Python モジュールのインストール

リリース *2.4*

Greg Ward
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、エンドユーザの視点に立ち、サードパーティ製のモジュールや拡張モジュールの構築やインストールによって標準の Python に機能を追加する方法について述べます。

Contents

1 はじめに

Python の広範な標準ライブラリは、プログラミングにおける多くの要求をカバーしていますが、時には何らかの新たな機能をサードパーティ製モジュールの形で追加する必要に迫られます。自分がプログラムを書くときのサポートとして必要な場合もあるし、自分が使いたいアプリケーションがたまたま Python で書かれていて、そのサポートとして必要な場合もあるでしょう。

以前は、すでにインストール済みの Python に対して、サードパーティ製モジュールを追加するためのサポートはほとんどありませんでした。しかし Python 配布ユーティリティ (Python Distribution Utilities, 略して Distutils) が Python 2.0 から取り入れられ、状況は変わりました。

このドキュメントが主要な対象としているのは、サードパーティモジュールをインストールする必要がある人たち: 単に何らかの Python アプリケーションを稼動させたいだけのエンドユーザやシステム管理者、そしてすでに Python プログラマであって、新たな道具を自分のツールキットに加えたいと思っている人たちです。このドキュメントを読むために、 Python について知っておく必要はありません; インストールしたモジュールを調べるために Python の対話モードにちょっとだけ手を出す必要がありますが、それだけです。自作の Python モジュールを他人が使えるようにするために配布する方法を探しているのなら、*Python* **モジュールの配布** マニュアルを参照してください。

1.1 もっとも簡単な場合: ありふれたインストール作業

最も楽なのは、インストールしたいモジュール配布物の特殊なバージョンをインストールしたいプラットフォーム向けに誰かがすでに用意してくれていて、他のアプリケーションと同じようにインストールするだけであるような場合です。例えば Windows ユーザ向けには実行可能形式のインストーラ、RPM ベースの Linux システム (Red Hat, SuSE, Mandrake その他多数) 向けには RPM パッケージ、 Debian ベースの Linux システム向けには Debian パッケージといった具合に、モジュール開発者はビルド済み配布物を作成しているかもしれません。

このような場合、自分のプラットフォームに合ったインストーラをダウンロードして、実行可能形式なら実行し、RPM なら `rpm -install` するといった、分かりきった作業をするだけです。Python を起動したり、 setup スクリプトを実行する必要はなく、何もコンパイルする必要はありません — 説明書きを読む必要すら全くないかもしれません (とはいえ、説明書きを読むのはよいことですが)。

もちろん、いつもこう簡単とは限りません。自分のプラットフォーム向けのお手軽なインストーラがないモジュール配布物に興味を持つこともあるでしょう。そんな場合には、モジュールの作者やメンテナがリリースしているソース配布物から作業をはじめねばなりません。ソース配布物からのインストールは、モジュールが標準的な方法でパッケージ化されている限りさほど大変ではありません。このドキュメントの大部分は、標準的なソース配布物からのビルドとインストールに関するものです。

1.2 新しい標準: Distutils

モジュールのソースコード配布物をダウンロードしたら、配布物が標準のやり方、すなわち Distutils のやり方に従ってパッケージされて配布されているかどうかすぐに分かります。Distutils の場合、まず配布物の名前とバージョン番号が、例えば 'foo-1.0.tar.gz' や 'widget-0.9.7.zip' のように、ダウンロードされたアーカイブファイルの名前にはっきりと反映されます。次に、アーカイブは同様の名前のディレクトリ、例えば 'foo-1.0' や 'widget-0.9.7' に展開されます。さらに、配布物には setup スクリプト 'setup.py' が入っています。また、'README.txt' 場合によっては 'README' という名前のファイルも入っていて、そこには、モジュール配布物の構築とインストールは簡単で、

```
python setup.py install
```

とするだけだ、という説明が書かれているはずです。

上記の全てが当てはまるなら、ダウンロードしたものをビルドしてインストールする方法はすでに知っていることになります: 上記のコマンドを実行するだけです。非標準の方法でインストールを行ったり、ビルドプロセスをカスタマイズ行いたいのでない限り、このマニュアルは必要ありません。別の言葉で言えば、上のコマンドこそが、このマニュアルから習得すべき全てということになります。

2　標準的なビルド・インストール作業

1.2 節で述べたよいうに、 Distutils を使ったモジュール配布物のビルドとインストールは、通常は単純なコマンド:

```
python setup.py install
```

で行います。

UNIX では、このコマンドをシェルプロンプトで行います; Windows では、コマンドプロンプトウィンドウ (“DOS ボックス”) を開いて、そこで行います; Mac OS の場合、作業はもう少しだけ複雑です (下記参照)

2.1　プラットフォームによる違い

setup コマンドは常に配布物ルートディレクトリ、すなわちモジュールのソース配布物を展開した際のトップレベルのサブディレクトリ内で実行しなければなりません。例えば、あるモジュールのソース配布物 ‘foo-1.0.tar.gz’ を UNIX システム上にダウンロードしたなら、通常は以下の操作を行います:

```
gunzip -c foo-1.0.tar.gz | tar xf -    # unpacks into directory foo-1.0
cd foo-1.0
python setup.py install
```

Windows では、おそらく ‘foo-1.0.zip’ をダウンロードしているでしょう。アーカイブファイルを ‘C:\Temp’ にダウンロードしたのなら、(WinZip のような) グラフィカルユーザインタフェースつきのアーカイブ操作ソフトや、 (**unzip** や **pkunzip** のような) コマンドラインツールを使ってアーカイブを展開します。次に、コマンドプロンプトウィンドウ (“DOS ボックス”) を開いて、以下を実行します:

```
cd c:\Temp\foo-1.0
python setup.py install
```

