Extending and Embedding Python

リリース *2.6ja2*

Guido van Rossum
Fred L. Drake, Jr., editor

2011 年 11 月 06 日

Python Software Foundation
Email: docs@python.org

目 次

第 1 章　C や C++ による Python の拡張 **3**
1.1　簡単な例 . . . 3
1.2　幕間小話: エラーと例外 . . . 5
1.3　例に戻る . . . 7
1.4　モジュールのメソッドテーブルと初期化関数 . . . 8
1.5　コンパイルとリンク . . . 9
1.6　C から Python 関数を呼び出す . . . 10
1.7　拡張モジュール関数でのパラメタ展開 . . . 12
1.8　拡張モジュール関数のキーワードパラメタ . . . 13
1.9　任意の値を構築する . . . 14
1.10　参照カウント法 . . . 15
1.11　C++での拡張モジュール作成 . . . 20
1.12　拡張モジュールに C API を提供する . . . 20

第 2 章　新しい型を定義する **25**
2.1　基本的なこと . . . 25
2.2　タイプメソッド . . . 52

第 3 章　distutils による C および C++ 拡張モジュールのビルド **65**
3.1　拡張モジュールの配布 . . . 66

第 4 章　Windows 上での C および C++ 拡張モジュールのビルド **69**
4.1　型どおりのアプローチ . . . 69
4.2　Unix と Windows の相違点 . . . 72
4.3　DLL 使用の実際 . . . 73

第 5 章　他のアプリケーションへの Python の埋め込み **75**
5.1　高水準の埋め込み . . . 76
5.2　超高水準の埋め込みから踏み出す: 概要 . . . 76
5.3　純粋な埋め込み . . . 77
5.4　埋め込まれた Python の拡張 . . . 79

5.5 C++による Python の埋め込み . . . 80
5.6 リンクに関する要件 . . . 80

第 6 章 このドキュメントについて **81**
6.1 翻訳者一覧 (敬称略) . . . 81

付 録 A 用語集 **83**

付 録 B このドキュメントについて **91**
B.1 Python ドキュメント 貢献者 . . . 91

付 録 C History and License **93**
C.1 Python の歴史 . . . 93
C.2 Terms and conditions for accessing or otherwise using Python . . . 94
C.3 Licenses and Acknowledgements for Incorporated Software . . . 97

付 録 D Copyright **105**

索引 **107**

Release 2.6

Date 2011 年 11 月 06 日

このドキュメントでは、Python インタプリタを拡張するために C/C++ でモジュールを書く方法について述べます。拡張モジュールでは、新たな関数を定義できるだけでなく、新たなオブジェクト型とそのメソッドも定義できます。このドキュメントではまた、Python インタプリタを別のアプリケーションに埋め込み (embedding)、拡張言語として使う方法についても述べます。また、動的に (実行時に) 拡張モジュールをロードする機能を OS がサポートしている場合に、動的ロード可能な拡張モジュールをコンパイルしてリンクする方法を示します。

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。

第 1 章

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 (*extention 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。

もし C ライブラリ関数やシステムコールを呼び出すような使い方を考えているなら、C のコードをいちいち書く前に ctypes モジュールの使用を検討してください。ctypes モジュールを使うと C のコードを扱う Python のコードが書けるようになるだけでなく、拡張モジュールを書きコンパイルして CPython に縛られてしまうよりも Python の実装間での互換性を高めることができます。

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 インタフェー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。)

[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。

このファイルの最初の行は以下のようにします:

```
#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)。

ノート: 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` 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ければ、 `"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;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", 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 引数は C 関数が Python の関数ではなく組み込みメソッドを実装している場合にのみ使われます。この例ではメソッドではなく関数を定義しているので、 *self* は常に *NULL* ポインタになります。 (これは、インタプリタが二つの異なる形式の C 関数を理解しなくてもよくするためです。)

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。

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、それぞれ Python の変数 `sys.exc_type`、`sys.exc_value` `` および `` `sys.exc_traceback` 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とり、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_*()` 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 **自らを呼び出したコード** に返すことになりますが、同様に `PyErr_*()` 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_*()` 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_*()` 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 `PyInt_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 (initspam()) の中で、例外オブジェクトを使って初期化します (ここではエラーチェックを省略しています):

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

    m = Py_InitModule("spam", SpamMethods);
    if (m == NULL)
        return;

    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    Py_INCREF(SpamError);
    PyModule_AddObject(m, "error", SpamError);
}
```

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

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、`PyArg_ParseTuple()` の逆ともいうべき関数 `Py_BuildValue()` を使います: `Py_BuildValue()` は書式化文字列と任意の数の C の値を引数にとり、新たな Python オブジェクトを返します。`Py_BuildValue()` に関する詳しい情報は後で示します。

```
return Py_BuildValue("i", 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 Pyhton 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。

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() 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; 0 は旧式の PyArg_ParseTuple() の変化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 initname() 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
initspam(void)
{
    (void) Py_InitModule("spam", SpamMethods);
}
```

PyMODINIT_FUNC は関数の戻り値を void 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 プログラムがモジュール spam を初めて import するとき、initspam() が呼び出されます。(Python の埋め込みに関するコメントは下記を参照してください。) initspam() は Py_InitModule() を呼び出して “モジュールオブジェクト” を生成し (オブジェクトは "spam" をキーとして辞書 sys.modules に挿入されます)、第二引数として与えたメソッドテーブル (PyMethodDef 構造体の配列) の情報に基づいて、組み込み関数オブジェクトを新たなモジュールに挿入していきます。 Py_InitModule() は、自らが生成した (この段階ではまだ未使用の) モジュールオブジェクトへのポインタを返します。Py_InitModule() は、幾つかのエラーでは致命的エラーで abort し、それ以外のモジュールが満足に初期化できなかった場合は *NULL* を返します。

Python を埋め込む場合には、 _PyImport_Inittab テーブルのエントリ内に initspam() がない限り、initspam() は自動的には呼び出されません。この問題を解決する最も簡単な方法は、Py_Initialize()

や PyMac_Initialize() を呼び出した後に initspam() を直接呼び出し、静的にリンクしておいたモジュールを静的に初期化してしまうというものです:

```
int
main(int argc, char *argv[])
{
    /* Python インタプリタに argv[0] を渡す */
    Py_SetProgramName(argv[0]);

    /* Python インタプリタを初期化する。必ず必要。 */
    Py_Initialize();

    /* 静的モジュールを追加する */
    initspam();
```

Python ソース配布物中の Demo/embed/demo.c ファイル内に例があります。

ノート: 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.module 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。また、 reload() 関数を拡張モジュールに対して利用でき、この場合はモジュール初期化関数 (initspam()) は呼び出されますが、モジュールが動的にロード可能なオブジェクトファイル (Unix では .so 、Windows では .dll) から読み出された場合にはモジュールファイルを再読み込みしないので注意してください。

より実質的なモジュール例は、Python ソース配布物に Modules/xxmodule.c という名前で入っています。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。ソース配布物や Windows にインストールされた Python に入っている **modulator.py** では、拡張モジュールで実装しなければならない関数やオブジェクトを宣言し、実装部分を埋めて作成するためのテンプレートを生成できるような、簡単なグラフィカルユーザインタフェースを提供しています。このスクリプトは Tools/modulator/ ディレクトリにあります; 詳しくはディレクトリ内の README ファイルを参照してください。

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; 詳しくは、拡張モジュールのビルドに関する章 (*distutils* **による** *C* **および** *C++* **拡張モジュールのビルド** 章) や、Windows におけるビルドに関係する追加情報の章 (:ref :*building-on-windows* 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くなるでしょう。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らりません。(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
```

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際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るでしょう。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);         /* 新たなコールバックへの参照を追加 */
        Py_XDECREF(my_callback);  /* 以前のコールバックを捨てる */
        my_callback = temp;       /* 新たなコールバックを記憶 */
        /* "None" を返す際の定型句 */
        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_XINCREF() 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() の書式化文字列中に、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;
...
/* ここでコールバックを呼ぶ */
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、このタプルは呼び出し直後に Py_DECREF() しています。

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ても (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。例外の送出が不可能だったり、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; /* エラーを返す */
...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; /* エラーを返す */
/* 場合によってはここで結果を使うかもね */
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; /* エラーを返す */
/* 場合によってはここで結果を使うかもね */
Py_DECREF(result);
```

1.7 拡張モジュール関数でのパラメタ展開

PyArg_ParseTuple() は、以下のように宣言されています:

```
int PyArg_ParseTuple(PyObject *arg, 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* 引数は書式化文字列で、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:

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

```
int size;

ok = PyArg_ParseTuple(args, ""); /* 引数なし */
    /* Python での呼び出し: f() */


ok = PyArg_ParseTuple(args, "s", &s); /* 文字列 */
    /* Python での呼び出し例: f('whoops!') */


ok = PyArg_ParseTuple(args, "lls", &k, &l, &s);
    /* 二つの long と文字列 */
    /* Python での呼び出し例: f(1, 2, 'three') */


ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size);
    /* 二つの int と文字列、文字列のサイズも返す */
    /* Python での呼び出し例: f((1, 2), 'three') */


{
    const char *file;
    const char *mode = "r";
    int bufsize = 0;
    ok = PyArg_ParseTuple(args, "s|si", &file, &mode, &bufsize);
    /* 文字列、オプションとして文字列がもう一つと整数が一つ */
    /* Python での呼び出し例:
       f('spam')
       f('spam', 'w')
       f('spam', 'wb', 100000) */
}


{
    int left, top, right, bottom, h, v;
    ok = PyArg_ParseTuple(args, "((ii)(ii))(ii)",
             &left, &top, &right, &bottom, &h, &v);
    /* 矩形と点を表現するデータ */
    /* Python での呼び出し例:
       f(((0, 0), (400, 300)), (10, 10)) */
}


{
    Py_complex c;
    ok = PyArg_ParseTuple(args, "D:myfunction", &c);
    /* 複素数。エラー発生時用に関数名も指定 */
    /* Python での呼び出し例: myfunction(1+2j) */
}
```

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,
                                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。

ノート: 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。

```
#include "Python.h"

static PyObject *
keywdarg_parrot(PyObject *self, PyObject *args, PyObject *keywds)
{
    int voltage;
    char *state = "a stiff";
    char *action = "voom";
    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_INCREF(Py_None);

    return Py_None;
}

static PyMethodDef keywdarg_methods[] = {
    /* PyCFunction の値は PyObject* パラメタを二つだけしか引数に
     * 取らないが、 keywordarg_parrot() は三つとるので、キャストが
     * 必要。
     */
    {"parrot", (PyCFunction)keywdarg_parrot, METH_VARARGS | METH_KEYWORDS,
     "Print a lovely skit to standard output."},
    {NULL, NULL, 0, NULL}   /* センティネル値 */
};

void
initkeywdarg(void)
{
  /* モジュールを作成して関数を追加する */
  Py_InitModule("keywdarg", keywdarg_methods);
}
```

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(char *format, ...);
```

Py_BuildValue() は、 PyArg_ParseTuple() の認識する一連の書式化単位に似た書式化単位を認識します。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() は書式化文字列中に書式化単位が二つかそれ以上入っている場合にのみタプルを構築します。書式化文字列が空なら、 None を返します。きっかり一つの書式化単位なら、その書式化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、書式化文字列を丸括弧で囲います。

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):

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

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。

循環参照検出機構は、ごみとなった循環参照を検出し、Python で実装された後始末関数 (finalizer、`__del__()` メソッド) が定義されていないかぎり、それらのメモリを再利用できます。後始末関数がある場合、検出機構は検出した循環参照を `gc` モジュールに (具体的にはこのモジュールの `garbage` 変数内) に公開します。 `gc` モジュールではまた、検出機構 (`collect()` 関数) を実行する方法や設定用のインタフェース、実行時に検出機構を無効化する機能も公開しています。循環参照検出機構はオプションの機構とみなされています; デフォルトで入ってはいますが、Unix プラットフォーム (Mac OS X も含みます) ではビルド時に **configure** スクリプトの *`--without-cycle-gc`* オプションを使って、他のプラットフォームでは `pyconfig.h` ヘッダの `WITH_CYCLE_GC` 定義をはずして無効にできます。こうして循環参照検出機構を無効化すると、 `gc` モジュールは利用できなくなります。

1.10.1 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を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、参照が必要なくなった際に `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)。

1.10.2 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、`PyInt_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、`PyInt_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。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。

[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!

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。

1.10.3 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, PyInt_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! buf() 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のようになります:

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

    Py_INCREF(item);
    PyList_SetItem(list, 1, PyInt_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んでいて、 `__del__()` メソッドがどうして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
    ... ブロックが起こる何らかの I/O 呼び出し...
    Py_END_ALLOW_THREADS
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

1.10.4 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、こうした関数群はしばしば受け取った関数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。

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。

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なことがあります。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 関数を持っ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: CObject を提供しています。CObject はポインタ (`void*`) を記憶する Python のデータ型です。 CObject 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

[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

しできます。とりわけ、CObject 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 CObject 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 CObject が使われます。エクスポートされているそれぞれの名前を使うと、CObject 自体や、CObject が公表しているアドレスで示される配列内に収められた全ての C API ポインタを得られます。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。

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、 CObject 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);
    return Py_BuildValue("i", 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:

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

を必ず追加してください。

`#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
initspam(void)
{
    PyObject *m;
    static void *PySpam_API[PySpam_API_pointers];
    PyObject *c_api_object;

    m = Py_InitModule("spam", SpamMethods);
    if (m == NULL)
        return;

    /* C API ポインタ配列を初期化する */
    PySpam_API[PySpam_System_NUM] = (void *)PySpam_System;

    /* API ポインタ配列のアドレスが入った CObject を生成する */
    c_api_object = PyCObject_FromVoidPtr((void *)PySpam_API, NULL);

    if (c_api_object != NULL)
        PyModule_AddObject(m, "_C_API", c_api_object);
}
```

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、 initspam() 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

/* spammodule のヘッダファイル */

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

/* C API ポインタの総数 */
#define PySpam_API_pointers 1


#ifdef SPAM_MODULE
/* この部分は spammodule.c をコンパイルする際に使われる */

static PySpam_System_RETURN PySpam_System PySpam_System_PROTO;

#else
/* この部分は spammodule の API を使うモジュール側で使われる */

static void **PySpam_API;

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

/* エラーによる例外の場合には -1 を、成功すると 0 を返す */
static int
import_spam(void)
{
    PyObject *c_api_object;
    PyObject *module;
```

```
    module = PyImport_ImportModule("spam");
    if (module == NULL)
        return -1;

    c_api_object = PyObject_GetAttrString(module, "_C_API");
    if (c_api_object == NULL) {
        Py_DECREF(module);
        return -1;
    }
    if (PyCObject_Check(c_api_object))
        PySpam_API = (void **)PyCObject_AsVoidPtr(c_api_object);

    Py_DECREF(c_api_object);
    Py_DECREF(module);
    return 0;
}

#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
initclient(void)
{
    PyObject *m;

    m = Py_InitModule("client", ClientMethods);
    if (m == NULL)
        return;
    if (import_spam() < 0)
        return;
    /* さらなる初期化処理はここに置ける */
}
```

