Python モジュールの配布

Greg Ward
日本語訳: Python ドキュメント翻訳プロジェクト

September 20, 2004
Email: distutils-sig@python.org

Abstract

このドキュメントでは、 Python モジュール配布ユーティリティ(Python Distribution Utilities, “Distutils”) について、モジュール開発者の視点に立ち、多くの人々がビルド/リリース/インストールの負荷をほとんどかけずに Python モジュールや拡張モジュールを入手できるようにする方法について述べます。

Contents

1 はじめに

このドキュメントで扱っている内容は、 Distutils を使った Python モジュールの配布で、とりわけ開発者/配布者の役割に重点を置いています: Python モジュールのインストールに関する情報を探しているのなら、*Installing Python Modules* マニュアルを参照してください。

2 概念と用語

Distutils の使い方は、モジュール開発者とサードパーティ製のモジュールをインストールするユーザ/管理者のどちらにとってもきわめて単純です。開発者側のやるべきことは (もちろん、しっかりした実装で、詳しく文書化され、よくテストされたコードを書くことは別として！) 以下の項目になります:

- setup スクリプト ('setup.py' という名前にするのがならわし) を書く
- (必要があれば) setup 設定ファイルを書く
- ソースコード配布物を作成する
- (必要があれば) 一つまたはそれ以上のビルド済み (バイナリ) 形式の配布物を作成する

これらの作業については、いずれもこのドキュメントで扱っています。

全てのモジュール開発者が複数の実行プラットフォームを利用できるわけではないので、全てのプラットフォーム向けにビルド済みの配布物を提供してもらえると期待するわけにはいきません。ですから、仲介を行う人々、いわゆる **パッケージ作成者** *(packager)* がこの問題を解決すべく立ち上がってくれることが望ましいでしょう。パッケージ作成者はモジュール開発者がリリースしたソースコード配布物を、一つまたはそれ以上のプラットフォーム上でビルドして、得られたビルド済み配布物をリリースすることになります。したがって、ほとんどの一般的なプラットフォームにおけるユーザは、setup スクリプト一つ実行せず、コードを一行たりともコンパイルしなくても、使っているプラットフォーム向けのきわめて普通の方法でほとんどの一般的な Python モジュール配布物をインストールできるでしょう。

2.1 簡単な例

setup スクリプトは通常単純なものですが、Python で書かれているため、スクリプト中で何かを処理しようと考えたとき特に制限はありません。とはいえ、 setup スクリプト中に何かコストの大きな処理を行うときは十分注意してください。autoconf 形式の設定スクリプトとは違い、 setup スクリプトはモジュール配布物をビルドしてインストールする中で複数回実行されることがあります。

'foo.py' という名前のファイルに収められている `foo` という名前のモジュールを配布したいだけなら、setup スクリプトは以下のような単純なものになります:

```
from distutils.core import setup
setup(name="foo",
      version="1.0",
      py_modules=["foo"])
```

以下のことに注意してください:

- Distutils に与えなければならない情報のほとんどは、`setup()` 関数のキーワード引数として与えます。
- キーワード引数は二つのカテゴリ: パッケージのメタデータ (パッケージ名、バージョン番号) 、パッケージに何が収められているかの情報 (上の場合は pure Python モジュールのリスト)、に行き着きます。
- モジュールはファイル名ではなく、モジュール名で指定します (パッケージと拡張モジュールについても同じです)
- 作者名、電子メールアドレス、プロジェクトの URL といった追加のメタデータを入れておくよう奨めます (3 の例を参照してください)

このモジュールのソースコード配布物を作成するには、上記のコードが入った setup スクリプト 'setup.py' を作成して、以下のコマンド:

```
python setup.py sdist
```

を実行します。

この操作を行うと、アーカイブファイル (例えば UNIX では tarball、Windows では ZIP ファイル) を作成します。アーカイブファイルには、setup スクリプト 'setup.py' と、配布したいモジュール 'foo.py' が入っています。アーカイブファイルの名前は 'foo-1.0.targ.gz' (または '.zip') になり、展開するとディレクトリ 'foo-1.0' を作成します。

エンドユーザが `foo` モジュールをインストールしたければ、'foo-1.0.tar.gz' (または '.zip') をダウンロードし、パッケージを展開して、以下のスクリプトを — 'foo-1.0' ディレクトリ中で — 実行します:

```
python setup.py install
```

この操作を行うと、インストールされている Python での適切なサードパーティ製モジュール置き場に 'foo.py' を完璧にコピーします.

ここで述べた簡単な例では、 Distutils の基本的な概念のいくつかを示しています。まず、開発者とインストール作業者は同じ基本インタフェース、すなわち setup スクリプトを使っています。二人の作業の違いは、使っている Distutils **コマンド** *(command)* にあります: `sdist` コマンドは、ほぼ完全に開発者だけが対象となる一方、`install` はどちらかというとインストール作業者向けです (とはいえ、ほとんどの開発者は自分のコードをインストールしたくなることがあるでしょう)。

ユーザにとって本当に簡単なものにしたいのなら、一つまたはそれ以上のビルド済み配布物を作ってあげられます。例えば、Windows マシン上で作業をしていて、他の Windows ユーザにとって簡単な配布物を提供したいのなら、実行可能な形式の (このプラットフォーム向けのビルド済み配布物としてはもっとも適切な) インストーラを作成できます。これには `bdist_wininst` を使います。例えば:

```
python setup.py bdist_wininst
```

とすると、実行可能なインストーラ形式、'foo-1.0.win32.exe' が現在のディレクトリに作成されます。

その他の有用な配布形態としては、`bdist_rpm` に実装されている RPM 形式、 Solaris **pkgtool** (`bdist_pkgtool`) 、 HP-UX **swinstall** (`bdist_sdux`) があります。例えば、以下のコマンドを実行すると、'foo-1.0.noarch.rpm' という名前の RPM ファイルを作成します:

```
python setup.py bdist_rpm
```

(`bdist_rpm` コマンドは `rpm` コマンドを使うため、Red Hat Linux や SuSE Linux、 Mandrake Linux といった RPM ベースのシステムで実行しなければなりません)

どの配布形式が利用できるかは、

```
python setup.py bdist --help-formats
```

を実行すれば分かります。

2.2 Python 一般の用語

このドキュメントを読んでいるのなら、モジュール (module)、拡張モジュール (extension) などが何を表すのかをよく知っているかもしれません。とはいえ、読者がみな共通のスタートポイントに立って Distutils の操作を始められるように、ここで一般的な Python 用語について以下のような用語集を示しておきます:

モジュール (module) Python においてコードを再利用する際の基本単位: すなわち、他のコードから import されるひとかたまりのコードです。ここでは、三種類のモジュール: pure Python モジュール、拡張モジュール、パッケージが関わってきます。

pure Python モジュール Python で書かれ、単一の '.py' ファイル内に収められたモジュールです ('.pyc' かつ/または '.pyo' ファイルと関連があります) 。 "pure モジュール (pure module)" と呼ばれることもあります。

拡張モジュール (extension module) Python を実装している低水準言語: Python の場合は C/C++ 、 Jython の場合は Java 、で書かれたモジュールです。通常は、動的にロードできるコンパイル済みの単一のファイルに入っています。例えば、UNIX 向け Python 拡張のための共有オブジェクト ('.so') 、Windows 向け Python 拡張のための DLL ('.pyd' という拡張子が与えられています)、 Jython 拡張のための Java クラスといった具合です。(現状では、 Distutils は Python 向けの C/C++ 拡張モジュールしか扱わないので注意してください。)

パッケージ (package) 他のモジュールが入っているモジュールです; 通常、ファイルシステム内のあるディレクトリに収められ、'__init__.py' が入っていることで通常のディレクトリと区別できます。

ルートパッケージ (root package) 階層的なパッケージの根 (root) の部分にあたるパッケージです。(この部分には '__init__.py' ファイルがないので、本当のパッケージではありませんが、便宜上そう呼びます。) 標準ライブラリの大部分はルートパッケージに入っています、また、多くの小規模な単体のサードパーティモジュールで、他の大規模なモジュールコレクションに属していないものもここに入ります。正規のパッケージと違い、ルートパッケージ上のモジュールの実体は様々なディレクトリにあります: 実際は、`sys.path` に列挙されているディレクトリ全てが、ルートパッケージに配置されるモジュールの内容に影響します。

2.3 Distutils 固有の用語

以下は Distutils を使って Python モジュールを配布する際に使われる特有の用語です:

モジュール配布物 (module distribution) 一個のファイルとしてダウンロード可能なリソースの形をとり、**一括して**インストールされることになっている形態で配られる Python モジュールのコレクションです。よく知られたモジュール配布物には、Numeric Python、PyXML、 PIL (the Python Imaging Library)、mxBase などがあります。(**パッケージ** *(package)* と呼ばれることもありますが、 Python 用語としてのパッケージとは意味が違います: 一つのモジュール配布物の中には、場合によりゼロ個、一つ、それ以上の Python パッケージが入っています。)

pure モジュール配布物 (pure module distribution) pure Python モジュールやパッケージだけが入ったモジュール配布物です。"pure 配布物 (pure distribution)" とも呼ばれます。

非 pure モジュール配布物 (non-pure module distribution) 少なくとも一つの拡張モジュールが入ったモジュール配布物です。"非 pure 配布物" とも呼びます。

配布物ルートディレクトリ (distribution root) ソースコードツリー (またはソース配布物) ディレクトリの最上階層で、‘setup.py’ のある場所です。一般的には、‘setup.py’ はこのディレクトリ上で実行します。