2.2　ビルド作業とインストール作業を分割する

`setup.py install` を実行すると、一度の実行で全てのモジュールをビルドしてインストールします。段階的に作業をしたい場合 — ビルドプロセスをカスタマイズしたり、作業がうまくいかない場合に特に便利です — には、setup スクリプトに一度に一つづつ作業を行わせるようにできます。この機能は、ビルドとインストールを異なるユーザで行う場合にも便利です — 例えば、モジュール配布物をビルドしておいてシステム管理者に渡して (または、自分でスーパユーザになって) 、インストールしたくなるかもしれません.

最初のステップでは全てをビルドしておき、次のステップで全てをインストールするには、 setup スクリプトを二度起動します:

```
python setup.py build
python setup.py install
```

この作業を行ってみれば、`install` コマンドを実行するとまず `build` コマンドを実行し、さらに — この場合には — ‘build’ ディレクトリの中が全て最新の状態になっているので、`build` は何も行わなくてよいと判断していることがわかるでしょう。

インターネットからダウンロードしたモジュールをインストールしたいだけなら、上のように作業を分割する機能は必要ないかもしれませんが、この機能はより進んだ使い方をする際にはとても便利です。自作の Python モジュールや拡張モジュールを配布することになれば、個々の Distutils コマンドを自分で何度も実行することになるでしょう。

2.3 ビルドの仕組み

上で示唆したように、 `build` コマンドは、インストールすべきファイルを **ビルドディレクトリ** *(build directory)* に置く働きがあります。デフォルトでは、ビルドディレクトリは配布物ルート下の 'build' になります; システムの処理速度に強いこだわりがあったり、ソースツリーに指一本触れたくないのなら、**--build-base** オプションを使ってビルドディレクトリを変更できます。例えば:

```
python setup.py build --build-base=/tmp/pybuild/foo-1.0
```

(あるいは、システム全体向け、あるいは個人用の Distutils 設定ファイルにディレクティブを書いて、永続的に設定を変えられます; 5 節を参照してください。) 通常は必要ない作業です。

ビルドツリーのデフォルトのレイアウトは以下のようになっています:

```
--- build/ --- lib/
または
--- build/ --- lib.<plat>/
               temp.<plat>/
```

`<plat>` は、現在の OS/ハードウェアプラットフォームと Python のバージョンを記述する短い文字列に展開されます。第一の 'lib' ディレクトリだけの形式は、 "pure モジュール配布物" — すなわち、pure Python モジュールだけの入ったモジュール配布物 — の場合に使われます。モジュール配布物に何らかの拡張モジュール (C/C++ で書かれたモジュール) が入っている場合、第二の `<plat>` 付きディレクトリが二つある形式が使われます。この場合、'temp.*plat*' ディレクトリには、コンパイル/リンク過程で生成され、実際にはインストールされない一時ファイルが収められます。どちらの場合にも、'lib' (または 'lib.*plat*') ディレクトリには、最終的にインストールされることになる全ての Python モジュール (pure Python モジュールおよび拡張モジュール) が入ります。

今後、 Python スクリプト、ドキュメント、バイナリ実行可能形式、その他 Python モジュールやアプリケーションのインストール作業に必要なディレクトリが追加されるかもしれません。

2.4 インストールの仕組み

`build` コマンドを実行した後 (明示的に実行した場合も、`install` コマンドが代わりに実行してくれた場合も) は、`install` コマンドの仕事は比較的単純なもの: 'build/lib' (または 'build/lib.*plat*') の下にあるもの全ての指定したインストールディレクトリへのコピー、になります。

インストールディレクトリを選ばなかった場合 — すなわち、`setup.py install` を実行しただけの場合 — には、`install` コマンドはサードパーティ製 Python モジュールを置くための標準の場所にインストールを行います。この場所は、プラットフォームや、Python 自体をどのようにビルド/インストールしたかで変わります。 UNIX や Mac OS では、インストールしようとするモジュール配布物が pure Python なのか、拡張モジュールを含む ("非 pure") のかによっても異なります:

プラットフォーム	標準のインストール場所	デフォルト値	注記
UNIX (pure)	*prefix*/lib/python2.4/site-packages	/usr/local/lib/python2.4/site-packages	(1)
UNIX (非 pure)	*exec-prefix*/lib/python2.4/site-packages	/usr/local/lib/python2.4/site-packages	(1)
Windows	*prefix*	C:\Python	(2)
Mac OS (pure)	*prefix*:Lib:site-packages	Python:Lib:site-packages	
Mac OS (非 pure)	*prefix*:Lib:site-packages	Python:Lib:site-packages	

注記:

(1) ほとんどの Linux ディストリビューションには、システムの標準インストール物として Python が入っているので、 Linux では普通、*prefix* や *exec-prefix* はどちらも '/usr' になります。 Linux (または UNIX ライクなシステム) 上で自分で Python をビルドした場合、デフォルトの *prefix* および *exec-prefix* は '/usr/local' になります。

