文書化することはこのドキュメントの範囲外ですので、[Python Packaging User's Guide](#) を参照してください。

2.2.2 基本のサンプルにデータとメソッドを追加する

この基本のサンプルにデータとメソッドを追加してみましょう。ついでに、この型を基底クラスとしても利用できるようにします。ここでは新しいモジュール custom2 をつくり、これらの機能を追加します:

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

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
```

(次のページに続く)

(前のページからの続き)

```

    int number;
} CustomObject;

static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwargs)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwargs)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwargs, "|OOi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
    }
}

```

(次のページに続く)

(前のページからの続き)

```

        Py_XDECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"first", T_OBJECT_EX, offsetof(CustomObject, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(CustomObject, last), 0,
     "last name"},
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom2.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),

```

(次のページに続く)

(前のページからの続き)

```

    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_new = Custom_new,
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom2",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom2(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}

```

このバージョンでは、いくつかの変更をおこないます。

以下の include を追加します:

```
#include <structmember.h>
```

すこしあとでふれますが、この include には属性を扱うための宣言が入っています。

Custom 型は その C 構造体に 3 つのデータ属性 *first*、*last*、および *number* をもつようになりました。*first* と *last* 属性はファーストネームとラストネームを格納した Python 文字列で、*number* 属性は C 言語での整数の値

です。

これにしたがうと、オブジェクトの構造体は次のようになります:

```
typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;
```

いまや管理すべきデータができたので、オブジェクトの割り当てと解放に際してはより慎重になる必要があります。最低限、オブジェクトの解放メソッドが必要です:

```
static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

この関数は `tp_dealloc` メンバに代入されます。

```
.tp_dealloc = (destructor) Custom_dealloc,
```

このメソッドは、まず二つの Python 属性の参照カウントをクリアします。 `Py_XDECREF()` は引数が `NULL` のケースを正しく扱えます (これは、`tp_new` が途中で失敗した場合に起こりえます)。このメソッドは、つぎにオブジェクトの型 (`Py_TYPE(self)` で算出します) のメンバ `tp_free` を呼び出し、オブジェクトのメモリを開放します。オブジェクトの型が `CustomType` であるとは限らない点に注意してください。なぜなら、オブジェクトはサブクラスのインスタンスかもしれないからです。

注釈: 上の destructor への明示的な型変換は必要です。なぜなら、`Custom_dealloc` が `CustomObject *` 引数をとると定義しましたが、`tp_dealloc` 関数のポインタは `PyObject *` 引数を受け取ることになっているからです。もし明示的に型変換をしなければ、コンパイラが警告を発するでしょう。これは、C におけるオブジェクト指向のポリモーフィズムです!

ファーストネームとラストネームを空文字列に初期化しておきたいので、`tp_new` の実装を追加することにししょう:

```
static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwargs)
{
    CustomObject *self;
```

(次のページに続く)

(前のページからの続き)

```

self = (CustomObject *) type->tp_alloc(type, 0);
if (self != NULL) {
    self->first = PyUnicode_FromString("");
    if (self->first == NULL) {
        Py_DECREF(self);
        return NULL;
    }
    self->last = PyUnicode_FromString("");
    if (self->last == NULL) {
        Py_DECREF(self);
        return NULL;
    }
    self->number = 0;
}
return (PyObject *) self;
}
    
```

そしてこれを `tp_new` メンバとしてインストールします:

```

.tp_new = Custom_new,
    
```

The `tp_new` handler is responsible for creating (as opposed to initializing) objects of the type. It is exposed in Python as the `__new__()` method. It is not required to define a `tp_new` member, and indeed many extension types will simply reuse `PyType_GenericNew()` as done in the first version of the `Custom` type above. In this case, we use the `tp_new` handler to initialize the `first` and `last` attributes to non-NULL default values.

`tp_new` is passed the type being instantiated (not necessarily `CustomType`, if a subclass is instantiated) and any arguments passed when the type was called, and is expected to return the instance created. `tp_new` handlers always accept positional and keyword arguments, but they often ignore the arguments, leaving the argument handling to initializer (a.k.a. `tp_init` in C or `__init__` in Python) methods.

注釈: `tp_new` は明示的に `tp_init` を呼び出してはいけません、これはインタプリタが自分で行うためです。

この `tp_new` の実装は、`tp_alloc` スロットを呼び出してメモリを割り当てます:

```

self = (CustomObject *) type->tp_alloc(type, 0);
    
```

Since memory allocation may fail, we must check the `tp_alloc` result against NULL before proceeding.

注釈: We didn't fill the `tp_alloc` slot ourselves. Rather `PyType_Ready()` fills it for us by inheriting it from our base class, which is `object` by default. Most types use the default allocation strategy.

注釈: もし協力的な `tp_new` (基底タイプの `tp_new` または `__new__()` を呼んでいるもの) を作りたいのならば、実行時のメソッド解決順序をつかってどのメソッドを呼び出すかを決定しようとしては **いけません**。つねに呼び出す型を静的に決めておき、直接その `tp_new` を呼び出すか、あるいは `type->tp_base->tp_new` を経由してください。こうしないと、あなたが作成したタイプの Python サブクラスが他の Python で定義されたクラスも継承している場合にうまく動かない場合があります。(とりわけ、そのようなサブクラスのインスタンスを `TypeError` を出さずに作ることが不可能になります。)

We also define an initialization function which accepts arguments to provide initial values for our instance:

```
static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwargs)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwargs, "|OOi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_XDECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_XDECREF(tmp);
    }
    return 0;
}
```

これは `tp_init` メンバに代入されます。

```
.tp_init = (initproc) Custom_init,
```

The `tp_init` slot is exposed in Python as the `__init__()` method. It is used to initialize an object after it's created. Initializers always accept positional and keyword arguments, and they should return either 0 on success or -1 on error.

Unlike the `tp_new` handler, there is no guarantee that `tp_init` is called at all (for example, the `pickle` module by default doesn't call `__init__()` on unpickled instances). It can also be called multiple times.

Anyone can call the `__init__()` method on our objects. For this reason, we have to be extra careful when assigning the new attribute values. We might be tempted, for example to assign the `first` member like this:

```
if (first) {
    Py_XDECREF(self->first);
    Py_INCREF(first);
    self->first = first;
}
```

But this would be risky. Our type doesn't restrict the type of the `first` member, so it could be any kind of object. It could have a destructor that causes code to be executed that tries to access the `first` member; or that destructor could release the *Global interpreter Lock* and let arbitrary code run in other threads that accesses and modifies our object.

To be paranoid and protect ourselves against this possibility, we almost always reassign members before decrementing their reference counts. When don't we have to do this?

- その参照カウントが 1 より大きいと確信できる場合
- when we know that deallocation of the object^{*1} will neither release the *GIL* nor cause any calls back into our type's code;
- when decrementing a reference count in a `tp_dealloc` handler on a type which doesn't support cyclic garbage collection^{*2}.

ここではインスタンス変数を属性として見えるようにしたいのですが、これにはいくつかの方法があります。もっとも簡単な方法は、メンバの定義を与えることです:

```
static PyMemberDef Custom_members[] = {
    {"first", T_OBJECT_EX, offsetof(CustomObject, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(CustomObject, last), 0,
     "last name"},
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};
```

そして、この定義を `tp_members` スロットに入れましょう:

```
.tp_members = Custom_members,
```

Each member definition has a member name, type, offset, access flags and documentation string. See the [総称的な属性を管理する](#) section below for details.

^{*1} これはそのオブジェクトが文字列や実数などの基本タイプであるような時に成り立ちます。

^{*2} We relied on this in the `tp_dealloc` handler in this example, because our type doesn't support garbage collection.

A disadvantage of this approach is that it doesn't provide a way to restrict the types of objects that can be assigned to the Python attributes. We expect the first and last names to be strings, but any Python objects can be assigned. Further, the attributes can be deleted, setting the C pointers to NULL. Even though we can make sure the members are initialized to non-NULL values, the members can be set to NULL if the attributes are deleted.

ここでは `Custom.name()` と呼ばれるメソッドを定義しましょう。これはファーストネーム `first` とラストネーム `last` を連結した文字列をそのオブジェクトの名前として返します。

```
static PyObject *
Custom_name(CustomObject *self)
{
    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }
    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}
```

このメソッドは C 関数として実装され、`Custom` (あるいは `Custom` のサブクラス) のインスタンスを第一引数として受けとります。メソッドはつねにそのインスタンスを最初の引数として受けとらなければなりません。しばしば位置引数とキーワード引数も受けとりますが、今回はなにも必要ないので、固定引数のタプルもキーワード引数の辞書も取らないことにします。このメソッドは Python の以下のメソッドと等価です:

```
def name(self):
    return "%s %s" % (self.first, self.last)
```

Note that we have to check for the possibility that our `first` and `last` members are NULL. This is because they can be deleted, in which case they are set to NULL. It would be better to prevent deletion of these attributes and to restrict the attribute values to be strings. We'll see how to do that in the next section.

さて、メソッドを定義したので、ここでメソッド定義用の配列を作成する必要があります:

```
static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};
```

(note that we used the `METH_NOARGS` flag to indicate that the method is expecting no arguments other than `self`)

and assign it to the `tp_methods` slot:

```
.tp_methods = Custom_methods,
```

Finally, we'll make our type usable as a base class for subclassing. We've written our methods carefully so far so that they don't make any assumptions about the type of the object being created or used, so all we need to do is to add the `Py_TPFLAGS_BASETYPE` to our class flag definition:

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
```

We rename `PyInit_custom()` to `PyInit_custom2()`, update the module name in the `PyModuleDef` struct, and update the full class name in the `PyTypeObject` struct.

Finally, we update our `setup.py` file to build the new module:

```
from distutils.core import setup, Extension
setup(name="custom", version="1.0",
      ext_modules=[
          Extension("custom", ["custom.c"]),
          Extension("custom2", ["custom2.c"]),
      ])

```

2.2.3 データ属性をこまかく制御する

In this section, we'll provide finer control over how the `first` and `last` attributes are set in the `Custom` example. In the previous version of our module, the instance variables `first` and `last` could be set to non-string values or even deleted. We want to make sure that these attributes always contain strings.

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

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last; /* last name */
    int number;
} CustomObject;

static void
Custom_dealloc(CustomObject *self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

```

(次のページに続く)

(前のページからの続き)

```

}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwargs)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwargs)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwargs, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}

```

(次のページに続く)

(前のページからの続き)

```
}

static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(CustomObject *self, PyObject *value, void *closure)
{
    PyObject *tmp;
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }
    tmp = self->first;
    Py_INCREF(value);
    self->first = value;
    Py_DECREF(tmp);
    return 0;
}

static PyObject *
Custom_getlast(CustomObject *self, void *closure)
{
    Py_INCREF(self->last);
    return self->last;
}

static int
Custom_setlast(CustomObject *self, PyObject *value, void *closure)
{
    PyObject *tmp;
    if (value == NULL) {
```

(次のページに続く)

(前のページからの続き)

```

        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
            "The last attribute value must be a string");
        return -1;
    }
    tmp = self->last;
    Py_INCREF(value);
    self->last = value;
    Py_DECREF(tmp);
    return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom3.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_new = Custom_new,
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_members = Custom_members,

```

(次のページに続く)

(前のページからの続き)

```

        .tp_methods = Custom_methods,
        .tp_getset = Custom_getsetters,
    };

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom3",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom3(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}

```

first 属性と last 属性をよりこまかく制御するためには、カスタムメイドの getter 関数と setter 関数を使います。以下は first 属性から値を取得する関数 (getter) と、この属性に値を格納する関数 (setter) です:

```

static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(CustomObject *self, PyObject *value, void *closure)
{
    PyObject *tmp;
    if (value == NULL) {

```

(次のページに続く)

(前のページからの続き)

```

    PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
    return -1;
}
if (!PyUnicode_Check(value)) {
    PyErr_SetString(PyExc_TypeError,
                    "The first attribute value must be a string");
    return -1;
}
tmp = self->first;
Py_INCREF(value);
self->first = value;
Py_DECREF(tmp);
return 0;
}

```

The getter function is passed a `Custom` object and a "closure", which is a void pointer. In this case, the closure is ignored. (The closure supports an advanced usage in which definition data is passed to the getter and setter. This could, for example, be used to allow a single set of getter and setter functions that decide the attribute to get or set based on data in the closure.)

The setter function is passed the `Custom` object, the new value, and the closure. The new value may be `NULL`, in which case the attribute is being deleted. In our setter, we raise an error if the attribute is deleted or if its new value is not a string.

ここでは `PyGetSetDef` 構造体の配列をつくります:

```

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL} /* Sentinel */
};

```

そしてこれを `tp_getset` スロットに登録します:

```

.tp_getset = Custom_getsetters,

```

The last item in a `PyGetSetDef` structure is the "closure" mentioned above. In this case, we aren't using a closure, so we just pass `NULL`.

また、メンバ定義からはこれらの属性を除いておきましょう:

```

static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},

```

(次のページに続く)

(前のページからの続き)

```

    {NULL} /* Sentinel */
};

```

また、ここでは `tp_init` ハンドラも渡されるものとして文字列のみを許可するように修正する必要があります*3:

```

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwds)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwds, "|UUi", kwlist,
                                     &first, &last,
                                     &self->number))
        return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}

```

With these changes, we can assure that the `first` and `last` members are never `NULL` so we can remove checks for `NULL` values in almost all cases. This means that most of the `Py_XDECREF()` calls can be converted to `Py_DECREF()` calls. The only place we can't change these calls is in the `tp_dealloc` implementation, where there is the possibility that the initialization of these members failed in `tp_new`.

さて、先ほどもしたように、このモジュール初期化関数と初期化関数内にあるモジュール名を変更しましょう。そして `setup.py` ファイルに追加の定義をくわえます。

*3 We now know that the `first` and `last` members are strings, so perhaps we could be less careful about decrementing their reference counts, however, we accept instances of string subclasses. Even though deallocating normal strings won't call back into our objects, we can't guarantee that deallocating an instance of a string subclass won't call back into our objects.

2.2.4 循環ガベージコレクションをサポートする

Python は、`term:循環ガベージコレクタ (GC) 機能<garbage collection>` ‘`term:循環ガベージコレクタ (GC) 機能<garbage collection>`’ をもっており、これは不要なオブジェクトを、たとえ参照カウントがゼロでなくても発見することができます。そのような状況はオブジェクトの参照が循環しているときに起こりえます。たとえば以下の例を考えてください:

```
>>> l = []
>>> l.append(l)
>>> del l
```

この例では、自分自身をふくむリストを作りました。たとえこのリストを 削除しても、それは自分自身への参照をまだ持ちつづけますから、参照カウントはゼロにはなりません。嬉しいことに Python には循環ガベージコレクタは最終的にはこのリストが不要であることを検出し、解放できます。

In the second version of the `Custom` example, we allowed any kind of object to be stored in the `first` or `last` attributes^{*4}. Besides, in the second and third versions, we allowed subclassing `Custom`, and subclasses may add arbitrary attributes. For any of those two reasons, `Custom` objects can participate in cycles:

```
>>> import custom3
>>> class Derived(custom3.Custom): pass
...
>>> n = Derived()
>>> n.some_attribute = n
```

To allow a `Custom` instance participating in a reference cycle to be properly detected and collected by the cyclic GC, our `Custom` type needs to fill two additional slots and to enable a flag that enables these slots:

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

typedef struct {
    PyObject_HEAD
    PyObject *first; /* first name */
    PyObject *last;  /* last name */
    int number;
} CustomObject;

static int
Custom_traverse(CustomObject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->first);
    Py_VISIT(self->last);
}
```

(次のページに続く)

^{*4} Also, even with our attributes restricted to strings instances, the user could pass arbitrary `str` subclasses and therefore still create reference cycles.

(前のページからの続き)

```

    return 0;
}

static int
Custom_clear(CustomObject *self)
{
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}

static void
Custom_dealloc(CustomObject *self)
{
    PyObject_GC_UnTrack(self);
    Custom_clear(self);
    Py_TYPE(self)->tp_free((PyObject *) self);
}

static PyObject *
Custom_new(PyTypeObject *type, PyObject *args, PyObject *kwargs)
{
    CustomObject *self;
    self = (CustomObject *) type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyUnicode_FromString("");
        if (self->first == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->last = PyUnicode_FromString("");
        if (self->last == NULL) {
            Py_DECREF(self);
            return NULL;
        }
        self->number = 0;
    }
    return (PyObject *) self;
}

static int
Custom_init(CustomObject *self, PyObject *args, PyObject *kwargs)
{
    static char *kwlist[] = {"first", "last", "number", NULL};
    PyObject *first = NULL, *last = NULL, *tmp;

    if (!PyArg_ParseTupleAndKeywords(args, kwargs, "|UUi", kwlist,

```

(次のページに続く)

(前のページからの続き)