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すみます。

最後に、CObject 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 *cobjects* 、および CObjects の実装部分 (Python ソースコード配布物中のファイル Include/cobject.h および Objects/cobject.c に述べられています。

脚注

第 2 章

新しい型を定義する

前の章でふれたように、Python では拡張モジュールを書くプログラマが Python のコードから操作できる、新しい型を定義できるようになっています。ちょうど Python の中核にある文字列やリストをつくれるようなものです。

これはそんなにむずかしくはありません。拡張型のためのコードにはすべて、一定のパターンが存在しています。しかし始める前に、いくつか細かいことを理解しておく必要があるでしょう。

ノート: Python 2.2 から、新しい型を定義する方法がかなり変わって (良くなって) います。この文書は Python 2.2 およびそれ以降で新しい型をどうやって定義するかについて述べています。古いバージョンの Python をサポートする必要がある場合は、 この文書の古い版 を参照してください。

2.1 基本的なこと

Python ランタイムでは、すべての Python オブジェクトは `PyObject*` 型の変数として扱います。`PyObject` はさほど大仰なオブジェクトではなく、単にオブジェクトに対する参照回数と、そのオブジェクトの「タイプオブジェクト (type object)」へのポインタを格納しているだけです。重要な役割を果たしているのはこのタイプオブジェクトです。つまりタイプオブジェクトは、例えばあるオブジェクトのある属性が参照されるとか、あるいは別のオブジェクトとの間で乗算を行うといったときに、どの (C の) 関数を呼び出すかを決定しているのです。これらの C 関数は「タイプメソッド (type method)」と呼ばれ、 `[].append` のようなもの (いわゆる「オブジェクトメソッド (object method)」) とは区別しています。

なので、新しいオブジェクトの型を定義したいときは、新しいタイプ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、ここに最小限の、しかし完全な、新しい型を定義するモジュールをあげておきます:

```
#include <Python.h>

typedef struct {
    PyObject_HEAD
    /* Type-specific fields go here. */
```

```
} noddy_NoddyObject;

static PyTypeObject noddy_NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0,                         /*ob_size*/
    "noddy.Noddy",             /*tp_name*/
    sizeof(noddy_NoddyObject), /*tp_basicsize*/
    0,                         /*tp_itemsize*/
    0,                         /*tp_dealloc*/
    0,                         /*tp_print*/
    0,                         /*tp_getattr*/
    0,                         /*tp_setattr*/
    0,                         /*tp_compare*/
    0,                         /*tp_repr*/
    0,                         /*tp_as_number*/
    0,                         /*tp_as_sequence*/
    0,                         /*tp_as_mapping*/
    0,                         /*tp_hash */
    0,                         /*tp_call*/
    0,                         /*tp_str*/
    0,                         /*tp_getattro*/
    0,                         /*tp_setattro*/
    0,                         /*tp_as_buffer*/
    Py_TPFLAGS_DEFAULT,        /*tp_flags*/
    "Noddy objects",           /* tp_doc */
};

static PyMethodDef noddy_methods[] = {
    {NULL}  /* Sentinel */
};

#ifndef PyMODINIT_FUNC        /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
initnoddy(void)
{
    PyObject* m;

    noddy_NoddyType.tp_new = PyType_GenericNew;
    if (PyType_Ready(&noddy_NoddyType) < 0)
        return;

    m = Py_InitModule3("noddy", noddy_methods,
                       "Example module that creates an extension type.");

    Py_INCREF(&noddy_NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&noddy_NoddyType);
}
```

さしあたって覚えておくことは以上ですが、これで前の章からすこしは説明がわかりやすくなっていることと思います。

最初に習うのは、つぎのようなものです:

```
typedef struct {
    PyObject_HEAD
} noddy_NoddyObject;
```

これが Noddy オブジェクトの内容です — このケースでは、ほかの Python オブジェクトが持っているものと何ら変わりはありません。つまり参照カウントと型オブジェクトへのポインタですね。これらは

PyObject_HEAD マクロによって展開されるメンバです。マクロを使う理由は、レイアウトを標準化するためと、デバッグ用ビルド時に特別なデバッグ用のメンバを定義できるようにするためです。この PyObject_HEAD マクロの後にはセミコロンがないことに注意してください。セミコロンはすでにマクロ内に含まれています。うっかり後にセミコロンをつけてしまわないように気をつけて。これはお使いの機種では何の問題も起こらないかもしれませんが、機種によっては、おそらく問題になるのです! (Windows 上では、MS Visual C がこの手のエラーを出し、コンパイルできないことが知られています)

比較のため、以下に標準的な Python の整数型の定義を見てみましょう:

```
typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```

では次にいってみます。かなめの部分、タイプオブジェクトです。

```
static PyTypeObject noddy_NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0,                         /*ob_size*/
    "noddy.Noddy",             /*tp_name*/
    sizeof(noddy_NoddyObject), /*tp_basicsize*/
    0,                         /*tp_itemsize*/
    0,                         /*tp_dealloc*/
    0,                         /*tp_print*/
    0,                         /*tp_getattr*/
    0,                         /*tp_setattr*/
    0,                         /*tp_compare*/
    0,                         /*tp_repr*/
    0,                         /*tp_as_number*/
    0,                         /*tp_as_sequence*/
    0,                         /*tp_as_mapping*/
    0,                         /*tp_hash */
    0,                         /*tp_call*/
    0,                         /*tp_str*/
    0,                         /*tp_getattro*/
    0,                         /*tp_setattro*/
    0,                         /*tp_as_buffer*/
    Py_TPFLAGS_DEFAULT,        /*tp_flags*/
    "Noddy objects",           /* tp_doc */
};
```

object.h の中にある PyTypeObject の定義を見ると、実際にはここに挙げた以上の数のメンバがあるとわかるでしょう。これ以外のメンバは C コンパイラによってゼロに初期化されるので、必要な時を除いてふつうはそれらの値を明示的には指定せずにおきます。

次のものは非常に重要なので、とくに最初の最初に見ておきましょう:

```
PyObject_HEAD_INIT(NULL)
```

これはちょっとぶっきらぼうですね。実際に書きたかったのはこうです:

```
PyObject_HEAD_INIT(&PyType_Type)
```

この場合、タイプオブジェクトの型は「type」という名前になりますが、これは厳密には C の基準に従っておらず、コンパイラによっては文句を言われます。幸いにも、このメンバは PyType_Ready() が埋めてくれます。

```
0,                          /* ob_size */
```

ヘッダ中の ob_size メンバは使われていません。これは歴史的な遺物であり、構造体中にこれが存在しているのは古いバージョンの Python 用にコンパイルされた拡張モジュールとのバイナリ上の互換性を保つためです。ここにはつねにゼロを指定してください。

```
"noddy.Noddy",              /* tp_name */
```

これは型の名前です。この名前はオブジェクトのデフォルトの表現形式と、いくつかのエラーメッセージ中で使われます。たとえば:

```
>>> "" + noddy.new_noddy()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot add type "noddy.Noddy" to string
```

注意: この名前はドットで区切られた名前で、モジュール名と、そのモジュール内での型名を両方ふくんでいます。この場合のモジュールは noddy で、型の名前は Noddy ですから、ここでの型名としては noddy.Noddy を指定するわけです。

```
sizeof(noddy_NoddyObject),  /* tp_basicsize */
```

これによって Python は PyObject_New() が呼ばれたときにどれくらいの量のメモリを割り当てればよいのか知ることができます。

ノート: 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。

```
0,                          /* tp_itemsize */
```

これはリストや文字列などの可変長オブジェクトのためのものです。今のところ無視しましょう。

このあとのいくつかのメソッドは使わないのでとばして、クラスのフラグ (flags) には Py_TPFLAGS_DEFAULT を入れます。

```
Py_TPFLAGS_DEFAULT,         /*tp_flags*/
```

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。これは現在のバージョンの Python で定義されているすべてのメンバを許可します。

この型の docstring は tp_doc に入れます。

```
"Noddy objects",            /* tp_doc */
```

ここからタイプメソッドに入るわけですが。ここがあなたのオブジェクトが他と違うところです。でも今回のバージョンでは、これらはどれも実装しないでおき、あとでこの例をより面白いものに改造すること

にしましょう。

とりあえずやりたいのは、この Noddy オブジェクトを新しく作れるようにすることです。オブジェクトの作成を許可するには、tp_new の実装を提供する必要があります。今回は、 API 関数によって提供されるデフォルトの実装 PyType_GenericNew() を使うだけにしましょう。これを単に tp_new スロットに代入すればよいのですが、これは互換上の理由からできません。プラットフォームやコンパイラによっては、構造体メンバの初期化に別の場所で定義されている C の関数を代入することはできないのです。なので、この tp_new の値はモジュール初期化用の関数で代入します。 PyType_Ready() を呼ぶ直前です:

```
noddy_NoddyType.tp_new = PyType_GenericNew;
if (PyType_Ready(&noddy_NoddyType) < 0)
    return;
```

これ以外のタイプメソッドはすべて *NULL* です。これらについては後ほどふれます。

このファイル中にある他のものは、どれもおなじみでしょう。 initnoddy() のこれを除いて:

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

この関数は、上で *NULL* に指定していた ob_type などのいくつものメンバを埋めて、 Noddy 型を初期化します。

```
PyModule_AddObject(m, "Noddy", (PyObject *)&noddy_NoddyType);
```

これはこの型をモジュール中の辞書に埋め込みます。これで、 Noddy クラスを呼べば Noddy インスタンスを作れるようになりました:

```
>>> import noddy
>>> mynoddy = noddy.Noddy()
```

これだけです! 残るはこれをどうやってビルドするかということです。上のコードを noddy.c というファイルに入れて、以下のものを setup.py というファイルに入れましょう。

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

そして、シェルから以下のように入力します。

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

これでサブディレクトリの下にファイル noddy.so が作成されます。このディレクトリに移動して Python を起動しましょう。import noddy して Noddy 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 Noddy 型はまだおもしろみに欠けています。何もデータを持ってないし、何もしてはくれません。継承してサブクラスを作ることさえできないのです。

2.1.1 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 noddy2 をつくり、以下の機能を追加します:

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

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

static void
Noddy_dealloc(Noddy* self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    self->ob_type->tp_free((PyObject*)self);
}

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL)
          {
            Py_DECREF(self);
            return NULL;
          }

        self->last = PyString_FromString("");
        if (self->last == NULL)
          {
            Py_DECREF(self);
            return NULL;
          }

        self->number = 0;
    }

    return (PyObject *)self;
}

static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|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;
}


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

static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }

    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);

    return result;
}

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

static PyTypeObject NoddyType = {
    PyObject_HEAD_INIT(NULL)
```

```
    0,                         /*ob_size*/
    "noddy.Noddy",             /*tp_name*/
    sizeof(Noddy),             /*tp_basicsize*/
    0,                         /*tp_itemsize*/
    (destructor)Noddy_dealloc, /*tp_dealloc*/
    0,                         /*tp_print*/
    0,                         /*tp_getattr*/
    0,                         /*tp_setattr*/
    0,                         /*tp_compare*/
    0,                         /*tp_repr*/
    0,                         /*tp_as_number*/
    0,                         /*tp_as_sequence*/
    0,                         /*tp_as_mapping*/
    0,                         /*tp_hash */
    0,                         /*tp_call*/
    0,                         /*tp_str*/
    0,                         /*tp_getattro*/
    0,                         /*tp_setattro*/
    0,                         /*tp_as_buffer*/
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /*tp_flags*/
    "Noddy objects",           /* tp_doc */
    0,                               /* tp_traverse */
    0,                               /* tp_clear */
    0,                               /* tp_richcompare */
    0,                               /* tp_weaklistoffset */
    0,                               /* tp_iter */
    0,                               /* tp_iternext */
    Noddy_methods,             /* tp_methods */
    Noddy_members,             /* tp_members */
    0,                         /* tp_getset */
    0,                         /* tp_base */
    0,                         /* tp_dict */
    0,                         /* tp_descr_get */
    0,                         /* tp_descr_set */
    0,                         /* tp_dictoffset */
    (initproc)Noddy_init,      /* tp_init */
    0,                         /* tp_alloc */
    Noddy_new,                 /* tp_new */
};

static PyMethodDef module_methods[] = {
    {NULL}  /* Sentinel */
};

#ifndef PyMODINIT_FUNC        /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
initnoddy2(void)
{
    PyObject* m;

    if (PyType_Ready(&NoddyType) < 0)
        return;

    m = Py_InitModule3("noddy2", module_methods,
                       "Example module that creates an extension type.");

    if (m == NULL)
      return;

    Py_INCREF(&NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&NoddyType);
```

```
}
```

このバージョンでは、いくつも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。

Noddy オブジェクトの構造体の名前は Noddy に縮めることにします。タイプオブジェクト名は NoddyType に縮めます。

これから Noddy 型は 3 つのデータ属性をもつようになります。*first* 、 *last* 、および *number* です。*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;
    PyObject *last;
    int number;
} Noddy;
```

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
Noddy_dealloc(Noddy* self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    self->ob_type->tp_free((PyObject*)self);
}
```

この関数は tp_dealloc メンバに代入されます。

```
(destructor)Noddy_dealloc, / *tp_dealloc* /
```

このメソッドでやっているのは、ふたつの Python 属性の参照カウントを減らすことです。 first メンバと last メンバが *NULL* かもしれないため、ここでは Py_XDECREF() を使いました。このあとそのオブジェクトのタイプメソッドである tp_free メンバを呼び出しています。ここではオブジェクトの型が NoddyType とは限らないことに注意してください。なぜなら、このオブジェクトはサブクラス化したインスタンスかもしれないからです。

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、新しいメソッドを追加することにしましょう:

```
static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL)
```

```
            {
              Py_DECREF(self);
              return NULL;
            }

        self->last = PyString_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:

```
Noddy_new,                  /* tp_new */
```

この新しいメンバはその型のオブジェクトを (初期化するのではなく) 作成する責任を負っています。Python ではこのメンバは __new__() メソッドとして見えています。__new__() メソッドについての詳しい議論は "Unifying types and classes in Python" という題名の論文を見てください。 new メソッドを実装する理由のひとつは、インスタンス変数の初期値を保証するためです。この例でやりたいのは new メソッドが first メンバと last メンバの値を *NULL* でないようにするということです。もしこれらの初期値が *NULL* でもよいのであれば、先の例でやったように、new メソッドとして PyType_GenericNew() を使うこともできたでしょう。 PyType_GenericNew() はすべてのインスタンス変数のメンバを *NULL* にします。

この new メソッドは静的なメソッドで、インスタンスを生成するときにその型と、型が呼び出されたときの引数が渡され、新しいオブジェクトを作成して返します。new メソッドはつねに、あらかじめ固定引数 (positional argument) とキーワード引数を取りますが、これらのメソッドはしばしばそれらの引数は無視して初期化メソッドにそのまま渡します。new メソッドはメモリ割り当てのために tp_alloc メンバを呼び出します。 tp_alloc をこちらで初期化する必要はありません。これは PyType_Ready() が基底クラス (デフォルトでは object) をもとに埋めるものです。ほとんどの型ではデフォルトのメモリ割り当てを使っています。

ノート: 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だ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。)

つぎに初期化用の関数を見てみましょう:

```
static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};
```

```
    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|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。

```
(initproc)Noddy_init,          /* tp_init */
```

Python では、 tp_init メンバは __init__() メソッドとして見えています。このメソッドは、オブジェクトが作成されたあとに、それを初期化する目的で使われます。 new メソッドとはちがって、初期化用のメソッドは必ず呼ばれるとは限りません。初期化用のメソッドは、インスタンスの初期値を提供するのに必要な引数を受けとります。このメソッドはつねに固定引数とキーワード引数を受けとります。

初期化メソッドは複数回呼び出される可能性があります。あなたのオブジェクトの __init__() メソッドは、誰にでも呼び出すことができるからです。このため、新しい値を代入するさいには特別な注意を払う必要があります。たとえば、 first メンバには以下のように代入したくなるかもしれません:

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

しかしこのやり方は危険です。このタイプでは first メンバに入るオブジェクトをなにも限定していないので、どんなオブジェクトでもとり得てしまうからです。それはこのコードが first メンバにアクセスしようとする前に、そのデストラクタが呼び出されてしまうかもしれないのです。このような可能性からパラノイア的に身をまもるため、ほとんどの場合メンバへの代入は, その参照カウントを減らす前におこなってください。こうする必要がないのはどんな場合でしょうか?

