Python モジュールの配布

リリース *2.4*

Greg Ward
Anthony Baxter
日本語訳: Python ドキュメント翻訳プロジェクト

May 11, 2006
Python Software Foundation
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

Distutils の紹介

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。

1 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 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 (4 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。

3 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。

4 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。

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 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 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、 11 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。 (Mac OS 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 Mac OS 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'))
```

5 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** 。)

6 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 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。

7 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。

7.1 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。

7.2 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。

7.3 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 `distutils.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。

7.4 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。)

7.5 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。

8 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。 デフォルトでは現在使っているインタプリタと置換しますが、オプション **--executable** (または**-e**) を指定することで、明示的にインタプリタのパスを指定して上書きすることができます。

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']
      )
```

9 パッケージデータをインストールする

しばしばパッケージに追加のファイルをインストールする必要があります。このファイルは、パッケージの実装に強く関連したデータや、そのパッケージを使うプログラマーが必要とするドキュメントなどです。これらのファイルを**パッケージデータ**と呼びます。

パッケージデータは 関数 `setup()` にキーワード引数 `package_data` を与えることで追加できます。この値はパッケージ名から、パッケージへコピーされる相対パス名リストへのマップである必要があります。それぞれのパスは対応するパッケージが含まれるディレクトリ (もし適切なら `package_dir` のマッピングが利用されます) からの相対パスとして扱われます。つまり、ファイルはソースディレクトリ中にパッケージの一部として存在すると仮定されています。この値にはグロブパターンを含むことができます。

パス名にはディレクトリ部分を含むことができます。必要なディレクトリはインストール時に作成されます。

たとえば、パッケージがいくつかのデータファイルを含むサブディレクトリを含んでいる場合、ソースツリーでは以下のように配置できます:

```
setup.py
src/
    mypkg/
        __init__.py
        module.py
        data/
            tables.dat
            spoons.dat
            forks.dat
```

対応する `setup()` 呼び出しは以下のようになります:

```
setup(...,
      packages=['mypkg'],
      package_dir={'mypkg': 'src/mypkg'},
      package_data={'mypkg': ['data/*.dat']},
      )
```

2.4 で追加された仕様です。

10 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。

11 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
```

12 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。

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

[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

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。

ソースコード配布物を作成する

2 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

13 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 オプションで指定されたスクリプト
- 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: 23 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'、'.svn' 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 — 4 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'、 'CVS' と、'.svn' 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。

14 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。

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” という言葉は主流のデスクトップシステム特有の用語です)

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。

15 ダム形式のビルド済み配布物を作成する

16 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@example.org>" \
                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 — 12 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。)

17 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 Mac OS 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。

17.1 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いくつかの機能を、追加の組み込み関数として利用することができます。

directory_created(*path*)

file_created(*path*)

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 *path* 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。

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。

例

18 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。

19 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。)

20 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。

Distutils の拡張

Distutils は様々な方法で拡張できます。 ほとんどの拡張は存在するコマンドを新しいコマンドで置換する形でおこなわれます。新しいコマンドはたとえば存在するコマンドを置換して、そのコマンドでパッケージをどう処理するかの細部を変更することでプラットフォーム特有のパッケージ形式をサポートするために書かれているかもしれません

ほとんどの distutils の拡張は存在するコマンドを変更したい 'setup.py' スクリプト中で行われます。ほとんどはパッケージにコピーされるファイル拡張子を '.py' の他に、いくつか追加するものです。

ほとんどの distutils のコマンド実装は `distutils.cmd` の `Command` クラスのサブクラスとして実装されています。新しいコマンドは `Command` を直接継承し、置換するコマンドでは置換対象のコマンドのサブクラスにすることで `Command` を間接的に継承します。コマンドは `Command` から派生したものである必要があります。

21 新しいコマンドの統合

新しいコマンド実装を統合するにはいくつかの方法があります。一番難しいものは新機能を distutils 本体に取り込み、それのサポートを提供する Python のバージョンが出ることを待つ (そして使う) ことです。これは様々な理由で本当に難しいことです。

もっとも一般的な、そしておそらくほとんどの場合にもっとも妥当な方法は、新しい実装をあなたの 'setup.py' スクリプトに取り込み、`distutils.core.setup()` 関数でそれらを使うようにすることです。

```
from distutils.command.build_py import build_py as _build_py
from distutils.core import setup

class build_py(_build_py):
    """Specialized Python source builder."""