3 setup スクリプトを書く

setup スクリプトは、Distutils を使ってモジュールをビルドし、配布し、インストールする際の全ての動作の中心になります。setup スクリプトの主な目的は、モジュール配布物について Distutils に伝え、モジュール配布を操作するための様々なコマンドを正しく動作させることにあります。 上の 2.1 の節で見てきたように、setup スクリプトは主に `setup()` の呼び出しからなり、開発者が distuils に対して与えるほとんどの情報は `setup()` のキーワード引数として指定されます。

ここではもう少しだけ複雑な例: Distutils 自体の setup スクリプト、を示します。これについては、以降の二つの節でフォローします。(Distutils が入っているのは Python 1.6 以降であり、 Python 1.5.2 ユーザが他のモジュール配布物をインストールできるようにするための独立したパッケージがあることを思い出してください。ここで示した、Distutils 自身の setup スクリプトは、Python 1.5.2 に Distutils パッケージをインストールする際に使います。)

```
#!/usr/bin/env python

from distutils.core import setup

setup(name="Distutils",
      version="1.0",
      description="Python Distribution Utilities",
      author="Greg Ward",
      author_email="gward@python.net",
      url="http://www.python.org/sigs/distutils-sig/",
      packages=['distutils', 'distutils.command'],
     )
```

上の例と、 2.1 で示したファイル一つからなる小さな配布物とは、違うところは二つしかありません: メタデータの追加と、モジュールではなくパッケージとして pure Python モジュール群を指定しているという点です。この点は重要です。というのも、 Distutils は 2 ダースものモジュールが (今のところ) 二つのパッケージに分かれて入っているからです; 各モジュールについていちいち明示的に記述したリストは、作成するのが面倒だし、維持するのも難しくなるでしょう。その他のメタデータについては、 3.7 を参照してください。

setup スクリプトに与えるパス名 (ファイルまたはディレクトリ) は、 UNIX におけるファイル名規約、つまりスラッシュ (’/’) 区切りで書かねばなりません。Distutils はこのプラットフォーム中立の表記を、実際にパス名として使う前に、現在のプラットフォームに適した表記に注意深く変換します。この機能のおかげで、setup スクリプトを異なるオペレーティングシステム間にわたって可搬性があるものにできます。言うまでもなく、これは Distutils の大きな目標の一つです。この精神に従って、このドキュメントでは全てのパス名をスラッシュ区切りにしています。 (MacOS プログラマは、先頭にスラッシュが **ない**場合は、 **相対パス** を表すということを心に留めておかねばなりません。この規約は、コロンを使った MacOS での規約と逆だからです。)

もちろん、この取り決めは Distutils に渡すパス名だけに適用されます。もし、例えば `glob.glob()` や `os.listdir()` のような、標準の Python 関数を使ってファイル群を指定するのなら、パス区切り文字 (path separator) をハードコーディングせず、以下のように可搬性のあるコードを書くよう注意すべきです:

```
glob.glob(os.path.join('mydir', 'subdir', '*.html'))
os.listdir(os.path.join('mydir', 'subdir'))
```

3.1 パッケージを全て列挙する

packages オプションは、 packages リスト中で指定されている各々のパッケージについて、パッケージ内に見つかった全ての pure Python モジュールを処理 (ビルド、配布、インストール、等) するよう Distutils に指

示します。このオプションを指定するためには、当然のことながら各パッケージ名はファイルシステム上のディレクトリ名と何らかの対応付けができなければなりません。デフォルトで使われる対応関係はきわめてはっきりしたものです。すなわち、パッケージ `distutils` が配布物ルートディレクトリからの相対パス 'distutils' で表されるディレクトリ中にあるというものです。つまり、setup スクリプト中で `packages = ['foo']` と指定したら、スクリプトの置かれたディレクトリからの相対パスで 'foo/__init__.py' を探し出せると Distutils に確約したことになります。この約束を裏切ると Distutils は警告を出しますが、そのまま壊れたパッケージの処理を継続します。

ソースコードディレクトリの配置について違った規約を使っていても、まったく問題はありません: 単に package_dir オプションを指定して、 Distutils に自分の規約を教えればよいのです。例えば、全ての Python ソースコードを 'lib' 下に置いて、"ルートパッケージ" 内のモジュール (つまり、どのパッケージにも入っていないモジュール) を 'lib' 内に入れ、`foo` パッケージを 'lib/foo' に入れる、といった具合にしたいのなら、

```
package_dir = {'': 'lib'}
```

を setup スクリプト内に入れます。辞書内のキーはパッケージ名で、空のパッケージ名はルートパッケージを表します。キーに対応する値はルートパッケージからの相対ディレクトリ名です、この場合、`packages = ['foo']` を指定すれば、'lib/foo/__init__.py' が存在すると Distutils に確約したことになります。

もう一つの規約のあり方は `foo` パッケージを 'lib' に置き換え、`foo.bar` パッケージが 'lib/bar' にある、などとするものです。このような規約は、 setup スクリプトでは

```
package_dir = {'foo': 'lib'}
```

のように書きます。package_dir 辞書に *package*`:` *dir* のようなエントリがあると、*package* の下にある全てのパッケージに対してこの規則が暗黙のうちに適用され、その結果 `foo.bar` の場合が自動的に処理されます。この例では、`packages = ['foo', 'foo.bar']` は、Distutils に 'lib/__init__.py' と 'lib/bar/__init__.py' を探すように指示します。(package_dir は再帰的に適用されますが、この場合 packages の下にある全てのパッケージを明示的に指定しなければならないことを心に留めておいてください: Distutils は '__init__.py' を持つディレクトリをソースツリーから再帰的にさがしたりは **しません** 。)

3.2 個々のモジュールを列挙する

小さなモジュール配布物の場合、パッケージを列挙するよりも、全てのモジュールを列挙するほうがよいと思うかもしれません — 特に、単一のモジュールが "ルートパッケージ" にインストールされる (すなわち、パッケージは全くない) ような場合がそうです。この最も単純なケースは 2.1 で示しました; ここではもうちょっと入り組んだ例を示します:

```
py_modules = ['mod1', 'pkg.mod2']
```

ここでは二つのモジュールについて述べていて、一方は "ルート" パッケージに入り、他方は `pkg` パッケージに入ります。ここでも、デフォルトのパッケージ/ディレクトリのレイアウトは、二つのモジュールが 'mod1.py' と 'pkg/mod2.py' にあり、'pkg/__init__.py' が存在することを暗示しています。また、パッケージ/ディレクトリの対応関係は package_dir オプションでも上書きできます。

3.3 拡張モジュールについて記述する

pure Python モジュールを書くより Python 拡張モジュールを書く方がちょっとだけ複雑なように、 Distutils での拡張モジュールに関する記述もちょっと複雑です。pure モジュールと違い、単にモジュールやパッケージを列挙して、Distutils が正しいファイルを見つけてくれると期待するだけでは十分ではありません; 拡張モジュールの名前、ソースコードファイル (群) 、そして何らかのコンパイル/リンクに関する必要事項 (include ディレクトリ、リンクすべきライブラリ、等) を指定しなければなりません。

こうした指定は全て、 `setup()` の別のキーワード引数、 extensions オプションを介して行えます。extensions は、 `Extensions` インスタンスからなるただのリストで、各インスタンスに一個の拡張モジュールを記述するようになっています。仮に、'foo.c' で実装された拡張モジュール `foo` が、配布物に一つだけ入ってるとします。コンパイラ/リンカに他の情報を与える必要がない場合、この拡張モジュールのための記述はきわめて単純です:

```
Extension("foo", ["foo.c"])
```

`Extension` クラスは、 `setup()` によって、`distutils.core` から import されます。従って、拡張モジュールが一つだけ入っていて、他には何も入っていないモジュール配布物を作成するための setup スクリプトは、以下のようになるでしょう:

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

`Explained` クラス (実質的には、`Explained` クラスの根底にある `build_ext` コマンドで実装されている、拡張モジュールをビルドする機構) は、Python 拡張モジュールをきわめて柔軟に記述できるようなサポートを提供しています。これについては後の節で説明します。

拡張モジュールの名前とパッケージ

`Extension` クラスのコンストラクタに与える最初の引数は、常に拡張モジュールの名前にします。これにはパッケージ名も含めます。例えば、

```
Extension("foo", ["src/foo1.c", "src/foo2.c"]p)
```

とすると、拡張モジュールをルートパッケージに置くことになります。一方、

```
Extension("pkg.foo", ["src/foo1.c", "src/foo2.c"])
```

は、同じ拡張モジュールを `pkg` パッケージの下に置くよう記述しています。ソースコードファイルと、作成されるオブジェクトコードはどちらの場合でも同じです; 作成された拡張モジュールがファイルシステム上のどこに置かれるか (すなわち Python の名前空間上のどこに置かれるか) が違うにすぎません。

同じパッケージ内に (または、同じ基底パッケージ下に) いくつもの拡張モジュールがある場合、ext_package キーワード引数を `setup()` に指定します。例えば、

```
setup(...
      ext_package="pkg",
      ext_modules=[Extension("foo", ["foo.c"]),
                   Extension("subpkg.bar", ["bar.c"])]
     )
```

とすると、 'foo.c' をコンパイルして `pkg.foo` にし、'bar.c' をコンパイルして `pkg.subpkg.bar` にします。

拡張モジュールのソースファイル

`Extension` コンストラクタの二番目の引数は、ソースファイルのリストです。 Distutils は現在のところ、C、C++、そして Objective-C の拡張しかサポートしていないので、引数は通常 C/C++/Objective-C ソースコードファイルになります。 (C++ソースコードファイルを区別できるよう、正しいファイル拡張子を使ってください: '.cc' や '.cpp' にすれば、 UNIX と Windows 用の双方のコンパイラで認識されるようです。)