- その参照カウントが 1 より大きいと確信できる場合。
- そのオブジェクトの解放があなたのタイプのコードにコールバックするようなことが決してない場合[1]。
- ガベージコレクションがサポートされていない場合に tp_dealloc ハンドラで参照カウントを減らすとき[2]。

[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 ここで出てきたタイプではガベージコレクションをサポートしていないので、この例では tp_dealloc ハンドラに依存しています。このハンドラはそのタイプがたとえガベージコレクションをサポートしている場合でも、そのオブジェクトの「追跡を解除する」ために呼ばれることがありますが、これは高度な話題でありここでは扱いません。

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、これにはいくつもの方法があります。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 Noddy_members[] = {
    {"first", T_OBJECT_EX, offsetof(Noddy, first), 0,
     "first name"},
    {"last", T_OBJECT_EX, offsetof(Noddy, last), 0,
     "last name"},
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL}  /* Sentinel */
};
```

そして、この定義を tp_members に入れましょう:

```
Noddy_members,              /* tp_members */
```

各メンバの定義はそれぞれ、メンバの名前、型、オフセット、アクセスフラグおよび docstring です。詳しくは後の “総称的な属性を管理する” (**総称的な属性を管理する**) の節をご覧ください。

この方法の欠点は、Python 属性に代入できるオブジェクトの型を制限する方法がないことです。ここではファーストネーム first とラストネーム last に、ともに文字列が入るよう期待していますが、今のやり方ではどんな Python オブジェクトも代入できてしまいます。加えてこの属性は削除 (del) できてしまい、その場合、C のポインタには *NULL* が設定されます。たとえもしメンバが *NULL* 以外の値に初期化されるようにしてあったとしても、属性が削除されればメンバは *NULL* になってしまいます。

ここでは 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 *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }

    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);

    return result;
}
```

このメソッドは C 関数として実装され、 Noddy (あるいは Noddy 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。しばしば固定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)
```

first メンバと last メンバがそれぞれ *NULL* かどうかチェックしなければならないことに注意してください。これらは削除される可能性があり、その場合値は *NULL* にセットされます。この属性の削除を禁止して、そこに入れられる値を文字列に限定できればなおいいでしょう。次の節ではこれについて扱います。

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

これを tp_methods スロットに入れましょう:

```
Noddy_methods,             /* tp_methods */
```

ここでの METH_NOARGS フラグは、そのメソッドが引数を取らないことを宣言するのに使われています。

最後に、この型を基底クラスとして利用可能にしましょう。上のメソッドは注意ぶかく書かれているので、これはそのオブジェクトの型が作成されたり利用される場合についてどんな仮定も置いていません。なので、ここですべきことは Py_TPFLAGS_BASETYPE をクラス定義のフラグに加えるだけです:

```
Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, / *tp_flags* /
```

initnoddy() の名前を initnoddy2() に変更し、 Py_InitModule3() に渡されるモジュール名を更新します。

さいごに setup.py ファイルを更新して新しいモジュールをビルドします。

```
from distutils.core import setup, Extension
setup(name="noddy", version="1.0",
      ext_modules=[
         Extension("noddy", ["noddy.c"]),
         Extension("noddy2", ["noddy2.c"]),
         ])
```

2.1.2 データ属性をこまかく制御する

この節では、 Noddy クラスの例にあった first と last の各属性にたいして、より精密な制御を提供します。以前のバージョンのモジュールでは、インスタンス変数の first と last には文字列以外のものも代入できてしまい、あまつさえ削除まで可能でした。ここではこれらの属性が必ず文字列を保持しているようにしましょう。

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

typedef struct {
    PyObject_HEAD
    PyObject *first;
    PyObject *last;
    int number;
} Noddy;

static void
Noddy_dealloc(Noddy* self)
{
    Py_XDECREF(self->first);
    Py_XDECREF(self->last);
    self->ob_type->tp_free((PyObject*)self);
}

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL)
          {
            Py_DECREF(self);
            return NULL;
          }

        self->last = PyString_FromString("");
        if (self->last == NULL)
          {
            Py_DECREF(self);
            return NULL;
          }

        self->number = 0;
    }

    return (PyObject *)self;
}

static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|SSi", 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;
}

static PyMemberDef Noddy_members[] = {
    {"number", T_INT, offsetof(Noddy, number), 0,
     "noddy number"},
    {NULL}  /* Sentinel */
};

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

static int
Noddy_setfirst(Noddy *self, PyObject *value, void *closure)
{
  if (value == NULL) {
    PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
    return -1;
  }

  if (! PyString_Check(value)) {
    PyErr_SetString(PyExc_TypeError,
                    "The first attribute value must be a string");
    return -1;
  }

  Py_DECREF(self->first);
  Py_INCREF(value);
  self->first = value;

  return 0;
}

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

static int
Noddy_setlast(Noddy *self, PyObject *value, void *closure)
{
  if (value == NULL) {
    PyErr_SetString(PyExc_TypeError, "Cannot delete the last attribute");
    return -1;
  }

  if (! PyString_Check(value)) {
    PyErr_SetString(PyExc_TypeError,
                    "The last attribute value must be a string");
```

```
    return -1;
  }

  Py_DECREF(self->last);
  Py_INCREF(value);
  self->last = value;

  return 0;
}

static PyGetSetDef Noddy_getseters[] = {
    {"first",
     (getter)Noddy_getfirst, (setter)Noddy_setfirst,
     "first name",
     NULL},
    {"last",
     (getter)Noddy_getlast, (setter)Noddy_setlast,
     "last name",
     NULL},
    {NULL}  /* Sentinel */
};

static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);

    return result;
}

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

static PyTypeObject NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0,                         /*ob_size*/
    "noddy.Noddy",             /*tp_name*/
    sizeof(Noddy),             /*tp_basicsize*/
    0,                         /*tp_itemsize*/
    (destructor)Noddy_dealloc, /*tp_dealloc*/
    0,                         /*tp_print*/
    0,                         /*tp_getattr*/
    0,                         /*tp_setattr*/
    0,                         /*tp_compare*/
    0,                         /*tp_repr*/
```

```
    0,                         /*tp_as_number*/
    0,                         /*tp_as_sequence*/
    0,                         /*tp_as_mapping*/
    0,                         /*tp_hash */
    0,                         /*tp_call*/
    0,                         /*tp_str*/
    0,                         /*tp_getattro*/
    0,                         /*tp_setattro*/
    0,                         /*tp_as_buffer*/
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /*tp_flags*/
    "Noddy objects",           /* tp_doc */
    0,                               /* tp_traverse */
    0,                               /* tp_clear */
    0,                               /* tp_richcompare */
    0,                               /* tp_weaklistoffset */
    0,                               /* tp_iter */
    0,                               /* tp_iternext */
    Noddy_methods,             /* tp_methods */
    Noddy_members,             /* tp_members */
    Noddy_getseters,           /* tp_getset */
    0,                         /* tp_base */
    0,                         /* tp_dict */
    0,                         /* tp_descr_get */
    0,                         /* tp_descr_set */
    0,                         /* tp_dictoffset */
    (initproc)Noddy_init,      /* tp_init */
    0,                         /* tp_alloc */
    Noddy_new,                 /* tp_new */
};

static PyMethodDef module_methods[] = {
    {NULL}  /* Sentinel */
};

#ifndef PyMODINIT_FUNC        /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
initnoddy3(void)
{
    PyObject* m;

    if (PyType_Ready(&NoddyType) < 0)
        return;

    m = Py_InitModule3("noddy3", module_methods,
                       "Example module that creates an extension type.");

    if (m == NULL)
      return;

    Py_INCREF(&NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&NoddyType);
}
```

`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:

```
Noddy_getfirst(Noddy *self, void *closure)
{
    Py_INCREF(self->first);
    return self->first;
```

```
}

static int
Noddy_setfirst(Noddy *self, PyObject *value, void *closure)
{
  if (value == NULL) {
    PyErr_SetString(PyExc_TypeError, "Cannot delete the first attribute");
    return -1;
  }

  if (! PyString_Check(value)) {
    PyErr_SetString(PyExc_TypeError,
                    "The first attribute value must be a string");
    return -1;
  }

  Py_DECREF(self->first);
  Py_INCREF(value);
  self->first = value;

  return 0;
}
```

getter 関数には Noddy オブジェクトと「閉包 (closure)」 (これは void 型のポインタです) が渡されます。今回のケースでは閉包は無視します。 (閉包とは定義データが渡される setter や getter の高度な利用をサポートするためのもので、これを使うとたとえば getter と setter をひとまとめにした関数に、閉包のデータにもとづいて属性を get するか set するか決めさせる、といったことができます。)

setter 関数には Noddy オブジェクトと新しい値、そして閉包が渡されます。新しい値は *NULL* かもしれず、その場合はこの属性が削除されます。ここでは属性が削除されたり、その値が文字列でないときにはエラーを発生させるようにします。

ここでは PyGetSetDef 構造体の配列をつくります:

```
static PyGetSetDef Noddy_getseters[] = {
    {"first",
     (getter)Noddy_getfirst, (setter)Noddy_setfirst,
     "first name",
     NULL},
    {"last",
     (getter)Noddy_getlast, (setter)Noddy_setlast,
     "last name",
     NULL},
    {NULL}  /* Sentinel */
};
```

そしてこれを tp_getset スロットに登録します:

```
Noddy_getseters,           /* tp_getset */
```

これで属性の getter と setter が登録できました。

PyGetSetDef 構造体の最後の要素が上で説明した閉包です。今回は閉包は使わないので *NULL* を渡しています。

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 Noddy_members[] = {
    {"number", T_INT, offsetof(Noddy, number), 0,
```

```
     "noddy number"},
    {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
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|SSi", 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;
}
```

これらの変更によって、 first メンバと last メンバが決して *NULL* にならないと保証できました。これでほとんどすべてのケースから *NULL* 値のチェックを除けます。これは Py_XDECREF() 呼び出しを Py_DECREF() 呼び出しに変えられることを意味します。唯一これを変えられないのはオブジェクト解放メソッド (deallocator) で、なぜならここではコンストラクタによるメンバ初期化が失敗している可能性があるからです。

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。

2.1.3 循環ガベージコレクションをサポートする

Python は循環ガベージコレクション機能をもっており、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。これ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:

[3] first および last メンバが文字列であるということはわかっているので、いまやそれらの参照カウントを減らすときにはそれほど注意する必要はないように思えるかもしれません。しかし文字列型のサブクラスは依然として受けつけられています。通常の文字列型ならば、解放時にあなたのオブジェクトがコールバックされることはありませんが、文字列型のサブクラスがそうしないという保証はありません。

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

この例では、自分自身をふくむリストをつくりました。たとえこのリストを del 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ション機能がありますから、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。

Noddy クラスの 2 番目の例では、 first 属性と last 属性にどんなオブジェクトでも格納できるようになっていました。[4] 。つまり、 Noddy オブジェクトの参照は循環しうるのです:

```
>>> import noddy2
>>> n = noddy2.Noddy()
>>> l = [n]
>>> n.first = l
```

これは実にばかげた例ですが、すくなくとも Noddy クラスに循環ガベージコレクション機能のサポートを加える口実を与えてくれます。循環ガベージコレクションをサポートするには 2 つのタイプスロットを埋め、これらのスロットを許可するようにクラス定義のフラグを設定する必要があります:

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

typedef struct {
    PyObject_HEAD
    PyObject *first;
    PyObject *last;
    int number;
} Noddy;

static int
Noddy_traverse(Noddy *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;
}

static int
Noddy_clear(Noddy *self)
{
    PyObject *tmp;

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

[4] 3 番目のバージョンでさえ、循環を回避できるという保証はされていません。たとえ通常の文字列型なら循環しない場合でも、文字列型のサブクラスをとることが許されていれば、そのタイプでは循環が発生しうるからです。

```
    tmp = self->last;
    self->last = NULL;
    Py_XDECREF(tmp);

    return 0;
}

static void
Noddy_dealloc(Noddy* self)
{
    Noddy_clear(self);
    self->ob_type->tp_free((PyObject*)self);
}

static PyObject *
Noddy_new(PyTypeObject *type, PyObject *args, PyObject *kwds)
{
    Noddy *self;

    self = (Noddy *)type->tp_alloc(type, 0);
    if (self != NULL) {
        self->first = PyString_FromString("");
        if (self->first == NULL)
          {
            Py_DECREF(self);
            return NULL;
          }

        self->last = PyString_FromString("");
        if (self->last == NULL)
          {
            Py_DECREF(self);
            return NULL;
          }

        self->number = 0;
    }

    return (PyObject *)self;
}

static int
Noddy_init(Noddy *self, PyObject *args, PyObject *kwds)
{
    PyObject *first=NULL, *last=NULL, *tmp;

    static char *kwlist[] = {"first", "last", "number", NULL};

    if (! PyArg_ParseTupleAndKeywords(args, kwds, "|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;
}