    # implement whatever needs to be different...

setup(cmdclass={'build_py': build_py},
      ...)
```

このアプローチは新実装をある特定のパッケージで利用したい時、そのパッケージに興味をもつ人全員がコマンドの新実装を必要とする時にもっとも価値があります。

Python 2.4 から、インストールされた Python を変更せずに、既存の 'setup.py' スクリプトをサポートするための 3 つめの選択肢が利用できるようになりました。これは追加のパッケージングシステムのサポートを追加するサードパーティ拡張を提供することを想定していますが、これらのコマンドは distutils が利用されている何にでも利用可能です。新しい設定オプション command_packages (コマンドラインオプション **--command-packages**) は、コマンド実装モジュールを検索する追加のパッケージを指定するために利用できます。 distutils の全てのオプションと同様に、このオプションもコマンドラインまたは設定ファイルで指定できます。このオプションは設定ファイル中では `[global]` セクションか、コマンドラインのコマンドより前でだけ設定できます。設定ファイル中で指定する場合、コマンドラインで上書きすることができます。空文字列を指定するとデフォルト値が使われます。これはパッケージと一緒に提供する設定ファイルでは指定しないでください。

この新オプションによってコマンド実装を探すためのパッケージをいくつでも追加することができます。複数のパッケージ名はコンマで区切って指定します。指定がなければ、検索は `distutils.command` パッケージのみで行われます。ただし 'setup.py' がオプション **--command-packages distcmds,buildcmds** で実行されている場合には、パッケージは `distutils.command`、`distcmds`、そして `buildcmds` を、この順番で検索します。新コマンドはコマンドと同じ名前のモジュールに、コマンドと同じ名前のクラスで実装されていると想定しています。上のコマドラインオプションの例では、コマンド `bdist_openpkg` は、`distcmds.bdist_openpkg.bdist_openpkg` か、`buildcmds.bdist_openpkg.bdist_openpkg` で実装されるかもしれません。

リファレンスマニュアル

22 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.

22.1 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。

22.2 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。

23 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: Mac OS ではコロン以外です。

API リファレンス

24 distutils.core — Distutils のコア機能

Distutils を使うためにインストールする必要がある唯一のモジュールが distutils.core モジュールです。setup() 関数 (セットアップスクリプトから呼び出されます) を提供します。間接的に distutils.dist.Distribution クラスと distutils.cmd.Command クラスを提供します。

setup(*arguments*)

全てを実行する基本的な関数で、Distutils でできるほとんどのことを実行します。XXXX を参照してください。

setup 関数はたくさんの引数をとります。以下のテーブルにまとめます。

argument name	value	type
name	パッケージの名前	文字列
version	パッケージのバージョン番号	distutils.version を参
description	1 行で書いたパッケージ解説	文字列
long_description	パッケージの長い解説	文字列
author	パッケージ作者の名前	文字列
author_email	パッケージ作者の email アドレス	文字列
maintainer	現在のメンテナの名前 (パッケージ作者と異なる場合)	文字列
maintainer_email	現在のメンテナの email アドレス (パッケージ作者と異なる場合)	
url	パッケージの URL(ホームページ)	URL
download_url	パッケージダウンロード用 URL	URL
packages	distutils が操作する Python パッケージのリスト	文字列のリスト
py_modules	distutils が操作する Python モジュールのリスト	文字列のリスト
scripts	ビルドおよびインストールする単体スクリプトファイルのリスト	文字列のリスト
ext_modules	ビルドする拡張モジュール	distutils.core.Exten
classifiers	パッケージの Trove カテゴリのリスト	XXX もっと良い定義へのリ
distclass	使用する Distribution クラス	distutils.core.Distr
script_name	setup.py スクリプトの名前 - デフォルトでは sys.argv[0]	文字列
script_args	セットアップスクリプトの引数	文字列のリスト
options	セットアップスクリプトのデフォルト引数	文字列
license	パッケージのライセンス	
keywords	説明用メタデータ。 PEP 314 を参照してください	
platforms		
cmdclass	コマンド名から Command サブクラスへのマッピング	辞書

run_setup(*script_name*[, *script_args*=*None*, *stop_after*='*run*'])

制御された環境でセットアップスクリプトを実行し、いろいろなものを操作する distutils.dist.Distribution クラスのインスタンスを返します。これはディストリビューションのメタデータ (キーワード引数 *script* として 関数 setup() に渡される) を参照したり、設定ファイルやコマンドラインの内容を調べる時に便利です。

script_name は execfile() で実行されるファイルです。*sys.argv[0]* は、 呼び出しのために *script_name* と置換されます。*script_args* は文字列のリストです。もし提供されていた場合、*sys.argv[1:]* は、呼び出しのために *script_args* で置換されます。

stop_after は いつ動作を停止するか関数 setup() に伝えます。とりうる値は:

値	説明
	Distribution インスタンスを作成し、キーワード引数を setup() に渡したあとに停止する。 設定ファイルをパーズしたあと停止する (そしてそのデータは Distribution インスタンスに保存される)。 コマンドライン (sys.argv[1:] または *script_args*) がパーズされたあとに停止する (そしてそのデータは Di 全てのコマンドを実行したあとに停止する (関数 setup() を通常の方法で呼び出した場合と同じ)。 デフォル

これに加えて、distutils.core モジュールは他のモジュールにあるいくつかのクラスを公開しています。

- Extension は distutils.extension から。
- Command は distutils.cmd から。
- Distribution は distutils.dist から。

それぞれの簡単な説明を以下に記します。完全な説明についてはそれぞれのモジュールをごらんください。

class Extension

Extension クラスは、セットアップスクリプト中で C または C++拡張モジュールを表します。コンストラクタで以下のキーワード引数をとります。

argument name	value
name	拡張のフルネーム (パッケージを含む) — ファイル名やパス名では**なく**、Python のピリオド区
sources	ソースファイル名のリスト。配布物ルートディレクトリ (setup スクリプトのある場所) からの
include_dirs	C/C++ヘッダファイルを検索するディレクトリのリスト (プラットフォーム独立のため UNIX
define_macros	定義するマクロのリスト; それぞれのマクロは 2 要素のタプルで定義されます。' 値' には定義
undef_macros	定義を消すマクロのリスト
library_dirs	リンク時に C/C++ライブラリを検索するディレクトリのリスト
libraries	リンクするライブラリ名のリスト (ファイル名やパスではない)
runtime_library_dirs	実行時 (shared extension では、拡張が読み込まれる時) に C/C++ライブラリを探索するディレ
extra_objects	追加でリンクするファイル ('sources' に対応するコードが含まれていないファイル、バイナリ
extra_compile_args	'sources' のソースをコンパイルする時に追加するプラットフォーム特有またはコンパイラ特有
extra_link_args	オブジェクトファイルをリンクして拡張 (または新しい Python インタプリタ) を作る時に追加
export_symbols	shared extension からエクスポートされるシンボルのリスト。全てのプラットフォームでは使え
depends	拡張が依存するファイルのリスト
language	拡張の言語 (例: `'c'`,`'c++'`,`'objc'`)。指定しなければソースの拡張子で検出される。

class `Distribution`

`Distribution` は Python ソフトウェアパッケージをどのようにビルド、インストール、パッケージするかを定義する。

`Distribution` のコンストラクタが取りうるキーワード引数のリストに関しては、`setup()` 関数を見てください。`setup()` は `Distribution` のインスタンスを作ります。

class `Command`

`Command` クラス (そのサブクラスのインスタンス) は distutils のあるコマンドを実装します。

25 `distutils.ccompiler` — CCompiler ベースクラス