ただし、SWIG インタフェース ('.i') ファイルはリストに含められます; `build_ext` コマンドは、 SWIG で書かれた拡張パッケージをどう扱えばよいか心得ています: `build_ext` は、インタフェースファイルを SWIG にかけ、得られた C/C++ ファイルをコンパイルして拡張モジュールを生成します。

プラットフォームによっては、コンパイラで処理され、拡張モジュールに取り込まれるような非ソースコードファイルを含められます。非ソースコードファイルとは、現状では Visual C++向けの Windows メッセージテキスト ('.mc') ファイルや、リソース定義 ('.rc') ファイルを指します。これらのファイルはバイナ

リリソース (‘.res’) ファイルにコンパイルされ、実行ファイルにリンクされます。

プリプロセッサオプション

Extension には三種類のオプション引数: include_dirs, define_macros, そして undef_macros があり、検索対象にするインクルードディレクトリを指定したり、プリプロセッサマクロを定義 (define)/定義解除 (undefine) したりする必要があるとき役立ちます。

例えば、拡張モジュールが配布物ルート下の ‘include’ ディレクトリにあるヘッダファイルを必要とするときには、include_dirs オプションを使います:

```
Extension("foo", ["foo.c"], include_dirs=["include"])
```

ここには絶対パスも指定できます; 例えば、自分の拡張モジュールが、‘/usr’ の下に X11R6 をインストールした UNIX システムだけでビルドされると知っていれば、

```
Extension("foo", ["foo.c"], include_dirs=["/usr/include/X11"])
```

のように書けます。

自分のコードを配布する際には、このような可搬性のない使い方は避けるべきです: おそらく、C のコードを

```
#include <X11/Xlib.h>
```

のように書いた方がましでしょう。

他の Python 拡張モジュール由来のヘッダを include する必要があるなら、Distutils の install_header コマンドが一貫した方法でヘッダファイルをインストールするという事実を活用できます。例えば、 Numerical Python のヘッダファイルは、 (標準的な Unix がインストールされた環境では) ‘/usr/local/include/python1.5/Numerical’ にインストールされます。(実際の場所は、プラットフォームやどの Python をインストールしたかで異なります。) Python の include ディレクトリ — 今の例では ‘/usr/local/include/python1.5’ — は、 Python 拡張モジュールをビルドする際に常にヘッダファイル検索パスに取り込まれるので、C コードを書く上でもっともよいアプローチは、

```
#include <Numerical/arrayobject.h>
```

となります。

‘Numerical’ インクルードディレクトリ自体をヘッダ検索パスに置きたいのなら、このディレクトリを Distutils の sysconfig モジュールを使って見つけさせられます:

```
from distutils.sysconfig import get_python_inc
incdir = os.path.join(get_python_inc(plat_specific=1), "Numerical")
setup(...,
      Extension(..., include_dirs=[incdir]))
```

この書き方も可搬性はあります — プラットフォームに関わらず、どんな Python がインストールされていても動作します — が、単に実践的な書き方で C コードを書く方が簡単でしょう。

define_macros および undef_macros オプションを使って、プリプロセッサマクロを定義 (define) したり、定義解除 (undefine) したりもできます。define_macros はタプル (name, value) からなるリストを引数にとります。name は定義したいマクロの名前 (文字列) で、value はその値です: value は文字列か None になります。(マクロ FOO を None にすると、C ソースコード内で #define FOO と書いたのと同じになります: こう書くと、ほとんどのコンパイラは FOO を文字列 1 に設定します。) undef_macros には、定義解除したいマクロ名からなるリストを指定します。

例えば、以下の指定:

```
Extension(...,
          define_macros=[('NDEBUG', '1'),
                         ('HAVE_STRFTIME', None)],
          undef_macros=['HAVE_FOO', 'HAVE_BAR'])
```

は、全ての C ソースコードファイルの先頭に、以下のマクロ:

```
#define NDEBUG 1
#define HAVE_STRFTIME
#undef HAVE_FOO
#undef HAVE_BAR
```

があるのと同じになります。

ライブラリオプション

拡張モジュールをビルドする際にリンクするライブラリや、ライブラリを検索するディレクトリも指定できます。`libraries` はリンクするライブラリのリストで、`library_dirs` はリンク時にライブラリを検索するディレクトリのリストです。また、`runtime_library_dirs` は、実行時に共有ライブラリ (動的にロードされるライブラリ) を検索するディレクトリのリストです。

例えば、ビルド対象システムの標準ライブラリ検索パスにあることが分かっているライブラリをリンクする時には、以下のようにします。

```
Extension(...,
          libraries=["gdbm", "readline"])
```

非標準のパス上にあるライブラリをリンクしたいなら、その場所を `library_dirs` に入れておかなければなりません:

```
Extension(...,
          library_dirs=["/usr/X11R6/lib"],
          libraries=["X11", "Xt"])
```

(繰り返しになりますが、この手の可搬性のない書き方は、コードを配布するのが目的なら避けるべきです。)

その他の操作

他にもいくつかオプションがあり、特殊な状況を扱うために使います。

extra_objects オプションには、リンカに渡すオブジェクトファイルのリストを指定します。ファイル名には拡張子をつけてはならず、コンパイラで使われているデフォルトの拡張子が使われます。

extra_compile_args および extra_link_args には、それぞれコンパイラとリンカに渡す追加のコマンドライン引数を指定します。

export_symbols は Windows でのみ意味があります。このオプションには、公開 (export) する (関数や変数の) シンボルのリストを入れられます。コンパイルして拡張モジュールをビルドする際には、このオプションは不要です: Distutils は公開するシンボルを自動的に `initmodule` に渡すからです。

3.4 スクリプトをインストールする

ここまでは、スクリプトから import され、それ自体では実行されないような pure Python モジュールおよび非 pure Python モジュールについて扱ってきました。

スクリプトとは、Python ソースコードを含むファイルで、コマンドラインから実行できるよう作られているものです。スクリプトは Distutils に複雑なことを一切させません。唯一の気の利いた機能は、スクリ

プトの最初の行が `#!` で始まっていて、 “python” という単語が入っていた場合、Distutils は最初の行を現在使っているインタプリタを参照するよう置き換えます。

scripts オプションには、単に上で述べた方法で取り扱うべきファイルのリストを指定するだけです。PyXML の setup スクリプトを例に示します:

```
setup (...
       scripts = ['scripts/xmlproc_parse', 'scripts/xmlproc_val']
      )
```

3.5 追加のファイルをインストールする

data_files オプションを使うと、モジュール配布物で必要な追加のファイル: 設定ファイル、メッセージカタログ、データファイル、その他これまで述べてきたカテゴリに収まらない全てのファイルを指定できます。

data_files には、(*directory*, *files*) のペアを以下のように指定します:

```
setup(...
      data_files=[('bitmaps', ['bm/b1.gif', 'bm/b2.gif']),
                  ('config', ['cfg/data.cfg']),
                  ('/etc/init.d', ['init-script'])]
     )
```

データファイルのインストール先ディレクトリ名は指定できますが、データファイル自体の名前の変更はできないので注意してください。

各々の (*directory*, *files*) ペアには、インストール先のディレクトリ名と、そのディレクトリにインストールしたいファイルを指定します。*directory* が相対パスの場合、インストールプレフィクス (installation prefix、pure Python パッケージなら `sys.prefix` 、拡張モジュールの入ったパッケージなら `sys.exec_prefix`) からの相対パスと解釈されます。*files* 内の各ファイル名は、パッケージソースコード配布物の最上階層の、‘setup.py’ のあるディレクトリからの相対パスと解釈されます。*files* に書かれたディレクトリ情報は、ファイルを最終的にどこにインストールするかを決めるときには使われません; ファイルの名前だけが使われます。

data_files オプションは、ターゲットディレクトリを指定せずに、単にファイルの列を指定できます。しかし、このやり方は推奨されておらず、指定すると `install` コマンドが警告を出力します。ターゲットディレクトリにデータファイルを直接インストールしたいなら、ディレクトリ名として空文字列を指定してください。

3.6 追加のメタデータ

setup スクリプトには、名前やバージョンにとどまらず、その他のメタデータを含められます。以下のような情報を含められます:

メタデータ	説明	値	注記
`name`	パッケージの名前	短い文字列	(1)
`version`	リリースのバージョン	短い文字列	(1)(2)
`author`	パッケージ作者の名前	短い文字列	(3)
`author_email`	パッケージ作者の電子メールアドレス	電子メールアドレス	(3)
`maintainer`	パッケージメンテナンス担当者の名前	短い文字列	(3)
`maintainer_email`	パッケージメンテナンス担当者の電子メールアドレス	電子メールアドレス	(3)
`url`	パッケージのホームページ	URL	(1)
`description`	パッケージについての簡潔な概要説明	短い文字列	
`long_description`	パッケージについての詳細な説明	長い文字列	
`download_url`	パッケージをダウンロードできる場所	URL	(4)
`classifiers`	Trove 分類語	文字列からなるリスト	(4)

注記:

(1) 必須のフィールドです。

(2) バージョン番号は *major.minor*[*.patch*[*.sub*]] の形式をとるよう奨めます。

(3) 作者かメンテナのどちらかは必ず区別してください。

(4) これらのフィールドは、2.2.3 および 2.3 より以前のバージョンの Python でも互換性を持たせたい場合には指定してはなりません。リストは PyPI ウェブサイト にあります。