(2) Windows での Python のデフォルトインストールディレクトリは、Python 1.6a1、 1.5.2、およびそれ以前のバージョンでは 'C:\Program Files\Python' です。

prefix および *exec-prefix* は、 Python がインストールされているディレクトリと、実行時にライブラリを探しにいく場所を表します。これらのディレクトリは、Windows と Mac OS では常に同じで、 UNIX でもほぼ常に同じです。自分の Python がどんな *prefix* や *exec-prefix* を使っているかは、Python を対話モードで起動して、単純なコマンドをいくつか入力すればわかります。Windows では、スタート ＞ (すべての) プログラム ＞ Python 2.4 ＞ Python (command line) を選びます。Mac OS 9 では、 'PythonInterpreter' を起動します。インタプリタを起動すると、プロンプトに Python コードを入力できます。例えば、作者の使っている Linux システムで、三つの Python 文を以下のように入力すると、出力から作者のシステムの *prefix* と *exec-prefix* を得られます:

```
Python 2.4 (#26, Aug  7 2004, 17:19:02)
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.prefix
'/usr'
>>> sys.exec_prefix
'/usr'
```

モジュールを標準の場所にインストールしたくない場合や、標準の場所にインストールするためのファイル権限を持っていない場合、 3 節にある、別の場所へのインストール方法の説明を読んでください。インストール先のディレクトリを大幅にカスタマイズしければ、 4 節のカスタムインストールに関する説明を読んでください。

3 別の場所へのインストール

しばしば、サードパーティ製 Python モジュールをインストールするための標準の場所以外にモジュールをインストールしなければならなかったり、単にそうしたくなるときがあります。例えば UNIX システムでは、標準のサードパーティ製モジュールディレクトリに対する書き込み権限を持っていないかもしれません。または、あるモジュールを、ローカルで使っている Python に標準のモジュールの一部として組み込む前にテストしてみたいと思うかもしれません。既存の配布物をアップグレードする際には特にそうでしょう: 実際にアップグレードする前に、既存のスクリプトの基本となる部分が新たなバージョンでも動作するか確認したいはずです。

Distutils の `install` コマンドは、別の場所へ配布物をインストールする作業を単純で苦労のない作業にするように設計されています。基本的なアイデアは、インストール先のベースディレクトリを指定しておき、`install` コマンドがそのベースディレクトリ下にファイル群をインストールするための一連のディレクトリ (**インストールスキーム**: *installation scheme*) を作成するというものです。詳細はプラットフォームによって異なるので、以下の節から自分のプラットフォームに当てはまるものを読んでください。

3.1 別の場所へのインストール: home スキーム

"home スキーム" の背後にある考え方は、Python モジュールを個人用のモジュール置き場でビルドし、維持するというものです。このスキームの名前は UNIX の「ホーム」ディレクトリの概念からとりました。というのも、 UNIX のユーザにとって、自分のホームディレクトリを '/usr/' や '/usr/local/' のようにレイアウトするのはよくあることだからです。とはいえ、このスキームはどのオペレーティングシステムのユーザでも使えます。新たなモジュールのインストールは単純で、

```
python setup.py install --home=<dir>
```

のようにします。このとき、**--home** オプションを使ってディレクトリを指定します。面倒臭がりの人は、単にチルダ (~) をタイプするだけでかまいません; `install` コマンドがチルダをホームディレクトリに展開してくれます:

```
python setup.py install --home=~
```

--home オプションは、インストールのベースディレクトリを指定します。ファイルはインストールベース下の以下のディレクトリに保存されます:

ファイルの種類	インストール先ディレクトリ	オーバライドするためのオプション
pure モジュール配布物	*home*/lib/python	**--install-purelib**
非 pure モジュール配布物	*home*/lib/python	**--install-platlib**
スクリプト	*home*/bin	**--install-scripts**
データ	*home*/share	**--install-data**

2.4 で変更された仕様: **--home** は UNIX でしかサポートされていませんでした

3.2 別の場所へのインストール: UNIX (prefix スキーム)

あるインストール済みの Python を使ってモジュールのビルド/インストールを (例えば setup スクリプトを実行して) 行いたいけれども、別のインストール済みの Python のサードパーティ製モジュール置き場 (あるいは、そう見えるようなディレクトリ構造) に、ビルドされたモジュールをインストールしたい場合には、“prefix スキーム” が便利です。そんな作業はまったくありえそうにない、と思うなら、確かにその通りです — “home スキーム” を先に説明したのもそのためです。とはいえ、prefix スキームが有用なケースは少なくとも二つあります。

まず、多くの Linux ディストリビューションは、 Python を ‘/usr/local’ ではなく ‘/usr’ に置いていることを考えてください。この場合は、Python はローカルの計算機ごとのアドオン (add-on) ではなく、“システム” の一部となっているので、‘/usr’ に置くのは全く正当なことです。しかしながら、 Python モジュールをソースコードからインストールしていると、モジュールを ‘/usr/lib/python2.*X*’ ではなく ‘/usr/local/lib/python2.*X*’ に置きたいと思うかもしれません。これを行うには

```
/usr/bin/python setup.py install --prefix=/usr/local
```

と指定します。

もう一つありえるのは、ネットワークファイルシステムにおいて、遠隔のディレクトリに対する読み出しと書き込みの際に違う名前を使う場合です。例えば、 ‘/usr/local/bin/python’ でアクセスするような Python インタプリタは、 ‘/usr/local/lib/python2.*X*’ からモジュールを探すでしょうが、モジュールは別の場所、例えば ‘/mnt/*@server*/export/lib/python2.*X*’ にインストールしなければならないかもしれません。この場合には、

```
/usr/local/bin/python setup.py install --prefix=/mnt/@server/export
```