このモジュールは `CCompiler` クラスの抽象ベースクラスを提供します。`CCompiler` のインスタンスはプロジェクトにおける全てのコンパイルおよびリンクに使われます。コンパイラのオプションを設定するためのメソッドが提供されます — マクロ定義、include ディレクトリ、リンクパス、ライブラリなど。

このモジュールは以下の関数を提供します。

`gen_lib_options` (*compiler, library_dirs, runtime_library_dirs, libraries*)

ライブラリを探索するディレクトリ、特定のライブラリとのリンクをするためのリンカオプションを生成します。*libraries* と *library_dirs* はそれぞれライブラリ名 (ファイル名ではありません!) のリストと、探索ディレクトリのリストです。compiler で利用できるコマンドラインオプションのリスト (指定されたフォーマット文字列に依存します) を返します。

`gen_preprocess_options` (*macros, include_dirs*)

C プリプロセッサオプション (-D, -U, -I) を生成します。これらは少なくとも 2 つのコンパイラで利用可能です。典型的な UNIX のコンパイラと、Visual C++です。*macros* は 1 または 2 要素のタプルで *(name,)* は *name* マクロの削除 (-U) を意味し、 *(name,value)* は *name* マクロを *value* として定義 (-D) します。*include_dirs* はディレクトリ名のリストで、ヘッダファイルのサーチパスに追加されます (-I)。UNIX のコンパイラと、Visual C++で利用できるコマンドラインオプションのリストを返します。

`get_default_compiler` (*osname, platform*)

指定されたプラットフォームのデフォルトコンパイラを返します。

問い合わせの *osname* は Python 標準の OS 名 (*os.name* で返されるもの) のひとつであるべきで、*platform* は *sys.platform* で返される共通の値です。

パラメータが指定されていない場合のデフォルト値は `os.name` と `sys.platform` です。

`new_compiler` (*plat=None, compiler=None, verbose=0, dry_run=0, force=0*)

指定されたプラットフォーム/コンパイラの組み合わせ向けに、CCompiler サブクラスのインスタンスを生成するファクトリ関数です。*plat* のデフォルト値は `os.name` (例: `'posix'`,`'nt'`), *compiler*)、*compiler* のデフォルト値はプラトフォームのデフォルトコンパイラです。現在は `'posix'` と `'nt'` だけがサポートされています、デフォルトのコンパイラは "traditional UNIX interface"(`UnixCCompiler` クラス) と、Visual C++(`MSVCCompiler` クラス) です。Windows で UNIX コンパイラオブジェクトを要求することも、UNIX で Microsoft コンパイラオブジェクトを要求することも可能です。*compiler* 引数を与えると *plat* は無視されます。

`show_compilers`()

利用可能なコンパイラのリストを表示します(build, build_ext, build_clib の、**--help-compiler** オプションで使われます。)

class CCompiler ([*verbose=0, dry_run=0, force=0*])

抽象ベースクラス CCompiler は実際のコンパイラクラスで実装される必要のあるインタフェースを定義しています。このクラスはコンパイラクラスで利用されるユーティリティメソッドも定義しています。

コンパイラ抽象クラスの基本的な前提は、各インスタンスはあるプロジェクトをビルドするときの全コンパイル/リンクで利用できるということです。そこで、コンパイルとリンクステップで共通する属性 — インクルードディレクトリ、マクロ定義、リンクするライブラリ など — はコンパイラインスタンスの属性になります。どのように各ファイルが扱われるかを変更できるように、ほとんどの属性はコンパイルごと、またはリンクごとに与えることができます。

各サブクラスのコンストラクタは Compiler クラスのインスタンスを作ります。フラグは *verbose*(冗長な出力を表示します)、*dry_run*(実際にはそのステップを実行しません)、そして *force*(依存関係を無視して全て再ビルドします) です。これらのフラグは全てデフォルト値が 0(無効) になっています。CCompiler またはサブクラスを直接インスタンス化したくない場合には、かわりに distutils.CCompiler.new_compiler() ファクトリ関数を利用してください。

以下のメソッドで、Compiler クラスのインスタンスが使うコンパイラオプションを手動で変更できます。

add_include_dir(*dir*)
dir をヘッダファイル探索ディレクトリのリストに追加します。コンパイラは add_include_dir() を呼び出した順にディレクトリを探索するよう指定されます。

set_include_dirs(*dirs*)
探索されるディレクトリのリストを *dirs* (文字列のリスト) に設定します。先に実行された add_include_dir() は上書きされます。後で実行する add_include_dir() は set_include_dirs() のリストにディレクトリを追加します。これはコンパイラがデフォルトで探索する標準インクルードディレクトリには影響しません。

add_library(*libname*)
libname をコンパイラオブジェクトによるリンク時に使われるライブラリのリストに追加します。*libname* はライブラリを含むファイル名ではなく、ライブラリそのものの名前です: 実際のファイル名はリンカ、コンパイラ、またはコンパイラクラス (プラットフォームに依存します) から推測されます。
リンカは add_library() と set_library() で渡された順にライブラリをリンクしようとします。ライブラリ名が重なることは問題ありません。リンカは指定された回数だけライブラリとリンクしようとします。

set_libraries(*libnames*)
コンパイラオブジェクトによるリンク時に使われるライブラリのリストを *libnames*(文字列のリスト) に設定します。これはリンカがデフォルトでリンクする標準のシステムライブラリには影響しません。

add_library_dir(*dir*)
add_library() と set_libraries() で指定されたライブラリを探索するディレクトリのリストに *dir* を追加します。リンカは add_library_dir() と set_library_dirs() で指定された順にディレクトリを探索されます。

set_library_dirs(*dirs*)
ライブラリを探索するディレクトリを *dirs*(文字列のリスト) に設定します。これはリンカがデフォルトで探索する標準ライブラリ探索パスには影響しません。

add_runtime_library_dir(*dir*)
実行時に共有ライブラリを探索するディレクトリのリストに *dir* を追加します。

set_runtime_library_dirs(*dirs*)
実行時に共有ライブラリを探索するディレクトリのリストを *dir* に設定します。これはランタイムリンカがデフォルトで利用する標準探索パスには影響しません。

define_macro(*name*[, *value=None*])
このコンパイラオブジェクトで実行される全てのコンパイルで利用されるプリプロセッサのマクロを定義します。省略可能なパラメータ *value* は文字列であるべきです。省略された場合は、マクロは特定の値をとらずに定義され、具体的な結果は利用されるコンパイラに依存します。(XXX 本当に? これについて ANSI で言及されている?)

undefine_macro(*name*)
このコンパイラオブジェクトで実行される全てのコンパイルで利用されるプリプロセッサのマクロ定義を消します。同じマクロを `define_macro()` で定義し、`undefine_macro()` で定義を削除した場合、後で呼び出されたものが優先される (複数の再定義と削除を含みます)。もしコンパイルごと (すなわち `compile()` の呼び出しごと) にマクロが再定義/削除される場合も後で呼び出されたものが優先されます。

add_link_object(*object*)
このコンパイラオブジェクトによる全てのリンクで利用されるオブジェクトファイル (または類似のライブラリファイルや “リソースコンパイラ” の出力) のリストに *object* を追加します。

set_link_objects(*objects*)
このコンパイラオブジェクトによる全てのリンクで利用されるオブジェクトファイル (または類似のもの) のリストを *objects* に設定します。これはリンカがデフォルト利用する標準オブジェクトファイル (システムライブラリなど) には影響しません。

以下のメソッドはコンパイラオプションの自動検出を実装しており、GNU **autoconf** に似たいくつかの機能を提供します。

detect_language(*sources*)
与えられたファイルまたはファイルのリストの言語を検出します。インスタンス属性 `language_map`(辞書) と、`language_order` (リスト) を仕事に使います。

find_library_file(*dirs, lib*[, *debug=0*])
指定されたディレクトリのリストから、スタティックまたは共有ライブラリファイル *lib* を探し、そのファイルのフルパスを返します。もし *debug* が真なら、(現在のプラットフォームで意味があれば) デバッグ版を探します。指定されたどのディレクトリでも *lib* が見つからなければ `None` を返します。

has_function(*funcname* [, *includes=None, include_dirs=None, libraries=None, library_dirs=None*])
funcname が現在のプラットフォームでサポートされているかどうかをブール値で返します。省略可能引数は追加のインクルードファイルやパス、ライブラリやパスを与えることでコンパイル環境を指定します。

library_dir_option(*dir*)