「短い文字列」 200 文字以内の一行のテキスト。

「長い文字列」 複数行からなり、ReStructuredText 形式で書かれたプレーンテキスト (`http://docutils.sf.net/` を参照してください)。

「文字列のリスト」 下記を参照してください。

これらの文字列はいずれも Unicode であってはなりません。

バージョン情報のコード化は、それ自体が一つのアートです。Python のパッケージは一般的に、*major.minor*[*.patch*][*sub*] というバージョン表記に従います。メジャー (major) 番号は最初は 0 で、これはソフトウェアが実験的リリースにあることを示します。メジャー番号は、パッケージが主要な開発目標を達成したとき、それを示すために加算されてゆきます。マイナー (minor) 番号は、パッケージに重要な新機能が追加されたときに加算されてゆきます。パッチ (patch) 番号は、バグフィクス版のリリースが作成されたときに加算されます。末尾にバージョン情報が追加され、サブリリースを示すこともあります。これは "a1,a2,...,aN" (アルファリリースの場合で、機能や API が変更されているとき)、 "b1,b2,...,bN" (ベータリリースの場合で、バグフィクスのみのとき) 、そして "pr1,pr2,...,prN" (プレリリースの最終段階で、リリーステストのとき) になります。以下に例を示します:

0.1.0 パッケージの最初の実験的なリリース

1.0.1a2 1.0 の最初のパッチバージョンに対する、2 回目のアルファリリース

classifiers は、 Python のリスト型で指定します:

```
setup(...
      classifiers = [
          'Development Status :: 4 - Beta',
          'Environment :: Console',
          'Environment :: Web Environment',
          'Intended Audience :: End Users/Desktop',
          'Intended Audience :: Developers',
          'Intended Audience :: System Administrators',
          'License :: OSI Approved :: Python Software Foundation License',
          'Operating System :: MacOS :: MacOS X',
          'Operating System :: Microsoft :: Windows',
          'Operating System :: POSIX',
          'Programming Language :: Python',
          'Topic :: Communications :: Email',
          'Topic :: Office/Business',
          'Topic :: Software Development :: Bug Tracking',
          ],
      )
```

'setup.py' に classifiers を入れておき、なおかつ 2.2.3 よりも以前のバージョンの Python と後方互換性を保ちたいなら、'setup.py' 中で `setup()` を呼び出す前に、以下のコードを入れます。

```
# patch distutils if it can't cope with the "classifiers" or
# "download_url" keywords
if sys.version < '2.2.3':
    from distutils.dist import DistributionMetadata
    DistributionMetadata.classifiers = None
    DistributionMetadata.download_url = None
```

3.7 setup スクリプトをデバッグする

setup スクリプトのどこかがまずいと、開発者の思い通りに動作してくれません。

Distutils は setup 実行時の全ての例外を捉えて、簡単なエラーメッセージを出力してからスクリプトを終了します。このような仕様にしているのは、Python にあまり詳しくない管理者がパッケージをインストールする際に混乱しなくてすむようにするためです。もし Distutils のはらわた深くからトレースバックした長大なメッセージを見たら、管理者はきっと Python のインストール自体がおかしくなっているのだと勘違いして、トレースバックを最後まで読み進んで実はファイルパーミッションの問題だったと気づいたりはしないでしょう。

しかし逆に、この仕様は開発者にとってはうまくいかない理由を見つける役には立ちません。そこで、DISTUTILS_DEBUG 環境変数を空文字以外の何らかの値に設定しておけば、 Distutils が何を実行しているか詳しい情報を出力し、例外が発生した場合には完全なトレースバックを出力するようにできます。

4 setup 設定ファイル (setup configuration file) を書く

時に、配布物をビルドする際に必要な全ての設定を**あらかじめ** 書ききれない状況が起きます: 例えば、ビルドを進めるために、ユーザに関する情報や、ユーザのシステムに関する情報を必要とするかもしれません。こうした情報が単純 — C ヘッダファイルやライブラリを検索するディレクトリのリストのように — であるかぎり、ユーザに設定ファイル (configuration file) 'setup.cfg' を提供して編集してもらうのが、安上がりで簡単な特定方法になります。設定ファイルはまた、あらゆるコマンドにおけるオプションにデフォルト値を与えておき、インストール作業者がコマンドライン上や設定ファイルの編集でデフォルト設定を上書きできるようにします。

setup 設定ファイルは setup スクリプト —理想的にはインストール作業者から見えないもの [1] —と、作者の手を離れて、全てインストール作業者次第となる setup スクリプトのコマンドライン引数との間を橋渡

[1] Distutils が自動設定機能 (auto-configuration) をサポートするまで、おそらくこの理想状態を達成することはないでしょう

しする中間層として有効です。実際、‘setup.cfg’ (と、ターゲットシステム上にある、その他の Distutils 設定ファイル) は、 setup スクリプトの内容より後で、かつコマンドラインで上書きする前に処理されます。この仕様の結果、いくつかの利点が生まれます:

- インストール作業者は、作者が ‘setup.py’ に設定した項目のいくつかを ‘setup.cfg’ を変更して上書きできます。
- ‘setu.py’ では簡単に設定できないような、標準でないオプションのデフォルト値を設定できます。
- インストール作業者は、‘setup.cfg’ に書かれたどんな設定も ‘setup.py’ のコマンドラインオプションで上書きできます。

設定ファイルの基本的な構文は簡単なものです:

```
[command]
option=value
...
```

ここで、 *command* は Distutils コマンドのうちの一つ (例えば `build_py`, `install`) で、*option* はそのコマンドでサポートされているオプションのうちの一つです。各コマンドには任意の数のオプションを設定でき、一つの設定ファイル中には任意の数のコマンドセクションを収められます。空白行は無視されます、‘#’ 文字で開始して行末まで続くコメントも同様に無視されます。長いオプション設定値は、継続行をインデントするだけで複数行にわたって記述できます。

あるコマンドがサポートしているオプションのリストは、**--help** オプションで調べられます。例えば以下のように。

```
> python setup.py --help build_ext
[...]
Options for 'build_ext' command:
  --build-lib (-b)     directory for compiled extension modules
  --build-temp (-t)    directory for temporary files (build by-products)
  --inplace (-i)       ignore build-lib and put compiled extensions into the
                       source directory alongside your pure Python modules
  --include-dirs (-I)  list of directories to search for header files
  --define (-D)        C preprocessor macros to define
  --undef (-U)         C preprocessor macros to undefine
[...]
```

コマンドライン上で **--foo-bar** と綴るオプションは、設定ファイル上では foo_bar と綴るので注意してください。

例えば、拡張モジュールを “インプレース (in-place)” でビルドしたいとします — すなわち、`pkg.ext` という拡張モジュールを持っていて、コンパイル済みの拡張モジュールファイル (例えば UNIX では ‘ext.so’) を pure Python モジュール `pkg.mod1` および `pkg.mod2` と同じソースディレクトリに置きたいとします。こんなときには、**--inplace** を使えば、確実にビルドを行えます。

```
python setup.py build_ext --inplace
```

しかし、この操作では、常に `build_ext` を明示的に指定しなければならず、 **--inplace** オプションを忘れずに与えなければなりません。こうした設定を “設定しっ放しにする” 簡単な方法は、‘setup.cfg’ に書いておくやり方で、設定ファイルは以下のようになります:

```
[build_ext]
inplace=1
```

この設定は、明示的に `build_ext` を指定するかどうかに関わらず、モジュール配布物の全てのビルドに影響します。ソース配布物に ‘setup.cfg’ を含めると、エンドユーザの手で行われるビルドにも影響します — このオプションの例に関しては ‘setup.cfg’ を含めるのはおそらくよくないアイデアでしょう。という

のは、拡張モジュールをインプレースでビルドすると常にインストールしたモジュール配布物を壊してしまうからです。とはいえ、ある特定の状況では、モジュールをインストールディレクトリの下に正しく構築できるので、機能としては有用だと考えられます。(ただ、インストールディレクトリ上でのビルドを想定するような拡張モジュールの配布は、ほとんどの場合よくない考え方です。)

もう一つ、例があります: コマンドによっては、実行時にほとんど変更されないたくさんのオプションがあります; 例えば、`bdist_rpm` には、RPM 配布物を作成する際に、“spec” ファイルを作成するために必要な情報を全て与えなければなりません。この情報には setup スクリプトから与えるものもあり、(インストールされるファイルのリストのように) Distutils が自動的に生成するものもあります。しかし、こうした情報の中には `bdist_rpm` のオプションとして与えるものがあり、毎回実行するごとにコマンドライン上で指定するのが面倒です。そこで、以下のような内容が Distutils 自体の ‘setup.cfg’ には入っています:

```
[bdist_rpm]
release = 1
packager = Greg Ward <gward@python.net>
doc_files = CHANGES.txt
            README.txt
            USAGE.txt
            doc/
            examples/
```

doc_files オプションは、単に空白で区切られた文字列で、ここでは可読性のために複数行をまたぐようにしています。

参考資料:

Installing Python Modules
(`../inst/config-syntax.html`)
設定ファイルに関する詳細情報は、システム管理者向けのこのマニュアルにあります。

5 ソースコード配布物を作成する

2.1 節で示したように、ソースコード配布物を作成するには `sdist` コマンドを使います。最も単純な例では、

```
python setup.py sdist
```

のようにします (ここでは、`sdist` に関するオプションを setup スクリプトや設定ファイル中で行っていないものと仮定します)。`sdist` は、現在のプラットフォームでのデフォルトのアーカイブ形式でアーカイブを生成します。デフォルトの形式は UNIX では gzip で圧縮された tar ファイル形式 (‘.tar.gz’) で、Windows では ZIP 形式です。