のようにします。

どちらの場合も、**--prefix** オプションでインストールベースディレクトリを決め、 **--exec-prefix** でプラットフォーム固有のファイル置き場名として使う、プラットフォーム固有インストールベースディレクトリを決めます。(プラットフォーム固有のファイルとは、現状では単に非 pure モジュール配布物のことを意味しますが、 C ライブラリやバイナリ実行可能形式などに拡張されるかもしれません。) **--exec-prefix** が指定されていなければ、デフォルトの **--prefix** になります。ファイルは以下のようにインストールされます:

ファイルの種類	インストール先ディレクトリ	オーバライドするためのオプション
pure モジュール配布物	*prefix*/lib/python2.*X*/site-packages	**--install-purelib**
非 pure モジュール配布物	*exec-prefix*/lib/python2.*X*/site-packages	**--install-platlib**
スクリプト	*prefix*/bin	**--install-scripts**
データ	*prefix*/share	**--install-data**

--prefix や **--exec-prefix** が実際に他のインストール済み Python の場所を指している必要はありません; 上に挙げたディレクトリがまだ存在しなければ、インストール時に作成されます。

ちなみに、prefix スキームが重要な本当の理由は、単に標準の UNIX インストールが prefix スキームを使っているからです。ただし、そのときには、 **--prefix** や **--exec-prefix** は Python 自体が `sys.prefix` や `sys.exec_prefix` を使って決めます。というわけで、読者は prefix スキームを決して使うことはあるまいと思っているかもしれませんが、`python setup.py install` をオプションを何もつけずに実行していれば、常に prefix スキームを使っていることになるのです。

拡張モジュールを別のインストール済み Python にインストールしても、拡張モジュールのビルド方法による影響を受けることはありません: 特に、拡張モジュールをコンパイルする際には、setup スクリプトを実行する際に使う Python インタプリタと一緒にインストールされている Python ヘッダファイル (‘Python.h’

とその仲間たち) を使います。上で述べてきたやり方でインストールされた拡張モジュールを実行するインタプリタと、インタプリタをビルドする際に用いた別のインタプリタとの互換性を保証するのはユーザの責任です。

これを行うには、二つのインタプリタが同じバージョンの Python (ビルドが違っていたり、同じビルドのコピーということもあり得ます) であるかどうかを確かめます。(もちろん、**--prefix** や**--exec-prefix** が別のインストール済み Python の場所すら指していなければどうにもなりません。)

3.3 別の場所へのインストール (prefix を使う方法): Windows

Windows はユーザのホームディレクトリという概念がなく、Windows 環境下で標準的にインストールされた Python は UNIX よりも単純な構成をしているので、 Windows で追加のパッケージを別の場所に入れる場合には、伝統的に **--prefix** が使われてきました。

```
python setup.py install --prefix="\Temp\Python"
```

とすると、モジュールを現在のドライブの '\Temp\Python' ディレクトリにインストールします

インストールベースディレクトリは、 **--prefix** オプションだけで決まります; **--exec-prefix** オプションは、Windows ではサポートされていません。ファイルは以下のような構成でインストールされます:

ファイルの種類	インストール先ディレクトリ	オーバライドするためのオプション
pure モジュール配布物	*prefix*	**--install-purelib**
非 pure モジュール配布物	*prefix*	**--install-platlib**
スクリプト	*prefix*\Scripts	**--install-scripts**
データ	*prefix*\Data	**--install-data**

4 カスタムのインストール

たまに、 3 節で述べたような別の場所へのインストールスキームが、自分のやりたいインストール方法と違うことがあります。もしかすると、同じベースディレクトリ下にあるディレクトリのうち、一つか二つだけをいじりたかったり、インストールスキームを完全に再定義したいと思うかもしれません。どちらの場合にせよ、こうした操作では**カスタムのインストールスキーム** を作成することになります。

別の場所へのインストールスキームに関するこれまでの説明で、"オーバライドするためのオプション" というコラムにお気づきかもしれません。このオプションは、カスタムのインストールスキームを定義するための手段です。各オーバライドオプションには、相対パスを指定しても、絶対パスを指定しても、インストールベースディレクトリのいずれかを明示的に指定してもかまいません。(インストールベースディレクトリは二種類あり、それら二つは通常は同じディレクトリです — UNIX の "prefix スキーム" を使っていて、**--prefix** と **--exec-prefix** オプションを使っているときだけ異なります。)

例えば、 UNIX 環境でモジュール配布物をホームディレクトリにインストールしたい — とはいえ、スクリプトは '~/bin' ではなく '~/scripts' に置きたい — とします。ご想像の通り、スクリプトを置くディレクトリは、**--install-scripts** オプションで上書きできます; この場合は相対パスで指定もでき、インストールベースディレクトリ (この場合にはホームディレクトリ) からの相対パスとして解釈されます:

```
python setup.py install --home=~ --install-scripts=scripts
```

UNIX 環境での例をもう一つ紹介します: インストール済みの Python が、'/usr/local/python' を prefix にしてビルドされ、インストールされていて、標準のインストールスクリプトは '/usr/local/python/bin' に入るようになっているとします。 '/usr/local/bin' に入るようにしたければ、絶対パスを **--install-scripts** オプションに与えて上書きすることになるでしょう:

```
python setup.py install --install-scripts=/usr/local/bin
```

(この操作を行うと、 "prefix スキーム" を使ったインストールになり、 prefix は Python インタプリタがインストールされている場所— この場合には '/usr/local/python' になります。)