dir をライブラリ探索ディレクトリに追加するコンパイラオプションを返します。

library_option(*lib*)
共有ライブラリまたは実行ファイルにリンクされるライブラリ一覧に *lib* を追加するコンパイラオプションを返します。

runtime_library_dir_option(*dir*)
ランタイムライブラリを検索するディレクトリのリストに *dir* を追加するコンパイラオプションを返します。

set_executables(***args*)
コンパイルのいろいろなステージで実行される実行ファイル (とその引数) を定義します。コンパイラクラス (の ’executables’ 属性) によって実行ファイルのセットは変わる可能性がありますが、ほとんどは以下のものを持っています:

attribute	description
compiler	C/C++ コンパイラ
linker_so	シェアードオブジェクト、ライブラリを作るために使うリンカ
linker_exe	バイナリ実行可能ファイルを作るために使うリンカ
archiver	静的ライブラリを作るアーカイバ

コマンドラインをもつプラットフォーム (UNIX, DOS/Windows) では、それぞれの文字列は実行ファイル名と (省略可能な) 引数リストに分割されます。(文字列の分割は UNIX のシェルが行うものに似ています: 単語はスペースで区切られますが、クォートとバックスラッシュでオーバーライドできます。`distutils.util.split_quoted()` をごらんください。)

以下のメソッドはビルドプロセスのステージを呼び出します。

compile(*sources*[, *output_dir=None, macros=None, include_dirs=None, debug=0, extra_preargs=None, extra_postargs=None, depends=None*])
1 つ以上のソースファイルをコンパイルします。オブジェクトファイルを生成 (たとえば ‘.c’ ファイルを ‘.o’ ファイルに変換) します。

sources はファイル名のリストである必要があります。おそらく C/C++ファイルですが、実際にはコンパイラとコンパイラクラスで扱えるもの (例: `MSVCCompiler` はリソースファイルを *sources* にとることができます) なら何でも指定できます。 *sources* のソースファイルひとつずつに対応するオブジェクトファイル名のリストを返します。実装に依存しますが、全てのソースファイルがコンパイルされる必要はありません。しかし全ての対応するオブジェクトファイル名が返ります。

もし *output_dir* が指定されていれば、オブジェクトファイルはその下に、オリジナルのパスを維持した状態で置かれます。 つまり、'foo/bar.c' は通常コンパイルされて 'foo/bar.o' になります (UNIX 実装の場合) が、もし *output_dir* が *build* であれば、'build/foo/bar.o' になります。

macros は (もし指定されていれば) マクロ定義のリストである必要があります。マクロ定義は *(name, value)* という形式の 2 要素のタプル、または *(name,)* という形式の 1 要素のタプルのどちらかです。前者はマクロを定義します。もし value が `None` であれば、マクロは特定の値をもたないで定義されます。1 要素のタプルはマクロ定義を削除します。後で実行された定義/再定義/削除が優先されます。

include_dirs は (もし指定されていれば) 文字列のリストである必要があります。このコンパイルだけで有効な、デフォルトのインクルードファイルの検索ディレクトリに追加するディレクトリ群を指定します。

debug はブーリアン値です。もし真なら、コンパイラはデバッグシンボルをオブジェクトファイルに (または別ファイルに) 出力します。

extra_postargs と *extra_postargs* は実装依存です。コマンドラインをもっているプラットフォーム (例 UNIX, DOS/Windows) では、おそらく文字列のリスト: コンパイラのコマンドライン引数の前/後に追加するコマンドライン引数です。他のプラットフォームでは、実装クラスのドキュメントを参照してください。どの場合でも、これらの引数は抽象コンパイラフレームワークが期待に沿わない時の脱出口として意図されています。

depends は (もし指定されていれば) ターゲットが依存しているファイル名のリストです。ソースファイルが依存しているファイルのどれかより古ければ、ソースファイルは再コンパイルされます。これは依存関係のトラッキングをサポートしていますが、荒い粒度でしか行われません。

失敗すると `CompileError` を起こします。

`create_static_lib` (*objects, output_libname*[, *output_dir*=`None`, *debug*=*0*, *target_lang*=`None`])
静的ライブラリファイルを作るために元ファイル群をリンクします。「元ファイル群」は *objects* で指定されたオブジェクトファイルのリストを基礎にしています。追加のオブジェクトファイルを `add_link_object()` および/または `set_link_objects()` で指定し、追加のライブラリを `add_library()` および/または `set_libraries()` で指定します。そして *libraries* で指定されたライブラリです。

output_libname はライブラリ名で、ファイル名ではありません; ファイル名はライブラリ名から作られます。*output_dir* はライブラリファイルが起かれるディレクトリです。

debug はブール値です。真なら、デバッグ情報がライブラリに含まれます (ほとんどのプラットフォームではコンパイルステップで意味をもちます: *debug* フラグは一貫性のためにここにもあります。)。

target_lang はオブジェクトがコンパイルされる対象になる言語です。これはその言語特有のリンク時の処理を可能にします。

失敗すると `LibError` を起こします。

`link` (*target_desc, objects, output_filename*[, *output_dir*=`None`, *libraries*=`None`, *library_dirs*=`None`, *runtime_library_dirs*=`None`, *export_symbols*=`None`, *debug*=*0*, *extra_preargs*=`None`, *extra_postargs*=`None`, *build_temp*=`None`, *target_lang*=`None`])
実行ファイルまたは共有ライブラリファイルを作るために元ファイル群をリンクします。

「元ファイル群」は *objects* で指定されたオブジェクトファイルのリストを基礎にしています。*output_filename* はファイル名です。もし *output_dir* が指定されていれば、それに対する相対パスとして *output_filename* は扱われます (必要ならば *output_filename* はディレクトリ名を含むことができます。)。

libraries はリンクするライブラリのリストです。これはファイル名ではなくライブラリ名で指定します。プラットフォーム依存の方式でファイル名に変換されます (例: *foo* は UNIX では 'libfoo.a' に、DOS/Windows では 'foo.lib' になります。)。ただしこれらはディレクトリ名を含むことができ、その場合はリンカは通常の場所全体を探すのではなく特定のディレクトリを参照します。

library_dirs はもし指定されるならば、修飾されていない (ディレクトリ名を含んでいない) ライブラリ名で指定されたライブラリを探索するディレクトリのリストです。これはシステムのデ

フォルトより優先され、add_library_dir() と/または set_library_dirs() に渡されます。*runtime_library_dirs* は共有ライブラリに埋め込まれるディレクトリのリストで、実行時にそれが依存する共有ライブラリのパスを指定します (これは UNIX でだけ意味があるかもしれません。)。

export_symbols は共有ライブラリがエクスポートするシンボルのリストです。(これは Windows だけで意味があるようです。)

debug は compile() や create_static_lib() と同じですが、少しだけ違いがあり、(create_static_lib() では *debug* フラグは形式をあわせるために存在していたのに対して) ほとんどのプラットフォームで意識されます。

extra_preargs と *extra_postargs* は compile() と同じですが、コンパイラではなくリンカへの引数として扱われます。

target_lang は指定されたオブジェクトがコンパイルされた対象言語です。リンク時に言語特有の処理を行えるようにします。

失敗すると LinkError が起きます。

link_executable (*objects, output_progname*[, *output_dir=None, libraries=None, library_dirs=None, runtime_library_dirs=None, debug=0, extra_preargs=None, extra_postargs=None, target_lang=None*])

実行ファイルをリンクします。*output_progname* は実行ファイルの名前です。*objects* はリンクされるオブジェクトのファイル名のリストです。他の引数は link メソッドと同じです。

link_shared_lib (*objects, output_libname*[, *output_dir=None, libraries=None, library_dirs=None, runtime_library_dirs=None, export_symbols=None, debug=0, extra_preargs=None, extra_postargs=None, build_temp=None, target_lang=None*])

共有ライブラリをリンクします。*output_libname* は出力先のライブラリ名です。*objects* はリンクされるオブジェクトのファイル名のリストです。他の引数は link メソッドと同じです。

