Distribuire moduli Python

Versione 2.3.4

Greg Ward
Anthony Baxter

26 aprile 2005

Python Software Foundation
Email: distutils-sig@python.org
Traduzione presso
http://www.zonapython.it
Email: zap@zonapython.it

Sommario

Questo documento descrive le utility di distribuzione Python (“Distutils”) dal punto di vista dello sviluppatore del modulo, descrivendo come usare Distutils per rendere i moduli e le estensioni Python facilmente disponibili ad un vasto numero di persone con solo una piccola aggiunta ai meccanismi di compilazione/rilascio/installazione.

Traduzione in italiano a cura di **Mauro Morichi** *mauro@teppisti.it.*

INDICE

INDICE

CAPITOLO

UNO

Un’introduzione a Distutils

Questo documento copre l’uso delle Distutils per la distribuzione di moduli Python, con particolare attenzione sul ruolo dello sviluppatore/distributore: se si stanno cercando informazioni sull’installazione di moduli Python, ci si deve riferire al manuale *Installare moduli Python*.

1.1 Concetti & terminologia

L’uso delle Distutil è piuttosto semplice, sia per gli sviluppatori dei moduli, sia per gli utenti/amministratori che devono installare moduli di terze parti. Come sviluppatore, le vostre responsabilità (al di là della scrittura di un solido, ben documentato e ben testato codice, ovviamente...) sono:

- scrivere uno script di setup (per convenzione ‘setup.py’)
- (facoltativo) scrivere un file di configurazione di setup
- creare una distribuzione di sorgenti
- (facoltativo) creare una o più distribuzioni precompilate (binari)

ognuno di questi passaggi viene trattato da questo documento.

Non tutti gli sviluppatori di moduli hanno accesso ad un vasto numero di piattaforme diverse tra loro, così non ci si può aspettare che possano creare una moltitudine di distribuzioni precompilate. Si spera che un gruppo di intermediari, chiamati *packagers*, coprano questa necessità. I *packagers* prelevano la distribuzione sorgente rilasciata dagli sviluppatori del modulo, la compilano per una o più piattaforme e rilasciano le risultanti distribuzioni compilate. Così gli utenti delle piattaforme più popolari saranno in grado di installare la maggior parte delle distribuzioni più comuni di moduli Python nel modo più naturale per le loro esigenze, senza la necessità di eseguire un singolo script di setup o di compilare neanche una riga di codice.

1.2 Un semplice esempio

Solitamente, lo script di setup è piuttosto semplice, visto che è scritto direttamente in Python e non esistono limiti arbitrari all’impl ementazione di nuove funzionalità da fargli compiere; altresì si deve sempre fare attenzione nell’inserire operazioni arbitrare complesse nello script di setup. Gli script di configurazione in stile Autoconf possono eseguire lo script di setup più volte, durante il procedimento di compilazione ed installazione del proprio modulo da distribuire.

Se tutto quello che si vuole è distribuire un modulo chiamato `foo`, contenuto in un file ‘foo.py’, lo script di setup potrebbe essere tanto semplice quanto quello che segue:

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

Alcune osservazioni:

- la maggior parte delle informazioni che vengono indicate a Distutils vengono fornite come argomenti chiave alla funzione `setup()`
- questi argomenti chiave rientrano in due categorie: metadata di package (nome, numero di versione) ed informazioni circa il contenuto del package (in questo caso, una lista di puri moduli Python)
- i moduli vengono specificati per nome di modulo, non per nome di file (la medesima cosa verrà fatta per package ed estensioni)
- Si raccomanda di fornire un insieme di metadata adeguati, tra cui, in particolare, il nome, l'indirizzo email ed un indirizzo URL per il progetto (vedete la sezione 2 per un esempio)

Per creare una distribuzione di sorgenti per questo modulo, si deve creare uno script di setup, 'setup.py', contenente il precedente codice, ed eseguire:

```
python setup.py sdist
```

che creerà un file archivio (per esempio un tarball in UNIX, un file ZIP in Windows) contenente lo script di setup 'setup.py' ed il proprio modulo 'foo.py'. Il file archivio verrà chiamato 'foo-1.0.tar.gz' (o '.zip') e scompattato in una directory 'foo-1.0'.