Windows 用の Python を管理しているのなら、サードパーティ製モジュールを *prefix* そのものの下ではな

く、 *prefix* の下にあるサブディレクトリに置きたいと考えるかもしれません。この作業は、インストールディレクトリのカスタマイズとほぼ同じくらい簡単です — 覚えておかねばならないのは、モジュールには二つのタイプ、pure モジュールと非 pure モジュール (非 pure モジュール配布物内のモジュール) があるということです。例えば以下のようにします:

```
python setup.py install --install-purelib=Site --install-platlib=Site
```

指定したインストール先ディレクトリは、*prefix* からの相対です。もちろん、*prefix* を '.pth' ファイルに入れるなどして、これらのディレクトリが Python のモジュール検索パス内に入るようにしなければなりません。 Python のモジュール検索パスを修正する方法は、 4.1 節を参照してください。

インストールスキーム全体を定義したいのなら、全てのインストールディレクトリオプションを指定しなければなりません。この作業には、相対パスを使った指定を勧めます; 例えば、全ての Python モジュール関連ファイルをホームディレクトリ下の 'python' ディレクトリの下に置き、そのホームディレクトリをマウントしている各プラットフォームごとに別のディレクトリを置きたければ、以下のようにインストールスキームを定義します:

```
python setup.py install --home=~ \
                        --install-purelib=python/lib \
                        --install-platlib=python/lib.$PLAT \
                        --install-scripts=python/scripts
                        --install-data=python/data
```

また、以下のようにも指定できます:

```
python setup.py install --home=~/python \
                        --install-purelib=lib \
                        --install-platlib='lib.$PLAT' \
                        --install-scripts=scripts
                        --install-data=data
```

`$PLAT` は、(必ずしも) 環境変数ではありません — この表記は、 Distutils がコマンドラインオプションの解釈と同じやり方で展開します。設定ファイルを解釈する際と同じです。

言うまでもないことですが、毎回新たなモジュール配布物をインストールする度にインストールスキーム全体の指定を行っていては面倒です。そこで、オプションは Distutils 設定ファイル (5 参照) にも指定できます:

```
[install]
install-base=$HOME
install-purelib=python/lib
install-platlib=python/lib.$PLAT
install-scripts=python/scripts
install-data=python/data
```

あるいは、以下のようにも指定できます:

```
[install]
install-base=$HOME/python
install-purelib=lib
install-platlib=lib.$PLAT
install-scripts=scripts
install-data=data
```

これら二つは、 setup スクリプトを異なるインストールベースディレクトリから実行した場合には同じには **ならない** ので注意してください。例えば、

```
python setup.py --install-base=/tmp
```

とすると、最初の書き方では pure モジュールが*/tmp/python/lib* に入り、二番目の書き方では */tmp/lib* に入ります。(二番目のケースでは、インストールベースを '/tmp/python' に指定しようと考えるでしょう。)

読者は、設定ファイル例で、入力値に `$HOME` や `$PLAT` を使っていることに気づいているかもしれませんね。これらは Distutils の設定変数で、環境変数を彷彿とさせます。実際、この表記が使えるプラットフォーム上では、設定ファイル中に環境変数を入れられますが、Distutils は他にも、例えば `$PLAT` のようにおそらくユーザの環境中にないような変数をいくつか持っています。(そしてもちろん、Mac OS 9 のような環境変数のないシステムでは、設定ファイル中で使える変数は Distutils が提供しているものだけです。)

4.1 Python サーチパスの変更

Python インタプリタが `import` 文を実行するとき、インタプリタは Python コードや拡張モジュールをモジュール検索パス中から探します。検索パスのデフォルト値は、インタプリタをビルドする際に Python のバイナリ内に設定されます。検索パスは、`sys` を import して、 `sys.path` を出力すればわかります。

```
$ python
Python 2.2 (#11, Oct  3 2002, 13:31:27)
[GCC 2.96 20000731 (Red Hat Linux 7.3 2.96-112)] on linux2
Type ``help'', ``copyright'', ``credits'' or ``license'' for more information.
>>> import sys
>>> sys.path
['', '/usr/local/lib/python2.3', '/usr/local/lib/python2.3/plat-linux2',
 '/usr/local/lib/python2.3/lib-tk', '/usr/local/lib/python2.3/lib-dynload',
 '/usr/local/lib/python2.3/site-packages']
>>>
```

`sys.path` 内の空文字列は、現在の作業ディレクトリを表します。

ローカルでインストールされるパッケージは、'.../site-packages/' ディレクトリに入るのが決まりですが、ユーザはどこか任意のディレクトリに Python モジュールをインストールしたいと思うかもしれません。例えば、自分のサイトでは、 web サーバに関連する全てのソフトウェアを '/www' に置くという決まりがあるかもしれません。そこで、アドオンの Python モジュールが '/www/python' 置かれることになると、モジュールを import するためにはディレクトリを `sys.path` に追加せねばなりません。ディレクトリを検索パスに追加するには、いくつかの異なる方法が存在します。

最も手軽な方法は、パス設定ファイルをすでに Python の検索パスに含まれるディレクトリ、通常は '.../site-packages/' ディレクトリに置くというものです。パス設定ファイルは拡張子が '.pth' で、ファイルには `sys.path` に追加するパスを一行に一つづつ記述しなければなりません。(新たなパスは今の `sys.path` の後ろに追加されるので、追加されたディレクトリ内にあるモジュールが標準のモジュールセットを上書きすることはありません。つまり、このメカニズムを使って、標準モジュールに対する修正版のインストールはできないということです。)