link_shared_object (*objects, output_filename*[, *output_dir=None, libraries=None, library_dirs=None, runtime_library_dirs=None, export_symbols=None, debug=0, extra_preargs=None, extra_postargs=None, build_temp=None, target_lang=None*])

共有オブジェクトをリンクします。*output_filename* は出力先の共有オブジェクト名です。*objects* はリンクされるオブジェクトのファイル名のリストです。他の引数は link メソッドと同じです。

preprocess (*source*[, *output_file=None, macros=None, include_dirs=None, extra_preargs=None, extra_postargs=None*])

source で指定されたひとつの C/C++ソースファイルをプリプロセスします。出力先のファイルは *output_file* か、もし *output_file* が指定されていなければ *stdout* になります。*macro* は compile() と同様にマクロ定義のリストで、define_macro() や undefine_macro() によって引数になります。*include_dirs* はデフォルトのリストに追加されるディレクトリ名のリストで、add_include_dir() と同じ方法で扱われます。

失敗すると PreprocessError が起きます。

以下のユーティリティメソッドは具体的なサブクラスで使うために、CCompiler クラスで定義されています。

executable_filename (*basename*[, *strip_dir=0, output_dir=''*])

basename で指定された実行ファイルのファイル名を返します。Windows 以外の典型的なプラットフォームでは basename そのままが、Windows では '.exe' が追加されたものが返ります。

library_filename (*libname*[, *lib_type='static', strip_dir=0, output_dir=''*])

現在のプラットフォームでのライブラリファイル名を返します。UNIX で *lib_type* が 'static' の場合、'liblibname.a' の形式を返し、*lib_type* が 'dynamic' の場合は 'liblibname.so' の形式を返します。

object_filenames (*source_filenames*[, *strip_dir=0, output_dir=''*])

指定されたソースファイルに対応するオブジェクトファイル名を返します。*source_filenames* はファイル名のリストです。

shared_object_filename (*basename*[, *strip_dir=0, output_dir=''*])

basename に対応する共有オブジェクトファイルのファイル名を返します。

execute (*func, args*[, *msg=None, level=1*])

distutils.util.execute() を呼びだします。このメソッドはログを取り、*dry_run* フラグを考慮にいれて、Python 関数 *func* に引数 *args* を与えて呼びだします。

spawn(*cmd*)
distutils.util.spawn() を呼び出します。これは指定したコマンドを実行する外部プロセスを呼び出します。

mkpath(*name*[, *mode*=*511*])
distutils.dir_util.mkpath() を呼び出します。これは親ディレクトリ込みでディレクトリを作成します。

move_file(*src, dst*)
distutils.file_util.move_file() を呼び出します。*src* を *dst* にリネームします。

announce(*msg*[, *level*=*1*])
distutils.log.debug() 関数を使ってメッセージを書き出します。

warn(*msg*)
警告メッセージ *msg* を標準エラー出力に書き出します。

debug_print(*msg*)
もしこの CCompiler インスタンスで *debug* フラグが指定されていれば *msg* を標準出力に出力し、そうでなければ何も出力しません。

26 distutils.unixccompiler — Unix C コンパイラ

このモジュールは UnixCCompiler クラスを提供します。CCompiler クラスのサブクラスで、典型的な UNIX スタイルのコマンドライン C コンパイラを扱います:

- マクロは **-D*name*[=value]** で定義されます。
- マクロは **-U*name*** で削除されます。
- インクルードファイルの探索ディレクトリは**-I*dir*** で指定されます。
- ライブラリは **-l*lib*** で指定されます。
- ライブラリの探索ディレクトリは **-L*dir*** で指定されます。
- コンパイルは **cc** (またはそれに似た) 実行ファイルに、**-c** オプションをつけて実行します: '.c' を '.o' にコンパイルします。
- 静的ライブラリは **ar** コマンドで処理されます (**ranlib** を使うかもしれません)
- 共有ライブラリのリンクは **cc -shared** で処理されます。

27 distutils.msvccompiler — Microsoft コンパイラ

このモジュールは MSVCCompiler クラスを提供します。抽象クラス CCompiler の具象クラスで Microsoft Visual Studio 向けのものです。.Net SDK の一部として無償で入手できるコンパイラを扱うこともできます。

28 distutils.bcppcompiler — Borland コンパイラ

このモジュールは BorlandCCompiler クラスを提供します。抽象クラス CCompiler の具象クラスで Borland C++ コンパイラ向けです。

29 distutils.cygwincompiler — Cygwin コンパイラ

このモジュールは CygwinCCompiler クラスを提供します。UnixCCompiler のサブクラスで Cygwin に移植された Windows 用の GNU C コンパイラ向けです。さらに Mingw32CCompiler クラスを含んでおり、これは mingw32 向けに移植された GCC (cygwin の no-cygwin モードと同じ) 向けです。

30 distutils.emxccompiler — OS/2 EMX コンパイラ

このモジュールは EMXCCompiler クラスを提供します。UnixCCompiler のサブクラスで GNU C コンパイラの OS/2 向け EMX ポートを扱います。

31 distutils.mwerkscompiler — Metrowerks CodeWarrior サポート

MWerksCompiler クラスを提供します。抽象クラス CCompiler の具象クラスで Macintosh の MetroWerks CodeWarrior 向けです。CW on Windows をサポートするには作業が必要です。

32 distutils.archive_util — アーカイブユーティリティ

このモジュールはアーカイブファイル (tar や zip) を作成する関数を提供します。

make_archive (*base_name, format*[, *root_dir=None, base_dir=None, verbose=0, dry_run=0*])
アーカイブファイル (例: zip や tar) を作成します。*base_name* は作成するファイル名からフォーマットの拡張子を除いたものです。*format* はアーカイブのフォーマットで zip、tar、ztar、gztar のいずれかです。*root_dir* はアーカイブのルートディレクトリになるディレクトリです: つまりアーカイブを作成する前に *root_dir* に *chdir* します。*base_dir* はアーカイブの起点となるディレクトリです: つまり *base_dir* はアーカイブ中の全ファイルおよびディレクトリの前につくディレクトリ名です。*root_dir* と *base_dir* はともにカレントディレクトリがデフォルト値です。アーカイブファイル名を返します。

警告: この関数は bz2 ファイルを扱えるように変更されるべきです

make_tarball (*base_name, base_dir*[, *compress='gzip', verbose=0, dry_run=0*])
base_dir 以下の全ファイルから、tar ファイルを作成 (オプションで圧縮) します。*compress* は'gzip'、'compress'、'bzip2'、または None である必要があります。'tar' と圧縮ユーティリティ*'compress'* にはパスが通っている必要があるので、これはおそらく UNIX だけで有効です。出力 tar ファイルは '*base_dir*.tar' という名前になり、圧縮によって拡張子がつきます ('.gz'、 '.bz2' または '.Z')。出力ファイル名が返ります。

警告: これは tarfile モジュールの呼び出しに置換されるべきです。

make_zipfile (*base_name, base_dir*[, *verbose=0, dry_run=0*])
base_dir 以下の全ファイルから、zip ファイルを作成します。出力される zip ファイルは *base_dir* + '.zip' という名前になります。zipfilePython モジュール (利用可能なら) または InfoZIP 'zip' ユーティリティ(インストールされていてパスが通っているなら) を使います。もしどちらも利用できなければ、DistutilsExecError が起きます。出力 zip ファイル名が返ります。

33 distutils.dep_util — 依存関係のチェック

このモジュールはシンプルなタイムスタンプを元にしたファイルやファイル群の依存関係を処理する関数を提供します。さらに、それらの依存関係解析を元にした関数を提供します。

newer (*source, target*)