--formats オプションを使えば、好きなだけ圧縮形式を指定できます。例えば:

```
python setup.py sdist --formats=gztar,zip
```

は、gzip された tarball と zip ファイルを作成します。利用可能な形式は以下の通りです:

形式	説明	注記
`zip`	zip ファイル (‘.zip’)	(1),(3)
`gztar`	gzip 圧縮された tar ファイル (‘.tar.gz’)	(2),(4)
`bztar`	bzip2 圧縮された tar ファイル (‘.tar.bz2’)	(4)
`ztar`	compress 圧縮された tar ファイル (‘.tar.Z’)	(4)
`tar`	tar ファイル (‘.tar’)	(4)

注記:

(1) Windows でのデフォルトです

(2) UNIX でのデフォルトです

(3) 外部ユーティリティの **zip** か、`zipfile` モジュールが必要です (Python 1.6 からは 標準ライブラリになっています)

(4) 外部ユーティリティ: **tar** 、場合によっては **gzip**、 **bzip2**、または **compress** も必要です

5.1 配布するファイルを指定する

明確なファイルのリスト (またはファイルリストを生成する方法) を明示的に与えなかった場合、`sdist` コマンドはソース配布物に以下のような最小のデフォルトのセットを含めます:

- py_modules と packages オプションに指定された Python ソースファイル全て
- ext_modules や libraries オプションに記載された C ソースファイル
- scripts で指定したスクリプト
- テストスクリプトと思しきファイル全て: 'test/test*.py' (現状では、Distutils はテストスクリプトをただソース配布物に含めるだけですが、将来は Python モジュール配布物に対するテスト標準ができるかもしれません)
- 'README.txt' (または 'README')、 'setup.py' (または setup スクリプトにしているもの) 、および 'setup.cfg'

上記のセットで十分なこともありますが、大抵他のファイルを配布物に含めたいと思うでしょう。普通は、 'MANIFEST.in' と呼ばれる**マニフェストテンプレート** *(manifest template)* を使ってこれを行います。マニフェストテンプレートは、ソース配布物に含めるファイルの正確なリストであるマニフェストファイル 'MANIFEST' をどうやって作成するか指示しているリストです。`sdist` コマンドはこのテンプレートを処理し、書かれた指示とファイルシステム上に見つかったファイルに基づいてマニフェストファイルを作成します。

自分用のマニフェストファイルを書きたいなら、その形式は簡単です: 一行あたり一つの通常ファイル (または通常ファイルに対するシンボリックリンク) だけを書きます。自分で 'MANIFEST' を提供する場合、全てを自分で指定しなければなりません: ただし、上で説明したデフォルトのファイルセットは、この中には含まれません。

マニフェストテンプレートには一行あたり一つのコマンドがあります。各コマンドはソース配布物に入れたり配布物から除外したりするファイルのセットを指定します。例えば、Distutils 自体のマニフェストテンプレートの話に戻ると:

```
include *.txt
recursive-include examples *.txt *.py
prune examples/sample?/build
```

各行はかなり明確に意味を取れるはずです: 上の指定では、`*.txt` にマッチする配布物ルート下の全てのファイル、'examples' ディレクトリ下にある `*.txt` か `*.py` にマッチする全てのファイルを含め、`examples/sample?/build` にマッチする全てのファイルを除外します。これらの処理はすべて、標準的に含められるファイルセットの評価よりも**後に** 行われるので、マニフェストテンプレートに明示的に指示をしておけば、標準セット中のファイルも除外できます。(**--no-defaults** オプションを設定して、標準セット自体を無効にもできます。) 他にも、このマニフェストテンプレート記述のためのミニ言語にはいくつかのコマンドがあります: 9.2 節を参照してください。

マニフェストテンプレート中のコマンドの順番には意味があります; 初期状態では、上で述べたようなデフォルトのファイルがあり、テンプレート中の各コマンドによって、逐次ファイルを追加したり除去したりしていいます。マニフェストテンプレートを完全に処理し終えたら、ソース配布物中に含めるべきでない以下のファイルをリストから除去します:

- Distutls の "build" (デフォルトの名前は 'build') ツリー下にある全てのファイル
- 'RCS' 'CVS' といった名前のディレクトリ下にある全てのファイル

こうして完全なファイルのリストができ、後で参照するためにマニフェストに書き込まれます。この内容は、ソース配布物のアーカイブを作成する際に使われます。

含めるファイルのデフォルトセットは **--no-defaults** で無効化でき、標準で除外するセットは **--no-prune** で無効化できます。

Distutils 自体のマニフェストテンプレートから、`sdist` コマンドがどのようにして Distutils ソース配布物に含めるファイルのリストを作成するか見てみましょう:

1. 'distutils' ディレクトリ、および 'distutils/command' サブディレクトリの下にある全ての Python ソースファイルを含めます (これらの二つのディレクトリが、setup スクリプト下の packages オプションに記載されているからです — 3 を参照してください)
2. 'README.txt', 'setup.py', および 'setup.cfg' (標準のファイルセット) を含めます
3. 'test/test*.py' (標準のファイルセット) を含めます
4. 配布物ルート下の '*.txt' を含めます (この処理で、'README.txt' がもう一度見つかりますが、こうした冗長性は後で刈り取られます)
5. 'examples' 下にあるサブツリー内で '*.txt' または '*.py' にマッチする全てのファイルを含めます
6. ディレクトリ名が 'examples/sample?/build' にマッチするディレクトリ以下のサブツリー内にあるファイル全てを除外します— この操作によって、上の二つのステップでリストに含められたファイルが除外されることがあるので、マニフェストテンプレート内では `recursive-include` コマンドの後に `prune` コマンドを持ってくることが重要です
7. 'build' ツリー全体、および 'RCS' と 'CVS' ディレクトリ全てを除外します

setup スクリプトと同様、マニフェストテンプレート中のディレクトリ名は常にスラッシュ区切りで表記します; Distutils は、こうしたディレクトリ名を注意深くプラットフォームでの標準的な表現に変換します。このため、マニフェストテンプレートは複数のオペレーティングシステムにわたって可搬性を持ちます。

5.2 マニフェスト (manifest) 関連のオプション

`sdist` コマンドが通常行う処理の流れは、以下のようになっています:

- マニフェストファイル 'MANIFEST' が存在しなければ、'MANIFEST.in' を読み込んでマニフェストファイルを作成します
- 'MANIFEST' も 'MANIFEST.in' もなければ、デフォルトのファイルセットだけでできたマニフェストファイルを作成します
- 'MANIFEST.in' または ('setup.py') が 'MANIFEST' より新しければ、'MANIFEST.in' を読み込んで 'MANIFEST' を生成します
- (生成されたか、読み出された) 'MANIFEST' 内にあるファイルのリストを使ってソース配布物アーカイブを作成します

上の動作は二種類のオプションを使って修正できます。まず、標準の "include" および "exclude" セットを無効化するには**--no-defaults** および **--no-prune** を使います

第二に、マニフェストファイルの再生成を強制できます — 例えば、現在マニフェストテンプレート内に指定しているパターンにマッチするファイルやディレクトリを追加したり削除したりすると、マニフェストを再生成しなくてはなりません:

```
python setup.py sdist --force-manifest
```

また、単にマニフェストを (再) 生成したいだけで、ソース配布物は作成したくない場合があるかもしれません:

```
python setup.py sdist --manifest-only
```

--manifest-only を行うと、 **--force-manifest** を呼び出します。**-o** は **--manifest-only** のショートカット、**-f** は **--force-manifest** のショートカットです。

6　ビルド済み配布物を作成する

“ビルド済み配布物” とは、おそらく皆さんが通常 “バイナリパッケージ” とか “インストーラ” (背景にしている知識によって違います) と考えているものです。とはいえ、配布物が必然的にバイナリ形式になるわけではありません。配布物には、 Python ソースコード、かつ/またはバイトコードが入るからです; また、我々はパッケージという呼び方もしません。すでに Python の用語として使っているからです (また、 “インストーラ” という言葉は Windows 特有の用語です)

ビルド済み配布物は、モジュール配布物をインストール作業者にとってできるだけ簡単な状況にする方法です: ビルド済み配布物は、RPM ベースの Linux システムユーザにとってはバイナリ RPM 、Windows ユーザにとっては実行可能なインストーラ、 Debian ベースの Linux システムでは Debian パッケージ、などといった具合です。当然のことながら、一人の人間が世の中にある全てのプラットフォーム用にビルド済み配布物を作成できるわけではありません。そこで、Distutils の設計は。開発者が自分の専門分野 — コードを書き、ソース配布物を作成する — に集中できる一方で、**パッケージ作成者** *(packager)* と呼ばれる、開発者とエンドユーザとの中間に位置する人々がソースコード配布物を多くのプラットフォームにおけるビルド済み配布物に変換できるようになっています。

もちろん、モジュール開発者自身がパッケージ作成者かもしれません; また、パッケージを作成するのはオリジナルの作成者が利用できないプラットフォームにアクセスできるような “外部の” ボランティアかもしれませんし、ソース配布物を定期的に取り込んで、アクセスできるかぎりのプラットフォーム向けにビルド済み配布物を生成するソフトウェアかもしれません。作業を行うのが誰であれ、パッケージ作成者は setup スクリプトを利用し、 `bdist` コマンドファミリを使ってビルド済み配布物を作成します。