パスは絶対パスでも相対パスでもよく、相対パスの場合には '.pth' ファイルのあるパスからの相対になります。検索パスにディレクトリが追加されると、今度はそのディレクトリに対して '.pth' ファイルを検索します。詳しくは *site* **モジュールのドキュメント** を読んでください。

やや便利さには欠けますが、Python の標準ライブラリ中にある 'site.py' ファイルを編集することでも、`sys.path` を変更できます。'site.py' は、 **-S** スイッチを与えて抑制しないかぎり、Python インタプリタが実行される際に自動的に import されます。ただし、設定するには、単に 'site.py' を編集して、例えば以下のような二行を加えます:

```
import sys
sys.path.append('/www/python/')
```

しかしながら、(例えば 2.2 から 2.2.2 にアップグレードするときのように) 同じメジャーバージョンの Python を再インストールすると、'site.py' は手持ちのバージョンで上書きされてしまいます。ファイルが変更されていることを覚えておき、インストールを行う前にコピーを忘れずとっておかねばなりません。

また、sys.path を修正できる二つの環境変数があります。PYTHONHOME を使うと、インストールされている Python のプレフィクスを別の値に設定できます。例えば、 PYTHONHOME を '/www/python' に設定すると、検索パスは ['', '/www/python/lib/python2.2/', '/www/python/lib/python2.3/plat-linux2', ...] といった具合になります。

PYTHONPATH を使うと、sys.path の先頭に一連のパスを追加できます。例えば、PYTHONPATH を '/www/python:/opt/py' に設定すると、検索パスは ['/www/python', '/opt/py'] から始まります。(sys.path にディレクトリを追加するには、そのディレクトリが実在しなければなりません; site は実在しないディレクトリを除去します。)

最後に、 sys.path はただの普通の Python のリストなので、どんな Python アプリケーションもエントリを追加したり除去したりといった修正を行えます。

5 Distutils 設定ファイル

上で述べたように、 Distutils 設定ファイルを使えば、任意の Distutils オプションに対して個人的な設定やサイト全体の設定を記録できます。すなわち、任意のコマンドの任意のオプションを二つか三つ (プラットフォームによって異なります) の設定ファイルに保存でき、コマンドラインを解釈する前にオプションを問い合わせさせるようにできます。つまり、設定ファイルはデフォルトの値を上書きし、さらにコマンドライン上で与えた値が設定ファイルの内容を上書きするわけです。さらに、複数の設定ファイルが適用されると、"先に" 適用されたファイルに指定されていた値は "後に" 適用されたファイル内の値で上書きされます。

5.1 設定ファイルの場所と名前

設定ファイルの名前と場所は、非常にわずかですがプラットフォーム間で異なります。UNIX では、三種類の設定ファイルは以下のようになります (処理される順に並んでいます):

設定ファイルのタイプ	場所とファイル名	注記
system	*prefix*/lib/python*ver*/distutils/distutils.cfg	(1)
personal	$HOME/.pydistutils.cfg	(2)
local	setup.cfg	(3)

Windows では設定ファイルは以下のようになります:

設定ファイルのタイプ	場所とファイル名	注記
system	*prefix*\Lib\distutils\distutils.cfg	(4)
personal	%HOME%\pydistutils.cfg	(5)
local	setup.cfg	(3)

Mac OS では以下のようになります:

設定ファイルのタイプ	場所とファイル名	注記
system	*prefix*:Lib:distutils:distutils.cfg	(6)
personal	N/A	
local	setup.cfg	(3)

注記:

(1) 厳密に言えば、システム全体向けの設定ファイルは、 Distutils がインストールされているディレクトリになります; UNIX の Python 1.6 以降では、表の通りの場所になります。 Python 1.5.2 では、 Distutils は通常 '*prefix*/lib/python1.5/site-packages/distutils' にインストールされるため、 Python 1.5.2 では設定ファイルをそこに置かなければなりません。

(2) UNIX では、環境変数 HOME が定義されていない場合、標準モジュール pwd の getpwuid() 関数を使ってユーザのホームディレクトリを決定します。

(3) 現在のディレクトリ (通常は setup スクリプトがある場所) です。

(4) (注記 (1) も参照してください) Python 1.6 およびそれ以降のバージョンでは、 Python のデフォルトの “インストールプレフィクス” は ‘C:\Python’ なので、システム設定ファイルは通常 ‘C:\Python\Lib\distutils\distutils.cfg’ になります。Python 1.5.2 ではデフォルトのプレフィクスは ‘C:\Program Files\Python’ であり、Distutils は標準ライブラリの一部ではありません — 従って、システム設定ファイルは、Windows 用の標準の Python 1.5.2 では ‘C:\Program Files\Python\distutils\distutils.cfg’ になります。

(5) Windows では、環境変数 HOME が設定されていない場合、個人用の設定ファイルはどこにもなく、使われることもありません。(言い換えれば、 Windows での Distutils はホームディレクトリがどこにあるか一切推測しようとしないということです。)

(6) (注釈 (1) と (4) も参照してください) デフォルトのインストールプレフィクスは単に ‘Python:’ なので、Python 1.6 以降では、通常は ‘Python:Lib:distutils:distutils.cfg’ になります。

5.2 設定ファイルの構文

Distutils 設定ファイルは、全て同じ構文をしています。設定ファイルはセクションでグループ分けされています。各 Distutils コマンドごとにセクションがあり、それに加えて全てのコマンドに影響するグローバルオプションを設定するための `global` セクションがあります。各セクションには `option=value` の形で、一行あたり一つのオプションを指定します。