source が存在して、*target* より最近変更されている、または *source* が存在して、*target* が存在していない場合は真を返します。両方が存在していて、*target* のほうが *source* より新しいか同じ場合には偽を返します。*source* が存在しない場合には DistutilsFileError を起こします。

newer_pairwise (*sources, targets*)
ふたつのファイル名リストを並列に探索して、それぞれのソースが対応するターゲットより新しいかをテストします。newer() の意味でターゲットよりソースが新しいペアのリスト (*sources*,*targets*) を返します。

newer_group (*sources, target*[, *missing='error'*])
target が *source* にリストアップされたどれかのファイルより古ければ真を返します。言い換えれば、*target* が存在して *sources* の全てより新しいなら偽を返し、そうでなければ真を返します。*missing* はソースファイルが存在しなかった時の振る舞いを決定します。デフォルト ('error') は os.stat()

で OSError 例外を起こします。もし'ignore' なら、単に存在しないソースファイルを無視します。もし'newer' なら、存在しないソースファイルについては *target* が古いとみなします (これは"dry-tun" モードで便利です: 入力がないのでコマンドは実行できませんが実際に実行しようとしていないので問題になりません)。

34 distutils.dir_util — ディレクトリツリーの操作

このモジュールはディレクトリとディレクトリツリーを操作する関数を提供します。

mkpath (*name*[, *mode=0777*, *verbose=0*, *dry_run=0*])
ディレクトリと、必要な親ディレクトリを作成します。もしディレクトリが既に存在している (*name* が空文字列の場合、カレントディレクトリを示すのでもちろん存在しています) 場合、何もしません。ディレクトリを作成できなかった場合 (例: ディレクトリと同じ名前のファイルが既に存在していた)、DistutilsFileError を起こします。もし *verbose* が真なら、それぞれの mkdir について 1 行、標準出力に出力します。実際に作成されたディレクトリのリストを返します。

create_tree (*base_dir*, *files*[, *mode=0777*, *verbose=0*, *dry_run=0*])
files を置くために必要な空ディレクトリを *base_dir* 以下に作成します。*base_dir* ディレクトリは存在している必要はありません。*files* はファイル名のリストで *base_dir* からの相対パスとして扱われます。*base_dir* + *files* のディレクトリ部分 が (既に存在していなければ) 作成されます。*mode*, *verbose* と *dry_run* フラグは mkpath() と同じです。

copy_tree (*src*, *dst*[*preserve_mode=1*, *preserve_times=1*, *preserve_symlinks=0*, *update=0*, *verbose=0*, *dry_run=0*])
src ディレクトリツリー全体を *dst* にコピーします。*src* と *dst* はどちらもディレクトリ名である必要があります。もし *src* がディレクトリでなければ、DistutilsFileError を起こします。もし *dst* が存在しなければ、*mkpath()* で作成されます。実行結果は、*src* 以下の全てのファイルが *dst* にコピーされ、*src* 以下の全てのディレクトリが *dst* に再帰的にコピーされます。コピーされた (またはされるはず) のファイルのリストを返します。返り値は *update* または *dry_run* に影響されません: *src* 以下の全ファイルを単に *dst* 以下に改名したリストが返されます。

preserve_mode と *preserve_times* は distutils.file_util の copy_file と同じです: 通常のファイルには適用されますが、ディレクトリには適用されません。もし *preserve_symlinks* が真なら、シンボリックリンクは (サポートされているシステムでは) シンボリックリンクとしてコピーされます。そうでなければ (デフォルト) シンボリックリンクは参照されている実体ファイルがコピーされます。*update* と *verbose* は copy_file() と同じです。

remove_tree (*directory*[*verbose=0*, *dry_run=0*])
再帰的に *directory* とその下の全ファイルを削除します。エラーは無視されます (*verbose* が真の時は stdout に出力されます)

35 distutils.file_util — 1 ファイルの操作

このモジュールはそれぞれのファイルを操作するユーティリティ関数を提供します。

copy_file (*src*, *dst*[*preserve_mode=1*, *preserve_times=1*, *update=0*, *link=None*, *verbose=0*, *dry_run=0*])
ファイル *src* を *dst* にコピーします。もし *dst* がディレクトリなら、*src* はそこへ同じ名前でコピーされます; そうでなければ、ファイル名として扱われます。 (もしファイルが存在するなら、上書きされます。) mosil*preserve_mode* が真 (デフォルト) なら、ファイルのモード (タイプやパーミッション、その他プラットフォームがサポートするもの) もコピーされます。もし *preserve_times* が真 (デフォルト) なら、最終更新、最終アクセス時刻もコピーされます。もし *update* が真なら、*src* は *dst* が存在しない場合か、*dst* が *src* より古い時にだけコピーします。

link は値を'hard' または'sym' に設定することでコピーのかわりにハードリンク (os.link を使います) またはシンボリックリンク (os.symlink を使います) を許可します。None(デフォルト) の時には、ファイルはコピーされます。*link* をサポートしていないシステムで有効にしないでください。copy_file() はハードリンク、シンボリックリンクが可能かチェックしていません。ファイルの内容をコピーするために*_copy_file_contents()* を利用しています。

'(dest_name, copied)' のタプルを返します: *dest_name* は出力ファイルの実際の名前、*copied* はファイルがコピーされた (*dry_run* が真の時にはコピーされることになった) 場合には真です。

move_file(*src, dst*[*verbose, dry_run*])
ファイル *src* を *dst* に移動します。もし *dst* がディレクトリなら、ファイルはそのディレクトリに同じ名前で移動されます。そうでなければ、*src* は *dst* に単にリネームされます。新しいファイルの名前を返します。**警告:** Unix では デバイスをまたがる移動は `copy_file()` を利用して扱っています。他のシステムではどうなっている ???

write_file(*filename, contents*)
filename を作成し、*contents*(行末文字がない文字列のシーケンス) を書き込みます。

36 `distutils.util` — その他のユーティリティ関数

このモジュールは他のユーティリティモジュールにあわないものを提供しています。

get_platform()
現在のプラットフォームを示す文字列を返します。これはプラットフォーム依存のビルドディレクトリやプラットフォーム依存の配布物を区別するために使われます。典型的には、('os.uname()' のように)OS の名前とバージョン、アーキテクチャを含みますが、厳密には OS に依存します。たとえば IRIX ではアーキテクチャはそれほど重要ではありません (IRIX は SGI のハードウェアだけで動作する) が、Linux ではカーネルのバージョンはそれほど重要ではありません。

返り値の例:

- `linux-i586`
- `linux-alpha`
- `solaris-2.6-sun4u`
- `irix-5.3`
- `irix64-6.2`

POSIX でないプラットフォームでは、今のところ単に `sys.platform` が返されます。

convert_path(*pathname*)
'pathname' をファイルシステムで利用できる名前にして返します。すなわち、'/' で分割し、現在のディレクトリセパレータで接続しなおします。セットアップスクリプト中のファイル名は Unix スタイルで提供され、実際に利用する前に変換する必要があるため、この関数が必要になります。もし *pathname* の最初または最後が スラッシュの場合、UNIX 的でないシステムでは `ValueError` が起きます。

change_root(*new_root, pathname*)
pathname の前に *new_root* を追加したものを返します。もし *pathname* が相対パスなら、'`os.path.join(new_root,pathname)`' と等価です。そうでなければ、*pathname* を相対パスに変換したあと接続します。これは DOS/Windows と Mac OS ではトリッキーな作業になります。

check_environ()
'os.environ' に、ユーザが config ファイル、コマンドラインオプションなどで利用できることを保証している環境変数があることを確認します。現在は以下のものが含まれています:

- HOME - ユーザのホームディレクトリ (UNIX のみ)
- PLAT - ハードウェアと OS を含む現在のプラットフォームの説明。(`get_platform()` を参照)

subst_vars(*s, local_vars*)
shell/Perl スタイルの変数置換を *s* について行います。全ての`$`に名前が続いたものは変数とみなされ、辞書 *local_vars* でみつかった値に置換されます。*local_vars* で見つからなかった場合には `os.environ` で置換されます。*os.environ* は最初にある値を含んでいることをチェックされます: `check_environ()` を参照。*local_vars* or `os.environ` のどちらにも値が見つからなかった場合、`ValueError` を起こします。