```

        &first, &last,
        &self->number))

    return -1;

    if (first) {
        tmp = self->first;
        Py_INCREF(first);
        self->first = first;
        Py_DECREF(tmp);
    }
    if (last) {
        tmp = self->last;
        Py_INCREF(last);
        self->last = last;
        Py_DECREF(tmp);
    }
    return 0;
}

static PyMemberDef Custom_members[] = {
    {"number", T_INT, offsetof(CustomObject, number), 0,
     "custom number"},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_getfirst(CustomObject *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
}

static int
Custom_setfirst(CustomObject *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The first attribute value must be a string");
        return -1;
    }
    Py_INCREF(value);
    Py_CLEAR(self->first);
    self->first = value;
}

```

(次のページに続く)

(前のページからの続き)

```

    return 0;
}

static PyObject *
Custom_getlast(CustomObject *self, void *closure)
{
    Py_INCREF(self->last);
    return self->last;
}

static int
Custom_setlast(CustomObject *self, PyObject *value, void *closure)
{
    if (value == NULL) {
        PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
        return -1;
    }
    if (!PyUnicode_Check(value)) {
        PyErr_SetString(PyExc_TypeError,
                        "The last attribute value must be a string");
        return -1;
    }
    Py_INCREF(value);
    Py_CLEAR(self->last);
    self->last = value;
    return 0;
}

static PyGetSetDef Custom_getsetters[] = {
    {"first", (getter) Custom_getfirst, (setter) Custom_setfirst,
     "first name", NULL},
    {"last", (getter) Custom_getlast, (setter) Custom_setlast,
     "last name", NULL},
    {NULL} /* Sentinel */
};

static PyObject *
Custom_name(CustomObject *self, PyObject *Py_UNUSED(ignored))
{
    return PyUnicode_FromFormat("%S %S", self->first, self->last);
}

static PyMethodDef Custom_methods[] = {
    {"name", (PyCFunction) Custom_name, METH_NOARGS,
     "Return the name, combining the first and last name"},
    {NULL} /* Sentinel */
};

```

(次のページに続く)

(前のページからの続き)

```

};

static PyTypeObject CustomType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "custom4.Custom",
    .tp_doc = "Custom objects",
    .tp_basicsize = sizeof(CustomObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
    .tp_new = Custom_new,
    .tp_init = (initproc) Custom_init,
    .tp_dealloc = (destructor) Custom_dealloc,
    .tp_traverse = (traverseproc) Custom_traverse,
    .tp_clear = (inquiry) Custom_clear,
    .tp_members = Custom_members,
    .tp_methods = Custom_methods,
    .tp_getset = Custom_getsetters,
};

static PyModuleDef custommodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "custom4",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_custom4(void)
{
    PyObject *m;
    if (PyType_Ready(&CustomType) < 0)
        return NULL;

    m = PyModule_Create(&custommodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&CustomType);
    if (PyModule_AddObject(m, "Custom", (PyObject *) &CustomType) < 0) {
        Py_DECREF(&CustomType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}

```

First, the traversal method lets the cyclic GC know about subobjects that could participate in cycles:

```
static int
Custom_traverse(CustomObject *self, visitproc visit, void *arg)
{
    int vret;
    if (self->first) {
        vret = visit(self->first, arg);
        if (vret != 0)
            return vret;
    }
    if (self->last) {
        vret = visit(self->last, arg);
        if (vret != 0)
            return vret;
    }
    return 0;
}
```

循環した参照に含まれるかもしれない各内部オブジェクトに対して、traversal メソッドに渡された `visit()` 関数を呼びます。`visit()` 関数は内部オブジェクトと、traversal メソッドに渡された追加の引数 `arg` を引数としてとります。この関数はこの値が非負の場合に返される整数の値を返します。

Python provides a `Py_VISIT()` macro that automates calling visit functions. With `Py_VISIT()`, we can minimize the amount of boilerplate in `Custom_traverse`:

```
static int
Custom_traverse(CustomObject *self, visitproc visit, void *arg)
{
    Py_VISIT(self->first);
    Py_VISIT(self->last);
    return 0;
}
```

注釈: The `tp_traverse` implementation must name its arguments exactly *visit* and *arg* in order to use `Py_VISIT()`.

Second, we need to provide a method for clearing any subobjects that can participate in cycles:

```
static int
Custom_clear(CustomObject *self)
{
    Py_CLEAR(self->first);
    Py_CLEAR(self->last);
    return 0;
}
```

Notice the use of the `Py_CLEAR()` macro. It is the recommended and safe way to clear data attributes of arbitrary types while decrementing their reference counts. If you were to call `Py_XDECREF()` instead on the attribute before setting it to `NULL`, there is a possibility that the attribute's destructor would call back into code that reads the attribute again (*especially* if there is a reference cycle).

注釈: You could emulate `Py_CLEAR()` by writing:

```
PyObject *tmp;
tmp = self->first;
self->first = NULL;
Py_XDECREF(tmp);
```

Nevertheless, it is much easier and less error-prone to always use `Py_CLEAR()` when deleting an attribute. Don't try to micro-optimize at the expense of robustness!

The deallocator `Custom_dealloc` may call arbitrary code when clearing attributes. It means the circular GC can be triggered inside the function. Since the GC assumes reference count is not zero, we need to untrack the object from the GC by calling `PyObject_GC_UnTrack()` before clearing members. Here is our reimplemented deallocator using `PyObject_GC_UnTrack()` and `Custom_clear`:

```
static void
Custom_dealloc(CustomObject *self)
{
    PyObject_GC_UnTrack(self);
    Custom_clear(self);
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

最後に、`Py_TPFLAGS_HAVE_GC` フラグをクラス定義のフラグに加えます:

```
.tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC,
```

これで完了です。`tp_alloc` スロットまたは `tp_free` ハンドラが書かれていれば、それらを循環ガベージコレクションに使えるよう修正すればよいのです。ほとんどの拡張機能は自動的に提供されるバージョンを使うでしょう。

2.2.5 他の型のサブクラスを作る

既存の型を継承した新しい拡張型を作成することができます。組み込み型から継承するのは特に簡単です。必要な `PyTypeObject` を簡単に利用できるからです。それに比べて、`PyTypeObject` 構造体を拡張モジュール間で共有するのは難しいです。

次の例では、ビルトインの `list` 型を継承した `SubList` 型を作成しています。新しい型は通常のリスト型と完全に互換性がありますが、追加で内部のカウンタを増やす `increment()` メソッドを持っています:

```
>>> import sublist
>>> s = sublist.SubList(range(3))
>>> s.extend(s)
>>> print(len(s))
6
>>> print(s.increment())
1
>>> print(s.increment())
2
```

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

typedef struct {
    PyListObject list;
    int state;
} SubListObject;

static PyObject *
SubList_increment(SubListObject *self, PyObject *unused)
{
    self->state++;
    return PyLong_FromLong(self->state);
}

static PyMethodDef SubList_methods[] = {
    {"increment", (PyCFunction) SubList_increment, METH_NOARGS,
     PyDoc_STR("increment state counter")},
    {NULL},
};

static int
SubList_init(SubListObject *self, PyObject *args, PyObject *kwds)
{
    if (PyList_Type.tp_init((PyObject *) self, args, kwds) < 0)
        return -1;
    self->state = 0;
    return 0;
}
```

(次のページに続く)

(前のページからの続き)

```

}

static PyTypeObject SubListType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    .tp_name = "sublist.SubList",
    .tp_doc = "SubList objects",
    .tp_basicsize = sizeof(SubListObject),
    .tp_itemsize = 0,
    .tp_flags = Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE,
    .tp_init = (initproc) SubList_init,
    .tp_methods = SubList_methods,
};

static PyModuleDef sublistmodule = {
    PyModuleDef_HEAD_INIT,
    .m_name = "sublist",
    .m_doc = "Example module that creates an extension type.",
    .m_size = -1,
};

PyMODINIT_FUNC
PyInit_sublist(void)
{
    PyObject *m;
    SubListType.tp_base = &PyList_Type;
    if (PyType_Ready(&SubListType) < 0)
        return NULL;

    m = PyModule_Create(&sublistmodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&SubListType);
    if (PyModule_AddObject(m, "SubList", (PyObject *) &SubListType) < 0) {
        Py_DECREF(&SubListType);
        Py_DECREF(m);
        return NULL;
    }

    return m;
}

```

見てわかるように、ソースコードは前の節の Custom の時と非常に似ています。違う部分をそれぞれを見ていきます。

```

typedef struct {
    PyListObject list;

```

(次のページに続く)

(前のページからの続き)

```

    int state;
} SubListObject;

```

The primary difference for derived type objects is that the base type's object structure must be the first value. The base type will already include the `PyObject_HEAD()` at the beginning of its structure.

When a Python object is a `SubList` instance, its `PyObject *` pointer can be safely cast to both `PyListObject *` and `SubListObject *`:

```

static int
SubList_init(SubListObject *self, PyObject *args, PyObject *kwargs)
{
    if (PyList_Type.tp_init((PyObject *) self, args, kwargs) < 0)
        return -1;
    self->state = 0;
    return 0;
}

```

We see above how to call through to the `__init__` method of the base type.

This pattern is important when writing a type with custom `tp_new` and `tp_dealloc` members. The `tp_new` handler should not actually create the memory for the object with its `tp_alloc`, but let the base class handle it by calling its own `tp_new`.

The `PyTypeObject` struct supports a `tp_base` specifying the type's concrete base class. Due to cross-platform compiler issues, you can't fill that field directly with a reference to `PyList_Type`; it should be done later in the module initialization function:

```

PyMODINIT_FUNC
PyInit_sublist(void)
{
    PyObject* m;
    SubListType.tp_base = &PyList_Type;
    if (PyType_Ready(&SubListType) < 0)
        return NULL;

    m = PyModule_Create(&sublistmodule);
    if (m == NULL)
        return NULL;

    Py_INCREF(&SubListType);
    if (PyModule_AddObject(m, "SubList", (PyObject *) &SubListType) < 0) {
        Py_DECREF(&SubListType);
        Py_DECREF(m);
        return NULL;
    }
}

```

(次のページに続く)

(前のページからの続き)

```

    return m;
}

```

PyType_Read() を呼ぶ前に、型の構造体の tp_base スロットは埋められていなければなりません。既存の型を継承する際には、tp_alloc スロットを PyType_GenericNew() で埋める必要はありません。-- 基底型のアロケーション関数が継承されます。

この後は、PyType_Ready() 関数を呼び、タイプオブジェクトをモジュールへ追加するのは、基本的な Custom の例と同じです。

脚注

2.3 Defining Extension Types: Assorted Topics

この節ではさまざまな実装可能なタイプメソッドと、それらが何をするものであるかについて、ざっと説明します。

以下は PyTypeObject の定義です。デバッグビルドでしか使われないいくつかのメンバは省いてあります:

```

typedef struct _typeobject {
    PyObject_VAR_HEAD
    const char *tp_name; /* For printing, in format "<module>.<name>" */
    Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    printfunc tp_print;
    getattrofunc tp_getattr;
    setattrofunc tp_setattr;
    PyAsyncMethods *tp_as_async; /* formerly known as tp_compare (Python 2)
                                   or tp_reserved (Python 3) */
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;

```

(次のページに続く)

(前のページからの続き)

```
reprfunc tp_str;
getattrofunc tp_getattro;
setattrofunc tp_setattro;

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

/* Flags to define presence of optional/expanded features */
unsigned long tp_flags;

const char *tp_doc; /* Documentation string */

/* call function for all accessible objects */
traverseproc tp_traverse;

/* delete references to contained objects */
inquiry tp_clear;

/* rich comparisons */
richcmpfunc tp_richcompare;

/* weak reference enabler */
Py_ssize_t tp_weaklistoffset;

/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;

/* Attribute descriptor and subclassing stuff */
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
struct _typeobject *tp_base;
PyObject *tp_dict;
descrgetfunc tp_descr_get;
descrsetfunc tp_descr_set;
Py_ssize_t tp_dictoffset;
initproc tp_init;
allocfunc tp_alloc;
newfunc tp_new;
freefunc tp_free; /* Low-level free-memory routine */
inquiry tp_is_gc; /* For PyObject_IS_GC */
PyObject *tp_bases;
PyObject *tp_mro; /* method resolution order */
PyObject *tp_cache;
PyObject *tp_subclasses;
PyObject *tp_weaklist;
```

(次のページに続く)

(前のページからの続き)

```

    destructor tp_del;

    /* Type attribute cache version tag. Added in version 2.6 */
    unsigned int tp_version_tag;

    destructor tp_finalize;

} PyTypeObject;

```

たくさんの メソッドがありますね。でもそんなに心配する必要はありません。定義したい型があるなら、実装するのはこのうちのごくわずかで済むことがほとんどです。

すでに予想されているでしょうが、この構造体について入念に見ていき、様々なハンドラについてより詳しい情報を提供します。しかしこれらのメンバが構造体中で定義されている順番は無視します。というのは、これらのメンバの現れる順序は歴史的な遺産によるものだからです。多くの場合いちばん簡単なのは、必要とするメンバがすべて含まれている例をとってきて、新しく作る型に合わせて値を変更することです。

```
const char *tp_name; /* For printing */
```

これは型の名前です。前の章で説明したように、これは色々な場面で現れ、ほとんどは診断目的で使われるものです。それなので、そのような場面で役に立つであろう名前を選んでください!

```
Py_ssize_t tp_basicsize, tp_itemsize; /* For allocation */
```

これらのフィールドは、この型のオブジェクトが新しく作成されるときにどれだけのメモリを割り当てればよいのかをランタイムに指示します。Python には可変長の構造体 (文字列やタプルなどを想像してください) に対する組み込みのサポートがある程度あり、ここで `tp_itemsize` メンバが使われます。これらについてはあとでふれます。

```
const char *tp_doc;
```

ここには Python スクリプトリファレンス `obj.__doc__` が doc string を返すときの文字列 (あるいはそのアドレス) を入れます。

では次に、型の基本的なメソッドに進みます。ほとんどの拡張の型がこのメソッドを実装します。

2.3.1 ファイナライズとメモリ解放

```
destructor tp_dealloc;
```

型のインスタンスの参照カウントがゼロになり、Python インタプリタがそれを潰して再利用したくなると、この関数が呼ばれます。解放すべきメモリをその型が保持していたり、それ以外にも実行すべき後処理がある場合は、それらをここに入れられます。オブジェクトそれ自体もここで解放される必要があります。この関数の例は、以下のようなものです:

```
static void
newdatatype_dealloc(newdatatypeobject *obj)
{
    free(obj->obj_UnderlyingDatatypePtr);
    Py_TYPE(obj)->tp_free(obj);
}
```

メモリ解放関数でひとつ重要なのは、処理待ちの例外にいっさい手をつけないことです。なぜなら、解放用の関数は Python インタプリタがスタックを元の状態に戻すときに呼ばれることが多いからです。そして (通常の関数からの復帰でなく) 例外のためにスタックが巻き戻されるときは、すでに発生している例外からメモリ解放関数を守るものはありません。解放用の関数がおこなう動作が追加の Python のコードを実行してしまうと、それらは例外が発生していることを検知するかもしれません。これはインタプリタが誤解させるエラーを発生させることにつながります。これを防ぐ正しい方法は、安全でない操作を実行する前に処理待ちの例外を保存しておき、終わったらそれを元に戻すことです。これは `PyErr_Fetch()` および `PyErr_Restore()` 関数を使うことによって可能になります:

```
static void
my_dealloc(PyObject *obj)
{
    PyObject *self = (PyObject *) obj;
    PyObject *cbresult;

    if (self->my_callback != NULL) {
        PyObject *err_type, *err_value, *err_traceback;

        /* This saves the current exception state */
        PyErr_Fetch(&err_type, &err_value, &err_traceback);

        cbresult = PyObject_CallObject(self->my_callback, NULL);
        if (cbresult == NULL)
            PyErr_WriteUnraisable(self->my_callback);
        else
            Py_DECREF(cbresult);

        /* This restores the saved exception state */
        PyErr_Restore(err_type, err_value, err_traceback);
    }
}
```

(次のページに続く)

(前のページからの続き)

```

        Py_DECREF(self->my_callback);
    }
    Py_TYPE(obj)->tp_free((PyObject*)self);
}

```

注釈: メモリ解放関数の中で安全に行えることにはいくつか制限があります。1つ目は、その型が (`tp_traverse` および `tp_clear` を使って) ガベージコレクションをサポートしている場合、`tp_dealloc` が呼び出されるまでに、消去されファイナライズされてしまうオブジェクトのメンバーが有り得ることです。2つ目は、`tp_dealloc` の中ではオブジェクトは不安定な状態にあることです: つまり参照カウントが0であるということです。(上の例にあるような) 複雑なオブジェクトや API の呼び出しでは、`tp_dealloc` を再度呼び出し、二重解放からクラッシュすることになるかもしれません。

Python 3.4 からは、複雑なファイナライズのコードは `tp_dealloc` に置かず、代わりに新しく導入された `tp_finalize` という型メソッドを使うことが推奨されています。

参考:

[PEP 442](#) で新しいファイナライズの仕組みが説明されています。

2.3.2 オブジェクト表現

Python では、オブジェクトの文字列表現を生成するのに 2 つのやり方があります: `repr()` 関数を使う方法と、`str()` 関数を使う方法です。(`print()` 関数は単に `str()` を呼び出します。) これらのハンドラはどちらも省略できます。