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

static PyObject *
Noddy_name(Noddy* self)
{
    static PyObject *format = NULL;
    PyObject *args, *result;

    if (format == NULL) {
        format = PyString_FromString("%s %s");
        if (format == NULL)
            return NULL;
    }

    if (self->first == NULL) {
        PyErr_SetString(PyExc_AttributeError, "first");
        return NULL;
    }

    if (self->last == NULL) {
        PyErr_SetString(PyExc_AttributeError, "last");
        return NULL;
    }

    args = Py_BuildValue("OO", self->first, self->last);
    if (args == NULL)
        return NULL;

    result = PyString_Format(format, args);
    Py_DECREF(args);

    return result;
}

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

static PyTypeObject NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0,                         /*ob_size*/
    "noddy.Noddy",             /*tp_name*/
    sizeof(Noddy),             /*tp_basicsize*/
```

```
    0,                         /*tp_itemsize*/
    (destructor)Noddy_dealloc, /*tp_dealloc*/
    0,                         /*tp_print*/
    0,                         /*tp_getattr*/
    0,                         /*tp_setattr*/
    0,                         /*tp_compare*/
    0,                         /*tp_repr*/
    0,                         /*tp_as_number*/
    0,                         /*tp_as_sequence*/
    0,                         /*tp_as_mapping*/
    0,                         /*tp_hash */
    0,                         /*tp_call*/
    0,                         /*tp_str*/
    0,                         /*tp_getattro*/
    0,                         /*tp_setattro*/
    0,                         /*tp_as_buffer*/
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC, /*tp_flags*/
    "Noddy objects",           /* tp_doc */
    (traverseproc)Noddy_traverse,   /* tp_traverse */
    (inquiry)Noddy_clear,           /* tp_clear */
    0,                          /* tp_richcompare */
    0,                          /* tp_weaklistoffset */
    0,                          /* tp_iter */
    0,                          /* tp_iternext */
    Noddy_methods,             /* tp_methods */
    Noddy_members,             /* tp_members */
    0,                         /* tp_getset */
    0,                         /* tp_base */
    0,                         /* tp_dict */
    0,                         /* tp_descr_get */
    0,                         /* tp_descr_set */
    0,                         /* tp_dictoffset */
    (initproc)Noddy_init,      /* tp_init */
    0,                         /* tp_alloc */
    Noddy_new,                 /* tp_new */
};

static PyMethodDef module_methods[] = {
    {NULL}  /* Sentinel */
};

#ifndef PyMODINIT_FUNC        /* declarations for DLL import/export */
#define PyMODINIT_FUNC void
#endif
PyMODINIT_FUNC
initnoddy4(void)
{
    PyObject* m;

    if (PyType_Ready(&NoddyType) < 0)
        return;

    m = Py_InitModule3("noddy4", module_methods,
                       "Example module that creates an extension type.");

    if (m == NULL)
      return;

    Py_INCREF(&NoddyType);
    PyModule_AddObject(m, "Noddy", (PyObject *)&NoddyType);
}
```

traversal メソッドは循環した参照に含まれる可能性のある内部オブジェクトへのアクセスを提供します:

```
static int
Noddy_traverse(Noddy *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 2.4 以降では、visit 関数の呼び出しを自動化する `Py_VISIT()` マクロが用意されています。`Py_VISIT()` を使えば、`Noddy_traverse()` は次のように簡略化できます:

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

ノート: 注意: `tp_traverse` の実装で `Py_VISIT()` を使うには、その引数に正確に *visit* および *arg* という名前をつける必要があります。これは、この退屈な実装に統一性を導入することを促進します。

また、循環した参照に含まれた内部オブジェクトを消去するためのメソッドも提供する必要があります。オブジェクト解放用のメソッドを再実装して、このメソッドに使いましょう:

```
static int
Noddy_clear(Noddy *self)
{
    PyObject *tmp;

    tmp = self->first;
    self->first = NULL;
    Py_XDECREF(tmp);

    tmp = self->last;
    self->last = NULL;
    Py_XDECREF(tmp);

    return 0;
}

static void
Noddy_dealloc(Noddy* self)
```

```
{
    Noddy_clear(self);
    self->ob_type->tp_free((PyObject*)self);
}
```

Noddy_clear() 中での一時変数の使い方に注目してください。ここでは、一時変数をつかって各メンバの参照カウントを減らす前にそれらに *NULL* を代入しています。これは次のような理由によります。すでにお話ししたように、もし参照カウントがゼロになると、このオブジェクトがコールバックされるようになってしまいます。さらに、いまやガベージコレクションをサポートしているため、ガベージコレクション時に実行されるコードについても心配しなくてはなりません。もしガベージコレクションが走っていると、あなたの tp_traverse ハンドラが呼び出される可能性があります。メンバの参照カウントがゼロになった場合に、その値が *NULL* に設定されていないと Noddy_traverse() が呼ばれる機会はありません。

Python 2.4 以降では、注意ぶかく参照カウントを減らすためのマクロ Py_CLEAR() が用意されています。Py_CLEAR() を使えば、 Noddy_clear() は次のように簡略化できます:

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

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:

```
Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE | Py_TPFLAGS_HAVE_GC, /* tp_flags */
```

これで完了です。 tp_alloc スロットまたは tp_free スロットが書かれていれば、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。

2.1.4 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 Shoddy 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 shoddy
>>> s = shoddy.Shoddy(range(3))
>>> s.extend(s)
>>> print len(s)
6
>>> print s.increment()
1
>>> print s.increment()
2
```

```
#include <Python.h>

typedef struct {
```

```
    PyListObject list;
    int state;
} Shoddy;


static PyObject *
Shoddy_increment(Shoddy *self, PyObject *unused)
{
    self->state++;
    return PyInt_FromLong(self->state);
}


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

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


static PyTypeObject ShoddyType = {
    PyObject_HEAD_INIT(NULL)
    0,                         /* ob_size */
    "shoddy.Shoddy",           /* tp_name */
    sizeof(Shoddy),            /* tp_basicsize */
    0,                         /* tp_itemsize */
    0,                         /* tp_dealloc */
    0,                         /* tp_print */
    0,                         /* tp_getattr */
    0,                         /* tp_setattr */
    0,                         /* tp_compare */
    0,                         /* tp_repr */
    0,                         /* tp_as_number */
    0,                         /* tp_as_sequence */
    0,                         /* tp_as_mapping */
    0,                         /* tp_hash */
    0,                         /* tp_call */
    0,                         /* tp_str */
    0,                         /* tp_getattro */
    0,                         /* tp_setattro */
    0,                         /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT |
      Py_TPFLAGS_BASETYPE,     /* tp_flags */
    0,                         /* tp_doc */
    0,                         /* tp_traverse */
    0,                         /* tp_clear */
    0,                         /* tp_richcompare */
    0,                         /* tp_weaklistoffset */
    0,                         /* tp_iter */
    0,                         /* tp_iternext */
    Shoddy_methods,            /* tp_methods */
    0,                         /* tp_members */
    0,                         /* tp_getset */
    0,                         /* tp_base */
```

```
    0,                         /* tp_dict */
    0,                         /* tp_descr_get */
    0,                         /* tp_descr_set */
    0,                         /* tp_dictoffset */
    (initproc)Shoddy_init,     /* tp_init */
    0,                         /* tp_alloc */
    0,                         /* tp_new */
};

PyMODINIT_FUNC
initshoddy(void)
{
    PyObject *m;

    ShoddyType.tp_base = &PyList_Type;
    if (PyType_Ready(&ShoddyType) < 0)
        return;

    m = Py_InitModule3("shoddy", NULL, "Shoddy module");
    if (m == NULL)
        return;

    Py_INCREF(&ShoddyType);
    PyModule_AddObject(m, "Shoddy", (PyObject *) &ShoddyType);
}
```

見てわかるように、ソースコードは前の節の Noddy 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;
    int state;
} Shoddy;
```

継承した型のオブジェクトの最初の違いは、親クラスのオブジェクト構造が最初に必要なことです。基底型が既に PyObject_HEAD() を構造体の先頭に持っています。

Python オブジェクトが Shoddy 型のインスタンスだった場合、その *PyObject** ポインタは *PyListObject** にも *Shoddy** にも安全にキャストできます。

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

この新しい型の __init__ メソッドで、基底型の __init__ メソッドを呼び出している様子を見ることができます。

このパターンは、カスタムの new と dealloc メソッドを実装するときには重要です。継承した型の new メソッドは、 tp_alloc を使ってメモリを割り当てるべきではありません。それは基底型の tp_new を呼出たときに処理されるからです。

Shoddy 型のために PyTypeObject() を埋めるとき、tp_base() スロットを見つけることができます。クロスプラットフォームのコンパイラに対応するために、直接そのスロットを PyList_Type() で埋めてはいけません。代わりに、後でモジュールの init() 関数の中で行うことができます。

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

    ShoddyType.tp_base = &PyList_Type;
    if (PyType_Ready(&ShoddyType) < 0)
        return;

    m = Py_InitModule3("shoddy", NULL, "Shoddy module");
    if (m == NULL)
        return;

    Py_INCREF(&ShoddyType);
    PyModule_AddObject(m, "Shoddy", (PyObject *) &ShoddyType);
}
```

PyType_Read() を呼ぶ前に、型の構造は tp_base スロットは埋められていなければなりません。継承している新しい型を作るとき、 tp_alloc スロットを PyType_GenericNew() で埋める必要はありません。 – 基底型のアロケート関数が継承されます。

その後、 PyType_Read() を呼んで、 Noddy の時と同じようにタイプオブジェクトをモジュールに追加します。

2.2 タイプメソッド

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
    char *tp_name; /* For printing, in format "<module>.<name>" */
    int tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    printfunc tp_print;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    cmpfunc tp_compare;
    reprfunc tp_repr;

    /* Method suites for standard classes */

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

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

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;
```

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

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

    char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

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

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    long tp_weaklistoffset;

    /* Added in release 2.2 */
    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

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

} PyTypeObject;
```

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す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、これらの多様なハンドラについて、これから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。型を初期化するさいに、これらのメンバを正しい順序で並べるよう、くれぐれも注意してください。ふつういちばん簡単なのは、必要なメンバがすべて含まれている (たとえそれらが `0` に初期化されていても) 例をとってきて、自分の型に合わせるよう変更をくわえることです。

```
char *tp_name; /* 表示用 */
```

これは型の名前です。前節で説明したように、これはいろいろな場面で現れ、ほとんどは診断用の目的で使われるものです。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。

```
int tp_basicsize, tp_itemsize; /* 割り当て用 */
```

これらのメンバは、この型のオブジェクトが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 (文字列やリスト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。

```
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、ほとんどの拡張型が実装するであろう基本的なタイプメソッドに入っていきます。

2.2.1 最終化 (finalization) と解放

```
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);
    obj->ob_type->tp_free(obj);
}
```

解放用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解放用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)
{
    MyObject *self = (MyObject *) obj;
    PyObject *cbresult;

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

```
        int have_error = PyErr_Occurred() ? 1 : 0;

        if (have_error)
            PyErr_Fetch(&err_type, &err_value, &err_traceback);

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

        if (have_error)
            PyErr_Restore(err_type, err_value, err_traceback);

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

2.2.2 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 3 つのやり方があります: `repr()` 関数 (あるいはそれと等価なバッククォートを用いた表現) を使う方法、 `str()` 関数を使う方法、そして `print` 文を使う方法です。ほとんどのオブジェクトで `print` 文は `str()` 関数と同じですが、必要な場合には特殊なケースとして `FILE*` にも表示できます。 `FILE*` への表示は、効率が問題となっている場合で、一時的な文字列オブジェクトを作成してファイルに書き込むのでは効率が悪すぎることがプロファイリングからも明らかな場合にのみ使うべきです。

これらのハンドラはどれも必須ではありません。ほとんどの型ではせいぜい `tp_str` ハンドラと `tp_repr` ハンドラを実装するだけですみます。

```
reprfunc tp_repr;
reprfunc tp_str;
printfunc tp_print;
```

`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 PyString_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 PyString_FromFormat("Stringified_newdatatype{{size:\%d}}",
                               obj->obj_UnderlyingDatatypePtr->size);
}
```

print ハンドラは Python がその型のインスタンスを「print する」必要のあるときに毎回呼ばれます。たとえば 'node' が TreeNode 型のインスタンスだとすると、print ハンドラは Python が以下を実行したときに呼ばれます:

```
print node
```

flags 引数には `Py_PRINT_RAW` というフラグがあり、これはその文字列をクォートやおそらくはエスケープシーケンスの解釈もなしで表示することを指示します。

この print 関数は `FILE*` オブジェクトを引数としてとります。たぶん、ここに出力することになるでしょう。

print 関数の例は以下のようになります:

```
static int
newdatatype_print(newdatatypeobject *obj, FILE *fp, int flags)
{
    if (flags & Py_PRINT_RAW) {
        fprintf(fp, "<{newdatatype object--size: %d}>",
                obj->obj_UnderlyingDatatypePtr->size);
    }
    else {
        fprintf(fp, "\"<{newdatatype object--size: %d}>\"",
                obj->obj_UnderlyingDatatypePtr->size);
    }
    return 0;
}
```

2.2.3 属性を管理する

属性をもつどのオブジェクトに対しても、その型は、それらオブジェクトの属性をどのように解決するか制御する関数を提供する必要があります。必要な関数としては、属性を (それが定義されていれば) 取り出すものと、もうひとつは属性に (それが許可されていれば) 値を設定するものです。属性を削除するのは特殊なケースで、この場合は新しい値としてハンドラに *NULL* が渡されます。

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。

```
getattrfunc  tp_getattr;        /* char * バージョン */
setattrfunc  tp_setattr;
/* ... */
getattrofunc tp_getattrofunc;   /* PyObject * バージョン */
setattrofunc tp_setattrofunc;
```

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

バージョン 2.2 で追加. 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。

`PyType_Ready()` が呼ばれると、これはそのタイプオブジェクトに参照されている 3 つのテーブルを使って、そのタイプオブジェクトの辞書中にデスクリプタ (*descriptor*) を作成します。各デスクリプタは、インスタンスオブジェクトの属性に対するアクセスを制御します。それぞれのテーブルはなくてもかまいません。もしこれら 3 つがすべて *NULL* だと、その型のインスタンスはその基底型から継承した属性だけを持つことになります。また、 `tp_getattro` および `tp_setattro` が *NULL* のままだった場合も、基底型にこれらの属性の操作がまかせられます。

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;
```

`tp_methods` が *NULL* でない場合、これは `PyMethodDef` 構造体への配列を指している必要があります。テーブル中の各エントリは、つぎのような構造体のインスタンスです:

```
typedef struct PyMethodDef {
    char         *ml_name;       /* メソッド名 */
    PyCFunction  ml_meth;        /* 実装する関数 */
    int          ml_flags;       /* flags */
    char         *ml_doc;        /* docstring */
} PyMethodDef;
```

その型が提供する各メソッドについてひとつのエントリを定義する必要があります。基底型から継承してきたメソッドについてはエントリは必要ありません。これの最後には、配列の終わりを示すための見張り番 (sentinel) として追加のエントリがひとつ必要です。この場合、 `ml_name` メンバが sentinel として使われ、その値は *NULL* でなければなりません。

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、アクセスを読み込み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 {
    char *name;
    int  type;
    int  offset;
    int  flags;
```

```
    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 (*descriptor*) 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。

Constant	Meaning
READONLY	絶対に変更できない。
RO	READONLY の短縮形。
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。

`tp_methods` テーブルと同じように、ここでも `name` メンバの値を *NULL* にした見張り用エントリが必要です。

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。これを紹介することは 2 つの意味をもっています。ひとつはどうやって、古いバージョンの Python と互換性のあるやり方で、基本的な属性管理をおこなうか。そしてもうひとつはハンドラの関数がどのようにして呼ばれるのか。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。

これを処理するありがちな方法は、(1) 一連の関数 (下の例の `newdatatype_getSize()` や `newdatatype_setSize()`) を実装する、(2) これらの関数を記録したメソッドテーブルを提供する、そして (3) そのテーブルの参照結果を返す getattr 関数を提供することです。メソッドテーブルはタイプオブジェクトの `tp_methods` メンバと同じ構造を持っています。

以下に例を示します:

```
static PyMethodDef newdatatype_methods[] = {
    {"getSize", (PyCFunction)newdatatype_getSize, METH_VARARGS,
     "Return the current size."},
    {"setSize", (PyCFunction)newdatatype_setSize, METH_VARARGS,
```

```
     "Set the size."},
    {NULL, NULL, 0, NULL}           /* 見張り */
};

static PyObject *
newdatatype_getattr(newdatatypeobject *obj, char *name)
{
    return Py_FindMethod(newdatatype_methods, (PyObject *)obj, name);
}
```

tp_setattr ハンドラは、クラスのインスタンスの __setattr__() または __delattr__() メソッドが呼ばれるであろう状況で呼び出されます。ある属性が削除されるとき、3 番目のパラメータは *NULL* になります。以下の例はたんに例外を発生させるものですが、もし本当にこれと同じことをしたいなら、tp_setattr ハンドラを *NULL* に設定すべきです。

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

2.2.4 オブジェクトの比較

```
cmpfunc tp_compare;
```

tp_compare ハンドラは、オブジェクトどうしの比較が必要で、そのオブジェクトに要求された比較をおこなうのに適した特定の拡張比較メソッドが実装されていないときに呼び出されます。(これが定義されているとき、PyObject_Compare() または PyObject_Cmp() が使われるとこれはつねに呼び出されます、また Python で cmp() が使われたときにも呼び出されます。) これは __cmp__() メソッドに似ています。この関数はもし *obj1* が *obj2* より「小さい」場合は -1 を返し、それらが等しければ 0 、そしてもし *obj1* が *obj2* より「大きい」場合は 1 を返す必要があります。(以前は大小比較の結果として、任意の大きさの負または正の整数を返せましたが、 Python 2.2 以降ではこれはもう許されていません。将来的には、上にあげた以外の返り値は別の意味をもつ可能性があります。)

tp_compare ハンドラは例外を発生させられます。この場合、この関数は負の値を返す必要があります。呼び出した側は PyErr_Occurred() を使って例外を検査しなければなりません。

以下はサンプル実装です:

```
static int
newdatatype_compare(newdatatypeobject * obj1, newdatatypeobject * obj2)
{
    long result;

    if (obj1->obj_UnderlyingDatatypePtr->size <
        obj2->obj_UnderlyingDatatypePtr->size) {
        result = -1;
    }
    else if (obj1->obj_UnderlyingDatatypePtr->size >
             obj2->obj_UnderlyingDatatypePtr->size) {
        result = 1;
    }
    else {
        result = 0;
```

```
    }
    return result;
}
```

2.2.5 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。

これら多数の抽象的なインターフェイスは、Python の実装が開発される初期の段階で定義されていました。とりわけ数値や辞書、そしてシーケンスなどのプロトコルは最初から Python の一部だったのです。それ以外のプロトコルはその後追加されました。型の実装にあるいくつかのハンドラルーチンに依存するようなプロトコルのために、古いプロトコルはハンドラの入ったオプションのブロックとして定義し、型オブジェクトから参照するようになりました。タイプオブジェクトの主部に追加のスロットをもつ新しいプロトコルについては、フラグ用のビットを立てることでそれらのスロットが存在しており、インタプリタがチェックすべきであることを指示できます。(このフラグ用のビットは、そのスロットの値が非 *NULL* であることを示しているわけではありません。フラグはスロットの存在を示すのに使えますが、そのスロットはまだ埋まっていないかもしれないのです。)

```
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、もし使うのならば、これはお使いの型のインスタンスのハッシュ番号を返すようにします。以下はやや的はずれな例ですが

```
static long
newdatatype_hash(newdatatypeobject *obj)
{
    long result;
    result = obj->obj_UnderlyingDatatypePtr->size;
    result = result * 3;
    return result;
}

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. *arg1* にはその呼び出しの対象となる、そのデータ型のインスタンスが入ります。たとえば呼び出しが `obj1('hello')` の場合、 *arg1* は `obj1` になります。

2. *arg2* 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. *arg3* はキーワード引数のための辞書です。これが *NULL* 以外でキーワード引数をサポートしているなら、 PyArg_ParseTupleAndKeywords() をつかって引数を取り出せます。キーワード引数をサポートしていないのにこれが *NULL* 以外の場合は、キーワード引数はサポートしていない旨のメッセージとともに TypeError を発生させてください。

以下はこの call 関数をてきとうに使った例です。

```
/* call 関数の実装。
 *    obj1 : 呼び出しを受けるインスタンス。
 *    obj2 : 呼び出しのさいの引数を格納するタプル、この場合は 3 つの文字列。
 */
static PyObject *
newdatatype_call(newdatatypeobject *obj, PyObject *args, PyObject *other)
{
    PyObject *result;
    char *arg1;
    char *arg2;
    char *arg3;

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

/* バージョン 2.2 以降で追加 */
/* Iterators */
getiterfunc tp_iter;
iternextfunc tp_iternext;
```

これらの関数はイテレータ用プロトコルをサポートします。オブジェクトが、その (ループ中に順に生成されていくかもしれない) 内容を巡回 (訳注: イテレータでひとつずつ要素をたどっていくこと) するイテレータをサポートしたい場合は、 tp_iter ハンドラを実装する必要があります。tp_iter ハンドラによって返されるオブジェクトは tp_iter と tp_iternext の両方を実装する必要があります。どちらのハンドラも、それが呼ばれたインスタンスをひとつだけ引数としてとり、新しい参照を返します。エラーが起きた場合には例外を設定してから *NULL* を返す必要があります。

巡回可能な要素を表現するオブジェクトに対しては、 tp_iter ハンドラがイテレータオブジェクトを返す必要があります。イテレータオブジェクトは巡回中の状態を保持する責任をもっています。お互いに干渉しない複数のイテレータの存在を許すようなオブジェクト (リストやタプルがそうです) の場合は、新しいイテレータを作成して返す必要があります。 (巡回の結果生じる副作用のために) 一回だけしか巡回できないオブジェクトの場合は、それ自身への参照を返すようなハンドラと、 tp_iternext ハンドラも実装する必要があります。ファイルオブジェクトはそのようなイテレータの例です。

イテレータオブジェクトは両方のハンドラを実装する必要があります。 tp_iter ハンドラはそのイテレータへの新しい参照を返します (これは破壊的にしか巡回できないオブジェクトに対する tp_iter ハンドラと同じです)。 tp_iternext ハンドラはその次のオブジェクトがある場合、それへの新しい参照を返します。巡回が終端に達したときは例外を出さずに *NULL* を返してもいいですし、StopIteration を放出

してもかまいません。例外を使わないほうがやや速度が上がるかもしれません。実際のエラーが起こったときには、例外を放出して *NULL* を返す必要があります。

2.2.6 弱参照 (Weak Reference) のサポート

Python の弱参照実装のひとつのゴールは、どのような（数値のような弱参照による利益を得ない）タイプでもオーバーヘッドなしで弱参照のメカニズムに組み込めるようにすることです。

弱参照可能なオブジェクトの拡張では、弱参照メカニズムのために `PyObject*` フィールドをインスタンス構造体に含む必要があります。これはオブジェクトのコンストラクタで *NULL* に初期化する必要があります。これは対応するタイプの `tp_weaklistoffset` フィールドをフィールドのオフセットに設定しなければいけません。たとえば、インスタンスタイプは以下の構造体で定義されます:

```
typedef struct {
    PyObject_HEAD
    PyClassObject *in_class;       /* The class object */
    PyObject      *in_dict;        /* A dictionary */
    PyObject      *in_weakreflist; /* List of weak references */
} PyInstanceObject;
```

インスタンス用に静的に宣言されたタイプオブジェクトはこのように定義されます:

```
PyTypeObject PyInstance_Type = {
    PyObject_HEAD_INIT(&PyType_Type)
    0,
    "module.instance",

    /* Lots of stuff omitted for brevity... */

    Py_TPFLAGS_DEFAULT,                         /* tp_flags */
    0,                                          /* tp_doc */
    0,                                          /* tp_traverse */
    0,                                          /* tp_clear */
    0,                                          /* tp_richcompare */
    offsetof(PyInstanceObject, in_weakreflist), /* tp_weaklistoffset */
};
```

タイプのコンストラクタは弱参照を *NULL* に初期化する責任があります:

```
static PyObject *
instance_new() {
    /* Other initialization stuff omitted for brevity */

    self->in_weakreflist = NULL;

    return (PyObject *) self;
}
```

さらに、デストラクタは弱参照を消すために弱参照のマネージャを呼ぶ必要があります。これはデストラクタのどの処理よりも先に実施される必要がありますが、弱参照リストが *NULL* でない場合にだけ必要です:

```
static void
instance_dealloc(PyInstanceObject *inst)
{
    /* Allocate temporaries if needed, but do not begin
       destruction just yet.
     */
```

```
    if (inst->in_weakreflist != NULL)
        PyObject_ClearWeakRefs((PyObject *) inst);

    /* Proceed with object destruction normally. */
}
```

2.2.7 その他いろいろ

上にあげたほとんどの関数は、その値として 0 を与えれば省略できることを忘れないでください。それぞれの関数で提供しなければならない型の定義があり、これらは Python の include 用ディレクトリの `object.h` というファイルにおさめられています。これは Python の配布ソースに含まれています。

新しいデータ型に何らかのメソッドを実装するやりかたを学ぶには、以下の方法がおすすめです: Python の配布されているソースをダウンロードして展開する。 `Objects` ディレクトリへ行き、C のソースファイルから「`tp_` 欲しい名前」の文字列で検索する (たとえば `tp_print` とか `tp_compare` のように)。こうすれば実装したい例がみつかるでしょう。

あるオブジェクトが、いま実装している型のインスタンスであるかどうかを確かめたい場合には、`PyObject_TypeCheck()` 関数を使ってください。使用例は以下のようなかんじです:

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

脚注

第 3 章

distutils による C および C++ 拡張モジュールのビルド

Python 1.4 になってから、動的にリンクされるような拡張モジュールをビルドするためのメイクファイルを作成するような、特殊なメイクファイルを Unix 向けに提供するようになりました。Python 2.0 からはこの機構 (いわゆる Makefile.pre.in および Setup ファイルの関係ファイル) はサポートされなくなりました。インタプリタ自体のカスタマイズはほとんど使われず、 distutils で拡張モジュールをビルドできるようになったからです。

distutils を使った拡張モジュールのビルドには、ビルドを行う計算機上に distutils をインストールしていることが必要です。 Python 2.x には distutils が入っており、 Python 1.5 用には個別のパッケージがあります。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見かけになっています:

```
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とります。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、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ファイルも入ります。 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、拡張モジュールのビルドについてのみ説明します。

駆動スクリプトをよりよく構成するために、決め打ちの引数を `setup()` に入れ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 = 'http://docs.python.org/extending/building',
       long_description = '''
This is really just a demo package.
''',
       ext_modules = [module1])
```

この例では、 `setup()` は追加のメタ情報と共に呼び出されます。配布パッケージを構築する際には、メタ情報の追加が推奨されています。拡張モジュール自体については、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/lo

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

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。

3.1 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

```
python setup.py install
```

を実行します。

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

```
python setup.py sdist
```

を実行します。

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 distutils のドキュメント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
```

第 4 章

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。

ノート: 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。

4.1 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。この節では、C や C++で書かれた Python 拡張モジュールを手動でビルドするアプローチについて述べます。

以下の説明に従って拡張モジュールをビルドするには、インストールされている Python と同じバージョンの Python のソースコードを持っていなければなりません。また、 Microsoft Visual C++ “Developer Studio” が必要になります; プロジェクトファイルは VC++ バージョン 7.1 向けのものが提供されていますが、以前のバージョンの VC++も使えます。Python 自体をビルドしたものと同じバージョンの VC++を使わなければならないことに注意しましょう。ここで述べる例題のファイルは、Python ソースコードと共に配布されており、 `PC\example_nt\` ディレクトリにあります。

1. **例題ファイルをコピーする** — example_nt ディレクトリは PC ディレクトリのサブディレクトリになっています。これは PC 関連の全てのファイルをソースコード配布物内の同じディレクトリに置くための措置です。とはいえ実際には、example_nt ディレクトリは PC の下では利用できません。そこで、まずこのディレクトリを一階層上にコピーして、example_nt が PC および Include と同じ階層のディレクトリになるようにします。以降の作業は、移動先の新しいディレクトリ内で行ってください。

2. **プロジェクトを開く** — VC++から、 **ファイル** → **ソリューションを開く**ダイアログメニューを選択します (**ファイル** → **開く**ではありません!)。ディレクトリ階層を辿って、 example_nt **をコピーしたディレクトリ** 内の example.sln を選択し、「開く」をクリックします。

3. **例題の DLL をビルドする** — 設定が全て正しく行われているか調べるために、ビルドしてみます:

4. ビルド構成を選びます。このステップは省略できます。**ビルド** → **設定マネージャー** → **アクティブなソリューションの設定**を選び、 **リリース** または **デバッグ** を選びます。このステップを飛ばすと、VC++ はデフォルトでデバッグ構成を使います。

5. DLL をビルドします。 **ビルド** → **ソリューションのビルド** を選びます。全ての中間ファイルおよび最終ファイルが、上のビルド構成で選んだ構成に従って Debug または Release という名前のディレクトリに生成されます。