単純な例として、Distutils ソースツリーから以下のコマンドを実行したとします:

```
python setup.py bdist
```

すると、Distutils はモジュール配布物 (ここでは Distutils 自体) をビルドし、“偽の (fake)” インストールを (‘build’ ディレクトリで) 行います。そして現在のプラットフォームにおける標準の形式でビルド済み配布物を生成します。デフォルトのビルド済み形式とは、UNIX では “ダム (dumb)” の tar ファイルで、 Windows ではシンプルな実行形式のインストーラになります。(tar ファイルは、特定の場所に手作業で解凍しないと動作しないので、 “ダム: 賢くない” 形式とみなします。)

従って、UNIX システムで上記のコマンドを実行すると、‘Distutils-1.0.*plat*.tar.gz’ を作成します; この tarball を正しい場所で解凍すると、ちょうどソース配布物をダウンロードして `python setup.py install` を実行したのと同じように、正しい場所に Distutils がインストールされます。(“正しい場所 (right place)” とは、ファイルシステムのルート下か、Python の *prefix* ディレクトリ下で、これは `bdist_dumb` に指定するコマンドで変わります; デフォルトの設定では、*prefix* からの相対パスにインストールされるダム配布物が得られます。)

言うまでもなく、pure Python 配布物の場合なら、`python setup.py install` するのに比べて大して簡単になったとは言えません—しかし、非 pure 配布物で、コンパイルの必要な拡張モジュールを含む場合、拡張モジュールを利用できるか否かという大きな違いになりえます。また、 RPM パッケージや Windows 用の実行形式インストーラのような “スマートな” ビルド済み配布物を作成しておけば、たとえ拡張モジュールが一切入っていなくてもユーザにとっては便利になります。

`bdist` コマンドには、 **--formats** オプションがあります。これは `sdist` コマンドの場合に似ていて、生成したいビルド済み配布物の形式を選択できます: 例えば、

```
python setup.py bdist --format=zip
```

とすると、UNIX システムでは、‘Distutils-1.0.*plat*.zip’ を作成します— 先にも述べたように、Distutils をインストールするには、このアーカイブ形式をルートディレクトリ下で展開します。

ビルド済み配布物として利用できる形式を以下に示します:

形式	説明	注記
gztar	gzip 圧縮された tar ファイル (‘.tar.gz’)	(1),(3)
ztar	compress 圧縮された tar ファイル (‘.tar.Z’)	(3)
tar	tar ファイル (‘.tar’)	(3)
zip	zip ファイル (‘.zip’)	(4)
rpm	RPM 形式	(5)
pkgtool	Solaris **pkgtool** 形式	
sdux	HP-UX **swinstall** 形式	
wininst	Windows 用の自己展開形式 ZIP ファイル	(2),(4)

注記:

(1) UNIX でのデフォルト形式です

(2) Windows でのデフォルト形式です

(3) 外部ユーティリティが必要です: **tar** と、**gzip** または **bzip2** または **compress** のいずれか

(4) 外部ユーティリティの **zip** か、`zipfile` モジュール (Python 1.6 からは標準 Python ライブラリの一部になっています) が必要です

(5) 外部ユーティリティの **rpm**、バージョン 3.0.4 以上が必要です (バージョンを調べるには、`rpm -version` とします)

`bdist` コマンドを使うとき、必ず **--formats** オプションを使わなければならないわけではありません; 自分の使いたい形式をダイレクトに実装しているコマンドも使えます。こうした `bdist` “サブコマンド (sub-command)” は、実際には類似のいくつかの形式を生成できます; 例えば、`bdist_dumb` コマンドは、全ての “ダム” アーカイブ形式 (`tar`, `ztar`, `gztar`, および `zip`) を作成できますし、`bdist_rpm` はバイナリ RPM とソース RPM の両方を生成できます。`bdist` サブコマンドと、それぞれが生成する形式を以下に示します:

コマンド	形式
bdist_dumb	tar, ztar, gztar, zip
bdist_rpm	rpm, srpm
bdist_wininst	wininst

`bdist_*` コマンドについては、以下の節で詳しく述べます。

6.1 ダム形式のビルド済み配布物を作成する

6.2 RPM パッケージを作成する

RPM 形式は、Red Hat, SuSE, Mandrake といった、多くの一般的な Linux ディストリビューションで使われています。普段使っているのがこれらの環境のいずれか (またはその他の RPM ベースの Linux ディストリビューション) なら、同じディストリビューションを使っている他のユーザ用に RPM パッケージを作成するのはとるに足らないことでしょう。一方、モジュール配布物の複雑さや、 Linux ディストリビューション間の違いにもよりますが、他の RPM ベースのディストリビューションでも動作するような RPM を作成できるかもしれません。

通常、モジュール配布物の RPM を作成するには、`bdist_rpm` コマンドを使います:

```
python setup.py bdist_rpm
```

あるいは、`bdist` コマンドを **--format** オプション付きで使います:

```
python setup.py bdist --formats=rpm
```

前者の場合、 RPM 特有のオプションを指定できます; 後者の場合、一度の実行で複数の形式を指定できます。両方同時にやりたければ、それぞれの形式について各コマンドごとにオプション付きで `bdist_*` コマンドを並べます:

```
python setup.py bdist_rpm --packager="John Doe <jdoe@python.net>" \
                bdist_wininst --target_version="2.0"
```

Distutils が setup スクリプトで制御されているのとほとんど同じく、RPM パッケージの作成は、‘.spec’ で制御されています。RPM の作成を簡便に解決するため、`bdist_rpm` コマンドでは通常、 setup スクリプトに与えた情報とコマンドライン、そして Distutils 設定ファイルに基づいて ‘.spec’ ファイルを作成します。‘.spec’ ファイルの様々なオプションやセクション情報は、以下のようにして setup スクリプトから取り出されます:

RPM ‘.spec’ ファイルのオプション またはセクション	**Distutils setup スクリプト内のオプション**
Name	name
Summary (preamble 内)	description
Version	version
Vendor	author と author_email, または maintainer と maintainer_email
Copyright	licence
Url	url
%description (セクション)	long_description

また、‘.spec’ ファイル内の多くのオプションは、 setup スクリプト中に対応するオプションがありません。これらのほとんどは、以下に示す `bdist_rpm` コマンドのオプションで扱えます:

RPM ‘.spec’ ファイルのオプション またはセクション	**`bdist_rpm` オプション**	**デフォルト値**
Release	release	“1”
Group	group	“Development/Libraries”
Vendor	vendor	(上記参照)
Packager	packager	(none)
Provides	provides	(none)
Requires	requires	(none)
Conflicts	conflicts	(none)
Obsoletes	obsoletes	(none)
Distribution	distribution_name	(none)
BuildRequires	build_requires	(none)
Icon	icon	(none)

言うまでもなく、こうしたオプションをコマンドラインで指定するのは面倒だし、エラーの元になりますから、普通は ‘setup.cfg’ に書いておくのがベストです — 4 節を参照してください。沢山の Python モジュール配布物を配布したりパッケージ化したりしているのなら、配布物全部に当てはまるオプションを個人用の Distutils 設定ファイル (‘~/.pydistutils.cfg’) に入れられます。

バイナリ形式の RPM パッケージを作成する際には三つの段階があり、Distutils はこれら全ての段階を自動的に処理します:

1. RPM パッケージの内容を記述する ‘.spec’ ファイルを作成します (‘.spec’ ファイルは setup スクリプトに似たファイルです; 実際、setup スクリプトのほとんどの情報が ‘.spec’ ファイルから引き揚げられます)
2. ソース RPM を作成します
3. “バイナリ (binary)” RPM を生成します (モジュール配布物に Python 拡張モジュールが入っているか否かで、バイナリコードが含まれることも含まれないこともあります)

通常、RPM は最後の二つのステップをまとめて行います; Distutils を使うと、普通は三つのステップ全てをまとめて行います。

望むなら、これらの三つのステップを分割できます。`bdist_rpm` コマンドに **--spec-only** を指定すれば、単に ‘.spec’ を作成して終了します; この場合、‘.spec’ ファイルは “配布物ディレクトリ (distribution directory)”— 通常は ‘dist/’ に作成されますが、 **--dist-dir** オプションで変更することもできます。(通常、

‘.spec’ ファイルは “ビルドツリー (build tree)”、すなわち `build_rpm` が作成する一時ディレクトリの中から引き揚げられます。)

自作の ‘.spec’ ファイルを **--spec-file** オプションで指定することもできます; **--spec-only** と併せて利用すれば、‘.spec’ ファイルを手作業でカスタマイズする機会が生まれます:

```
> python setup.py bdist_rpm --spec-only
# ... dist/FooBar-1.0.spec を編集
> python setup.py bdist_rpm --spec-file=dist/FooBar-1.0.spec
```

(とはいえ、‘.spec’ の内容をカスタマイズしたいのなら、標準の `bdist_rpm` を上書きして、自分の思い通りに ‘.spec’ ファイルを書かせる方がおそらくましでしょう。)

6.3 Windows インストーラを作成する

実行可能なインストーラは、Windows 環境ではごく自然なバイナリ配布形式です。インストーラは結構なグラフィカルユーザインタフェースを表示して、モジュール配布物に関するいくつかの情報を setup スクリプト内のメタデータから取り出して示し、ユーザがいくつかのオプションを選んだり、インストールを決行するか取りやめるか選んだりできるようにします。

メタデータは setup スクリプトから取り出されるので、Windows インストーラの作成は至って簡単で、以下を実行するだけです:

```
python setup.py bdist_wininst
```

あるいは、`bdist` コマンドを **--formats** オプション付きで実行します:

```
python setup.py bdist --formats=wininst
```

(pure Python モジュールとパッケージだけの入った) pure モジュール配布物の場合、作成されるインストーラは実行バージョンに依存しない形式になり、‘foo-1.0.win32.exe’ のような名前になります。pure モジュールの Windows インストーラは UNIX や MacOS といったプラットフォームでも作成できます。