```

reprfunc tp_repr;
reprfunc tp_str;

```

`tp_repr` ハンドラは呼び出されたインスタンスの文字列表現を格納した文字列オブジェクトを返す必要があります。簡単な例は以下のようなものです:

```

static PyObject *
newdatatype_repr(newdatatypeobject * obj)
{
    return PyUnicode_FromFormat("Repr-ified_newdatatype{{size:%d}}",
                                obj->obj_UnderlyingDatatypePtr->size);
}

```

`tp_repr` ハンドラが指定されていなければ、インタプリタはその型の `tp_name` とそのオブジェクトの一意な識別値をもちいて文字列表現を作成します。

`tp_str` ハンドラと `str()` の関係は、上の `tp_repr` ハンドラと `repr()` の関係に相当します。つまり、これは Python のコードがオブジェクトのインスタンスに対して `str()` を呼び出したときに呼ばれます。この関数の実装は `tp_repr` ハンドラのそれと非常に似ていますが、得られる文字列表現は人間が読むことを意図されています。`tp_str` が指定されていない場合、かわりに `tp_repr` ハンドラが使われます。

以下は簡単な例です:

```
static PyObject *
newdatatype_str(newdatatypeobject * obj)
{
    return PyUnicode_FromFormat("Stringified_newdatatype{size:%d}",
                                obj->obj_UnderlyingDatatypePtr->size);
}
```

2.3.3 属性を管理する

For every object which can support attributes, the corresponding type must provide the functions that control how the attributes are resolved. There needs to be a function which can retrieve attributes (if any are defined), and another to set attributes (if setting attributes is allowed). Removing an attribute is a special case, for which the new value passed to the handler is `NULL`.

Python は 2 つの属性ハンドラの組をサポートしています。属性をもつ型はどちらか一組を実装するだけでよく、それらの違いは一方の組が属性の名前を `char*` として受け取るのに対してもう一方の組は属性の名前を `PyObject*` として受け取る、というものです。それぞれの型はその実装にとって都合がよい方を使えます。

```
getattrofunc tp_getattr;      /* char * version */
setattrofunc tp_setattr;
/* ... */
getattrofunc tp_getattro;     /* PyObject * version */
setattrofunc tp_setattro;
```

オブジェクトの属性へのアクセスがつねに (すぐあとで説明する) 単純な操作だけならば、`PyObject*` を使って属性を管理する関数として、総称的 (generic) な実装を使えます。特定の型に特化した属性ハンドラの必要性は Python 2.2 からほとんど完全になくなりました。しかし、多くの例はまだ、この新しく使えるようになった総称的なメカニズムを使うよう更新されてはいません。

総称的な属性を管理する

ほとんどの型は **単純な** 属性を使うだけです。では、どのような属性が単純だといえるのでしょうか？それが満たすべき条件はごくわずかです:

1. `PyType_Ready()` が呼ばれたとき、すでに属性の名前がわかっていること。
2. 属性を参照したり設定したりするときに、特別な記録のための処理が必要でなく、また参照したり設定した値に対してどんな操作も実行する必要がないこと。

これらの条件は、属性の値や、値が計算されるタイミング、または格納されたデータがどの程度妥当なものであるかといったことになんら制約を課すものではないことに注意してください。

When `PyType_Ready()` is called, it uses three tables referenced by the type object to create *descriptors* which are placed in the dictionary of the type object. Each descriptor controls access to one attribute of the instance object. Each of the tables is optional; if all three are `NULL`, instances of the type will only have attributes that are inherited from their base type, and should leave the `tp_getattro` and `tp_setattro` fields `NULL` as well, allowing the base type to handle attributes.

テーブルはタイプオブジェクト中の 3 つのメンバとして宣言されています:

```
struct PyMethodDef *tp_methods;
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
```

If `tp_methods` is not `NULL`, it must refer to an array of `PyMethodDef` structures. Each entry in the table is an instance of this structure:

```
typedef struct PyMethodDef {
    const char *ml_name;        /* method name */
    PyCFunction ml_meth;        /* implementation function */
    int ml_flags;               /* flags */
    const char *ml_doc;         /* docstring */
} PyMethodDef;
```

One entry should be defined for each method provided by the type; no entries are needed for methods inherited from a base type. One additional entry is needed at the end; it is a sentinel that marks the end of the array. The `ml_name` field of the sentinel must be `NULL`.

2 番目のテーブルは、インスタンス中に格納されるデータと直接対応づけられた属性を定義するのに使います。いくつもの C の原始的な型がサポートされており、アクセスを読み出し専用にも読み書き可能にもできます。このテーブルで使われる構造体は次のように定義されています:

```
typedef struct PyMemberDef {
    const char *name;
    int type;
```

(次のページに続く)

(前のページからの続き)

```

    int         offset;
    int         flags;
    const char *doc;
} PyMemberDef;

```

このテーブルの各エントリに対して **デスクリプタ** が作成され、値をインスタンスの構造体から抽出しうる型に対してそれらが追加されます。type メンバは structmember.h ヘッダで定義された型のコードをひとつ含んでいる必要があります。この値は Python における値と C における値をどのように変換しあうかを定めるものです。flags メンバはこの属性がどのようにアクセスされるかを制御するフラグを格納するのに使われます。

以下のフラグ用定数は structmember.h で定義されており、これらはビットごとの OR を取って組み合わせられます。

定数	意味
READONLY	絶対に変更できない。
READ_RESTRICTED	制限モード (restricted mode) では参照できない。
WRITE_RESTRICTED	制限モード (restricted mode) では変更できない。
RESTRICTED	制限モード (restricted mode) では参照も変更もできない。

tp_members を使ったひとつの面白い利用法は、実行時に使われるデスクリプタを作成しておき、単にテーブル中にテキストを置いておくことによって、この方法で定義されたすべての属性に doc string を関連付けられるようにすることです。アプリケーションはこのイントロスペクション用 API を使って、クラスオブジェクトからデスクリプタを取り出し、その __doc__ 属性を使って doc string を得られます。

As with the tp_methods table, a sentinel entry with a name value of NULL is required.

特定の型に特化した属性の管理

話を単純にするため、ここでは char* を使ったバージョンのみを示します。name パラメータの型はインターフェイスとして char* を使うか PyObject* を使うかの違いしかありません。この例では、上の総称的な例と同じことを効率的にやりますが、Python 2.2 で追加された総称的な型のサポートを使わずにやります。これはハンドラの関数がどのようにして呼ばれるのかを説明します。これで、たとえその機能を拡張する必要があるとき、何をどうすればいいかわかるでしょう。

tp_getattr ハンドラはオブジェクトが属性への参照を要求するときに呼ばれます。これは、そのクラスの __getattr__ () メソッドが呼ばれるであろう状況と同じ状況下で呼び出されます。

以下に例を示します。:

```

static PyObject *
newdatatype_getattr(newdatatypeobject *obj, char *name)
{

```

(次のページに続く)

(前のページからの続き)

```

if (strcmp(name, "data") == 0)
{
    return PyLong_FromLong(obj->data);
}

PyErr_Format(PyExc_AttributeError,
             "'%.50s' object has no attribute '%.400s'",
             tp->tp_name, name);
return NULL;
}

```

The `tp_setattr` handler is called when the `__setattr__()` or `__delattr__()` method of a class instance would be called. When an attribute should be deleted, the third parameter will be `NULL`. Here is an example that simply raises an exception; if this were really all you wanted, the `tp_setattr` handler should be set to `NULL`.

```

static int
newdatatype_setattr(newdatatypeobject *obj, char *name, PyObject *v)
{
    PyErr_Format(PyExc_RuntimeError, "Read-only attribute: %s", name);
    return -1;
}

```

2.3.4 オブジェクトの比較

```
richcmpfunc tp_richcompare;
```

`tp_richcompare` ハンドラは比較処理が要求されたときに呼び出されます。 `__lt__()` のような 拡張比較メソッド に類似しており、 `PyObject_RichCompare()` と `PyObject_RichCompareBool()` から呼び出されます。

This function is called with two Python objects and the operator as arguments, where the operator is one of `Py_EQ`, `Py_NE`, `Py_LE`, `Py_GT`, `Py_LT` or `Py_GE`. It should compare the two objects with respect to the specified operator and return `Py_True` or `Py_False` if the comparison is successful, `Py_NotImplemented` to indicate that comparison is not implemented and the other object's comparison method should be tried, or `NULL` if an exception was set.

これは内部ポインタのサイズが等しければ等しいと見なすデータ型のサンプル実装です:

```

static PyObject *
newdatatype_richcmp(PyObject *obj1, PyObject *obj2, int op)
{
    PyObject *result;
    int c, size1, size2;
}

```

(次のページに続く)

```

/* code to make sure that both arguments are of type
   newdatatype omitted */

size1 = obj1->obj_UnderlyingDatatypePtr->size;
size2 = obj2->obj_UnderlyingDatatypePtr->size;

switch (op) {
case Py_LT: c = size1 < size2; break;
case Py_LE: c = size1 <= size2; break;
case Py_EQ: c = size1 == size2; break;
case Py_NE: c = size1 != size2; break;
case Py_GT: c = size1 > size2; break;
case Py_GE: c = size1 >= size2; break;
}
result = c ? Py_True : Py_False;
Py_INCREF(result);
return result;
}

```

2.3.5 抽象的なプロトコルのサポート

Python はいくつもの **抽象的な** “プロトコル” をサポートしています。これらを使用する特定のインターフェイスについては `abstract` で解説されています。

A number of these abstract interfaces were defined early in the development of the Python implementation. In particular, the number, mapping, and sequence protocols have been part of Python since the beginning. Other protocols have been added over time. For protocols which depend on several handler routines from the type implementation, the older protocols have been defined as optional blocks of handlers referenced by the type object. For newer protocols there are additional slots in the main type object, with a flag bit being set to indicate that the slots are present and should be checked by the interpreter. (The flag bit does not indicate that the slot values are non-NULL. The flag may be set to indicate the presence of a slot, but a slot may still be unfilled.)

```

PyNumberMethods  *tp_as_number;
PySequenceMethods *tp_as_sequence;
PyMappingMethods *tp_as_mapping;

```

お使いのオブジェクトを数値やシーケンス、あるいは辞書のようにふるまうようにしたいならば、それぞれに C の `PyNumberMethods` 構造体、`PySequenceMethods` 構造体、または `PyMappingMethods` 構造体のアドレスを入れます。これらに適切な値を入れても入れなくてもかまいません。これらを使った例は Python の配布ソースにある `Objects` でみつけることができるでしょう。

```
hashfunc tp_hash;
```

この関数は、もし使うことにしたならば、データ型のインスタンスのハッシュ番号を返すようにします。次のは単純な例です:

```
static Py_hash_t
newdatatype_hash(newdatatypeobject *obj)
{
    Py_hash_t result;
    result = obj->some_size + 32767 * obj->some_number;
    if (result == -1)
        result = -2;
    return result;
}
```

`Py_hash_t` is a signed integer type with a platform-varying width. Returning `-1` from `tp_hash` indicates an error, which is why you should be careful to avoid returning it when hash computation is successful, as seen above.

```
ternaryfunc tp_call;
```

この関数は、その型のインスタンスが「関数として呼び出される」ときに呼ばれます。たとえばもし `obj1` にそのインスタンスが入っていて、Python スクリプトで `obj1('hello')` を実行したとすると、`tp_call` ハンドラが呼ばれます。

この関数は 3 つの引数をとります:

1. *self* は呼び出しの対象となるデータ型のインスタンスです。たとえば呼び出しが `obj1('hello')` の場合、*self* は `obj1` になります。
2. *args* は呼び出しの引数を格納しているタプルです。ここから引数を取り出すには `PyArg_ParseTuple()` を使います。
3. *kwds* is a dictionary of keyword arguments that were passed. If this is non-NULL and you support keyword arguments, use `PyArg_ParseTupleAndKeywords()` to extract the arguments. If you do not want to support keyword arguments and this is non-NULL, raise a `TypeError` with a message saying that keyword arguments are not supported.

以下は `tp_call` の簡易な実装です:

```
static PyObject *
newdatatype_call(newdatatypeobject *self, PyObject *args, PyObject *kwds)
{
    PyObject *result;
    const char *arg1;
    const char *arg2;
```

(次のページに続く)

(前のページからの続き)

```

const char *arg3;

if (!PyArg_ParseTuple(args, "sss:call", &arg1, &arg2, &arg3)) {
    return NULL;
}
result = PyUnicode_FromFormat(
    "Returning -- value: [%d] arg1: [%s] arg2: [%s] arg3: [%s]\n",
    obj->obj_UnderlyingDatatypePtr->size,
    arg1, arg2, arg3);
return result;
}
    
```

```

/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;
    
```

These functions provide support for the iterator protocol. Both handlers take exactly one parameter, the instance for which they are being called, and return a new reference. In the case of an error, they should set an exception and return NULL. `tp_iter` corresponds to the Python `__iter__()` method, while `tp_iternext` corresponds to the Python `__next__()` method.

Any *iterable* object must implement the `tp_iter` handler, which must return an *iterator* object. Here the same guidelines apply as for Python classes:

- For collections (such as lists and tuples) which can support multiple independent iterators, a new iterator should be created and returned by each call to `tp_iter`.
- Objects which can only be iterated over once (usually due to side effects of iteration, such as file objects) can implement `tp_iter` by returning a new reference to themselves -- and should also therefore implement the `tp_iternext` handler.

Any *iterator* object should implement both `tp_iter` and `tp_iternext`. An iterator's `tp_iter` handler should return a new reference to the iterator. Its `tp_iternext` handler should return a new reference to the next object in the iteration, if there is one. If the iteration has reached the end, `tp_iternext` may return NULL without setting an exception, or it may set `StopIteration` *in addition* to returning NULL; avoiding the exception can yield slightly better performance. If an actual error occurs, `tp_iternext` should always set an exception and return NULL.

2.3.6 弱参照 (Weak Reference) のサポート

One of the goals of Python's weak reference implementation is to allow any type to participate in the weak reference mechanism without incurring the overhead on performance-critical objects (such as numbers).

参考:

Documentation for the `weakref` module.

For an object to be weakly referencable, the extension type must do two things:

1. Include a `PyObject*` field in the C object structure dedicated to the weak reference mechanism. The object's constructor should leave it `NULL` (which is automatic when using the default `tp_alloc`).
2. Set the `tp_weaklistoffset` type member to the offset of the aforementioned field in the C object structure, so that the interpreter knows how to access and modify that field.

Concretely, here is how a trivial object structure would be augmented with the required field:

```
typedef struct {
    PyObject_HEAD
    PyObject *weakreflist; /* List of weak references */
} TrivialObject;
```

And the corresponding member in the statically-declared type object:

```
static PyTypeObject TrivialType = {
    PyVarObject_HEAD_INIT(NULL, 0)
    /* ... other members omitted for brevity ... */
    .tp_weaklistoffset = offsetof(TrivialObject, weakreflist),
};
```

The only further addition is that `tp_dealloc` needs to clear any weak references (by calling `PyObject_ClearWeakRefs()`) if the field is non-`NULL`:

```
static void
Trivial_dealloc(TrivialObject *self)
{
    /* Clear weakrefs first before calling any destructors */
    if (self->weakreflist != NULL)
        PyObject_ClearWeakRefs((PyObject *) self);
    /* ... remainder of destruction code omitted for brevity ... */
    Py_TYPE(self)->tp_free((PyObject *) self);
}
```

2.3.7 その他いろいろ

In order to learn how to implement any specific method for your new data type, get the *CPython* source code. Go to the `Objects` directory, then search the C source files for `tp_` plus the function you want (for example, `tp_richcompare`). You will find examples of the function you want to implement.

When you need to verify that an object is a concrete instance of the type you are implementing, use the `PyObject_TypeCheck()` function. A sample of its use might be something like the following:

```
if (!PyObject_TypeCheck(some_object, &MyType)) {
    PyErr_SetString(PyExc_TypeError, "arg #1 not a mything");
    return NULL;
}
```

参考:

Download CPython source releases. <https://www.python.org/downloads/source/>

The CPython project on GitHub, where the CPython source code is developed. <https://github.com/python/cpython>

2.4 C および C++ 拡張のビルド

CPython の C 拡張は **初期化関数** をエクスポートした共有ライブラリ (例、Linux の `.so` ファイルや Windows の `.pyd` ファイル) です。

インポートできるように、共有ライブラリは使える状態で `PYTHONPATH` 上になければならず、ファイル名をモジュール名に揃え、適切な拡張子になっていなければいけません。distutils を使っているときは、自動的に正しいファイル名が生成されます。

初期化関数のシグネチャは次のとおりです:

```
PyObject* PyInit_modulename(void)
```

この関数は初期化がモジュールか、`PyModuleDef` インスタンスを返します。詳しいことは `initializing-modules` を参照してください。

名前に ASCII しか使っていないモジュールの場合、関数名は `PyInit_<modulename>` の `<modulename>` をモジュール名で置き換えたものでなければなりません。multi-phase-initialization を使っているときは、モジュール名に ASCII 以外の文字も使えます。この場合、初期化関数の名前は `PyInitU_<modulename>` で、`<modulename>` はハイフンをアンダースコアで置き換えて Python の *punycode* エンコーディングでエンコードしたのになります。Python で書くと次のような処理になります:

```
def initfunc_name(name):
    try:
        suffix = b'_' + name.encode('ascii')
    except UnicodeEncodeError:
        suffix = b'U_' + name.encode('punycode').replace(b'-', b'_')
    return b'PyInit' + suffix
```

1 つの共有ライブラリに複数の初期化関数を定義することで、複数のモジュールをエクスポートすることは可能です。しかし、デフォルトではファイル名に対応した関数しか見付けようとしないので、複数のモジュールをインポートさせるにはシンボリックリンクか独自のインポーターを使う必要があります。詳しいことは [PEP 489](#) の “Multiple modules in one library” 節を参照してください。

2.4.1 distutils による C および C++ 拡張モジュールのビルド

拡張モジュールは Python に含まれる distutils を使ってビルドできます。distutils はバイナリパッケージの作成もサポートしているので、ユーザが拡張モジュールをインストールする際に、必ずしもコンパイラや distutils が必要というわけではありません。

distutils ベースのパッケージには、駆動スクリプト (driver script) となる `setup.py` が入っています。`setup.py` は普通の Python プログラムファイルで、ほとんどの場合以下のような内容になっています:

```
from distutils.core import setup, Extension

module1 = Extension('demo',
                    sources = ['demo.c'])

setup (name = 'PackageName',
      version = '1.0',
      description = 'This is a demo package',
      ext_modules = [module1])
```

この `setup.py` とファイル `demo.c` があるとき、以下のコマンド

```
python setup.py build
```

を実行すると、`demo.c` をコンパイルして、`demo` という名前の拡張モジュールを `build` ディレクトリ内に生成します。システムによってはモジュールファイルは `build/lib.system` サブディレクトリに生成され、`demo.so` や `demo.pyd` といった名前になることがあります。

`setup.py` 内では、コマンドの実行はすべて `setup` 関数を呼び出して行います。この関数は可変個のキーワード引数を取ります。上の例ではその一部を使っているにすぎません。もっと具体的にいうと、この例の中ではパッケージをビルドするためのメタ情報と、パッケージの内容を指定しています。通常、パッケージには Python ソースモジュールやドキュメント、サブパッケージ等といった別のモジュールも入ります。distutils の機能に関する詳細は、distutils-index に書かれている distutils のドキュメントを参照してください; この節では拡張モジュール

ルのビルドについてのみ説明します。

駆動スクリプトの構成をよりよくするために、`setup()` への引数を前もって計算しておくことがよくあります。上の例では、`setup()` の `ext_modules` は拡張モジュールのリストで、リストの各々の要素は `Extension` クラスのインスタンスになっています。上の例では、`demo` という名の拡張モジュールを定義していて、単一のソースファイル `demo.c` をコンパイルしてビルドするよう定義しています。

多くの場合、拡張モジュールのビルドはもっと複雑になります。というのは、プリプロセッサ定義やライブラリの追加指定が必要になることがあるからです。例えば以下のファイルがその実例です。

```
from distutils.core import setup, Extension

module1 = Extension('demo',
                    define_macros = [('MAJOR_VERSION', '1'),
                                     ('MINOR_VERSION', '0')],
                    include_dirs = ['/usr/local/include'],
                    libraries = ['tcl83'],
                    library_dirs = ['/usr/local/lib'],
                    sources = ['demo.c'])

setup (name = 'PackageName',
      version = '1.0',
      description = 'This is a demo package',
      author = 'Martin v. Loewis',
      author_email = 'martin@v.loewis.de',
      url = 'https://docs.python.org/extending/building',
      long_description = '''
This is really just a demo package.
''',
      ext_modules = [module1])
```

この例では、メタ情報が追加された状態で `setup()` が呼び出されていますが、配布パッケージを構築するにあたっては、メタ情報を付けておくことが推奨されます。拡張モジュール自体についてのメタ情報には、プリプロセッサ定義、インクルードファイルのディレクトリ、ライブラリのディレクトリ、ライブラリといった指定があります。distutils はこの情報をコンパイラに応じて異なるやり方で引渡します。例えば Unix では、上の設定は以下のようなコンパイルコマンドになるかもしれません

```
gcc -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -fPIC -DMAJOR_VERSION=1 -DMINOR_VERSION=0 -I/usr/
local/include -I/usr/local/include/python2.2 -c demo.c -o build/temp.linux-i686-2.2/demo.o

gcc -shared build/temp.linux-i686-2.2/demo.o -L/usr/local/lib -ltcl83 -o build/lib.linux-i686-2.2/
demo.so
```

これらのコマンドラインは実演目的で書かれたものです; distutils のユーザは distutils が正しくコマンドを実行すると信用してください。

2.4.2 拡張モジュールの配布

拡張モジュールをうまくビルドできたら、三通りの使い方があります。

エンドユーザは普通モジュールをインストールしようと考えます; これには、次を実行します

```
python setup.py install
```

モジュールメンテナはソースパッケージを作成します; これには、次を実行します

```
python setup.py sdist
```

場合によってはソース配布物に追加のファイルを含める必要があります; これには `MANIFEST.in` ファイルを使います; 詳しくは `manifest` を参照してください。

ソースコード配布物をうまく構築できたら、メンテナはバイナリ配布物も作成できます。プラットフォームに応じて、以下のコマンドのいずれかを使います。

```
python setup.py bdist_wininst
python setup.py bdist_rpm
python setup.py bdist_dumb
```

2.5 Windows 上での C および C++ 拡張モジュールのビルド

この章では Windows 向けの Python 拡張モジュールを Microsoft Visual C++ を使って作成する方法について簡単に述べ、その後に拡張モジュールのビルドがどのように動作するのかについて詳しい背景を述べます。この説明は、Python 拡張モジュールを作成する Windows プログラマと、Unix と Windows の双方でうまくビルドできるようなソフトウェアの作成に興味がある Unix プログラマの双方にとって有用です。

モジュールの作者には、この節で説明している方法よりも、`distutils` によるアプローチで拡張モジュールをビルドするよう勧めます。また、Python をビルドした際に使われた C コンパイラが必要です; 通常は Microsoft Visual C++ です。

注釈: この章では、Python のバージョン番号が符号化されて入っているたくさんのファイル名について触れます。これらのファイル名は `XY` で表されるバージョン名付きで表現されます; 'X' は使っている Python リリースのメジャーバージョン番号、'Y' はマイナーバージョン番号です。例えば、Python 2.2.1 を使っているなら、`XY` は実際には 22 になります。

2.5.1 型どおりのアプローチ

Windows での拡張モジュールのビルドには、Unix と同じように、`distutils` パッケージを使ったビルド作業の制御と手動の二通りのアプローチがあります。`distutils` によるアプローチはほとんどの拡張モジュールでうまくいきます; `distutils` を使った拡張モジュールのビルドとパッケージ化については、`distutils-index` にあります。これらを本当に手動で行わなければならないとわかった場合、標準ライブラリモジュールの `winsound` のプロジェクトファイルが学習に有益かもしれません。

2.5.2 Unix と Windows の相違点

Unix と Windows では、コードの実行時読み込みに全く異なるパラダイムを用いています。動的ロードされるようなモジュールをビルドしようとする前に、自分のシステムがどのように動作するか知っておいてください。

Unix では、共有オブジェクト (`.so`) ファイルにプログラムが使うコード、そしてプログラム内で使う関数名やデータが入っています。ファイルがプログラムに結合されると、これらの関数やデータに対するファイルのコード内の全ての参照は、メモリ内で関数やデータが配置されている、プログラム中の実際の場所を指すように変更されます。これは基本的にはリンク操作にあたります。

Windows では、動的リンクライブラリ (`.dll`) ファイルにはぶら下がり参照 (dangling reference) はありません。その代わり、関数やデータへのアクセスはルックアップテーブルを介します。従って DLL コードの場合、実行時にポインタがプログラムメモリ上の正しい場所を指すように修正する必要はありません; その代わり、コードは常に DLL のルックアップテーブルを使い、ルックアップテーブル自体は実行時に実際の関数やデータを指すように修正されます。

Unix には、唯一のライブラリファイル形式 (`.a`) しかありません。`.a` ファイルには複数のオブジェクトファイル (`.o`) 由来のコードが入っています。共有オブジェクトファイル (`.so`) を作成するリンク処理の段階中に、リンクは定義場所の不明な識別子に遭遇することがあります。このときリンクはライブラリ内のオブジェクトファイルを検索します; もし識別子が見つかると、リンクはそのオブジェクトファイルから全てのコードを取り込みます。

Windows では、二つの形式のライブラリ、静的ライブラリとインポートライブラリがあります (どちらも `.lib` と呼ばれています)。静的ライブラリは Unix における `.a` ファイルに似ています; このファイルには、必要に応じて取り込まれるようなコードが入っています。インポートライブラリは、基本的には特定の識別子が不正ではなく、DLL がロードされた時点で存在することを保証するためにだけ使われます。リンクはインポートライブラリからの情報を使ってルックアップテーブルを作成し、DLL に入っていない識別子を使えるようにします。アプリケーションや DLL がリンクされるさい、インポートライブラリが生成されることがあります。このライブラリは、アプリケーションや DLL 内のシンボルに依存するような、将来作成される全ての DLL で使うために必要になります。

二つの動的ロードモジュール、B と C を作成し、別のコードブロック A を共有するとします。Unix では、`A.a` を `B.so` や `C.so` をビルドするときのリンクに渡したりは **しません**; そんなことをすれば、コードは二度取り込まれ、B と C のそれぞれが自分用のコピーを持ってしまう。Windows では、`A.dll` をビルドすると `A.lib` もビルドされます。B や C のリンクには `A.lib` を渡します。`A.lib` にはコードは入っていません; 単に A のコー

ドにアクセスするするために実行時に用いられる情報が入っているだけです。

Windows ではインポートライブラリの使用は `import spam` とするようなものです; この操作によって `spam` の名前にアクセスできますが、コードのコピーを個別に作成したりはしません。Unix では、ライブラリとのリンクはむしろ `from spam import *` に似ています; この操作では個別にコードのコピーを生成します。

2.5.3 DLL 使用の実際

Windows 版の Python は Microsoft Visual C++ でビルドされています; 他のコンパイラを使うと、うまく動作したり、しなかったりします (Borland も一見うまく動作しません)。この節の残りの部分は MSVC++ 向けの説明です。

Windows で DLL を作成する際は、`pythonXY.lib` をリンカに渡さねばなりません。例えば二つの DLL、`spam` と `ni` (`spam` の中には C 関数が入っているとします) をビルドするには、以下のコマンドを実行します:

```
cl /LD /I/python/include spam.c ../libs/pythonXY.lib
cl /LD /I/python/include ni.c spam.lib ../libs/pythonXY.lib
```

最初のコマンドで、三つのファイル: `spam.obj`、`spam.dll` および `spam.lib` ができます。`Spam.dll` には (`PyArg_ParseTuple()` のような) Python 関数は全く入っていませんが、`pythonXY.lib` のおかげで Python コードを見つけることはできます。

二つ目のコマンドでは、`ni.dll` (および `.obj` と `.lib`) ができ、このライブラリは `spam` と Python 実行形式中の必要な関数をどうやって見つければよいか知っています。

全ての識別子がルックアップテーブル上に公開されるわけではありません。他のモジュール (Python 自体を含みます) から、自作の識別子が見えるようにするには、`void _declspec(dllexport) initspam(void)` や `PyObject _declspec(dllexport) *NiGetSpamData(void)` のように、`_declspec(dllexport)` で宣言せねばなりません。

Developer Studio は必要もなく大量のインポートライブラリを DLL に突っ込んで、実行形式のサイズを 100K も大きくしてしまいます。不要なライブラリを追い出したければ、「プロジェクトのプロパティ」ダイアログを選び、「リンカ」タブに移動して、**インポートライブラリの無視** を指定します。その後、適切な `msvcrtxx.lib` をライブラリのリストに追加してください。

大規模なアプリケーションへの PYTHON ランタイムの埋め込み

Python インタプリタの中でメインアプリケーションとして実行される拡張を作るのではなく、CPython をより大きなアプリケーションの中に埋め込む方が望ましいことがあります。この節ではその上手い埋め込みに関わる詳細について説明します。

3.1 他のアプリケーションへの Python の埋め込み

前章では、Python を拡張する方法、すなわち C 関数のライブラリを Python に結びつけて機能を拡張する方法について述べました。同じようなことを別の方法でも実行できます: それは、自分の C/C++ アプリケーションに Python を埋め込んで機能を強化する、というものです。埋め込みを行うことで、アプリケーションの何らかの機能を C や C++ の代わりに Python で実装できるようになります。埋め込みは多くの用途で利用できます; ユーザが Python でスクリプトを書き、アプリケーションを自分好みに仕立てられるようにする、というのがその一例です。プログラマが、特定の機能を Python でより楽に書ける場合に自分自身のために埋め込みを行うこともできます。

Python の埋め込みは Python の拡張と似ていますが、全く同じというわけではありません。その違いは、Python を拡張した場合にはアプリケーションのメインプログラムは依然として Python インタプリタである一方、Python を組み込み込んだ場合には、メインプログラムには Python が関係しない --- その代わりに、アプリケーションのある一部分が時折 Python インタプリタを呼び出して何らかの Python コードを実行させる --- かもしれない、ということです。

従って、Python の埋め込みを行う場合、自作のメインプログラムを提供しなければなりません。メインプログラムがやらなければならないことの一つに、Python インタプリタの初期化があります。とにかく少なくとも関数 `Py_Initialize()` を呼び出さねばなりません。オプションとして、Python 側にコマンドライン引数を渡すために関数呼び出しを行います。その後、アプリケーションのどこでもインタプリタを呼び出せるようになります。

インタプリタを呼び出すには、異なるいくつかの方法があります: Python 文が入った文字列を `PyRun_SimpleString()` に渡す、stdio ファイルポインタとファイル名 (これはエラーメッセージ内でコードを識別するためだけのものです) を `PyRun_SimpleFile()` に渡す、といった具合です。これまでの各章で説明した低水準の操作を呼び出して、Python オブジェクトを構築したり使用したりもできます。

参考:

c-api-index Python C インタフェースの詳細はこのマニュアルに書かれています。必要な情報の大部分はここにあるはずです。

3.1.1 高水準の埋め込み

Python の埋め込みの最も簡単な形式は、超高水準インタフェースの利用です。このインタフェースは、アプリケーションとやり取りする必要がない Python スクリプトを実行するためのものです。例えばこれは、一つのファイル上で何らかの操作を実現するのに利用できます。

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

int
main(int argc, char *argv[])
{
    wchar_t *program = Py_DecodeLocale(argv[0], NULL);
    if (program == NULL) {
        fprintf(stderr, "Fatal error: cannot decode argv[0]\n");
        exit(1);
    }
    Py_SetProgramName(program); /* optional but recommended */
    Py_Initialize();
    PyRun_SimpleString("from time import time,ctime\n"
                      "print('Today is', ctime(time()))\n");
    if (Py_FinalizeEx() < 0) {
        exit(120);
    }
    PyMem_RawFree(program);
    return 0;
}
```

Py_SetProgramName() は Py_Initialize() の前に呼び出す必要があります。これによりインタプリタにランタイムライブラリへのパスを伝えることが出来ます。続いて、Python インタプリタを Py_Initialize() で初期化し、続いてハードコードされた Python スクリプトで日付と時間の出力を実行します。その後、Py_FinalizeEx() の呼び出しでインタプリタを終了し、プログラムの終了に続きます。実際のプログラムでは、Python スクリプトを他のソース、おそらくテキストエディタルーチンやファイル、データベースから取り出したいと考えるかもしれません。Python コードをファイルから取り出すには、PyRun_SimpleFile() 関数を使うのがよいでしょう。この関数はメモリを確保して、ファイルの内容をロードする手間を省いてくれます。

3.1.2 超高水準の埋め込みから踏み出す: 概要

高水準インタフェースは、断片的な Python コードをアプリケーションから実行できるようにしてくれますが、アプリケーションと Python コードの間でのデータのやり取りは、控えめに言っても煩わしいものです。データのやり取りをしたいなら、より低水準のインタフェース呼び出しを利用しなくてはなりません。より多く C コードを書かねばならない代わりに、ほぼ何でもできるようになります。

Python の拡張と埋め込みは、趣旨こそ違え、同じ作業であるということに注意せねばなりません。これまでの章で議論してきたトピックのほとんどが埋め込みでもあてはまります。これを示すために、Python から C への拡張を行うコードが実際には何をするか考えてみましょう:

1. データ値を Python から C に変換する。
2. 変換された値を使って C ルーチンの関数呼び出しを行い、
3. 呼び出しで得られたデータ値 C から Python に変換する。

Python を埋め込む場合には、インタフェースコードが行う作業は以下のようになります:

1. データ値を C から Python に変換する。
2. 変換された値を使って Python インタフェースルーチンの関数呼び出しを行い、
3. 呼び出しで得られたデータ値 Python から C に変換する。

一見して分かるように、データ変換のステップは、言語間でデータを転送する方向が変わったのに合わせて単に入れ替えただけです。唯一の相違点は、データ変換の間にあるルーチンです。拡張を行う際には C ルーチン呼び出ししますが、埋め込みの際には Python ルーチン呼び出しします。