6. **デバッグモードの DLL をテストする** — デバッグビルドが成功したら、コマンドプロンプトを起動し、example_nt\Debug ディレクトリに移動してください。以下のセッション通りにコマンドを実行できるはずです (C> は DOS コマンドのプロンプト、 >>> は Python のプロンプトです; ビルド情報や様々なデバッグ出力は、ここに示したスクリーン出力と一致しないこともあるので注意して下さい):

   ```
   C>..\..\PCbuild\python_d
   Adding parser accelerators ...
   Done.
   Python 2.2 (#28, Dec 19 2001, 23:26:37) [MSC 32 bit (Intel)] on win32
   Type "copyright", "credits" or "license" for more information.
   >>> import example
   [4897 refs]
   >>> example.foo()
   Hello, world
   [4903 refs]
   >>>
   ```

   おめでとうございます! とうとう初めての Python 拡張モジュールのビルドに成功しましたね。

7. **自分用にプロジェクトを作成する** — プロジェクト用のディレクトリを適当な名前で作成してください。自作の C ソースコードをディレクトリ内にコピーします。モジュールのソースコードファイル名は必ずしもモジュール名と一致している必要はありませんが、初期化関数の名前はモジュール名と一致していなければなりません — 初期化関数の名前が initspam() なら、モジュールは spam という名前でしか import できません。 initspam() は第一引数を "spam" にして、 Py_InitModule() を呼び出します (このディレクトリにある、最小限の内容が書かれている example.c を手がかりにするとよいでしょう)。ならわしとして、ファイルは spam.c または spammodule.c という名前にしておきます。出力ファイル名はリリースモードでは spam.pyd、デバッグモードでは spam_d.pyd、になるはずです。.pyd という拡張子は、システムライブラリの spam.dll と作成した拡張モジュールの間での混乱を避けるために選ばれました。バージョン 2.5 で変更. さて、やり方は二通りあります:

8. example.dsw と example.vcproj をコピーし、 spam.* に名前を変えて、手作業で編集する

9. 新しくプロジェクトを作成する; 説明は下にあります。

   どちらの場合も、 `example_nt\example.def` を `spam\spam.def` にコピーして、新たにできた `spam.def` を編集し、二行目に 'initspam' が入るようにします。自分で新たなプロジェクトを作成したのなら、ここで `spam.def` をプロジェクトに追加しておいてください (このファイルはたった二行しかない目障りなファイルです。 `.def` ファイルを全く無視するという方法もあり、それには */exprt:initspam* を「プロジェクトのプロパティ」ダイアログにあるリンク設定のどこかに手動で追加します)。

10. **新しくプロジェクトを作成する** — **ファイル** → **新規作成** → **プロジェクト** ダイアログを使って、新たなプロジェクト用ワークスペースを作成します。 *Visual C++* **プロジェクト***/Win32/Win32* **プロジェクト**を選択し、名前 (spam) を入れ、「場所」が先ほど作成した `spam` ディレクトリの親ディレクトリに (Python ビルドツリーの直下のサブディレクトリで、 `Include` および `PC` と同じディレクトリになるはずです) あることを確かめます。「作成」をクリックします。

    前の節で述べた `spam.def` をここで作成しておかねばなりません。その後、**追加** → **ファイルをプロジェクトに追加** ダイアログを選びます。「ファイルの種類」を `*.*` にして、 `spam.c` と `spam.def` を選び、 OK をクリックします (一つ一つファイルを追加してもかまいません)。

    プロジェクト → *spam* **のプロパティ** ダイアログを開きます。ほんのいくつかですが、設定の変更が必要です。 **構成** ドロップダウンリストに **すべての構成**が設定されているか確かめてください。 C/C++ タブを選び、ポップアップメニューから「一般」カテゴリを選びます。以下のテキスト:

    ```
    ..\Include,..\PC
    ```

    を、 **追加のインクルードディレクトリ** とラベルされたエントリボックスに入力します

    次に、「リンカ」タブの「一般」カテゴリを選び、

    ```
    ..\PCbuild
    ```

    を **追加のライブラリディレクトリ** と書かれたテキストボックスに入力します。

    さて、構成ごとに特有の設定をいくつか行う必要があります:

    「構成」ドロップダウンリストから、 **リリース** を選んでください。「リンク」タブをクリックし、「入力」カテゴリを選んで、「追加の依存ファイル」ボックス内のリストに `pythonXY.lib` を追加します。

    「構成」ドロップダウンリストから、 **デバッグ** に切り替え、「追加の依存ファイル」ボックス内のリストに `pythonXY_d.lib` を追加します。次に C/C++ タブをクリックして、**コード生成** をカテゴリから選び、 **ラインタイムライブラリ** に対して**マルチスレッドデバッグ** *DLL* を選びます。

    「構成」ドロップダウンリストから **リリース** に切り替えなおします。 **ラインタイムライブラリ** に対して**マルチスレッド** *DLL* を選びます。

作っているモジュールが新たな型を作成するのなら、以下の行:

```
PyObject_HEAD_INIT(&PyType_Type)
```

がうまくいかないはずです。そこで:

```
PyObject_HEAD_INIT(NULL)
```

に変更してください。また、以下の行をモジュール初期化関数に加えます:

```
MyObject_Type.ob_type = &PyType_Type;
```

この操作を行う詳しい理由は、 Python FAQ の第 3 節を参照してください。

4.2 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、リンカは定義場所の不明な識別子に遭遇することがあります。このときリンカ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、リンカ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。リンカ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カに渡したりは **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います。 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。

4.3 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。不用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。

第 5 章

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。

Python の埋め込みを行っている簡単なデモは、ソース配布物の `Demo/embed/` ディレクトリにあります。

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。

5.1 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。

```
#include <Python.h>

int
main(int argc, char *argv[])
{
  Py_Initialize();
  PyRun_SimpleString("from time import time,ctime\n"
                     "print 'Today is',ctime(time())\n");
  Py_Finalize();
  return 0;
}
```

上のコードでは、まず Python インタプリタを `Py_Initialize()` で起動し、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_Finalize()` 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。

5.2 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 ルーチンを呼び出しますが、埋め込みの際には Python ルーチンを呼び出します。

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。

5.3 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:

```
#include <Python.h>

int
main(int argc, char *argv[])
{
    PyObject *pName, *pModule, *pDict, *pFunc;
    PyObject *pArgs, *pValue;
    int i;

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

    Py_Initialize();
    pName = PyString_FromString(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 = PyInt_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", PyInt_AsLong(pValue));
                Py_DECREF(pValue);
            }
            else {
                Py_DECREF(pFunc);
                Py_DECREF(pModule);
```

```
                PyErr_Print();
                fprintf(stderr,"Call failed\n");
                return 1;
            }
        }
        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;
    }
    Py_Finalize();
    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します:

```
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、

```
Py_Initialize();
pName = PyString_FromString(argv[1]);
/* pName のエラーチェックは省略している */
pModule = PyImport_Import(pName);
```

から始まる部分です。

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、データ変換ルーチン `PyString_FromString()` で構築します。

```
pFunc = PyObject_GetAttrString(pModule, argv[2]);
/* pFunc は新たな参照 */

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。

5.4 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す。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;

/* アプリケーションのコマンドライン引数の個数を返す */
static PyObject*
emb_numargs(PyObject *self, PyObject *args)
{
    if(!PyArg_ParseTuple(args, ":numargs"))
        return NULL;
    return Py_BuildValue("i", numargs);
}

static PyMethodDef EmbMethods[] = {
    {"numargs", emb_numargs, METH_VARARGS,
     "Return the number of arguments received by the process."},
    {NULL, NULL, 0, NULL}
};
```

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()` の直後に挿入します:

```
numargs = argc;
Py_InitModule("emb", EmbMethods);
```

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

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

のようなことができます。

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。

5.5 C++による Python の埋め込み

C++ プログラム中にも Python を埋め込めます; 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。

5.6 リンクに関する要件

Python ソースと一緒についてくる **configure** スクリプトは動的にリンクされる拡張モジュールが必要とするシンボルを公開するようただしく Python をビルドしますが、この機能は Python ライブラリを静的に埋め込むようなアプリケーションには継承されません。少なくとも Unix ではそうです。これは、アプリケーションが静的な実行時ライブラリ (`libpython.a`) にリンクされていて、かつ (`.so` ファイルとして実装されている) 動的ロードされるような拡張モジュールをロードする必要がある場合に起きる問題です。

問題になるのは、拡張モジュールが使うあるエントリポイントが Python ランタイムだけで定義されているという状況です。埋め込みを行うアプリケーション側がこうしたエントリポイントを全く使わない場合、リンカによってはエントリポイントを最終的に生成される実行可能形式のシンボルテーブル内に含めません。こうした場合、リンカに追加のオプションを与えて、これらのシンボルを除去しないよう教える必要があります。

プラットフォームごとに正しいオプションを決めるのはかなり困難です、とはいえ、幸運なことに、オプションは Python のビルド設定内にすでにあります。インストール済みの Python インタプリタからオプションを取り出すには、対話インタプリタを起動して、以下のような短いセッションを実行します:

```
>>> import distutils.sysconfig
>>> distutils.sysconfig.get_config_var('LINKFORSHARED')
'-Xlinker -export-dynamic'
```

表示された文字列の内容が、ビルド時に使うべきオプションです。文字列が空であれば、特に追加すべきオプションはありません。`LINKFORSHARED` の定義内容は、 Python のトップレベル `Makefile` 内の同名の変数に対応しています。

第 6 章

このドキュメントについて

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

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

または、Google Project Hosting 上の Issue Tracker

http://code.google.com/p/python-doc-ja/issues/list

までご報告ください。

6.1 翻訳者一覧 (敬称略)

- Yasushi MASUDA
- Yusuke SHINYAMA
- Kazuo Moriwaka
- TAKAGI Masahiro
- INADA Naoki <inada-n at klab.jp>

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。インタラクティブに実行されるコードサンプルとしてよく出てきます。

... インタラクティブシェルにおける、インデントされたコードブロックや対応する括弧 (丸括弧 ()、角括弧 []、curly brace{}) の内側で表示されるデフォルトのプロンプト。

2to3 Python 2.x のコードを Python 3.x のコードに変換するツール。ソースコードを解析して、その解析木を巡回 (traverse) して、非互換なコードの大部分を処理する。

2to3 は、`lib2to3` モジュールとして標準ライブラリに含まれています。スタンドアローンのツールとして使うときのコマンドは `Tools/scripts/2to3` として提供されています。 *2to3-reference* を参照してください。

abstract base class (抽象基底クラス) Abstract Base Classes (ABCs と略されます) は *duck-typing* を補完するもので、 `hasattr()` などの別のテクニックでは不恰好になる場合にインタフェースを定義する方法を提供します。Python は沢山のビルトイン ABCs を、(`collections` モジュールで) データ構造、(`numbers` モジュールで) 数値型、(`io` モジュールで) ストリーム型で提供いています。`abc` モジュールを利用して独自の ABC を作成することもできます。

argument (引数) 関数やメソッドに渡された値。関数の中では、名前の付いたローカル変数に代入されます。

関数やメソッドは、その定義中に位置指定引数 (positional arguments, 訳注: `f(1, 2)` のように呼び出し側で名前を指定せず、引数の位置に引数の値を対応付けるもの) とキーワード引数 (keyword arguments, 訳注: `f(a=1, b=2)` のように、引数名に引数の値を対応付けるもの) の両方を持つことができます。位置指定引数とキーワード引数は可変長です。関数定義や呼び出しは、 `*` を使って、不定数個の位置指定引数をシーケンス型に入れて受け取ったり渡したりすることができます。同じく、キーワード引数は `**` を使って、辞書に入れて受け取ったり渡したりできます。

引数リスト内では任意の式を使うことができ、その式を評価した値が渡されます。

attribute (属性) オブジェクトに関連付けられ、ドット演算子を利用して名前で参照される値。例えば、オブジェクト *o* が属性 *a* を持っているとき、その属性は *o.a* で参照されます。

BDFL 慈悲ぶかき独裁者 (Benevolent Dictator For Life) の略です。Python の作者、 Guido van Rossum のことです。

bytecode (バイトコード) Python のソースコードはバイトコードへとコンパイルされます。バイトコードは Python プログラムのインタプリタ内部での形です。バイトコードはまた、 `.pyc` や `.pyo` ファイルにキャッシュされ、同じファイルを二度目に実行した際により高速に実行できるようにします (ソースコードからバイトコードへの再度のコンパイルは回避されます)。このバイトコードは、各々のバイトコードに対応するサブルーチンを呼び出すような “仮想計算機 (*virtual machine*)” で動作する “中間言語 (intermediate language)” といえます。

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

classic class (旧スタイルクラス) `object` を継承していないクラス全てを指します。新スタイルクラス (*new-style class*) も参照してください。旧スタイルクラスは Python 3.0 で削除されます。

coercion (型強制) 同じ型の 2 つの引数を要する演算の最中に、ある型のインスタンスを別の型に暗黙のうちに変換することです。例えば、 `int(3.15)` は浮動小数点数を整数の `3` にします。しかし、 `3+4.5` の場合、各引数は型が異なっていて (一つは整数、一つは浮動小数点数)、加算をする前に同じ型に変換しなければいけません。そうでないと、 `TypeError` 例外が投げられます。2 つの被演算子間の型強制は組み込み関数の `coerce` を使って行えます。従って、 `3+4.5` は `operator.add(*coerce(3, 4.5))` を呼び出すことに等しく、`operator.add(3.0, 4.5)` という結果になります。型強制を行わない場合、たとえ互換性のある型であっても、すべての引数はプログラマーが、単に `3+4.5` とするのではなく、`float(3)+4.5` というように、同じ型に正規化しなければいけません。

complex number (複素数) よく知られている実数系を拡張したもので、すべての数は実部と虚部の和として表されます。虚数は虚数単位元 (`-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、工業では `j` と書かれます。

Python は複素数に組込みで対応し、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` となります。`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ー) `with` 文で扱われる、環境を制御するオブジェクト。`__enter__()` と `__exit__()` メソッドを定義することで作られる。

PEP 343 を参照。

CPython Python プログラミング言語の基準となる実装。CPython という単語は、この実装を Jython や IronPython といった他の実装と区別する必要が有る文脈で利用されます。

decorator (デコレータ) 関数を返す関数。通常、 `@wrapper` という文法によって関数を変換するのに利用されます。デコレータの一般的な利用レとして、 `classmethod()` と `staticmethod()` があります。