これは完全な文字列挿入関数ではないことに注意してください。`$variable` の名前には大小英字、数字、アンダーバーだけを含むことができます。{ } や を使った引用形式は利用できません。

grok_environment_error(*exc*[, *prefix*=`''error: ''`])
例外オブジェクト `EnvironmentError` (`IOError` または `OSError`) から、エラーメッセージを生成します。Python 1.5.1 またはそれ以降の形式を扱い、ファイル名を含んでいない例外オブジェクトも扱います。このような状況はエラーが 2 つのファイルに関係する操作、たとえば `rename()` や `link()` で発生します。*prefix* をプレフィクスに持つエラーメッセージを返します。

split_quoted(*s*)
文字列を Unix のシェルのようなルール (引用符やバックスラッシュの扱い) で分割します。つまり、バックスラッシュでエスケープされるか、引用符で囲まれていなければ各語はスペースで区切られます。一重引用符と二重引用符は同じ意味です。引用符もバックスラッシュでエスケープできます。2 文字でのエスケープシーケンスに使われているバックスラッシュは削除され、エスケープされていた文字だけが残ります。引用符は文字列から削除されます。語のリストが返ります。

execute(*func, args*[, *msg=None, verbose=0, dry_run=0*])
外部に影響するいくつかのアクション (たとえば、ファイルシステムへの書き込み) を実行します。そのようなアクションは *dry_run* フラグで無効にする必要があるので特別です。この関数はその繁雑な処理を行います。関数と引数のタプル、(実行する「アクション」をはっきりさせるための) 表示に使われる任意のメッセージを渡してください。

strtobool(*val*)
真偽値をあらわす文字列を真 (1) または偽 (0) に変換します。

真の値は `y`, `yes`, `t`, `true`, `on` そして `1` です。偽の値は `n`, `no`, `f`, `false`, `off` そして `0` です。*val* が上のどれでもない時は `ValueError` を起こします。

byte_compile(*py_files*[, *optimize=0, force=0, prefix=None, base_dir=None, verbose=1, dry_run=0, direct=None*])
Python ソースファイル群をバイトコンパイルして '.pyc' または '.pyo' ファイルを同じディレクトリに作成します。*py_files* はコンパイルされるファイルのリストです。'.py' でおわっていないファイルはスキップされます。*optimize* は以下のどれかです:

- `0` - 最適化しない ('.pyc' ファイルを作成します)
- `1` - 通常の最適化 ('`python -O`' のように)
- `2` - さらに最適化 ('`python -OO`' のように)

もし *force* が真なら、全てのファイルがタイムスタンプに関係なく再コンパイルされます。

バイトコードファイルにエンコードされるソースファイル名は、デフォルトでは *py_files* が使われます。これを *prefix* と *base_dir* で変更することができます。*prefix* はそれぞれのソースファイル名から削除される文字列で、*base_dir* は (*prefix* を削除したあと) 先頭に追加されるディレクトリ名です。任意に *prefix* と *base_dir* のどちらか、両方を与える (与えない) ことができます。

もし *dry_run* が真なら、ファイルシステムに影響することは何もされません。

バイトコンパイルは現在のインタプリタプロセスによって標準の `py_compile` モジュールを使って直接行われるか、テンポラリスクリプトを書いて間接的に行われます。通常は `byte_compile()` に直接かそうでないかをまかせます (詳細についてはソースをごらんください)。*direct* フラグは関節モードで作成されたスクリプトで使用されます。何をやっているか理解していない時は `None` のままにしておいてください。

rfc822_escape(*header*)
RFC 822 ヘッダに含められるよう加工した *header* を返します。改行のあとには 8 つのスペースが追加されます。この関数は文字列に他の変更はしません。

37 `distutils.dist` — Distribution クラス

このモジュールは `Distribution` クラスを提供します。これは構築/インストール/配布される配布物をあらわします。

38 `distutils.extension` — Extension クラス

このモジュールは `Extension` クラスを提供します。C/C++拡張モジュールをセットアップスクリプトで表すために使われます。

39 `distutils.debug` — Distutils デバッグモード

このモジュールは DEBUG フラグを提供します。

40 distutils.errors — Distutils 例外

distutils のモジュールで使用される例外を提供します。distutils のモジュールは標準的な例外を起こします。特に、 SystemExit はエンドユーザによる失敗 (コマンドライン引数の間違いなど) で起きます。

このモジュールは 'from ... import *' で安全に使用することができます。このモジュールは Distutils ではじまり、Error で終わるシンボルしか export しません。

41 distutils.fancy_getopt — 標準 getopt モジュールのラッパ

このモジュールは以下の機能を標準の getopt モジュールに追加するラッパを提供します:

- 短いオプションと長いオプションを関連づけます
- オプションはヘルプ文字列を持ちます。可能性としては fancy_getopt に完全な利用方法サマリを作らせることができます。
- オプションは渡されたオブジェクトの属性を設定します。
- 真偽値をとるオプションは "負のエイリアス" を持ちます。— たとえば **--quiet** の "負のエイリアス" が**--verbose** の場合、コマンドラインで **--quiet** を指定すると *verbose* は偽になります。

fancy_getopt(*options, negative_opt, object, args*)
ラッパ関数。*options* は FancyGetopt のコンストラクタで説明されている '(long_option, short_option, help_string)' の 3 要素タプルのリストです。*negative_opt* はオプション名からオプション名のマッピングになっている辞書で、キー、値のどちらも *options* リストに含まれている必要があります。*object* は値を保存するオブジェクト (FancyGetopt クラスの getopt() メソッドを参照してください) です。*args* は引数のリストです。*args* として None を渡すと、sys.argv[1:] が使われます。

wrap_text(*text, width*)
text を *width* 以下の幅で折り返します。

警告: textwrap で置き換えられるべき (Python 2.3 以降で利用可能)。

class **FancyGetopt**([*option_table*=*None*])
option_table は 3 つ組タプルのリストです。'(long_option, short_option, help_string)'

もしオプションが引数を持つなら、*long_option* に'=' を追加する必要があります。*short_option* は一文字のみで、':' はどの場合にも不要です。*long_option* に対応する *short_option* がない場合、*short_option* は None にしてください。全てのオプションタプルは長い形式のオプションを持つ必要があります。

FancyGetopt クラスは以下のメソッドを提供します:

getopt([*args*=*None*, *object*=*None*])
args のコマンドラインオプションを解析します。*object* に属性として保存します。

もし *args* が None もしくは与えられない場合には、sys.argv[1:] を使います。もし *object* が None もしくは与えられない場合には、新しく OptionDummy インスタンスを作成し、オプションの値を保存したのち '(args, object)' のタプルを返します。もし *object* が提供されていれば、その場で変更され、getopt() は *args* のみを返します。どちらのケースでも、返された *args* は渡された *args* リスト (これは変更されません) の変更されたコピーです。

get_option_order()
直前に実行された getopt() が処理した '(option, value)' タプルのリストを返します。getopt() がまだ呼ばれていない場合には RuntimeError を起こします。

generate_help([*header*=*None*])
この FancyGetopt オブジェクトのオプションテーブルからヘルプテキスト (出力の一行に対応する文字列のリスト) を生成します。

もし与えられていれば、*header* をヘルプの先頭に出力します。

42 distutils.filelist — FileList クラス

このモジュールはファイルシステムを見て、ファイルのリストを構築するために使われる FileList クラスを提供します。

43 distutils.log — シンプルな PEP 282 スタイルのロギング

警告: 標準の logging モジュールに置き換えられるべき

44 distutils.spawn — サブプロセスの生成

このモジュールは spawn() 関数を提供します。これは様々なプラットフォーム依存の他プログラムをサブプロセスとして実行する関数に対するフロントエンドになっています。与えられた実行ファイルの名前からパスを探索する find_executable() 関数も提供しています。