この章では、Python から C へ、そしてその逆へとデータを変換する方法については議論しません。また、正しい参照の使い方やエラーの扱い方についてすでに理解しているものと仮定します。これらの側面についてはインタプリタの拡張と何ら変わるところがないので、必要な情報については以前の章を参照できます。

3.1.3 純粋な埋め込み

最初に例示するプログラムは、Python スクリプト内の関数を実行するためのものです。超高水準インタフェースに関する節で挙げた例と同様に、Python インタプリタはアプリケーションと直接やりとりはしません (が、次の節でやりとりするよう変更します)。

Python スクリプト内で定義されている関数を実行するためのコードは以下のようになります:

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

int
main(int argc, char *argv[])
```

(次のページに続く)

(前のページからの続き)

```

{
    PyObject *pName, *pModule, *pFunc;
    PyObject *pArgs, *pValue;
    int i;

    if (argc < 3) {
        fprintf(stderr, "Usage: call pythonfile funcname [args]\n");
        return 1;
    }

    Py_Initialize();
    pName = PyUnicode_DecodeFSDefault(argv[1]);
    /* Error checking of pName left out */

    pModule = PyImport_Import(pName);
    Py_DECREF(pName);

    if (pModule != NULL) {
        pFunc = PyObject_GetAttrString(pModule, argv[2]);
        /* pFunc is a new reference */

        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyLong_FromLong(atoi(argv[i + 3]));
                if (!pValue) {
                    Py_DECREF(pArgs);
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert argument\n");
                    return 1;
                }
                /* pValue reference stolen here: */
                PyTuple_SetItem(pArgs, i, pValue);
            }
            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);
            if (pValue != NULL) {
                printf("Result of call: %ld\n", PyLong_AsLong(pValue));
                Py_DECREF(pValue);
            }
            else {
                Py_DECREF(pFunc);
                Py_DECREF(pModule);
                PyErr_Print();
                fprintf(stderr, "Call failed\n");
                return 1;
            }
        }
    }
}

```

(次のページに続く)

(前のページからの続き)

```

    }
    else {
        if (PyErr_Occurred())
            PyErr_Print();
        fprintf(stderr, "Cannot find function \"%s\"\n", argv[2]);
    }
    Py_XDECREF(pFunc);
    Py_DECREF(pModule);
}
else {
    PyErr_Print();
    fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
    return 1;
}
if (Py_FinalizeEx() < 0) {
    return 120;
}
return 0;
}

```

このコードは `argv[1]` を使って Python スクリプトをロードし、`argv[2]` 内に指定された名前の関数を呼び出します。関数の整数引数は `argv` 配列中の他の値になります。このプログラムを [コンパイルしてリンク](#) し (できた実行可能形式を `call` と呼びましょう)、以下のような Python スクリプトを実行することになります:

```

def multiply(a,b):
    print("Will compute", a, "times", b)
    c = 0
    for i in range(0, a):
        c = c + b
    return c

```

実行結果は以下のようになるはずです:

```

$ call multiply multiply 3 2
Will compute 3 times 2
Result of call: 6

```

この程度の機能を実現するにはプログラムがいささか大きすぎますが、ほとんどは Python から C へのデータ変換やエラー報告のためのコードです。Python の埋め込みという観点から最も興味深い部分は以下のコードから始まる部分です:

```

Py_Initialize();
pName = PyUnicode_DecodeFSDefault(argv[1]);
/* Error checking of pName left out */
pModule = PyImport_Import(pName);

```

インタプリタの初期化後、スクリプトは `PyImport_Import()` を使って読み込まれます。このルーチンは Python 文字列を引数に取る必要があります、データ変換ルーチン `PyUnicode_FromString()` で構築します。

```
pFunc = PyObject_GetAttrString(pModule, argv[2]);
/* pFunc is a new reference */

if (pFunc && PyCallable_Check(pFunc)) {
    ...
}
Py_XDECREF(pFunc);
```

ひとたびスクリプトが読み込まれると、`PyObject_GetAttrString()` を使って必要な名前を取得できます。名前がスクリプト中に存在し、取得したオブジェクトが呼び出し可能オブジェクトであれば、このオブジェクトが関数であると考えて差し支えないでしょう。そこでプログラムは定石どおりに引数のタプル構築に進みます。その後、Python 関数を以下のコードで呼び出します:

```
pValue = PyObject_CallObject(pFunc, pArgs);
```

関数が処理を戻す際、`pValue` は `NULL` になるか、関数の戻り値への参照が入っています。値を調べた後には忘れずに参照を解放してください。

3.1.4 埋め込まれた Python の拡張

ここまでは、埋め込み Python インタプリタはアプリケーション本体の機能にアクセスする手段がありませんでした。Python API を使うと、埋め込みインタプリタを拡張することでアプリケーション本体へのアクセスを可能にします。つまり、アプリケーションで提供されているルーチンを使って、埋め込みインタプリタを拡張するので。複雑なことのように思えますが、それほどひどいわけではありません。さしあたって、アプリケーションが Python インタプリタを起動したということをちょっと忘れてみてください。その代わり、アプリケーションがサブルーチンの集まりで、あたかも普通の Python 拡張モジュールを書くかのように、Python から各ルーチンにアクセスできるようにするグルー (glue, 糊) コードを書くと考えてください。例えば以下のようにです:

```
static int numargs=0;

/* Return the number of arguments of the application command line */
static PyObject*
emb_numargs(PyObject *self, PyObject *args)
{
    if(!PyArg_ParseTuple(args, ":numargs"))
        return NULL;
    return PyLong_FromLong(numargs);
}

static PyMethodDef EmbMethods[] = {
    {"numargs", emb_numargs, METH_VARARGS,
```

(次のページに続く)

(前のページからの続き)

```

    "Return the number of arguments received by the process."},
    {NULL, NULL, 0, NULL}
};

static PyModuleDef EmbModule = {
    PyModuleDef_HEAD_INIT, "emb", NULL, -1, EmbMethods,
    NULL, NULL, NULL, NULL
};

static PyObject*
PyInit_emb(void)
{
    return PyModule_Create(&EmbModule);
}

```

上のコードを `main()` 関数のすぐ上に挿入します。また、以下の二つの文を `Py_Initialize()` の呼び出しの前に挿入します:

```

numargs = argc;
PyImport_AppendInittab("emb", &PyInit_emb);

```

これら二つの行は `numargs` 変数を初期化し、埋め込み Python インタプリタから `emb.numargs()` 関数にアクセスできるようにします。これらの拡張モジュール関数を使うと、Python スクリプトは以下のようなことができます。

```

import emb
print("Number of arguments", emb.numargs())

```

実際のアプリケーションでは、こうしたメソッドでアプリケーション内の API を Python に公開することになります。

3.1.5 C++ による Python の埋め込み

C++ プログラム中にも Python を埋め込みます; 厳密に言うと、どうやって埋め込むかは使っている C++ 処理系の詳細に依存します; 一般的には、メインプログラムを C++ で書き、C++ コンパイラを使ってプログラムをコンパイル・リンクする必要があるでしょう。Python 自体を C++ でコンパイルしなおす必要はありません。

3.1.6 Unix 系システムにおけるコンパイルとリンク

Python インタプリタをアプリケーションに埋め込むためにコンパイラ (とリンカ) に渡すべき正しいフラグを見出すのは簡単でないかもしれません。これは特に、Python がライブラリモジュールに対してリンクされた C 動的拡張 (.so ファイル) として実装されたものをロードする必要があるためです。

必要なコンパイル・リンクのオプションを知るために、pythonX.Y-config スクリプトが使えます (これは Python インストール時に生成されたもので、python3-config スクリプトも利用出来るかもしれません)。このスクリプトにはオプションが多くありますが、直接的に有用なのはこれでしょう:

- pythonX.Y-config --cflags は推奨のコンパイルオプションを出力します:

```
$ /opt/bin/python3.4-config --cflags
-I/opt/include/python3.4m -I/opt/include/python3.4m -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-
↳ prototypes
```

- pythonX.Y-config --ldflags は推奨のリンクオプションを出力します:

```
$ /opt/bin/python3.4-config --ldflags
-L/opt/lib/python3.4/config-3.4m -lpthread -ldl -lutil -lm -lpython3.4m -Xlinker -export-
↳ dynamic
```

注釈: 複数 Python バージョン共存 (とりわけシステムの Python とあなた自身でビルドした Python) での混乱を避けるために、上での例のように pythonX.Y-config は絶対パスで起動したほうが良いです。

もしこの手順でうまくいかなければ (たしかにこれは全ての Unix 的なプラットフォームで動作することを保障するものではないですが、bug reports は歓迎です)、あなたのシステムのダイナミックリンクについてのドキュメントを読み、Python の Makefile のコンパイルオプションを調べる必要があるでしょう (Makefile の場所を調べるには sysconfig.get_makefile_filename() を使ってください)。この場合、sysconfig モジュールが役に立つ道具になります。これによってあなたが付け加えたいコンパイル・リンクのオプション構成をプログラマ的に抽出できます。例えば:

```
>>> import sysconfig
>>> sysconfig.get_config_var('LIBS')
'-lpthread -ldl -lutil'
>>> sysconfig.get_config_var('LINKFORSHARED')
'-Xlinker -export-dynamic'
```

用語集

>>> インタラクティブシェルにおけるデフォルトの Python プロンプトです。インタプリタでインタラクティブに実行されるコード例でよく出てきます。

... インタラクティブシェルにおいて、インデントされたコードブロック、対応する左右の区切り文字の組 (丸括弧、角括弧、波括弧、三重引用符) の内側、デコレーターの後に、コードを入力する際に表示されるデフォルトの Python プロンプトです。

2to3 Python 2.x のコードを Python 3.x のコードに変換するツールです。ソースコードを解析してその解析木を巡回 (traverse) することで検知できる、非互換性の大部分を処理します。

2to3 は標準ライブラリの `lib2to3` として利用できます。単体のツールとしての使えるスクリプトが `Tools/scripts/2to3` として提供されています。2to3-reference を参照してください。

abstract base class (抽象基底クラス) 抽象基底クラスは *duck-typing* を補完するもので、`hasattr()` などの別のテクニックでは不恰好であったり微妙に誤る (例えば `magic methods` の場合) 場合にインタフェースを定義する方法を提供します。ABC は仮想 (virtual) サブクラスを導入します。これは親クラスから継承しませんが、それでも `isinstance()` や `issubclass()` に認識されます; `abc` モジュールのドキュメントを参照してください。Python には、多くの組み込み ABC が同梱されています。その対象は、(`collections.abc` モジュールで) データ構造、(`numbers` モジュールで) 数、(`io` モジュールで) ストリーム、(`importlib.abc` モジュールで) インポートファインダ及びローダーです。`abc` モジュールを利用して独自の ABC を作成できます。

annotation (アノテーション) 変数、クラス属性、関数のパラメータや返り値に関係するラベルです。慣例により *type hint* として使われています。

ローカル変数のアノテーションは実行時にはアクセスできませんが、グローバル変数、クラス属性、関数のアノテーションはそれぞれモジュール、クラス、関数の `__annotations__` 特殊属性に保持されています。

機能の説明がある *variable annotation*, *function annotation*, [PEP 484](#), [PEP 526](#) を参照してください。

引数 (argument) (実引数) 関数を呼び出す際に、関数 (または *メソッド*) に渡す値です。実引数には2種類あります:

- **キーワード引数:** 関数呼び出しの際に引数の前に識別子がついたもの (例: `name=`) や、`**` に続けた辞書の中の値として渡された引数。例えば、次の `complex()` の呼び出しでは、3 と 5 がキーワード引数です:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- **位置引数:** キーワード引数以外の引数。位置引数は引数リストの先頭を書くことができ、また `*` に続けた *iterable* の要素として渡すことができます。例えば、次の例では 3 と 5 は両方共位置引数です:

```
complex(3, 5)
complex(*(3, 5))
```

実引数は関数の実体において名前付きのローカル変数に割り当てられます。割り当てを行う規則については `calls` を参照してください。シンタックスにおいて実引数を表すためにあらゆる式を使うことが出来ます。評価された値はローカル変数に割り当てられます。

仮引数、FAQ の 実引数と仮引数の違いは何ですか?、**PEP 362** を参照してください。

asynchronous context manager (非同期コンテキストマネージャ) `__aenter__()` と `__aexit__()` メソッドを定義することで `async with` 文内の環境を管理するオブジェクトです。**PEP 492** で導入されました。

asynchronous generator (非同期ジェネレータ) *asynchronous generator iterator* を返す関数です。`async def` で定義されたコルーチン関数に似ていますが、`yield` 式を持つ点で異なります。`yield` 式は `async for` ループで使用できる値の並びを生成するのに使用されます。

通常は非同期ジェネレータ関数を指しますが、文脈によっては **非同期ジェネレータイテレータ** を指す場合があります。意図された意味が明らかでない場合、明瞭化のために完全な単語を使用します。

非同期ジェネレータ関数には、`async for` 文や `async with` 文だけでなく `await` 式もあることがあります。

asynchronous generator iterator (非同期ジェネレータイテレータ) *asynchronous generator* 関数で生成されるオブジェクトです。

これは、`__anext__()` メソッドを使って呼び出されたときに awaitable オブジェクトを返す *asynchronous iterator* です。この awaitable オブジェクトは、次の `yield` 式までの非同期ジェネレータ関数の本体を実行します。

`yield` にくるたびに、その位置での実行状態 (ローカル変数と保留状態の `try` 文) 処理は一時停止されます。`__anext__()` で返された他の awaitable によって **非同期ジェネレータイテレータ** が実際に再開されたとき、中断した箇所を取得します。**PEP 492** および **PEP 525** を参照してください。

asynchronous iterable (非同期イテラブル) `async for` 文の中で使用できるオブジェクトです。自身の `__aiter__()` メソッドから *asynchronous iterator* を返さなければなりません。**PEP 492** で導入されました。

asynchronous iterator (非同期イテレータ) `__aiter__()` と `__anext__()` メソッドを実装したオブジェクトです。`__anext__` は *awaitable* オブジェクトを返さなければなりません。`async for` は `StopAsyncIteration` 例外を送出するまで、非同期イテレータの `__anext__()` メソッドが返す *awaitable* を解決します。**PEP 492** で導入されました。

属性 (属性) オブジェクトに関連付けられ、ドット表記式によって名前で参照される値です。例えば、オブジェクト *o* が属性 *a* を持っているとき、その属性は *o.a* で参照されます。

awaitable (待機可能) `await` 式で使うことが出来るオブジェクトです。*coroutine* か、`__await__()` メソッドがあるオブジェクトです。**PEP 492** を参照してください。

BDFL 慈悲深き終身独裁者 (Benevolent Dictator For Life) の略です。Python の作者、Guido van Rossum のことです。

binary file (バイナリファイル) *bytes-like* オブジェクト の読み込みおよび書き込みができる **ファイルオブジェクト** です。バイナリファイルの例は、バイナリモード ('rb', 'wb' or 'rb+') で開かれたファイル、`sys.stdin.buffer`、`sys.stdout.buffer`、`io.BytesIO` や `gzip.GzipFile` のインスタンスです。

`str` オブジェクトの読み書きができるファイルオブジェクトについては、*text file* も参照してください。

bytes-like object `bufferobjects` をサポートしていて、C 言語の意味で 連続した contiguous バッファを提供可能なオブジェクト。`bytes`、`bytearray`、`array.array` や、多くの一般的な `memoryview` オブジェクトがこれに当たります。*bytes-like* オブジェクトは、データ圧縮、バイナリファイルへの保存、ソケットを経由した送信など、バイナリデータを要求するいろいろな操作に利用することができます。

幾つかの操作ではバイナリデータを変更する必要があります。その操作のドキュメントではよく ”読み書き可能な *bytes-like* オブジェクト” に言及しています。変更可能なバッファオブジェクトには、`bytearray` と `bytearray` の `memoryview` などが含まれます。また、他の幾つかの操作では不変なオブジェクト内のバイナリデータ (”読み出し専用の *bytes-like* オブジェクト”) を必要します。それには `bytes` と `bytes` の `memoryview` オブジェクトが含まれます。

bytecode (バイトコード) Python のソースコードは、Python プログラムの CPython インタプリタの内部表現であるバイトコードへとコンパイルされます。バイトコードは `.pyc` ファイルにキャッシュされ、同じファイルが二度目に実行される時はより高速になります (ソースコードからバイトコードへの再度のコンパイルは回避されます)。この ”中間言語 (intermediate language)” は、各々のバイトコードに対応する機械語を実行する **仮想マシン** で動作するといえます。重要な注意として、バイトコードは異なる Python 仮想マシン間で動作することや、Python リリース間で安定であることは期待されていません。

バイトコードの命令一覧は `dis` モジュール にあります。

クラス (クラス) ユーザー定義オブジェクトを作成するためのテンプレートです。クラス定義は普通、そのクラスのインスタンス上の操作をするメソッドの定義を含みます。

class variable (クラス変数) クラス上に定義され、クラスレベルで (つまり、クラスのインスタンス上ではなしに) 変更されることを目的としている変数です。

coercion (型強制) 同じ型の 2 引数を伴う演算の最中に行われる、ある型のインスタンスの別の型への暗黙の変換です。例えば、`int(3.15)` は浮動小数点数を整数 3 に変換します。しかし `3+4.5` では、各引数は型が異なり (一つは整数、一つは浮動小数点数)、加算をする前に同じ型に変換できなければ `TypeError` 例外が投げられます。型強制がなかったら、すべての引数は、たとえ互換な型であっても、単に `3+4.5` ではなく `float(3)+4.5` というように、プログラマーが同じ型に正規化しなければいけません。

complex number (複素数) よく知られている実数系を拡張したもので、すべての数は実部と虚部の和として表されます。虚数は虚数単位 (-1 の平方根) に実数を掛けたもので、一般に数学では i と書かれ、工学では j と書かれます。Python は複素数に組み込みで対応し、後者の表記を取っています。虚部は末尾に j をつけて書きます。例えば `3+1j` です。`math` モジュールの複素数版を利用するには、`cmath` を使います。複素数の使用はかなり高度な数学の機能です。必要性を感じなければ、ほぼ間違いなく無視してしまってよいでしょう。

context manager (コンテキストマネージャ) `__enter__()` と `__exit__()` メソッドを定義することで `with` 文内の環境を管理するオブジェクトです。[PEP 343](#) を参照してください。

context variable (コンテキスト変数) コンテキストに依存して異なる値を持つ変数。これは、ある変数の値が各々の実行スレッドで異なり得るスレッドローカルストレージに似ています。しかしコンテキスト変数では、1 つの実行スレッドにいくつかのコンテキストがあり得、コンテキスト変数の主な用途は並列な非同期タスクの変数の追跡です。`contextvars` を参照してください。

contiguous (隣接、連続) バッファが厳密に **C-連続** または **Fortran 連続** である場合に、そのバッファは連続しているとみなせます。ゼロ次元バッファは C 連続であり Fortran 連続です。一次元の配列では、その要素は必ずメモリ上で隣接するように配置され、添字がゼロから始まり増えていく順序で並びます。多次元の C-連続な配列では、メモリアドレス順に要素を巡る際には最後の添え字が最初に変わるのに対し、Fortran 連続な配列では最初の添え字が最初に動きます。

コルーチン (コルーチン) コルーチンはサブルーチンのより一般的な形式です。サブルーチンには決められた地点から入り、別の決められた地点から出ます。コルーチンには多くの様々な地点から入る、出る、再開することができます。コルーチンは `async def` 文で実装できます。[PEP 492](#) を参照してください。

coroutine function (コルーチン関数) *coroutine* オブジェクトを返す関数です。コルーチン関数は `async def` 文で実装され、`await`、`async for`、および `async with` キーワードを持つことが出来ます。これらは [PEP 492](#) で導入されました。

CPython python.org で配布されている、Python プログラミング言語の標準的な実装です。”CPython” という単語は、この実装を Jython や IronPython といった他の実装と区別する必要がある場合に利用されます。

decorator (デコレータ) 別の関数を返す関数で、通常、`@wrapper` 構文で関数変換として適用されます。デコレータの一般的な利用例は、`classmethod()` と `staticmethod()` です。

デコレータの文法はシンタックスシュガーです。次の 2 つの関数定義は意味的に同じものです:

```
def f(...):
    ...
```

(次のページに続く)

(前のページからの続き)

```
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

同じ概念がクラスにも存在しますが、あまり使われません。デコレータについて詳しくは、関数定義 および クラス定義 のドキュメントを参照してください。

descriptor (デスクリプタ) メソッド `__get__()`、`__set__()`、あるいは `__delete__()` を定義しているオブジェクトです。あるクラス属性がデスクリプタであるとき、属性探索によって、束縛されている特別な動作が呼び出されます。通常、`get`、`set`、`delete` のために `a.b` と書くと、`a` のクラス辞書内でオブジェクト `b` を検索しますが、`b` がデスクリプタであればそれぞれのデスクリプタメソッドが呼び出されます。デスクリプタの理解は、Python を深く理解する上で鍵となります。というのは、デスクリプタこそが、関数、メソッド、プロパティ、クラスメソッド、静的メソッド、そしてスーパークラスの参照といった多くの機能の基盤だからです。

デスクリプタのメソッドに関して詳しくは、`descriptors` を参照してください。

dictionary (辞書) 任意のキーを値に対応付ける連想配列です。`__hash__()` メソッドと `__eq__()` メソッドを実装した任意のオブジェクトをキーにできます。Perl ではハッシュ (hash) と呼ばれています。

dictionary view (辞書ビュー) `dict.keys()`、`dict.values()`、`dict.items()` が返すオブジェクトです。辞書の項目の動的なビューを提供します。すなわち、辞書が変更されるとビューはそれを反映します。辞書ビューを強制的に完全なリストにするには `list(dictview)` を使用してください。`dict-views` を参照してください。

docstring クラス、関数、モジュールの最初の式である文字列リテラルです。そのスイートの実行時には無視されますが、コンパイラによって識別され、そのクラス、関数、モジュールの `__doc__` 属性として保存されます。イントロスペクションできる (訳注: 属性として参照できる) ので、オブジェクトのドキュメントを書く標準的な場所です。

duck-typing あるオブジェクトが正しいインタフェースを持っているかを決定するのにオブジェクトの型を見ないプログラミングスタイルです。代わりに、単純にオブジェクトのメソッドや属性が呼ばれたり使われたりします。「アヒルのように見えて、アヒルのように鳴けば、それはアヒルである。」) インタフェースを型より重視することで、上手くデザインされたコードは、ポリモーフィックな代替を許して柔軟性を向上させます。ダックタイピングは `type()` や `isinstance()` による判定を避けます。(ただし、ダックタイピングを **抽象基底クラス** で補完することもできます。) その代わりに、典型的に `hasattr()` 判定や **EAFP** プログラミングを利用します。

EAFP 「認可をとるより許しを請う方が容易 (easier to ask for forgiveness than permission、マーフィーの法則)」の略です。この Python で広く使われているコーディングスタイルでは、通常は有効なキーや属性が存在するものと仮定し、その仮定が誤っていた場合に例外を捕捉します。この簡潔で手早く書けるコーディングスタイルには、`try` 文および `except` 文がたくさんあるのが特徴です。このテクニックは、C のよう

な言語でよく使われている *LBYL* スタイルと対照的なものです。

expression (式) 何かの値と評価される、一まとまりの構文 (a piece of syntax) です。言い換えると、式とはリテラル、名前、属性アクセス、演算子や関数呼び出しなど、値を返す式の要素の積み重ねです。他の多くの言語と違い、Python では言語の全ての構成要素が式というわけではありません。while のように、式としては使えない **文** もあります。代入も式ではなく文です。

extension module (拡張モジュール) C や C++ で書かれたモジュールで、Python の C API を利用して Python コアやユーザーコードとやりとりします。

f-string 'f' や 'F' が先頭に付いた文字列リテラルは "f-string" と呼ばれ、これは フォーマット済み文字列リテラルの短縮形の名称です。PEP 498 も参照してください。

file object (ファイルオブジェクト) 下位のリソースへのファイル志向 API (read() や write() メソッドを持つもの) を公開しているオブジェクトです。ファイルオブジェクトは、作成された手段によって、実際のディスク上のファイルや、その他のタイプのストレージや通信デバイス (例えば、標準入出力、インメモリバッファ、ソケット、パイプ、等) へのアクセスを媒介できます。ファイルオブジェクトは *file-like objects* や *streams* とも呼ばれます。

ファイルオブジェクトには実際には 3 種類あります: 生の **バイナリーファイル**、バッファされた **バイナリーファイル**、そして **テキストファイル** です。インターフェイスは io モジュールで定義されています。ファイルオブジェクトを作る標準的な方法は open() 関数を使うことです。

file-like object *file object* と同義です。

finder (ファインダ) インポートされているモジュールの *loader* の発見を試行するオブジェクトです。

Python 3.3 以降では 2 種類のファインダがあります。sys.meta_path で使用される *meta path finder* と、sys.path_hooks で使用される *path entry finder* です。

詳細については PEP 302、PEP 420 および PEP 451 を参照してください。

floor division 一番近い小さい整数に丸める数学除算。floor division 演算子は // です。例えば、11 // 4 は 2 になり、float の true division の結果 2.75 と異なります。(-11) // 4 は -2.75 を **小さい方に丸める**ので -3 になることに注意してください。PEP 238 を参照してください。

function (関数) 呼び出し側に値を返す一連の文のことです。関数には 0 以上の **実引数** を渡すことが出来ます。実体の実行時に引数を使用することが出来ます。**仮引数**、**メソッド**、function を参照してください。

function annotation (関数アノテーション) 関数のパラメータや戻り値の *annotation* です。

関数アノテーションは、通常は **型ヒント** のために使われます: 例えば、この関数は 2 つの int 型の引数を取ると期待され、また int 型の戻り値を持つと期待されています。

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

関数アノテーションの文法は function の節で解説されています。

機能の説明がある *variable annotation* と **PEP 484** を参照してください。

`__future__` 互換性のない新たな言語機能を現在のインタプリタで有効にするためにプログラマが利用できる擬似モジュールです。

`__future__` モジュールを `import` してその変数を評価すれば、新たな機能が初めて追加されたのがいつで、いつ言語デフォルトの機能になるかわかります:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection (ガベージコレクション) これ以降使われることのないメモリを解放する処理です。Python は、参照カウントと、循環参照を検出し破壊する循環ガベージコレクタを使ってガベージコレクションを行います。ガベージコレクタは `gc` モジュールを使って操作できます。

ジェネレータ (ジェネレータ) *generator iterator* を返す関数です。通常関数に似ていますが、`yield` 式を持つ点で異なります。`yield` 式は、`for` ループで使用できたり、`next()` 関数で値を 1 つずつ取り出したりできる、値の並びを生成するのに使用されます。

通常はジェネレータ関数を指しますが、文脈によっては **ジェネレータイテレータ** を指す場合があります。意図された意味が明らかでない場合、明瞭化のために完全な単語を使用します。

generator iterator (ジェネレータイテレータ) *generator* 関数で生成されるオブジェクトです。

`yield` のたびに局所実行状態 (局所変数や未処理の `try` 文などを含む) を記憶して、処理は一時的に中断されます。**ジェネレータイテレータ** が再開されると、中断した位置を取得します (通常関数が実行のたびに新しい状態から開始するのと対照的です)。

generator expression (ジェネレータ式) イテレータを返す式です。普通の式に、ループ変数を定義する `for` 節、範囲、そして省略可能な `if` 節がつづいているように見えます。こうして構成された式は、外側の関数に向けて値を生成します:

```
>>> sum(i*i for i in range(10))           # sum of squares 0, 1, 4, ... 81
285
```

generic function (ジェネリック関数) 異なる型に対し同じ操作をする関数群から構成される関数です。呼び出し時にどの実装を用いるかはディスパッチアルゴリズムにより決定されます。

single dispatch、`functools.singledispatch()` デコレータ、**PEP 443** を参照してください。

GIL *global interpreter lock* を参照してください。

global interpreter lock (グローバルインタプリタロック) *CPython* インタプリタが利用している、一度に Python の **バイトコード** を実行するスレッドは一つだけであることを保証する仕組みです。これにより (`dict` などの重要な組み込み型を含む) オブジェクトモデルが同時アクセスに対して暗黙的に安全になるので、

CPython の実装がシンプルになります。インタプリタ全体をロックすることで、マルチプロセッサマシンが生じる並列化のコストと引き換えに、インタプリタを簡単にマルチスレッド化できるようになります。

ただし、標準あるいは外部のいくつかの拡張モジュールは、圧縮やハッシュ計算などの計算の重い処理をするときに GIL を解除するように設計されています。また、I/O 処理をする場合 GIL は常に解除されます。

過去に ” 自由なマルチスレッド化 ” したインタプリタ (供用されるデータを細かい粒度でロックする) が開発されましたが、一般的なシングルスプロセッサの場合のパフォーマンスが悪かったので成功しませんでした。このパフォーマンスの問題を克服しようとする、実装がより複雑になり保守コストが増加すると考えられています。

hash-based pyc (ハッシュベース pyc ファイル) 正当性を判別するために、対応するソースファイルの最終更新時刻ではなくハッシュ値を使用するバイトコードのキャッシュファイルです。

hashable (ハッシュ可能) **ハッシュ可能** なオブジェクトとは、生存期間中変わらないハッシュ値を持ち (`__hash__()` メソッドが必要)、他のオブジェクトと比較ができる (`__eq__()` メソッドが必要) オブジェクトです。同値なハッシュ可能オブジェクトは必ず同じハッシュ値を持つ必要があります。

ハッシュ可能なオブジェクトは辞書のキーや集合のメンバーとして使えます。辞書や集合のデータ構造は内部でハッシュ値を使っているからです。

Python のイミュータブルな組み込みオブジェクトは、ほとんどがハッシュ可能です。(リストや辞書のような) ミュータブルなコンテナはハッシュ不可能です。(タプルや `frozenset` のような) イミュータブルなコンテナは、要素がハッシュ可能であるときのみハッシュ可能です。ユーザー定義のクラスのインスタンスであるようなオブジェクトはデフォルトでハッシュ可能です。それらは全て (自身を除いて) 比較結果は非等価であり、ハッシュ値は `id()` より得られます。

IDLE Python の統合開発環境 (Integrated DeveLopment Environment) です。IDLE は Python の標準的な配布に同梱されている基本的な機能のエディタとインタプリタ環境です。

immutable (イミュータブル) 固定の値を持ったオブジェクトです。イミュータブルなオブジェクトには、数値、文字列、およびタプルなどがあります。これらのオブジェクトは値を変えられません。別の値を記憶させる際には、新たなオブジェクトを作成しなければなりません。イミュータブルなオブジェクトは、固定のハッシュ値が必要となる状況で重要な役割を果たします。辞書のキーがその例です。

import path *path based finder* が `import` するモジュールを検索する場所 (または *path entry*) のリスト。`import` 中、このリストは通常 `sys.path` から来ますが、サブパッケージの場合は親パッケージの `__path__` 属性からも来ます。

importing あるモジュールの Python コードが別のモジュールの Python コードで使えるようにする処理です。

importer モジュールを探してロードするオブジェクト。*finder* と *loader* のどちらでもあるオブジェクト。

interactive (対話的) Python には対話的インタプリタがあり、文や式をインタプリタのプロンプトに入力すると即座に実行されて結果を見ることができます。`python` と何も引数を与えずに実行してください。(コンピュータのメインメニューから Python の対話的インタプリタを起動できるかもしれません。) 対話的イン

タプリタは、新しいアイデアを試してみたり、モジュールやパッケージの中を覗いてみる (`help(x)` を覚えておいてください) のに非常に便利なツールです。

interpreted Python はインタプリタ形式の言語であり、コンパイラ言語の対極に位置します。(バイトコードコンパイラがあるために、この区別は曖昧ですが。) ここでのインタプリタ言語とは、ソースコードのファイルを、まず実行可能形式にしてから実行させるといった操作なしに、直接実行できることを意味します。インタプリタ形式の言語は通常、コンパイラ形式の言語よりも開発／デバッグのサイクルは短いものの、プログラムの実行は一般に遅いです。**対話的** も参照してください。

interpreter shutdown Python インタープリターはシャットダウンを要請された時に、モジュールやすべてのクリティカルな内部構造をなどの、すべての確保したリソースを段階的に開放する、特別なフェーズに入ります。このフェーズは **ガベージコレクタ** を複数回呼び出します。これによりユーザー定義のデストラクターや `weakref` コールバックが呼び出されることがあります。シャットダウンフェーズ中に実行されるコードは、それが依存するリソースがすでに機能しない (よくある例はライブラリーモジュールや `warning` 機構です) ために様々な例外に直面します。

インタープリタがシャットダウンする主な理由は `__main__` モジュールや実行されていたスクリプトの実行が終了したことです。

iterable (反復可能オブジェクト) 要素を一度に 1 つずつ返せるオブジェクトです。反復可能オブジェクトの例には、(`list`, `str`, `tuple` といった) 全てのシーケンス型や、`dict` や **ファイルオブジェクト** といった幾つかの非シーケンス型、あるいは *Sequence* 意味論を実装した `__iter__()` メソッドか `__getitem__()` メソッドを持つ任意のクラスのインスタンスが含まれます。

反復可能オブジェクトは `for` ループ内やその他多くのシーケンス (訳注: ここでのシーケンスとは、シーケンス型ではなくただの列という意味) が必要となる状況 (`zip()`, `map()`, ...) で利用できます。反復可能オブジェクトを組み込み関数 `iter()` の引数として渡すと、オブジェクトに対するイテレータを返します。このイテレータは一連の値を引き渡す際に便利です。通常は反復可能オブジェクトを使う際には、`iter()` を呼んだりイテレータオブジェクトを自分で操作する必要はありません。`for` 文ではこの操作を自動的に行い、一時的な無名の変数を作成してループを回している間イテレータを保持します。**イテレータ**、**シーケンス**、**ジェネレータ** も参照してください。

iterator (イテレータ) データの流れを表現するオブジェクトです。イテレータの `__next__()` メソッドを繰り返し呼び出す (または組み込み関数 `next()` に渡す) と、流れの中の要素を一つずつ返します。データがなくなると、代わりに `StopIteration` 例外を送出します。その時点で、イテレータオブジェクトは尽きており、それ以降は `__next__()` を何度呼んでも `StopIteration` を送じます。イテレータは、そのイテレータオブジェクト自体を返す `__iter__()` メソッドを実装しなければならないので、イテレータは他の `iterable` を受理するほとんどの場所で利用できます。はっきりとした例外は複数の反復を行うようなコードです。(`list` のような) コンテナオブジェクトは、自身を `iter()` 関数にオブジェクトに渡したり `for` ループ内で使うたびに、新たな未使用のイテレータを生成します。これをイテレータで行おうとすると、前回のイテレーションで使用済みの同じイテレータオブジェクトを単純に返すため、空のコンテナのようになってしまいます。