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

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

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

デコレータについてのより詳しい情報は、*the documentation for function definition* を参照してください。

descriptor (デスクリプタ) メソッド `__get__()`, `__set__()`, あるいは `__delete__()` が定義されている **新スタイル** *(new-style)* のオブジェクトです。あるクラス属性がデスクリプタである場合、その属性を参照するときに、そのデスクリプタに束縛されている特別な動作を呼び出します。通常、get,set,delete のために *a.b* と書くと、 *a* のクラス辞書内でオブジェクト *b* を検索しますが、 *b* がデスクリプタの場合にはデスクリプタで定義されたメソッドを呼び出します。デスクリプタの理解は、 Python を深く理解する上で鍵となります。というのは、デスクリプタこそが、関数、メソッド、プロパティ、クラスメソッド、静的メソッド、そしてスーパクラスの参照といった多くの機能の基盤だからです。

dictionary (辞書) 任意のキーを値に対応付ける連想配列です。`dict` の使い方は `list` に似ていますが、ゼロから始まる整数に限らず、`__hash__()` 関数を実装している全て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。

docstring クラス、関数、モジュールの最初の式となっている文字列リテラルです。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正しい場所です。

duck-typing Python 的なプログラムスタイルではオブジェクトの型を（型オブジェクトとの関係ではなく）メソッドや属性といったシグネチャを見ることで判断します。（「もしそれがガチョウのようにみえて、ガチョウのように鳴けば、それはガチョウである」）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 (polymorphic な置換を許可することによって) 柔軟性を増すことができます。duck-typing は `type()` や `isinstance()` を避けます。(ただし、duck-typing を抽象ベースクラス (*abstract base class*) で補完することもできます。) その代わりに `hasattr()` テストや *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。 Python で広く使われているコーディングスタイルでは、通常は有効なキーや属性が存在するものと仮定し、その仮定が誤っていた場合に例外を捕捉します。この簡潔で手早く書けるコーディングスタイルには、 `try` 文および `except` 文がたくさんあるのが特徴です。このテクニックは、C のような言語でよく使われている *LBYL* スタイルと対照的なものです。

expression (式) 何かの値に評価される、一つづきの構文 (a piece of syntax). 言い換えると、リテラル、名前、属性アクセス、演算子や関数呼び出しといった、値を返す式の要素の組み合わせ。他の多くの言語と違い、Python は言語の全ての構成要素が式というわけではありません。`print` や `if` のように、式にはならない、文 (*statement*) もあります。代入も式ではなく文です。

extension module (拡張モジュール) C や C++で書かれたモジュール。ユーザーコードや Python のコアとやりとりするために、Python の C API を利用します。

finder モジュールの *loader* を探すオブジェクト。`find_module()` という名前のメソッドを実装していなければなりません。詳細については **PEP 302** を参照してください。

function (関数) 呼び出し側に値を返す、一連の文。ゼロ個以上の引数を受け取り、それを関数の本体を実行するときに諒できます。*argument* や *method* も参照してください。

__future__ 互換性のない新たな機能を現在のインタプリタで有効にするためにプログラマが利用できる擬似モジュールです。例えば、式 `11/4` は現状では `2` になります。この式を実行しているモジュールで

```
from __future__ import division
```

を行って **真の除算操作** *(true division)* を有効にすると、式 `11/4` は `2.75` になります。実際に `__future__` モジュールを import してその変数を評価すれば、新たな機能が初めて追加されたのがいつで、いつデフォルトの機能になる予定かわかります。

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

garbage collection (ガベージコレクション) もう使われなくなったメモリを開放する処理。Python は、Python は参照カウントと循環参照を見つけて破壊する循環参照コレクタを使ってガベージコレクションを行います。

generator (ジェネレータ) イテレータを返す関数です。 `return` 文の代わりに `yield` 文を使って呼び出し側に要素を返す他は、通常の関数と同じに見えます。

よくあるジェネレータ関数は一つまたはそれ以上の `for` ループや `while` ループを含んでおり、ループの呼び出し側に要素を返す (`yield`) ようになっています。ジェネレータが返すイテレータを使って関数を実行すると、関数は `yield` キーワードで (値を返して) 一旦停止し、 `next()` を呼んで次の要素を要求するたびに実行を再開します。

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、ループ変を定義している `for` 式、範囲、そして省略可能な `if` 式がつづいているように見えます。こうして構成された式は、外側の関数に対して値を生成します。:

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

GIL グローバルインタプリタロック (*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* の VM(*virtual machine*) の中で一度に 1 つのスレッドだけが動作することを保証するために使われているロックです。このロックによって、同時に同じメモリにアクセスする 2 つのプロセスは存在しないと保証されているので、CPython を単純な構造にできるのです。インタプリタ全体にロックをかけると、多重プロセサ計算機における並列性の恩恵と引き換えにインタプリタの多重スレッド化を簡単に行えます。かつて “スレッド自由な (free-threaded)” インタプリタを作ろうと努力したことがありましたが、広く使われている単一プロセッサの場合にはパフォーマンスが低下するという事態に悩まされました。

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

辞書のキーや集合型のメンバーは、内部でハッシュ値を使っているので、ハッシュ可能オブジェクトである必要があります。

Python の全ての不変 (*immutable*) なビルドインオブジェクトはハッシュ可能です。リストや辞書といった変更可能なコンテナ型はハッシュ可能ではありません。

ユーザー定義クラスのインスタンスはデフォルトでハッシュ可能です。それらは、比較すると常に不等で、ハッシュ値は `id()` になります。

IDLE Python の組み込み開発環境 (Integrated DeveLopment Environment) です。IDLE は Python の標準的な配布物についてくる基本的な機能のエディタとインタプリタ環境です。初心者に向いている点として、 IDLE はよく洗練され、複数プラットフォームで動作する GUI アプリケーションを実装したい人むけの明解なコード例にもなっています。

immutable (不変オブジェクト) 固定の値を持ったオブジェクトです。変更不能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。不変オブジェクトは、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。辞書におけるキーがその例です。

integer division (整数除算) 剰余を考慮しない数学的除算です。例えば、式 `11/4` は現状では `2.75` ではなく `2` になります。これは **切り捨て除算** *(floor division)* とも呼ばれます。二つの整数間で除算を行うと、結果は (端数切捨て関数が適用されて) 常に整数になります。しかし、被演算子の一方が (`float` のような) 別の数値型の場合、演算の結果は共通の型に型強制されます (型強制 (*coercion*) 参照)。例えば、浮動小数点数で整数を除算すると結果は浮動小数点になり、場合によっては端数部分を伴います。 `//` 演算子を `/` の代わりに使うと、整数除算を強制できます。*__future__* も参照してください。

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

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

iterable (反復可能オブジェクト) 要素を一つずつ返せるオブジェクトです。

反復可能オブジェクトの例には、(`list`, `str`, `tuple` といった) 全てのシーケンス型や、 `dict` や `file` といった幾つかの非シーケンス型、あるいは `__iter__()` か `__getitem__()` メソッドを実装したクラスのインスタンスが含まれます。

反復可能オブジェクトは `for` ループ内やその他多くのシーケンス (訳注: ここでのシーケンスとは、シーケンス型ではなくただの列という意味) が必要となる状況 (`zip()`, `map()`, ...) で利用できます。

反復可能オブジェクトを組み込み関数 `iter()` の引数として渡すと、オブジェクトに対するイテレータを返します。このイテレータは一連の値を引き渡す際に便利です。反復可能オブジェクトを使う際には、通常 `iter()` を呼んだり、イテレータオブジェクトを自分で扱う必要はありません。`for` 文ではこの操作を自動的に行い、無名の変数を作成してループの間イテレータを記憶します。イテレータ (*iterator*) シーケンス (*sequence*), およびジェネレータ (*generator*) も参照してください。

iterator 一連のデータ列 (stream) 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と、データ列中の要素を一つずつ返します。後続のデータがなくなると、データの代わりに `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全てのオブジェクトを出し尽くしており、それ以降は `next()` を何度呼んでも `StopIteration` を送出します。イテレータは、そのイテレータオブジェクト自体を返す `__iter__()` メソッドを実装しなければならなくなっており、そのため全てのイテレータは他の反復可能オブジェクトを受理できるほとんどの場所

で利用できます。著しい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では、 `iter()` 関数にオブジェクトを渡したり、 `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。このイテレータをさらに別の場所でイテレータとして使おうとすると、前回のイテレーションパスで使用された同じイテレータオブジェクトを返すため、空のコンテナのように見えます。

より詳細な情報は *typeiter* にあります。

keyword argument (キーワード引数) 呼び出し時に、 `variable_name=` が手前にある引数。変数名は、その値が関数内のどのローカル変数に渡されるかを指定します。キーワード引数として辞書を受け取ったり渡したりするために `**` を使うことができます。 *argument* も参照してください。

lambda (ラムダ) 無名のインライン関数で、関数が呼び出されたときに評価される 1 つの式 (*expression*) を持ちます。ラムダ関数を作る構文は、 `lambda [arguments]: 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。

list (リスト) Python のビルトインのシーケンス型 (*sequence*) です。リストという名前ですが、リンクリストではなく、他の言語で言う配列 (array) と同種のもので、要素へのアクセスは O(1) です。

list comprehension (リスト内包表記) シーケンス内の全てあるいは一部の要素を処理して、その結果からなるリストを返す、コンパクトな書き方です。`result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` とすると、 0 から 255 までの偶数を 16 進数表記 (0x..) した文字列からなるリストを生成します。`if` 節はオプションです。 `if` 節がない場合、`range(256)` の全ての要素が処理されます。

loader モジュールをロードするオブジェクト。`load_module()` という名前のメソッドを定義していなければなりません。詳細は **PEP 302** を参照してください。

mapping (マップ) 特殊メソッド `__getitem__()` を使って、任意のキーに対する検索をサポートする (`dict` のような) コンテナオブジェクトです。

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

method クラス内で定義された関数。クラス属性として呼び出された場合、メソッドはインスタンスオブジェクトを第一引数 (*argument*) として受け取ります (この第一引数は普段 `self` と呼ばれます)。*function* と *nested scope* も参照してください。

mutable (変更可能オブジェクト) 変更可能なオブジェクトは、 `id()` を変えることなく値を変更できます。変更不能 (*immutable*) も参照してください。

named tuple (名前付きタプル) タプルに似ていて、インデックスによりアクセスする要素に名前付き属性としてもアクセス出来るクラス。(例えば、 `time.localtime()` はタプルに似たオブジェクトを返し、その *year* には `t[0]` のようなインデックスによるアクセスと、`t.tm_year` のような名前付き要素としてのアクセスが可能です。)

名前付きタプルには、 time.struct_time のようなビルトイン型もありますし、通常のクラス定義によって作成することもできます。名前付きタプルを collections.namedtuple() ファクトリ関数で作成することもできます。最後の方法で作った名前付きタプルには自動的に、Employee(name='jones', title='programmer') のような自己ドキュメント表現 (self-documenting representation) 機能が付いてきます。

namespace (名前空間) 変数を記憶している場所です。名前空間は辞書を用いて実装されています。名前空間には、ローカル、グローバル、組み込み名前空間、そして (メソッド内の) オブジェクトのネストされた名前空間があります。例えば、関数 __builtin__.open() と os.open() は名前空間で区別されます。名前空間はまた、ある関数をどのモジュールが実装しているかをはっきりさせることで、可読性やメンテナンス性に寄与します。例えば、 random.seed(), itertools.izip() と書くことで、これらの関数がそれぞれ random モジュールや itertools モジュールで実装されていることがはっきりします。

nested scope (ネストされたスコープ) 外側で定義されている変数を参照する機能。具体的に言えば、ある関数が別の関数の中で定義されている場合、内側の関数は外側の関数中の変数を参照できます。ネストされたスコープ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。変数の代入は、常に最も内側のスコープにある変数に対する書き込みになります。同様に、グローバル変数を使うとグローバル名前空間の値を読み書きします。

new-style class (新スタイルクラス) object から継承したクラス全てを指します。これには list や dict のような全ての組み込み型が含まれます。 __slots__(), デスクリプタ、プロパティ、__getattribute__() といった、Python の新しい機能を使えるのは新スタイルクラスだけです。

より詳しい情報は *newstyle* を参照してください。

object 状態 (属性や値) と定義された振る舞い (メソッド) をもつ全てのデータ。もしくは、全ての新スタイルクラス (*new-style class*) の基底クラスのこと。

positional argument (位置指定引数) 引数のうち、呼び出すときの順序で、関数やメソッドの中のどの名前に代入されるかが決定されるもの。複数の位置指定引数を、関数定義側が受け取ったり、渡したりするために、* を使うことができます。*argument* も参照してください。

Python 3000 Python の次のメジャーバージョンである Python 3.0 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繋がる、考え方やコード。例えば、Python の一般的なイディオムに iterable の要素を for 文を使って巡回することです。この仕組を持たない言語も多くあるので、Python に慣れ親しんでいない人は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]
```

これと対照的な、よりきれいな Pythonic な方法はこうなります。