例えば、以下は全てのコマンドに対してデフォルトでメッセージを出さないよう強制するための完全な設定ファイルです:

```
[global]
verbose=0
```

この内容のファイルがシステム全体用の設定ファイルとしてインストールされていれば、そのシステムの全てのユーザによる全ての Python モジュール配布物に対する処理に影響します。ファイルが (個人用の設定をサポートしているシステムで) 個人用の設定ファイルとしてインストールされていれば、そのユーザが処理するモジュール配布物にのみ影響します。この内容を特定のモジュール配布物の ‘setup.cfg’ として使えば、その配布物だけに影響します。

以下のようにして、デフォルトの “ビルドベース” ディレクトリをオーバライドしたり、`build*` コマンドが常に強制的にリビルドを行うようにもできます:

```
[build]
build-base=blib
force=1
```

この設定は、コマンドライン引数の

```
python setup.py build --build-base=blib --force
```

に対応します。ただし、後者ではコマンドライン上で `build` コマンドを含めて、そのコマンドを実行するよう意味しているところが違います。特定のコマンドに対するオプションを設定ファイルに含めると、このような関連付けの必要はなくなります; あるコマンドが実行されると、そのコマンドに対するオプションが適用されます。(また、設定ファイル内からオプションを取得するような他のコマンドを実行した場合、それらのコマンドもまた設定ファイル内の対応するオプションの値を使います。)

あるコマンドに対するオプションの完全なリストは、例えば以下のように、**--help** を使って調べます:

```
python setup.py build --help
```

グローバルオプションの完全なリストを得るには、コマンドを指定せずに**--help** オプションを使います:

```
python setup.py --help
```

“Python モジュールの配布” マニュアルの、“リファレンスマニュアル” の節も参照してください。

6 拡張モジュールのビルド: 小技と豆知識

Distutils は、可能なときにはいつでも、 ‘setup.py’ スクリプトを実行する Python インタプリタが提供する設定情報を使おうとします。例えば、拡張モジュールをコンパイルする際には、コンパイラやリンカのフラグには Python をコンパイルした際と同じものが使われます。通常、この設定はうまくいきますが、状況が複雑になると不適切な設定になることもあります。この節では、通常の Distutils の動作をオーバライドする方法について議論します。

6.1 コンパイラ/リンカのフラグをいじるには

C や C++ で書かれた Python 拡張をコンパイルする際、しばしば特定のライブラリを使ったり、特定の種類のオブジェクトコードを生成したりする上で、コンパイラやリンカに与えるフラグをカスタマイズする必要があります。ある拡張モジュールが自分のプラットフォームではテストされていなかったり、クロスコンパイルを行わねばならない場合にはこれが当てはまります。

最も一般的なケースでは、拡張モジュールの作者はすでに拡張モジュールのコンパイルが複雑になることを見越していて、‘Setup’ ファイルを提供して編集できるようにしています。‘Setup’ ファイルの編集は、モジュール配布物に多くの個別の拡張モジュールがあったり、コンパイラに拡張モジュールをコンパイルさせるために細かくフラグをセットする必要があるような場合にのみ行うことになるでしょう。

‘Setup’ ファイルが存在する場合、ビルドするべき拡張モジュールのリストを得るために解釈されます。‘Setup’ ファイルの各行には単一のモジュールを書きます。各行は以下のような構造をとります:

module `...` [*sourcefile* `...`] [*cpparg* `...`] [*library* `...`]

次に、各フィールドについて見てみましょう。

- *module* はビルドする拡張モジュールの名前で、Python の識別子名として有効でなければなりません。モジュールの名前変更は、このフィールドを変えるだけではできない (ソースコードの編集も必要です) ので、このフィールドに手を加えるべきではありません。
- *sourcefile* は、少なくともファイル名から何の言語で書かれているかがわかるようになっているソースコードファイル名です。‘.c’ で終わるファイルは C で書かれているとみなされ、‘.C’、‘.cc’、および ‘.c++’ で終わるファイルは C++ で書かれているとみなされます。‘.m’ や ‘.mm’ で終わるファイルは Objective C で書かれているとみなされます。
- *cpparg* は C プリプロセッサへの引数で、**-I**、 **-D**、**-U** または **-C** のいずれかから始まる文字列です。
- *library* は ‘.a’ で終わるか、**-l** または **-L** のいずれかから始まる文字列です。

特定のプラットフォームにおいて、プラットフォーム上の特殊なライブラリが必要な場合、‘Setup’ ファイルを編集して `python setup.py build` を実行すればライブラリを追加できます。例えば、以下の行

```
foo foomodule.c
```

で定義されたモジュールを、自分のプラットフォーム上の数学ライブラリ ‘libm.a’ とリンクしなければならない場合、‘Setup’ 内の行に **-lm** を追加するだけです:

```
foo foomodule.c -lm
```

コンパイラやリンカ向けの任意のスイッチオプションは、**-Xcompiler** *arg* や**-Xlinker** *arg* オプションで与えます:

```
foo foomodule.c -Xcompiler -o32 -Xlinker -shared -lm
```

-Xcompiler および **-Xlinker** の後にくるオプションは、それぞれ適切なコマンドラインに追加されます。従っ

て、上の例では、コンパイラには **-o32** オプションが渡され、リンカには **-shared** が渡されます。コンパイラオプションに引数が必要な場合、複数の **-Xcompiler** オプションを与えます; 例えば、 `-x c++` を渡すには、'Setup' ファイルには `-Xcompiler -x -Xcompiler c++` を渡さねばなりません。