詳細な情報は `typeiter` にあります。

key function (キー関数) キー関数、あるいは照合関数とは、ソートや順序比較のための値を返す呼び出し可能オブジェクト (callable) です。例えば、`locale.strxfrm()` をキー関数に使用すれば、ロケール依存のソートの慣習にのっとったソートキーを返します。

Python の多くのツールはキー関数を受け取り要素の並び順やグループ化を管理します。`min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, `itertools.groupby()` 等があります。

キー関数を作る方法はいくつかあります。例えば `str.lower()` メソッドを大文字小文字を区別しないソートを行うキー関数として使うことが出来ます。あるいは、`lambda r: (r[0], r[2])` のような `lambda` 式からキー関数を作ることができます。また、`operator` モジュールは `attrgetter()`, `itemgetter()`, `methodcaller()` という 3 つのキー関数コンストラクタを提供しています。キー関数の作り方と使い方の例は [Sorting HOW TO](#) を参照してください。

keyword argument [実引数](#) を参照してください。

lambda (ラムダ) 無名のインライン関数で、関数が呼び出されたときに評価される 1 つの [式](#) を含みます。ラムダ関数を作る構文は `lambda [parameters]: expression` です。

LBYL 「ころばぬ先の杖 (look before you leap)」の略です。このコーディングスタイルでは、呼び出しや検索を行う前に、明示的に前提条件 (pre-condition) 判定を行います。[EAFP](#) アプローチと対照的で、`if` 文がたくさん使われるのが特徴的です。

マルチスレッド化された環境では、LBYL アプローチは ” 見る ” 過程と ” 飛ぶ ” 過程の競合状態を引き起こすリスクがあります。例えば、`if key in mapping: return mapping[key]` というコードは、判定の後、別のスレッドが探索の前に `mapping` から `key` を取り除くと失敗します。この問題は、ロックするか EAFP アプローチを使うことで解決できます。

list (リスト) Python の組み込みの [シーケンス](#) です。リストという名前ですが、リンクリストではなく、他の言語で言う配列 (array) と同種のもので、要素へのアクセスは $O(1)$ です。

list comprehension (リスト内包表記) シーケンス中の全てあるいは一部の要素を処理して、その結果からなるリストを返す、コンパクトな方法です。`result = ['{:04x}'.format(x) for x in range(256) if x % 2 == 0]` とすると、0 から 255 までの偶数を 16 進数表記 (0x..) した文字列からなるリストを生成します。`if` 節はオプションです。`if` 節がない場合、`range(256)` の全ての要素が処理されます。

loader モジュールをロードするオブジェクト。`load_module()` という名前のメソッドを定義していなければなりません。ローダーは一般的に [finder](#) から返されます。詳細は [PEP 302](#) を、[abstract base class](#) については `importlib.abc.Loader` を参照してください。

magic method [special method](#) のくだけた同義語です。

mapping (マッピング) 任意のキー探索をサポートしていて、`Mapping` か `MutableMapping` の抽象基底クラスで指定されたメソッドを実装しているコンテナオブジェクトです。例えば、`dict`, `collections.defaultdict`, `collections.OrderedDict`, `collections.Counter` などです。

meta path finder `sys.meta_path` を検索して得られた *finder*. meta path finder は *path entry finder* と関係はありますが、別物です。

meta path finder が実装するメソッドについては `importlib.abc.MetaPathFinder` を参照してください。

metaclass (メタクラス) クラスのクラスです。クラス定義は、クラス名、クラスの辞書と、基底クラスのリストを作ります。メタクラスは、それら 3 つを引数として受け取り、クラスを作る責任を負います。ほとんどのオブジェクト指向言語は (訳注:メタクラスの) デフォルトの実装を提供しています。Python が特別なのはカスタムのメタクラスを作成できる点です。ほとんどのユーザーにとって、メタクラスは全く必要のないものです。しかし、一部の場面では、メタクラスは強力でエレガントな方法を提供します。たとえば属性アクセスのログを取ったり、スレッドセーフ性を追加したり、オブジェクトの生成を追跡したり、シングルトンを実装するなど、多くの場面で利用されます。

詳細は `metaclasses` を参照してください。

method (メソッド) クラス本体の中で定義された関数。そのクラスのインスタンスの属性として呼び出された場合、メソッドはインスタンスオブジェクトを第一 **引数** として受け取ります (この第一引数は通常 `self` と呼ばれます)。**関数** と **ネストされたスコープ** も参照してください。

method resolution order (メソッド解決順序) 探索中に基底クラスが構成要素を検索される順番です。2.3 以降の Python インタープリタが使用するアルゴリズムの詳細については [The Python 2.3 Method Resolution Order](#) を参照してください。

module (モジュール) Python コードの組織単位としてはたらくオブジェクトです。モジュールは任意の Python オブジェクトを含む名前空間を持ちます。モジュールは *importing* の処理によって Python に読み込まれます。

パッケージ を参照してください。

module spec モジュールをロードするのに使われるインポート関連の情報を含む名前空間です。`importlib.machinery.ModuleSpec` のインスタンスです。

MRO *method resolution order* を参照してください。

mutable (ミュータブル) ミュータブルなオブジェクトは、`id()` を変えることなく値を変更できます。**イミュータブル** も参照してください。

named tuple ”名前付きタプル”という用語は、タプルを継承していて、インデックスが付く要素に対し属性を使ってのアクセスもできる任意の型やクラスに適用されています。その型やクラスは他の機能も持っていることもあります。

`time.localtime()` や `os.stat()` の返り値を含むいくつかの組み込み型は名前付きタプルです。他の例は `sys.float_info` です:

```
>>> sys.float_info[1]                # indexed access
1024
>>> sys.float_info.max_exp            # named field access
1024
>>> isinstance(sys.float_info, tuple) # kind of tuple
True
```

(上の例のように) いくつかの名前付きタプルは組み込み型になっています。その他にも名前付きタプルは、通常のクラス定義で `tuple` を継承し、名前のフィールドを定義して作成できます。そのようなクラスは手動で書いたり、`collections.namedtuple()` ファクトリ関数で作成したりできます。後者の方法は、手動で書いた名前付きタプルや組み込みの名前付きタプルには無い付加的なメソッドを追加できます。

namespace (名前空間) 変数が格納される場所です。名前空間は辞書として実装されます。名前空間にはオブジェクトの (メソッドの) 入れ子になったものだけでなく、局所的なもの、大域的なもの、そして組み込みのものがあります。名前空間は名前の衝突を防ぐことによってモジュール性をサポートする。例えば関数 `builtins.open` と `os.open()` は名前空間で区別されています。また、どのモジュールが関数を実装しているか明示することによって名前空間は可読性と保守性を支援します。例えば、`random.seed()` や `itertools.islice()` と書くと、それぞれモジュール `random` や `itertools` で実装されていることが明らかです。

namespace package (名前空間パッケージ) サブパッケージのコンテナとして提供される **PEP 420 package**。Namespace package はおそらく物理表現を持たず、`__init__.py` ファイルがないため、*regular package* と異なります。

module を参照してください。

nested scope (ネストされたスコープ) 外側で定義されている変数を参照する機能です。例えば、ある関数が別の関数の中で定義されている場合、内側の関数は外側の関数中の変数を参照できます。ネストされたスコープはデフォルトでは変数の参照だけができ、変数の代入はできないので注意してください。ローカル変数は、最も内側のスコープで変数を読み書きします。同様に、グローバル変数を使うとグローバル名前空間の値を読み書きします。`nonlocal` で外側の変数に書き込みます。

new-style class (新スタイルクラス) 今では全てのクラスオブジェクトに使われている味付けの古い名前です。以前の Python のバージョンでは、新スタイルクラスのみが `__slots__`、デスクリプタ、`__getattr__()`、クラスメソッド、そして静的メソッド等の Python の新しい、多様な機能を利用できました。

object (オブジェクト) 状態 (属性や値) と定義された振る舞い (メソッド) をもつ全てのデータ。もしくは、全ての **新スタイルクラス** の究極の基底クラスのこと。

package (パッケージ) サブモジュールや再帰的にサブパッケージを含むことの出来る *module* のことです。専門的には、パッケージは `__path__` 属性を持つ Python オブジェクトです。

regular package と *namespace package* を参照してください。

parameter (仮引数) 名前付の実体で **関数** (や **メソッド**) の定義において関数が受ける **実引数** を指定します。仮

引数には5種類あります:

- **位置またはキーワード:** **位置** または **キーワード引数** として渡すことができる引数を指定します。これはたとえば以下の *foo* や *bar* のように、デフォルトの仮引数の種類です:

```
def func(foo, bar=None): ...
```

- **位置のみ:** 位置によってのみ与えられる引数を指定します。Python に引数が位置のみであることを定義する文法はありませんが、組み込み関数には位置のみの引数を持つもの (例: *abs()*) があります。
- **キーワード専用:** キーワードによってのみ与えられる引数を指定します。キーワード専用の引数を定義できる場所は、例えば以下の *kw_only1* や *kw_only2* のように、関数定義の仮引数リストに含めた可変長位置引数または裸の *** の後です:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- **可変長位置:** (他の仮引数で既に受けられた任意の位置引数に加えて) 任意の個数の位置引数が与えられることを指定します。このような仮引数は、以下の *args* のように仮引数名の前に *** をつけることで定義できます:

```
def func(*args, **kwargs): ...
```

- **可変長キーワード:** (他の仮引数で既に受けられた任意のキーワード引数に加えて) 任意の個数のキーワード引数が与えられることを指定します。このような仮引数は、上の例の *kwargs* のように仮引数名の前に **** をつけることで定義できます。

仮引数はオプションと必須の引数のどちらも指定でき、オプションの引数にはデフォルト値も指定できます。

仮引数、FAQ の **実引数と仮引数の違いは何ですか?**、`inspect.Parameter` クラス、`function` セクション、**PEP 362** を参照してください。

path entry *path based finder* が `import` するモジュールを探す *import path* 上の1つの場所です。

path entry finder `sys.path_hooks` にある callable (つまり *path entry hook*) が返した *finder* です。与えられた *path entry* にあるモジュールを見つける方法を知っています。

パスエントリーファインダが実装するメソッドについては `importlib.abc.PathEntryFinder` を参照してください。

path entry hook `sys.path_hook` リストにある callable で、指定された *path entry* にあるモジュールを見つける方法を知っている場合に *path entry finder* を返します。

path based finder デフォルトの *meta path finder* の1つは、モジュールの *import path* を検索します。

path-like object (path-like オブジェクト) ファイルシステムパスを表します。path-like オブジェクトは、パスを表す `str` オブジェクトや `bytes` オブジェクト、または `os.PathLike` プロトコルを実装したオブジェクト

のどれかです。os.PathLike プロトコルをサポートしているオブジェクトは os.fspath() を呼び出すことで str または bytes のファイルシステムパスに変換できます。os.fsdecode() と os.fsencode() はそれぞれ str あるいは bytes になるのを保証するのに使えます。PEP 519 で導入されました。

PEP Python Enhancement Proposal. PEP は、Python コミュニティに対して情報を提供する、あるいは Python の新機能やその過程や環境について記述する設計文書です。PEP は、機能についての簡潔な技術的仕様と提案する機能の論拠 (理論) を伝えるべきです。

PEP は、新機能の提案にかかる、コミュニティによる問題提起の集積と Python になされる設計決断の文書化のための最上位の機構となることを意図しています。PEP の著者にはコミュニティ内の合意形成を行うこと、反対意見を文書化することの責務があります。

PEP 1 を参照してください。

portion PEP 420 で定義されている、namespace package に属する、複数のファイルが (zip ファイルに格納されている場合もある) 1 つのディレクトリに格納されたもの。

位置引数 (positional argument) 実引数 を参照してください。

provisional API (暫定 API) 標準ライブラリの後方互換性保証から計画的に除外されたものです。そのようなインタフェースへの大きな変更は、暫定であるとされている間は期待されていませんが、コア開発者によって必要とみなされれば、後方非互換な変更 (インタフェースの削除まで含まれる) が行われえます。このような変更はむやみに行われるものではありません -- これは API を組み込む前には見落とされていた重大な欠陥が露呈したときにのみ行われます。

暫定 API についても、後方互換性のない変更は「最終手段」とみなされています。問題点が判明した場合でも後方互換な解決策を探すべきです。

このプロセスにより、標準ライブラリは問題となるデザインエラーに長い間閉じ込められることなく、時代を超えて進化を続けられます。詳細は PEP 411 を参照してください。

provisional package provisional API を参照してください。

Python 3000 Python 3.x リリースラインのニックネームです。(Python 3 が遠い将来の話だった頃に作られた言葉です。) "Py3k" と略されることもあります。

Pythonic 他の言語で一般的な考え方で書かれたコードではなく、Python の特に一般的なイディオムに従った考え方やコード片。例えば、Python の一般的なイディオムでは for 文を使ってイテラブルのすべての要素に渡ってループします。他の多くの言語にはこの仕組みはないので、Python に慣れていない人は代わりに数値のカウンターを使うかもしれません:

```
for i in range(len(food)):
    print(food[i])
```

これに対し、きれいな Pythonic な方法は:

```
for piece in food:
    print(piece)
```

qualified name (修飾名) モジュールのグローバルスコープから、そのモジュールで定義されたクラス、関数、メソッドへの、“パス”を表すドット名表記です。**PEP 3155** で定義されています。トップレベルの関数やクラスでは、修飾名はオブジェクトの名前と同じです:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

モジュールへの参照で使われると、**完全修飾名** (*fully qualified name*) はすべての親パッケージを含む全体のドット名表記、例えば `email.mime.text` を意味します:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count (参照カウント) あるオブジェクトに対する参照の数。参照カウントが 0 になったとき、そのオブジェクトは破棄されます。参照カウントは通常は Python のコード上には現れませんが、*CPython* 実装の重要な要素です。`sys` モジュールは、プログラマーが任意のオブジェクトの参照カウントを知るための `getrefcount()` 関数を提供しています。

regular package 伝統的な、`__init__.py` ファイルを含むディレクトリとしての *package*。

namespace package を参照してください。

`__slots__` クラス内での宣言で、インスタンス属性の領域をあらかじめ定義しておき、インスタンス辞書を排除することで、メモリを節約します。これはよく使われるテクニックですが、正しく扱うには少しトリッキーなので、稀なケース、例えばメモリが死活問題となるアプリケーションでインスタンスが大量に存在する、といったときを除き、使わないのがベストです。

sequence (シーケンス) 整数インデックスによる効率的な要素アクセスを `__getitem__()` 特殊メソッドを通じてサポートし、長さを返す `__len__()` メソッドを定義した *iterable* です。組み込みシーケンス型には、`list`, `str`, `tuple`, `bytes` などがあります。`dict` は `__getitem__()` と `__len__()` もサポートしますが、検索の際に整数ではなく任意の *immutable* なキーを使うため、シーケンスではなくマッピング (mapping) とみなされているので注意してください。

`collections.abc.Sequence` 抽象基底クラスは `__getitem__()` や `__len__()` だけでなく `count()`、`index()`、`__contains__()`、`__reversed__()` よりも豊富なインターフェイスを定義しています。この拡張されたインターフェイスを実装している型は `register()` を使用することで明示的に登録することが出来ます。

single dispatch *generic function* の一種で実装は一つの引数の型により選択されます。

slice (スライス) 一般に **シーケンス** の一部を含むオブジェクト。スライスは、添字表記 `[]` で与えられた複数の数の間にコロンを書くことで作られます。例えば、`variable_name[1:3:5]` です。角括弧 (添字) 記号は `slice` オブジェクトを内部で利用しています。

special method (特殊メソッド) ある型に特定の操作、例えば加算をするために Python から暗黙に呼び出されるメソッド。この種類のメソッドは、メソッド名の最初と最後にアンダースコア 2 つがついています。特殊メソッドについては `specialnames` で解説されています。

statement (文) 文はスイート (コードの”ブロック”) に不可欠な要素です。文は **式** かキーワードから構成されるもののどちらかです。後者には `if`、`while`、`for` があります。

text encoding ユニコード文字列をエンコードするコーデックです。

text file (テキストファイル) `str` オブジェクトを読み書きできる *file object* です。しばしば、テキストファイルは実際にバイト指向のデータストリームにアクセスし、**テキストエンコーディング** を自動的に行います。テキストファイルの例は、`sys.stdin`、`sys.stdout`、`io.StringIO` インスタンスなどをテキストモード (`'r'` or `'w'`) で開いたファイルです。

bytes-like **オブジェクト** を読み書きできるファイルオブジェクトについては、**バイナリファイル** も参照してください。

triple-quoted string (三重クォート文字列) 3つの連続したクォート記号 (`'''`) かアポストロフィー (`'`) で囲まれた文字列。通常の (一重) クォート文字列に比べて表現できる文字列に違いはありませんが、幾つかの理由で有用です。1つか2つの連続したクォート記号をエスケープ無しに書くことができますし、行継続文字 (`\`) を使わなくても複数行にまたがるできるので、ドキュメンテーション文字列を書く時に特に便利です。

type (型) Python オブジェクトの型はオブジェクトがどのようなものかを決めます。あらゆるオブジェクトは型を持っています。オブジェクトの型は `__class__` 属性でアクセスしたり、`type(obj)` で取得したり出来ます。

type alias (型エイリアス) 型の別名で、型を識別子に代入して作成します。

型エイリアスは **型ヒント** を単純化するのに有用です。例えば:

```
from typing import List, Tuple

def remove_gray_shades(
    colors: List[Tuple[int, int, int]] -> List[Tuple[int, int, int]]:
    pass
```