```
for piece in food:
    print piece
```

reference count (参照カウント) あるオブジェクトに対する参照の数。参照カウントが 0 になったとき、そのオブジェクトは破棄されます。参照カウントは通常は Python のコード上には現れませんが、*CPython* 実装の重要な要素です。sys モジュールは、プログラマーが任意のオブジェクトの参照カウントを知るための getrefcount() 関数を提供しています。

__slots__ 新スタイルクラス (*new-style class*) 内で、インスタンス属性の記憶に必要な領域をあらかじめ定義しておき、それとひきかえにインスタンス辞書を排除してメモリの節約を行うための宣言です。これはよく使われるテクニックですが、正しく動作させるのには少々手際を要するので、例えばメモリが死活問題となるようなアプリケーション内にインスタンスが大量に存在するといった稀なケースを除き、使わないのがベストです。

sequence (シーケンス) 特殊メソッド `__getitem__()` で整数インデックスによる効率的な要素へのアクセスをサポートし、 `len()` で長さを返すような反復可能オブジェクト (*iterable*) です。組み込みシーケンス型には、 `list`, `str`, `tuple`, `unicode` などがあります。`dict` は `__getitem__()` と `__len__()` もサポートしますが、検索の際に任意の変更不能 (*immutable*) なキーを使うため、シーケンスではなくマップ (mapping) とみなされているので注意してください。

slice (スライス) 多くの場合、シーケンス (*sequence*) の一部を含むオブジェクト。スライスは、添字記号 `[]` で数字の間にコロンを書いたときに作られます。例えば、`variable_name[1:3:5]` です。添字記号は `slice` オブジェクトを内部で利用しています。(もしくは、古いバージョンの、`__getslice__()` と `__setslice__()` を利用します。)

special method (特殊メソッド) ある型に対する特定の動作をするために、 Python から暗黙的に呼ばれるメソッド。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 つを持ちます。特殊メソッドについては *specialnames* で解説されています。

statement (文) 文は一種のコードブロックです。文は *expression* か、それ以外のキーワードにより構成されます。例えば `if`, `while`, `print` は文です。

triple-quoted string (三重クォート文字列) 3 つの連続したクォート記号 (") かアポストロフィー (') で囲まれた文字列。通常の (一重) クォート文字列に比べて表現できる文字列に違いはありませんが、幾つかの理由で有用です。1 つか 2 つの連続したクォート記号をエスケープ無しに書くことができますし、行継続文字 (\) を使わなくても複数行にまたがることができるので、ドキュメンテーション文字列を書く時に特に便利です。

type (型) Python のオブジェクトの型は、そのオブジェクトの種類を決定します。全ての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__` 属性からアクセスしたり、`type(obj)` で取得することができます。

virtual machine (仮想マシン) ソフトウェアにより定義されたコンピュータ。Python の仮想マシンは、バイトコードコンパイラが出力したバイトコード (*bytecode*) を実行します。

Zen of Python (Python の悟り) Python を理解し利用する上での導きとなる、Python の設計原則と哲学をリストにしたものです。対話プロンプトで "`import this`" とするとこのリストを読めます。

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、 docs@python.org メーリングリスト上で行われています。私たちは常に、一緒にドキュメントの開発をしてくれるボランティアを探しています。気軽にこのメーリングリストにメールしてください。

多大な感謝を:

- Fred L. Drake, Jr., the creator of the original Python documentation toolset and writer of much of the content;
- the Docutils project for creating reStructuredText and the Docutils suite;
- Fredrik Lundh for his Alternative Python Reference project from which Sphinx got many good ideas.

このドキュメントや、 Python 自体のバグ報告については、 *reporting-bugs* を参照してください。

ノート: 訳注: 日本語訳の問題については、 Google Project Hosting 上の Issue Tracker <http://code.google.com/p/python-doc-ja/issues/list> に登録するか、python-doc-jp@python.jp にメールで報告をお願いします。

B.1 Python ドキュメント 貢献者

この節では、Python ドキュメントに何らかの形で貢献した人をリストアップしています。このリストは完全ではありません – もし、このリストに載っているべき人を知っていたら、docs@python.org にメールで教えてください。私たちは喜んでその問題を修正します。

Aahz, Michael Abbott, Steve Alexander, Jim Ahlstrom, Fred Allen, A. Amoroso, Pehr Anderson, Oliver Andrich, Jes s Cea Avi n, Daniel Barclay, Chris Barker, Don Bashford, Anthony Baxter, Alexander Belopolsky, Bennett Benson, Jonathan Black, Robin Boerdijk, Michal Bozon, Aaron Brancotti, Georg Brandl, Keith Briggs, Ian Bruntlett, Lee Busby, Lorenzo M. Catucci, Carl Cerecke, Mauro Cicognini, Gilles Civario, Mike Clarkson, Steve

Clift, Dave Cole, Matthew Cowles, Jeremy Craven, Andrew Dalke, Ben Darnell, L. Peter Deutsch, Robert Donohue, Fred L. Drake, Jr., Josip Dzolonga, Jeff Epler, Michael Ernst, Blame Andy Eskilsson, Carey Evans, Martijn Faassen, Carl Feynman, Dan Finnie, Hern n Mart nez Foffani, Stefan Franke, Jim Fulton, Peter Funk, Lele Gaifax, Matthew Gallagher, Ben Gertzfield, Nadim Ghaznavi, Jonathan Giddy, Shelley Gooch, Nathaniel Gray, Grant Griffin, Thomas Guettler, Anders Hammarquist, Mark Hammond, Harald Hanche-Olsen, Manus Hand, Gerhard H ring, Travis B. Hartwell, Tim Hatch, Janko Hauser, Thomas Heller, Bernhard Herzog, Magnus L. Hetland, Konrad Hinsen, Stefan Hoffmeister, Albert Hofkamp, Gregor Hoffleit, Steve Holden, Thomas Holenstein, Gerrit Holl, Rob Hooft, Brian Hooper, Randall Hopper, Michael Hudson, Eric Huss, Jeremy Hylton, Roger Irwin, Jack Jansen, Philip H. Jensen, Pedro Diaz Jimenez, Kent Johnson, Lucas de Jonge, Andreas Jung, Robert Kern, Jim Kerr, Jan Kim, Greg Kochanski, Guido Kollerie, Peter A. Koren, Daniel Kozan, Andrew M. Kuchling, Dave Kuhlman, Erno Kuusela, Thomas Lamb, Detlef Lannert, Piers Lauder, Glyph Lefkowitz, Robert Lehmann, Marc-Andr Lemburg, Ross Light, Ulf A. Lindgren, Everett Lipman, Mirko Liss, Martin von L wis, Fredrik Lundh, Jeff MacDonald, John Machin, Andrew MacIntyre, Vladimir Marangozov, Vincent Marchetti, Laura Matson, Daniel May, Rebecca McCreary, Doug Mennella, Paolo Milani, Skip Montanaro, Paul Moore, Ross Moore, Sjoerd Mullender, Dale Nagata, Ng Pheng Siong, Koray Oner, Tomas Oppelstrup, Denis S. Otkidach, Zooko O'Whielacronx, Shriphani Palakodety, William Park, Joonas Paalasmaa, Harri Pasanen, Bo Peng, Tim Peters, Benjamin Peterson, Christopher Petrilli, Justin D. Pettit, Chris Phoenix, Fran ois Pinard, Paul Prescod, Eric S. Raymond, Edward K. Ream, Sean Reifschneider, Bernhard Reiter, Armin Rigo, Wes Rishel, Armin Ronacher, Jim Roskind, Guido van Rossum, Donald Wallace Rouse II, Mark Russell, Nick Russo, Chris Ryland, Constantina S., Hugh Sasse, Bob Savage, Scott Schram, Neil Schemenauer, Barry Scott, Joakim Sernbrant, Justin Sheehy, Charlie Shepherd, Michael Simcich, Ionel Simionescu, Michael Sloan, Gregory P. Smith, Roy Smith, Clay Spence, Nicholas Spies, Tage Stabell-Kulo, Frank Stajano, Anthony Starks, Greg Stein, Peter Stoehr, Mark Summerfield, Reuben Sumner, Kalle Svensson, Jim Tittsler, Ville Vainio, Martijn Vries, Charles G. Waldman, Greg Ward, Barry Warsaw, Corran Webster, Glyn Webster, Bob Weiner, Eddy Welbourne, Jeff Wheeler, Mats Wichmann, Gerry Wiener, Timothy Wild, Collin Winter, Blake Winton, Dan Wolfe, Steven Work, Thomas Wouters, Ka-Ping Yee, Rory Yorke, Moshe Zadka, Milan Zamazal, Cheng Zhang.

Python がこの素晴らしいドキュメントを持っているのは、Python コミュニティによる情報提供と貢献のおかげです。 – ありがとう！

History and License

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, http://www.cwi.nl/ 参照) で Guido van Rossum によって ABC と呼ばれる言語の後継言語として生み出されました。その後多くの人々が Python に貢献していますが、 Guido は今日でも Python 製作者の先頭に立っています。

1995 年、 Guido は米国ヴァージニア州レストンにある Corporation for National Reserch Initiatives (CNRI, http://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, http://www.zope.com/ 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、 http://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 http://www.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	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
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes

次のページに続く

表 C.1 – 前のページからの続き

リリース	ベース	年	権利	GPL 互換
2.2.1	2.2	2002	PSF	yes
2.2.2	2.2.1	2002	PSF	yes
2.2.3	2.2.2	2002-2003	PSF	yes
2.3	2.2.2	2002-2003	PSF	yes
2.3.1	2.3	2002-2003	PSF	yes
2.3.2	2.3.1	2003	PSF	yes
2.3.3	2.3.2	2003	PSF	yes
2.3.4	2.3.3	2004	PSF	yes
2.3.5	2.3.4	2005	PSF	yes
2.4	2.3	2004	PSF	yes
2.4.1	2.4	2005	PSF	yes
2.4.2	2.4.1	2005	PSF	yes
2.4.3	2.4.2	2006	PSF	yes
2.4.4	2.4.3	2006	PSF	yes
2.5	2.4	2006	PSF	yes
2.5.1	2.5	2007	PSF	yes
2.5.2	2.5.1	2008	PSF	yes
2.5.3	2.5.2	2008	PSF	yes
2.6	2.5	2008	PSF	yes
2.6.1	2.6	2008	PSF	yes
2.6.2	2.6.1	2009	PSF	yes
2.6.3	2.6.2	2009	PSF	yes
2.6.4	2.6.3	2009	PSF	yes
2.6.5	2.6.4	2010	PSF	yes

ノート: 「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してくださった多くのボランティアのみなさんに感謝します。

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

PSF LICENSE AGREEMENT FOR PYTHON 2.6ja2

1. This LICENSE AGREEMENT is between the Python Software Foundation (“PSF”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 2.6ja2 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 deriva-

tive works, distribute, and otherwise use Python 2.6ja2 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright 2001-2010 Python Software Foundation; All Rights Reserved" are retained in Python 2.6ja2 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.6ja2 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.6ja2.

4. PSF is making Python 2.6ja2 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.6ja2 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.6ja2 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.6ja2, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.6ja2, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

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

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT

OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

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

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

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

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

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

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.keio.ac.jp/ matumoto/MT2002/emt19937ar.html . The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).
```

```
Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
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.keio.ac.jp/matumoto/emt.html
email: matumoto@math.keio.ac.jp
```

C.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
GAI_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 GAI_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 GAI_ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN GAI_ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

```
  ---------------------------------------------------------------------
 /                      Copyright (c) 1996.                            \
|          The Regents of the University of California.                 |
|                        All rights reserved.                           |
|                                                                       |
|   Permission to use, copy, modify, and distribute this software for   |
|   any purpose without fee is hereby granted, provided that this en-   |
|   tire notice is included in all copies of any software which is or   |
|   includes  a  copy  or  modification  of  this software and in all   |
|   copies of the supporting documentation for such software.           |
|                                                                       |
|   This  work was produced at the University of California, Lawrence   |
|   Livermore National Laboratory under  contract  no.  W-7405-ENG-48   |
|   between  the  U.S.  Department  of  Energy and The Regents of the   |
|   University of California for the operation of UC LLNL.              |
|                                                                       |
|                             DISCLAIMER                                |
|                                                                       |
|   This  software was prepared as an account of work sponsored by an   |
|   agency of the United States Government. Neither the United States   |
|   Government  nor the University of California nor any of their em-   |
|   ployees, makes any warranty, express or implied, or  assumes  any   |
|   liability  or  responsibility  for the accuracy, completeness, or   |
|   usefulness of any information,  apparatus,  product,  or  process   |
|   disclosed,   or  represents  that  its  use  would  not  infringe   |
|   privately-owned rights. Reference herein to any specific  commer-   |
|   cial  products,  process,  or  service  by trade name, trademark,   |
|   manufacturer, or otherwise, does not  necessarily  constitute  or   |
|   imply  its endorsement, recommendation, or favoring by the United   |
|   States Government or the University of California. The views  and   |
|   opinions  of authors expressed herein do not necessarily state or   |
|   reflect those of the United States Government or  the  University   |
|   of  California,  and shall not be used for advertising or product   |
 \  endorsement purposes.                                              /
  ---------------------------------------------------------------------
```

C.3.4 MD5 message digest algorithm

The source code for the `md5` module contains the following notice:

```
Copyright (C) 1999, 2002 Aladdin Enterprises.  All rights reserved.

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.
```

```
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

L. Peter Deutsch
ghost@aladdin.com

Independent implementation of MD5 (RFC 1321).

This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
      http://www.ietf.org/rfc/rfc1321.txt
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A.  It does not include
any code or documentation that is identified in the RFC as being
copyrighted.

The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>.  Other authors are noted in the change history
that follows (in reverse chronological order):

2002-04-13 lpd Removed support for non-ANSI compilers; removed
      references to Ghostscript; clarified derivation from RFC 1321;
      now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
      added conditionalization for C++ compilation from Martin
      Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
```

C.3.5 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

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

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
```

```
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.6 Cookie management

The `Cookie` 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.3.7 Profiling

The `profile` and `pstats` modules contain the following notice:

```
Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software
and its associated documentation for any purpose (subject to the
restriction in the following sentence) 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 InfoSeek not be used in
advertising or publicity pertaining to distribution of the software
without specific, written prior permission.  This permission is
explicitly restricted to the copying and modification of the software
to remain in Python, compiled Python, or other languages (such as C)
wherein the modified or derived code is exclusively imported into a
Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION 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 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.9 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.3.10 XML Remote Procedure Calls

The xmlrpclib 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 MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.11 test_epoll

The test_epoll contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.12 Select kqueue

The `select` and contains the following notice for the kqueue interface:

```
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.
```

Copyright

Python and this documentation is:

Copyright 2001-2010 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.

Japanese translation is: Copyright 2003-2009 Python Document Japanese Translation Project. All rights reserved.

ライセンスおよび許諾に関する完全な情報は、 *History and License* を参照してください。

索引

..., **83**
__future__, **85**
__slots__, **90**
>>>, **83**
2to3, **83**

abstract base class, **83**
argument, **83**
attribute, **83**

BDFL, **83**
bytecode, **84**

class, **84**
classic class, **84**
coercion, **84**
complex number, **84**
context manager, **84**
CPython, **84**

deallocation, object, 54
decorator, **84**
descriptor, **85**
dictionary, **85**
distutils.sysconfig
    モジュール, 80
docstring, **85**
duck-typing, **85**

EAFP, **85**
expression, **85**
extension module, **85**

finalization, of objects, 54
finder, **85**
function, **85**

garbage collection, **86**
generator, 86, **86**
generator expression, 86, **86**
GIL, **86**
global interpreter lock, **86**

hashable, **86**

IDLE, **87**
immutable, **87**
importer, **87**
integer division, **87**
interactive, **87**
interpreted, **87**
iterable, **87**
iterator, **87**

keyword argument, **88**

lambda, **88**
LBYL, **88**
list, **88**
list comprehension, **88**
loader, **88**

mapping, **88**
metaclass, **88**
method, **88**
mutable, **88**

named tuple, **88**
namespace, **89**
nested scope, **89**
new-style class, **89**

object, **89**

deallocation, 54
finalization, 54

Philbrick, Geoff, 14
positional argument, **89**
PyArg_ParseTuple(), 12
PyArg_ParseTupleAndKeywords(), 13
PyErr_Fetch(), 54
PyErr_Restore(), 54
PyObject_CallObject(), 11
Python 3000, **89**
Python Enhancement Proposals
PEP 302, 85, 88
PEP 343, 84
Pythonic, **89**

READ_RESTRICTED, 58
READONLY, 58
reference count, **89**
repr
組み込み関数, 55
RESTRICTED, 58
RO, 58

sequence, **90**
slice, **90**
special method, **90**
statement, **90**
str
組み込み関数, 55

triple-quoted string, **90**
type, **90**

virtual machine, **90**

WRITE_RESTRICTED, 58

Zen of Python, **90**

組み込み関数
repr, 55
str, 55

モジュール
distutils.sysconfig, 80