非 pure 配布物の場合、拡張モジュールは Windows プラットフォーム上だけで作成でき、Python のバージョンに依存したインストーラになります。インストーラのファイル名もバージョン依存性を反映して、‘foo-1.0.win32-py2.0.exe’ のような形式になります。従って、サポートしたい全てのバージョンの Python に対して、別々のインストーラを作成しなければなりません。

インストーラは、ターゲットとなるシステムにインストールを実行した後、pure モジュールを通常 (normal) モードと最適化 (optimizing) モードでコンパイルしようと試みます。何らかの理由があってコンパイルさせたくなければ、`bdist_wininst` コマンドを**--no-target-compile** かつ/または**--no-target-optimize** オプション付きで実行します。

デフォルトでは、インストーラは実行時にクールな “Python Powered” ロゴを表示しますが、自作のビットマップ画像も指定できます。画像は Windows の ‘.bmp’ ファイル形式でなくてはならず、**--bitmap** オプションで指定します。

インストーラを起動すると、デスクトップの背景ウィンドウ上にでっかいタイトルも表示します。タイトルは配布物の名前とバージョン番号から作成します。**--title** オプションを使えば、タイトルを別のテキストに変更できます。

インストーラファイルは “配布物ディレクトリ (distribution directory)” — 通常は ‘dist/’ に作成されますが、**--dist-dir** オプションで指定することもできます。

インストール後実行スクリプト (postinstallation script)

Python 2.3 からは、インストール実行後スクリプトを**--install-script** オプションで指定できるようになりました。スクリプトはディレクトリを含まないベースネーム (basename) で指定しなければならず、スクリプトファイル名は setup 関数の scripts 引数中に挙げられていなければなりません。

指定したスクリプトは、インストール時、ターゲットとなるシステム上で全てのファイルがコピーされ

た後に実行されます。このとき argv[1] を “-install” に設定します。また、アンインストール時には、ファイルを削除する前に argv[1] を “-remove” に設定して実行します。

Windows インストーラでは、インストールスクリプトは埋め込みで実行され、全ての出力 (sys.stdout、sys.stderr) はバッファにリダイレクトされ、スクリプトの終了後に GUI 上に表示されます。

インストール後実行スクリプトでは、インストール/アンインストールのコンテキストで特に有用な機能をいくつか使えます。

```
directory_created(pathname)
file_created(pathname)
```

これらの関数は、インストール後実行スクリプトがディレクトリやファイルを作成した際に呼び出さなければなりません。この関数はアンインストーラに作成されたパス名を登録し、配布物をアンインストールする際にファイルが消されるようにします。安全を期すために、ディレクトリは空の時にのみ削除されます。

```
get_special_folder_path(csidl_string)
```

この関数は、「スタートメニュー」や「デスクトップ」といった、Windows における特殊なフォルダ位置を取得する際に使えます。この関数はフォルダのフルパスを返します。 ’csidl_string’ は以下の文字列のいずれかでなければなりません:

```
"CSIDL_APPDATA"

"CSIDL_COMMON_STARTMENU"
"CSIDL_STARTMENU"

"CSIDL_COMMON_DESKTOPDIRECTORY"
"CSIDL_DESKTOPDIRECTORY"

"CSIDL_COMMON_STARTUP"
"CSIDL_STARTUP"

"CSIDL_COMMON_PROGRAMS"
"CSIDL_PROGRAMS"

"CSIDL_FONTS"
```

該当するフォルダを取得できなかった場合、 OSError が送出されます。

どの種類のフォルダを取得できるかは、特定の Windows のバージョンごとに異なります。また、おそらく設定によっても異なるでしょう。詳細については、 `SHGetSpecialFolderPath` 関数に関する Microsoft の ドキュメントを参照してください。

```
create_shortcut(target, description, filename[, arguments[,
                workdir[, iconpath[, iconindex]]]])
```

この関数はショートカットを作成します。*target* はショートカットによって起動されるプログラムへのパスです。*description* はショートカットに対する説明です。*filename* はユーザから見えるショートカットの名前です。コマンドライン引数があれば、*arguments* に指定します。*workdir* はプログラムの作業ディレクトリです。*iconpath* はショートカットのためのアイコンが入ったファイルで、*iconindex* はファイル *iconpath* 中のアイコンへのインデクスです。これについても、詳しくは `IShellLink` インタフェースに関する Microsoft の ドキュメントを参照してください。

7　パッケージインデクスに登録する

Python パッケージインデクス (Python Package Index, PyPI) は、distutils でパッケージ化された配布物に関するメタデータを保持しています。配布物のメタデータをインデクスに提出するには、Distutils のコマンド `register` を使います。`register` は以下のように起動します:

```
python setup.py register
```

Distutils は以下のようなプロンプトを出します:

```
running register
We need to know who you are, so please choose either:
 1. use your existing login,
 2. register as a new user,
 3. have the server generate a new password for you (and email it to you), or
 4. quit
Your selection [default 1]:
```

注意: ユーザ名とパスワードをローカルの計算機に保存しておくと、このメニューは表示されません。

まだ PyPI に登録したことがなければ、まず登録する必要があります。この場合選択肢 2 番を選び、リクエストされた詳細情報を入力してゆきます。詳細情報を提出し終えると、登録情報の承認を行うためのメールを受け取るはずです。

すでに登録を行ったことがあれば、選択肢 1 を選べます。この選択肢を選ぶと、PyPI ユーザ名とパスワードを入力するよう促され、`register` がメタデータをインデクスに自動的に提出します。

配布物の様々なバージョンについて、好きなだけインデクスへの提出を行ってかまいません。特定のバージョンに関するメタデータを入れ替えたければ、再度提出を行えば、インデクス上のデータが更新されます。

PyPI は提出された配布物の (名前、バージョン) の各組み合わせについて記録を保持しています。ある配布物名について最初に情報を提出したユーザが、その配布物名のオーナ (owner) になります。オーナは `register` コマンドか、web インタフェースを介して変更を提出できます。オーナは他のユーザをオーナやメンテナとして指名できます。メンテナはパッケージ情報を編集できますが、他の人をオーナやメンテナに指名することはできません。

デフォルトでは、 PyPI はあるパッケージについて全てのバージョンを表示します。特定のバージョンを非表示にしたければ、パッケージの Hidden プロパティを yes に設定します。この値は web インタフェースで編集しなければなりません。

8　例

8.1　pure Python 配布物 (モジュール形式)

単に二つのモジュール、特定のパッケージに属しないモジュールを配布するだけなら、setup スクリプト中で py_modules オプションを使って個別に指定できます。

もっとも単純なケースでは、二つのファイル: setup スクリプト自体と、配布したい単一のモジュール、この例では 'foo.py' について考えなければなりません:

```
<root>/
        setup.py
        foo.py
```

(この節の全ての図において、 `<root>` は配布物ルートディレクトリを参照します。) この状況を扱うための最小の setup スクリプトは以下のようになります:

```
from distutils.core import setup
setup(name = "foo", version = "1.0",
      py_modules = ["foo"])
```

配布物の名前は name オプションで個々に指定し、配布されるモジュールの一つと配布物を同じ名前にする必要はないことに注意してください (とはいえ、この命名方法はよいならわしでしょう)。ただし、配布物名はファイル名を作成するときに使われるので、文字、数字、アンダースコア、ハイフンだけで構成しなければなりません。

py_modules はリストなので、もちろん複数のモジュールを指定できます。例えば、モジュール `foo` と `bar` を配布しようとしているのなら、 setup スクリプトは以下のようになります:

```
<root>/
        setup.py
        foo.py
        bar.py
```

また、セットアップスクリプトは以下のようになります.

```
from distutils.core import setup
setup(name = "foobar", version = "1.0",
      py_modules = ["foo", "bar"])
```

モジュールのソースファイルは他のディレクトリに置けますが、そうしなければならないようなモジュールを沢山持っているのなら、モジュールを個別に列挙するよりもパッケージを指定した方が簡単でしょう。

8.2 pure Python 配布物 (パッケージ形式)

二つ以上のモジュールを配布する場合、とりわけ二つのパッケージに分かれている場合、おそらく個々のモジュールよりもパッケージ全体を指定する方が簡単です。たとえモジュールがパッケージ内に入っていなくても状況は同じで、その場合はルートパッケージにモジュールが入っていると Distutils に教えることができ、他のパッケージと同様にうまく処理されます (ただし、'__init__.py' があってはなりません)。

最後の例で挙げた setup スクリプトは、

```
from distutils.core import setup
setup(name = "foobar", version = "1.0",
      packages = [""])
```

のようにも書けます (空文字はルートパッケージを意味します)

これら二つのファイルをサブディレクトリ下に移動しておいて、インストール先はルートパッケージのままにしておきたい、例えば:

```
<root>/
        setup.py
        src/      foo.py
                  bar.py
```

のような場合には、パッケージ名にはルートパッケージをそのまま指定しておきますが、ルートパッケージに置くソースファイルがどこにあるかを Distutils に教えなければなりません:

```
from distutils.core import setup
setup(name = "foobar", version = "1.0",
      package_dir = {"": "src"},
      packages = [""])
```

もっと典型的なケースでは、複数のモジュールを同じパッケージ (またはサブパッケージ) に入れて配布しようと思うでしょう。例えば、`foo` と `bar` モジュールがパッケージ `foobar` に属する場合、ソースツリーをレイアウトする一案として、以下が考えられます。

```
<root>/
        setup.py
        foobar/
                __init__.py
                foo.py
                bar.py
```