これは次のようにより読みやすくなります:

```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

機能の説明がある `typing` と [PEP 484](#) を参照してください。

type hint (型ヒント) 変数、クラス属性、関数のパラメータや戻り値の期待される型を指定する *annotation* です。

型ヒントは必須ではなく Python では強制ではありませんが、静的型解析ツールにとって有用であり、IDE のコード補完とリファクタリングの手助けになります。

グローバル変数、クラス属性、関数で、ローカル変数でないものの型ヒントは `typing.get_type_hints()` で取得できます。

機能の説明がある `typing` と [PEP 484](#) を参照してください。

universal newlines テキストストリームの解釈法の一つで、以下のすべてを行末と認識します: Unix の行末規定 `'\n'`、Windows の規定 `'\r\n'`、古い Macintosh の規定 `'\r'`。利用法について詳しくは、[PEP 278](#) と [PEP 3116](#)、さらに `bytes.splitlines()` も参照してください。

variable annotation (変数アノテーション) 変数あるいはクラス属性の *annotation*。

変数あるいはクラス属性に注釈を付けたときは、代入部分は任意です:

```
class C:
    field: 'annotation'
```

変数アノテーションは通常は **型ヒント** のために使われます: 例えば、この変数は `int` の値を取ることを期待されています:

```
count: int = 0
```

変数アノテーションの構文については [annassign](#) 節で解説しています。

この機能について解説している *function annotation*, [PEP 484](#), [PEP 526](#) を参照してください。

virtual environment (仮想環境) 協調的に切り離された実行環境です。これにより Python ユーザとアプリケーションは同じシステム上で動いている他の Python アプリケーションの挙動に干渉することなく Python パッケージのインストールと更新を行うことができます。

`venv` を参照してください。

virtual machine (仮想マシン) 完全にソフトウェアにより定義されたコンピュータ。Python の仮想マシンは、バイトコードコンパイラが出力した **バイトコード** を実行します。

Zen of Python (Python の悟り) Python を理解し利用する上での導きとなる、Python の設計原則と哲学をリストにしたものです。対話プロンプトで `"import this"` とするとこのリストを読めます。

このドキュメントについて

このドキュメントは、Python のドキュメントを主要な目的として作られた ドキュメントプロセッサの [Sphinx](#) を利用して、[reStructuredText](#) 形式のソースから生成されました。

ドキュメントとそのツール群の開発は、Python 自身と同様に完全にボランティアの努力です。もしあなたが貢献したいなら、どのようにすればよいかについて [reporting-bugs](#) ページをご覧ください。新しいボランティアはいつでも歓迎です! (訳注: 日本語訳の問題については、GitHub 上の [Issue Tracker](#) で報告をお願いします。)

多大な感謝を:

- Fred L. Drake, Jr., オリジナルの Python ドキュメントツールセットの作成者で、ドキュメントの多くを書きました。
- [Docutils](#) プロジェクト. [reStructuredText](#) と [docutils](#) ツールセットを作成しました。
- Fredrik Lundh の [Alternative Python Reference](#) プロジェクトから Sphinx は多くのアイデアを得ました。

B.1 Python ドキュメント 貢献者

多くの方々が Python 言語、Python 標準ライブラリ、そして Python ドキュメンテーションに貢献してくれています。ソース配布物の [Misc/ACKS](#) に、それら貢献してくれた人々を部分的にはありますがリストアップしてあります。

Python コミュニティからの情報提供と貢献がなければこの素晴らしいドキュメンテーションは生まれませんでした -- ありがとう!

歴史とライセンス

C.1 Python の歴史

Python は 1990 年代の始め、オランダにある Stichting Mathematisch Centrum (CWI, <https://www.cwi.nl/> 参照) で Guido van Rossum によって ABC と呼ばれる言語の後継言語として生み出されました。その後多くの人々が Python に貢献していますが、Guido は今日でも Python 製作者の先頭に立っています。

1995 年、Guido は米国ヴァージニア州レストンにある Corporation for National Research Initiatives (CNRI, <https://www.cnri.reston.va.us/> 参照) で Python の開発に携わり、いくつかのバージョンをリリースしました。

2000 年 3 月、Guido と Python のコア開発チームは BeOpen.com に移り、BeOpen PythonLabs チームを結成しました。同年 10 月、PythonLabs チームは Digital Creations (現在の Zope Corporation, <https://www.zope.org/> 参照) に移りました。そして 2001 年、Python に関する知的財産を保有するための非営利組織 Python Software Foundation (PSF, <https://www.python.org/psf/> 参照) を立ち上げました。このとき Zope Corporation は PSF の賛助会員になりました。

Python のリリースは全てオープンソース (オープンソースの定義は <https://opensource.org/> を参照してください) です。歴史的にみて、ごく一部を除くほとんどの Python リリースは GPL 互換になっています; 各リリースについては下表にまとめてあります。

リリース	ベース	西暦年	権利	GPL 互換
0.9.0 - 1.2	n/a	1991-1995	CWI	yes
1.3 - 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 以降	2.1.1	2001-現在	PSF	yes

注釈: 「GPL 互換」という表現は、Python が GPL で配布されているという意味ではありません。Python のライセンスは全て、GPL と違い、変更したバージョンを配布する際に変更をオープンソースにしなくてもかまいません。GPL 互換のライセンスの下では、GPL でリリースされている他のソフトウェアと Python を組み合わせられますが、それ以外のライセンスではそうではありません。

Guido の指示の下、これらのリリースを可能にくださった多くのボランティアのみなさんに感謝します。

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

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.7.17

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 3.7.17 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 3.7.17 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All Rights Reserved" are retained in Python 3.7.17 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 3.7.17 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 3.7.17.
4. PSF is making Python 3.7.17 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 3.7.17 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.7.17 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.7.17, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 3.7.17, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").
2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative

(次のページに続く)

(前のページからの続き)

- works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.
3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
 4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
 5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
 6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
 7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All

(次のページに続く)

(前のページからの続き)

Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: <http://hdl.handle.net/1895.22/1013>."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.
4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

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

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

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

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` module includes code based on a download from <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html>. The following are the verbatim comments from the original code:

A C-program for MT19937, with initialization improved 2002/1/26.

Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using `init_genrand(seed)`
or `init_by_array(init_key, key_length)`.

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

(次のページに続く)

(前のページからの続き)

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The names of its contributors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.

<http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html>

email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)

C.3.2 ソケット

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, <http://www.wide.ad.jp/>.

Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.

All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors

(次のページに続く)

(前のページからの続き)

```
may be used to endorse or promote products derived from this software
without specific prior written permission.
```

```
THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Asynchronous socket services

The `asyncio` and `asynchat` modules contain the following notice:

```
Copyright 1996 by Sam Rushing
```

```
    All Rights Reserved
```

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

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


C.3.4 Cookie management

The `http.cookies` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>
```

```
All Rights Reserved
```

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

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

C.3.5 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err... reserved and offered to the public under the terms of the
Python 2.2 license.
```

```
Author: Zooko O'Whielacronx
```

```
http://zooko.com/
```

```
mailto:zooko@zooko.com
```

```
Copyright 2000, Mojam Media, Inc., all rights reserved.
```

```
Author: Skip Montanaro
```

```
Copyright 1999, Bioreason, Inc., all rights reserved.
```

```
Author: Andrew Dalke
```

```
Copyright 1995-1997, Automatrix, Inc., all rights reserved.
```

```
Author: Skip Montanaro
```

(次のページに続く)

(前のページからの続き)

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of neither Automatrix, Bioreason or Mojam Media be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

C.3.6 UUencode and UUdecode functions

The uu module contains the following notice:

Copyright 1994 by Lance Ellinghouse

Cathedral City, California Republic, United States of America.

All Rights Reserved

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

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

Modified by Jack Jansen, CWI, July 1995:

- Use binascii module to do the actual line-by-line conversion between ascii and binary. This results in a 1000-fold speedup. The C version is still 5 times faster, though.
- Arguments more compliant with Python standard

C.3.7 XML Remote Procedure Calls

The `xmlrpc.client` module contains the following notice:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
 Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

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

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

C.3.8 test_epoll

`test_epoll` モジュールは次の告知を含んでいます:

Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be

(次のページに続く)

(前のページからの続き)

included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

C.3.9 Select kqueue

select モジュールは kqueue インターフェースについての次の告知を含んでいます:

Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

C.3.10 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski's implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
    https://github.com/majek/csiphash/

Solution inspired by code from:
    Samuel Neves (supercop/crypto_auth/siphash24/little)
    djb (supercop/crypto_auth/siphash24/little2)
    Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod と dtoa

The file `Python/dtoa.c`, which supplies C functions `dtoa` and `strtod` for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from <http://www.netlib.org/fp/>. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/******
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
```

(次のページに続く)

(前のページからの続き)

```
*
* THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
* WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
* REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
* OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
*
*****/
```

C.3.12 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

LICENSE ISSUES

=====

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

OpenSSL License

```
/* =====
 * Copyright (c) 1998-2008 The OpenSSL Project. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in
 * the documentation and/or other materials provided with the
 * distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 * software must display the following acknowledgment:
 * "This product includes software developed by the OpenSSL Project
 * for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
```

(次のページに続く)

(前のページからの続き)

```

*
* 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
*     endorse or promote products derived from this software without
*     prior written permission. For written permission, please contact
*     openssl-core@openssl.org.
*
* 5. Products derived from this software may not be called "OpenSSL"
*     nor may "OpenSSL" appear in their names without prior written
*     permission of the OpenSSL Project.
*
* 6. Redistributions of any form whatsoever must retain the following
*     acknowledgment:
*     "This product includes software developed by the OpenSSL Project
*     for use in the OpenSSL Toolkit (http://www.openssl.org/)"
*
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com).  This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

```

Original SSLeay License

```

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
*
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
*
* This library is free for commercial and non-commercial use as long as
* the following conditions are aheared to.  The following conditions

```

(次のページに続く)

```
* apply to all code found in this distribution, be it the RC4, RSA,
* lhash, DES, etc., code; not just the SSL code. The SSL documentation
* included with this distribution is covered by the same copyright terms
* except that the holder is Tim Hudson (tjh@cryptsoft.com).
*
* Copyright remains Eric Young's, and as such any Copyright notices in
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
* notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
* notice, this list of conditions and the following disclaimer in the
* documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
* must display the following acknowledgement:
* "This product includes cryptographic software written by
* Eric Young (eay@cryptsoft.com)"
* The word 'cryptographic' can be left out if the routines from the library
* being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
* the apps directory (application code) you must include an acknowledgement:
* "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/
```


C.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                        and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

The `_ctypes` extension is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

```
Copyright (c) 1996-2008 Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
```

(次のページに続く)

(前のページからの続き)

```
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

The `zlib` extension is built using an included copy of the `zlib` sources if the `zlib` version found on the system is too old to be used for the build:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler
```

```
This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.
```

```
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:
```

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

```
Jean-loup Gailly      Mark Adler
jloup@gzip.org        madler@alumni.caltech.edu
```

C.3.16 cfuhash

The implementation of the hash table used by the `tracemalloc` で使用しているハッシュテーブルの実装は、cfuhash プロジェクトのものに基づきます:

```
Copyright (c) 2005 Don Owens  
All rights reserved.
```

```
This code is released under the BSD license:
```

```
Redistribution and use in source and binary forms, with or without  
modification, are permitted provided that the following conditions  
are met:
```

- * Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- * Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- * Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

```
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS  
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT  
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS  
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE  
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,  
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES  
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR  
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)  
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,  
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)  
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED  
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

The `_decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2016 Stefan Krah. All rights reserved.
```

```
Redistribution and use in source and binary forms, with or without  
modification, are permitted provided that the following conditions  
are met:
```

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

```
THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND  
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE  
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE  
ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE  
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL  
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS  
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)  
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT  
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY  
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF  
SUCH DAMAGE.
```

付録

D

COPYRIGHT

Python and this documentation is:

Copyright © 2001-2023 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

ライセンスおよび許諾に関する完全な情報は、[歴史とライセンス](#) を参照してください。

索引

アルファベット以外

..., 89
 2to3, 89
 >>>, 89
 __future__, 95
 __slots__, 103
 クラス, 91
 コルーチン, 92
 ジェネレータ, 95
 位置引数 (*positional argument*), 102
 環境変数
 PYTHONPATH, 74
 組み込み関数
 repr, 65

A

abstract base class, 89
 annotation, 89
 asynchronous context manager, 90
 asynchronous generator, 90
 asynchronous generator iterator, 90
 asynchronous iterable, 90
 asynchronous iterator, 91
 awaitable, 91

B

BDFL, 91
 binary file, 91
 bytecode, 91
 bytes-like object, 91

C

C-contiguous, 92
 class variable, 91
 coercion, 92
 complex number, 92
 context manager, 92
 context variable, 92
 contiguous, 92
 coroutine function, 92
 CPython, 92

D

deallocation, object, 64
 decorator, 92
 descriptor, 93
 dictionary, 93
 dictionary view, 93
 docstring, 93
 duck-typing, 93

E

EAFP, 93
 expression, 94
 extension module, 94

F

f-string, 94
 file object, 94
 file-like object, 94
 finalization, of objects, 64
 finder, 94
 floor division, 94
 Fortran contiguous, 92
 function, 94
 function annotation, 94

G

garbage collection, 95
 generator, 95
 generator expression, 95
 generator iterator, 95
 generic function, 95
 GIL, 95
 global interpreter lock, 95

H

hash-based pyc, 96
 hashable, 96

I

IDLE, 96
 immutable, 96
 import path, 96
 importer, 96
 importing, 96
 interactive, 96
 interpreted, 97
 interpreter shutdown, 97
 iterable, 97
 iterator, 97

K

key function, 98
 keyword argument, 98

L

lambda, 98
 LBYL, 98

list, 98
list comprehension, 98
loader, 98

M

magic
 method, 98
magic method, 98
mapping, 98
meta path finder, 99
metaclass, 99
method, 99
 magic, 98
 special, 104
method resolution order, 99
module, 99
module spec, 99
MRO, 99
mutable, 99

N

named tuple, 99
namespace, 100
namespace package, 100
nested scope, 100
new-style class, 100

O

object, 100
 deallocation, 64
 finalization, 64

P

package, 100
parameter, 100
path based finder, 101
path entry, 101
path entry finder, 101
path entry hook, 101
path-like object, 101
PEP, 102
Philbrick, Geoff, 18
portion, 102
provisional API, 102
provisional package, 102
PyArg_ParseTuple(), 16
PyArg_ParseTupleAndKeywords(), 18
PyErr_Fetch(), 64
PyErr_Restore(), 64
PyInit_modulename (*C の関数*), 74
PyObject_CallObject(), 15
Python 3000, 102
Python Enhancement Proposals
 PEP 1, 102
 PEP 238, 94
 PEP 278, 105
 PEP 302, 94, 98
 PEP 343, 92
 PEP 362, 90, 101
 PEP 411, 102
 PEP 420, 94, 100, 102
 PEP 442, 65
 PEP 443, 95
 PEP 451, 94
 PEP 484, 89, 95, 105

PEP 489, 13, 75
PEP 492, 9092
PEP 498, 94
PEP 519, 102
PEP 525, 90
PEP 526, 89, 105
PEP 3116, 105
PEP 3155, 103

Pythonic, 102
PYTHONPATH, 74

Q

qualified name, 103

R

READ_RESTRICTED, 68
READONLY, 68
reference count, 103
regular package, 103
repr
 組み込み関数, 65
RESTRICTED, 68

S

sequence, 103
single dispatch, 104
slice, 104
special
 method, 104
special method, 104
statement, 104
string
 object representation, 65

T

text encoding, 104
text file, 104
triple-quoted string, 104
type, 104
type alias, 104
type hint, 105

U

universal newlines, 105

V

variable annotation, 105
virtual environment, 105
virtual machine, 105
属性, 91
引数 (*argument*), 89

W

WRITE_RESTRICTED, 68

Z

Zen of Python, 106