コンパイラフラグは、環境変数 CFLAGS の設定でも与えられます。CFLAGS が設定されていれば、'Setup' ファイル内で指定されているコンパイラフラグに CFLAGS の内容が追加されます。

6.2 Windows で非 Microsoft コンパイラを使ってビルドするには

Borland C++

この小節では、 Borland C++ コンパイラのバージョン 5.5 で Distutils を使うために必要な手順について述べています。

まず、 Borland のオブジェクトファイル形式 (OMF) は、Python 公式サイトや ActiveState の Web サイトからダウンロードできるバージョンの Python が使っている形式とは違うことを知っておかねばなりません (Python は通常、 Microsoft Visual C++ でビルドされています。Microsoft Visual C++ は COFF をオブジェクトファイル形式に使います。) このため、以下のようにして、 Python のライブラリ 'python24.lib' を Borland の形式に変換する必要があります:

```
coff2omf python24.lib python20_bcpp.lib
```

'coff2omf' プログラムは、 Borland コンパイラに付属しています。'python24.lib' は Python インストールディレクトリの 'Libs' ディレクトリ内にあります。拡張モジュールで他のライブラリ (zlib, ...) を使っている場合、それらのライブラリも変換しなければなりません。

変換されたファイルは、通常のライブラリと同じディレクトリに置かねばなりません。

さて、 Distutils は異なる名前を持つこれらのライブラリをどのように扱うのでしょうか? 拡張モジュールで (例えば 'foo' という名の) ライブラリが必要な場合、 Distutils はまず '_bcpp' が後ろに付いたライブラリ (例えば 'foo_bcpp.lib') が見つかるかどうか調べ、あればそのライブラリを使います。該当するライブラリがなければ、デフォルトの名前 ('foo.lib') を使います[1]。

Borland C++ を使って Distutils に拡張モジュールをコンパイルさせるには、以下のように入力します:

```
python setup.py build --compiler=bcpp
```

Borland C++ コンパイラをデフォルトにしたいなら、自分用、またはシステム全体向けに、 Distutils の設定ファイルを書くことを検討した方がよいでしょう (5 節を参照してください)。

参考資料:

C++Builder Compiler
(http://www.borland.com/bcppbuilder/freecompiler/)
Borland によるフリーの C++ コンパイラに関する情報で、コンパイラのダウンロードページへのリンクもあります。

Creating Python Extensions Using Borland's Free Compiler
(http://www.cyberus.ca/%7eg_will/pyExtenDL.shtml)
Borland 製のフリーのコマンドライン C++ を使って Python をビルドする方法について述べたドキュメントです。

GNU C / Cygwin / MinGW

この節では、 Cygwin や MinGW [2]配布物中の GNU C/C++ コンパイラで Distutils を使うために必要な手順について述べます。Cygwin 向けにビルドされている Python インタプリタを使っているなら、以下の手順をとらなくても Distutils はまったく問題なく動作します。

上記のコンパイラは、いくつかの特殊なライブラリを必要とします。この作業は Borland の C++ よりもやや複雑です。というのは、ライブラリを変換するためのプログラムが存在しないからです。

[1] つまり、全ての既存の COFF ライブラリを同名の OMF ライブラリに置き換えてもかまわないということです
[2] 詳しくは http://sources.redhat.com/cygwin/ や http://www.mingw.org/ を参照してください

まず、 Python DLL が公開している全てのシンボルからなるリストを作成しなければなりません。 (この作業むけのプログラムは、http://starship.python.net/crew/kernr/mingw32/Notes.html にあります。そのページで PExports 0.42h を探してください。)

```
pexports python24.dll >python24.def
```

これで、上で得られた情報をもとに、 gcc 用の import ライブラリを作成できます。

```
dlltool --dllname python24.dll --def python24.def --output-lib libpython24.a
```

出来上がったライブラリは、 'python24.lib' と同じディレクトリ (Python インストールディレクトリの 'libs' ディレクトリになるはずです) に置かなければなりません。

拡張モジュールが他のライブラリ (zlib, ...) を必要とする場合、それらのライブラリも変換しなければなりません。変換されたファイルは、それぞれ通常のライブラリが置かれているのと同じディレクトリに置かねばなりません。

Cygwin を使って Distutils に拡張モジュールをコンパイルさせるには、

```
python setup.py build --compiler=cygwin
```

のように入力します。また、非 cygwin モードの Cygwin [3] や MinGW では、

```
python setup.py build --compiler=mingw32
```

のように入力します。

上記のオプションやコンパイラをデフォルトにしたいなら、自分用、またはシステム全体向けに、Distutils の設定ファイルを書くことを検討した方がよいでしょう (5 節を参照してください)。

参考資料:

Building Python modules on MS Windows platform with MinGW
(http://www.zope.org/Members/als/tips/win32_mingw_modules)
MinGW 環境で必要なライブラリのビルドに関する情報があります。

http://pyopengl.sourceforge.net/ftp/win32-stuff/
Cygwin/MinGW および Borland 形式に変換済みの import ライブラリと、Distutils がビルド済みの Python の場所を特定するために必要なレジストリエントリを作成するためのスクリプトがあります。

7 日本語訳について

7.1 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 Installing 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。

7.2 翻訳者

2.3.3 和訳 Yasushi Masuda ⟨ y.masuda@acm.org ⟩ (April 2, 2004)

[3] このモードでは POSIX エミュレーションを利用できませんが、'cygwin1.dll' も必要なくなります。