実際、 Distutils ではこれをデフォルトのレイアウトとして想定していて、setup スクリプトを書く際にも最小限の作業しか必要ありません:

```
from distutils.core import setup
setup(name = "foobar", version = "1.0",
      packages = ["foobar"])
```

モジュールを入れるディレクトリをパッケージの名前にしたくない場合、ここでも package_dir オプションを使う必要があります。例えば、パッケージ `foobar` のモジュールが 'src' に入っているとします:

```
<root>/
        setup.py
        src/
                __init__.py
                foo.py
                bar.py
```

適切な setup スクリプトは、

```
from distutils.core import setup
setup(name = "foobar", version = "1.0",
      package_dir = {"foobar" : "src"},
      packages = ["foobar"])
```

のようになるでしょう。

また、メインパッケージ内のモジュールを配布物ルート下に置くことがあるかもしれません:

```
<root>/
        setup.py
        __init__.py
        foo.py
        bar.py
```

この場合、 setup スクリプトは

```
from distutils.core import setup
setup(name = "foobar", version = "1.0",
      package_dir = {"foobar" : ""},
      packages = ["foobar"])
```

のようになるでしょう。(空文字列も現在のディレクトリを表します。)

サブパッケージがある場合、packages で明示的に列挙しなければなりませんが、package_dir はサブパッケージへのパスを自動的に展開します。(別の言い方をすれば、 Distutils はソースツリーを**走査せず**、どのディレクトリが Python パッケージに相当するのかを '__init__.py' files. を探して調べようとします。) このようにして、デフォルトのレイアウトはサブパッケージ形式に展開されます:

```
<root>/
        setup.py
        foobar/
                __init__.py
                foo.py
                bar.py
                subfoo/
                        __init__.py
                        blah.py
```

対応する setup スクリプトは以下のようになります。

```
from distutils.core import setup
setup(name = "foobar", version = "1.0",
      packages = ["foobar", "foobar.subfoo"])
```

(ここでも、package_dir を空文字列にすると現在のディレクトリを表します。)

8.3 単体の拡張モジュール

拡張モジュールは、ext_modules オプションを使って指定します。package_dir は、拡張モジュールのソースファイルをどこで探すかには影響しません; pure Python モジュールのソースのみに影響します。もっとも単純なケースでは、単一の C ソースファイルで書かれた単一の拡張モジュールは:

```
<root>/
        setup.py
        foo.c
```

になります。

`foo` 拡張をルートパッケージ下に所属させたい場合、 setup スクリプトは

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

になります。

同じソースツリーレイアウトで、この拡張モジュールを `foopkg` の下に置き、拡張モジュールの名前を変えるには:

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

のようにします。

9 リファレンスマニュアル

9.1 モジュールをインストールする: `install` コマンド群

`install` コマンドは最初にビルドコマンドを実行済みにしておいてから、サブコマンド `install_lib` を実行します。`install_data` and `install_scripts`.

`install_data`

このコマンドは配布物中に提供されている全てのデータファイルをインストールします。

`install_scripts`

このコマンドは配布物中の全ての (Python) スクリプトをインストールします。

9.2 ソースコード配布物を作成する: `sdist` command

マニフェストテンプレート関連のコマンドを以下に示します:

コマンド	説明
`include` *pat1 pat2* `...`	列挙されたパターンのいずれかにマッチする全てのファイルを対象に含めます
`exclude` *pat1 pat2* `...`	列挙されたパターンのいずれかにマッチする全てのファイルを対象から除外します
`recursive-include` *dir pat1 pat2* `...`	*dir* 下にある、列挙されたパターンのいずれかにマッチする全てのファイルを対象に含めます
`recursive-exclude` *dir pat1 pat2* `...`	*dir* 下にある、列挙されたパターンのいずれかにマッチする全てのファイルを対象から除外します
`global-include` *pat1 pat2* `...`	ソースツリー下にある、列挙されたパターンのいずれかにマッチする全てのファイルを対象に含めます
`global-exclude` *pat1 pat2* `...`	ソースツリー下にある、列挙されたパターンのいずれかにマッチする全てのファイルを対象から除外します
`prune` *dir*	*dir* 下の全てのファイルを除外します
`graft` *dir*	*dir* 下の全てのファイルを含めます

ここでいうパターンとは、UNIX 式の “glob” パターンです: `*` は全ての正規なファイル名文字列に一致し、`?` は正規なファイル名文字一字に一致します。また、`[`*range*`]` は、*range* の範囲 (例えば、`a=z`、`a-zA-Z`、`a-f0-9_.`) 内にある、任意の文字にマッチします。“正規なファイル名文字” の定義は、プラットフォームごとに特有のものです: UNIX ではスラッシュ以外の全ての文字です: Windows では、バックラッシュとコロン以外です: MacOS ではコロン以外です。

10 `distutils.sysconfig` — システム設定情報

`distutils.sysconfig` モジュールでは、 Python の低水準の設定情報へのアクセス手段を提供しています。どの設定情報変数にアクセスできるかは、プラットフォームと設定自体に大きく左右されます。また、特定の変数は、使っている Python のバージョンに対するビルドプロセスに左右されます; こうした変数は、UNIX システムでは、‘Makefile’ や Python と一緒にインストールされる設定ヘッダから探し出されます。設定ファイルのヘッダは、2.2 以降のバージョンでは ‘pyconfig.h’ 、それ以前のバージョンでは ‘config.h’ です。

他にも、`distutils` パッケージの別の部分を操作する上で便利な関数がいくつか提供されています。

PREFIX
`os.path.normpath(sys.prefix)` の結果です。

EXEC_PREFIX
`os.path.normpath(sys.exec_prefix)` の結果です。

get_config_var(*name*)
ある一つの設定変数に対する値を返します。`get_config_vars().get(`*name*`)` と同じです。

get_config_vars(*…*)
定義されている変数のセットを返します。引数を指定しなければ、設定変数名を変数の値に対応付けるマップ型を返します。引数を指定する場合、引数の各値は文字列でなければならず、戻り値は引数に関連付けられた各設定変数の値からなる配列になります。引数に指定した名前の設定変数に値がない場合、その変数値には `None` が入ります。

get_config_h_filename()
設定ヘッダのフルパス名を返します。 UNIX の場合、このヘッダファイルは **configure** スクリプトによって生成されるヘッダファイル名です; 他のプラットフォームでは、ヘッダは Python ソース配布物中で直接与えられています。ファイルはプラットフォーム固有のテキストファイルです。

get_makefile_filename()
Python をビルドする際に用いる 'Makefile' のフルパスを返します。UNIX の場合、このファイルは **configure** スクリプトによって生成されます; 他のプラットフォームでは、この関数の返す値の意味は様々です。有意なファイル名を返す場合、ファイルはプラットフォーム固有のテキストファイル形式です。この関数は POSIX プラットフォームでのみ有用です。

get_python_inc([*plat_specific*[, *prefix*]])
C インクルードファイルディレクトリについて、一般的なディレクトリ名か、プラットフォーム依存のディレクトリ名のいずれかを返します。*plat_specific* が真であれば、プラットフォーム依存のインクルードディレクトリ名を返します; *plat_specific* が偽か、省略された場合には、プラットフォームに依存しないディレクトリを返します。 *prefix* が指定されていれば、`PREFIX` の代わりに用いられます。また、 *plat_specific* が真の場合、`EXEC_PREFIX` の代わりに用いられます。

get_python_lib([*plat_specific*[, *standard_lib*[, *prefix*]]])
ライブラリディレクトリについて、一般的なディレクトリ名か、プラットフォーム依存のディレクトリ名のいずれかを返します。*plat_specific* が真であれば、プラットフォーム依存のライブラリディレクトリ名を返します; *plat_specific* が偽か、省略された場合には、プラットフォームに依存しないディレクトリを返します。*prefix* が指定されていれば、`PREFIX` の代わりに用いられます。また、 *plat_specific* が真の場合、`EXEC_PREFIX` の代わりに用いられます。*standard_lib* が真であれば、サードパーティ製の拡張モジュールをインストールするディレクトリの代わりに、標準ライブラリのディレクトリを返します。

以下の関数は、`distutils` パッケージ内の使用だけを前提にしています。

customize_compiler(*compiler*)
`distutils.ccompiler.CCompiler` インスタンスに対して、プラットフォーム固有のカスタマイズを行います。

この関数は現在のところ、UNIX だけで必要ですが、将来の互換性を考慮して一貫して常に呼び出されます。この関数は様々な UNIX の変種ごとに異なる情報や、Python の 'Makefile' に書かれた情報をインスタンスに挿入します。この情報には、選択されたコンパイラやコンパイラ/リンカのオプション、そして共有オブジェクトを扱うためにリンカに指定する拡張子が含まれます。

この関数はもっと特殊用途向けで、Python 自体のビルドプロセスでしか使われません。

set_python_build()
`distutils.sysconfig` モジュールに、モジュールが Python のビルドプロセスの一部として使われることを知らせます。これによって、ファイルコピー先を示す相対位置が大幅に変更され、インストール済みの Python ではなく、ビルド作業領域にファイルが置かれるようになります。

11 日本語訳について

11.1 このドキュメントについて

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

スト
http://www.python.jp/mailman/listinfo/python-doc-jp
または、プロジェクトのバグ管理ページ
http://sourceforge.jp/tracker/?atid=116\&group_id=11\&func=browse
までご報告ください。

11.2 翻訳者

2.3.3 和訳 Yasushi Masuda (March 12, 2004)
2.3.4 差分 Yasushi Masuda (September 20, 2004)