45 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シーケンスになります。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。

46 distutils.text_file — TextFile クラス

このモジュールは TextFile クラスを提供します。これはテキストファイルへのインタフェースを提供し、コメントの削除、空行の無視、バックスラッシュでの行の連結を任意に行えます。

class TextFile([*filename*=*None*, *file*=*None*, ***options*])
このクラスはファイルのようなオブジェクトを提供します。これは行指向のテキストファイルを処理する時に共通して必要となる処理を行います: (#がコメント文字なら) コメントの削除、空行のスキップ、(行末のバックスラッシュでの) 改行のエスケープによる行の連結、先頭/末尾の空白文字の削除。これらは全て独立して任意に設定できます。

クラスは warn() メソッドを提供しており、物理行つきの警告メッセージを生成することができます。この物理行は論理行が複数の物理行にまたがっていても大丈夫です。また unreadline() メソッドが一行先読みを実装するために提供されています。

TextFile のインスタンスは *filename*、*file*、またはその両方をとって作成されます。両方が None の場合 RuntimeError が起きます。*filename* は文字列、*file* はファイルオブジェクト (または readline() と close() のメソッドを提供する何か) である必要があります。TextFile が生成する警告メッセージに含めることができるので、*filename* を与えることが推奨されます、もし *file* が提供されなければ、TextFile は組み込みの *open()* を利用して自分で作成します。

オプションは全て真偽値で、*readline()* で返される値に影響します。

オプション名	説明
strip_comments	バックスラッシュでエスケープされていない限り、'#' から行末までと、'#' の先にある空白文字の
lstrip_ws	行を返す前に先頭の空白文字の並びを削除します。
rstrip_ws	行を返す前に行末の空白文字 (改行文字を含みます!) の並びを削除します。
skip_blanks	コメントと空白を除いた*あとで*内容がない行をスキップします。(もし lstrip_ws と rstrip_ws が
join_lines	もしコメントと空白文字を削除したあとで、バックスラッシュが最後の改行文字でない文字なら、
collapse_join	前の行と接続するとき、行頭の空白文字を削除します。'(join_lines and not lstrip_ws

rstrip_ws は行末の改行を削除するので、readline() のセマンティクスが組み込みファイルオブジェクトの readline() メソッドとは変わってしまいます! 特に、 *rstrip_ws* が真で *skip_blanks* が偽のとき、readline() はファイルの終端で None を返し、空文字列を返したときは空行 (または全て空白文字の行) です。

open(*filename*)
新しいファイル *filename* を開きます。これはコンストラクタ引数の *file* と *filename* を上書きします。

close()
現在のファイルを閉じ、(ファイル名や現在の行番号を含め) 現在のファイルについての情報を全て消します。

warn(*msg*[,*line*=*None*])
標準エラー出力に現在のファイルの論理行に結びついた警告メッセージを出力します。もし現在の論理行が複数の物理行に対応するなら、警告メッセージは以下のように全体を参照します: '"lines 3-5"'。 もし *line* が与えられていれば、現在の行番号を上書きします; 物理行のレンジをあらわすリストまたはタプル、もしくはある物理行をあらわす整数のどれでも与えられます。

readline()
現在のファイル (または unreadline() で"unread" を直前に行っていればバッファ) から論理行を 1 行読み込んで返します。もし *join_lines* オプションが真なら、このメソッドは複数の物理行を読み込んで接続した文字列を返します。現在の行番号を更新します。そのため readline() のあとに warn() を呼ぶと丁度読んだ行についての警告を出します。*rstrip_ws* が真で、*strip_blanks* が偽のとき空文字列が返るので、ファイルの終端では None を返します。

readlines()
現在のファイルで残っている全ての論理行のリストを読み込んで返します。行番号を、ファイルの最後の行に更新します。

unreadline(*line*)
line(文字列) を次の readline() 用に、内部バッファに push します。行の先読みを必要とするパーサを実装する時に便利です。unreadline で "unread" された行は readline で読み込む際に再度処理 (空白の除去など) されません。もし unreadlinee を、readline を呼ぶ前に複数回実行すると、最後に unread した行から返されます。

47 distutils.version — バージョン番号クラス

48 distutils.cmd — Distutils コマンドの抽象クラス

このモジュールは抽象ベースクラス Command を提供します。

class Command(*dist*)
コマンドクラスを定義するための抽象ベースクラス — distutils の「働きバチ」— です。コマンドクラスは *options* とよばれるローカル変数を持ったサブルーチンと考えることができます。オプションは initialize_options() で宣言され、finalize_options() で定義さ (最終的な値を与えら) れます。どちらも全てのコマンドクラスで実装する必要があります。この 2 つの区別は必要です。なぜならオプションの値は外部 (コマンドライン、設定ファイルなど) から来るかもしれず、他のオプションに依存しているオプションは外部の影響を処理した後で計算される必要があるからです。そのため finalize_options() が存在します。サブルーチンの本体は全ての処理をオプションの値にもとづいて行う run() メソッドで、これも全てのコマンドクラスで実装される必要があります。

クラスのコンストラクタは Distribution のインスタンスである単一の引数 *dist* をとります。

69　新しい Distutils コマンドの作成

このセクションでは Distutils の新しいコマンドを作成する手順の概要をしめします。

新しいコマンドは `distutils.command` パッケージ中のモジュールに作られます。'command_template' というディレクトリにサンプルのテンプレートがあります。このファイルを実装しようとしているコマンドと同名の新しいモジュールにコピーしてください。このモジュールはモジュール (とコマンド) と同じ名前のクラスを実装する必要があります。そのため、`peel_banana` コマンド (ユーザは '`setup.py peel_banana`' と実行できます) を実装する際には、'command_template' を 'distutils/command/peel_banana.py' にコピーし、`distutils.cmd.Command` のサブクラス `peel_banana` クラスを実装するように編集してください。

`Command` のサブクラスは以下のメソッドを実装する必要があります。

initialize_options()（こ）
: のコマンドがサポートする全てのオプションのデフォルト値を設定します。これらのデフォルトは他のコマンドやセットアップスクリプト、設定ファイル、コマンドラインによって上書きされるかもしれません。そのためオプション間の依存関係を記述するには適切な場所ではありません。一般的に `initialize_options()` は単に '`self.foo = None`' のような定義だけを行います。

finalize_options()
: このコマンドがサポートする全てのオプションの最終的な値を設定します。これは可能な限り遅く呼び出されます。つまりコマンドラインや他のコマンドによるオプションの代入のあとに呼び出されます。そのため、オプション間の依存関係を記述するのに適した場所です。もし *foo* が *bar* に依存しており、かつまだ *foo* が `initialize_options()` で定義された値のままなら、*foo* を *bar* から代入しても安全です。

run()
: コマンドの本体です。実行するべきアクションを実装しています。`initialize_options()` で初期化され、他のコマンドされ、セットアップスクリプト、コマンドライン、設定ファイルでカスタマイズされ、`finalize_options()` で設定されたオプションがアクションを制御します。端末への出力とファイルシステムとのやりとりは全て `run()` が行います。

sub_commands は コマンドの "ファミリー" を定式化したものです。たとえば `install` は サブコマンド `install_lib`、`install_headers` などの親です。コマンドファミリーの親は *sub_commands* をクラス属性として持ちます。2 要素のタプル '`(command_name, predicate)`' のリストで、*command_name* には文字列、*predicate* には親コマンドのメソッドで、現在の状況がコマンド実行にふさわしいかどうか判断するものを指定します。(例えば `install_headers` はインストールするべき C ヘッダファイルがある時だけ有効です。) もし *predicate* が None なら、そのコマンドは常に有効になります。

sub_commands は 通常クラスの最後で定義されます。これは predicate は bound されていないメソッドになるので、全て先に定義されている必要があるためです。

標準的な例は `install` コマンドです。

Module Index

D

Index

T

U

W

日本語訳について

69.1 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。

69.2 翻訳者

2.3.3 和訳 Yasushi Masuda <y.masuda at acm.org> (March 12, 2004) 2.4 差分訳 Kazuo Moriwaka (May 8, 2006)