Se un utente finale desidera installare il modulo `foo`, le uniche cose che deve fare è scaricare 'foo-1.0.tar.gz' (o '.zip'), scompattarlo, e nella directory 'foo-1.0'—eseguire:

```
python setup.py install
```

che completa la copia di 'foo.py' nella giusta directory per i moduli di terze parti nella propria installazione Python.

Questo semplice esempio dimostra alcuni fondamentali concetti delle Distutils. Primo, sia lo sviluppatore che l'installatore si trovano davanti alla stessa interfaccia, come ad esempio lo script di setup. La differenza è quale *comando* delle Distutils usano: il comando `sdist` viene eseguito per lo più dallo sviluppatore, mentre `install` è più spesso eseguito dall'installatore (comunque la maggior parte degli sviluppatori, vogliono occasionalmente installare il proprio codice).

Se si desidera rendere la vita realmente facile ai propri utenti, si possono preparare una o più distribuzioni precompilate. Solitamente, se si sta lavorando su di una macchina Windows e si vogliono facilitare le cose per gli altri utenti Windows, si possono creare degli installer eseguibili (il tipo più appropriato di distribuzione precompilata per questa piattaforma) con il comando `bdistwininst`. Per esempio:

```
python setup.py bdist_wininst
```

creerà un installatore eseguibile, 'foo-1.0.win32.exe', nella directory corrente.

Altri tipici formati per distribuzioni precompilate sono gli RPM, implementati con il comando `bdist_rpm`, Solaris **pkgtool** (`bdist_pkgtool`) e HP_UX **swinstall** (`bdist_sdux`). Per esempio, il seguente comando creerà un file RPM chiamato 'foo-1.0.noarch.rpm':

```
python setup.py bdist_rpm
```

(Il comando `bdist_rpm` usa l'eseguibile `rpm`, che comunque deve essere eseguito su di un sistema basato sugli RPM come RedHat Linux, SuSe Linux o Mandrake Linux).

È possibile sapere quali formati di distribuzione sono disponibili, in qualsiasi momento, digitando:

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

1.3 Terminologia generica per Python

Se si sta leggendo questo documento, probabilmente si avrà già un'idea piuttosto precisa di cosa sono i moduli, le estensioni e via discorrendo. Ciononostante, per avere la certezza che ciascuno parta da un punto di inizio comune, è a disposizione il seguente glossario dei più comuni termini Python:

modulo l'unità di base di codice riusabile in Python: un blocco di codice importato da altro codice. In questo contesto ci interessano tre tipologie di moduli: moduli Python puri, moduli di estensione e package.

modulo in Python puro un modulo scritto in Python e contenuto in un singolo file '.py' (e possibilmente associato con un file '.pyc' e/o '.pyo'). Alcune volte viene citato come "modulo puro".

modulo di estensione un modulo scritto nel linguaggio di basso livello dell'implementazione Python: C/C++ per Python, Java per Jython. Tipicamente contenuto in un singolo file precompilato caricabile dinamicamente, per esempio un file oggetto condiviso ('.so') per le estensioni Python in ambiente UNIX, una DLL (indicata dal file con estensione '.pyd') per le estensioni Python in Windows, o un file di classe Java per le estensioni Jython. (Da notare che correntemente, le Distutils gestiscono solo estensioni C/C++ per Python).

package un modulo che contiene altri moduli: tipicamente è contenuto in una directory del filesystem e si distingue dalle altre directory dalla presenza di un file '__init__.py'.

root package l'apice della gerarchia dei package. (Questo non è realmente un package, in quanto non possiede un file '__init__.py', ma in qualche modo lo si doveva pure chiamare). La stragrande maggioranza della libreria standard è nel package principale, come ci sono tanti piccoli, autonomi, moduli di terze parti, che non dipendono da una più grande collezione di moduli. Diversamente dai package regolari, i moduli nel package principale possono essere suddivisi in tante directory: infatti, ogni directory elencata in `sys.path` aggiunge moduli al package principale.

1.4 Terminologia specifica per Distutils

I seguenti termini si applicano più specificatamente all'ambito della distribuzione di moduli Python utilizzando le Distutils:

distribuzione di moduli una collezione di moduli Python distribuiti insieme come una singola risorsa scaricabile e da considerarsi installabile *in blocco*. Esempi di alcune delle distribuzioni di moduli molto conosciute sono Numeric Python, PyXML, PIL (la libreria per la gestione delle immagini in Python) o mxBase. (Questa potrebbe essere chiamata *package*, solo che sono termini già sfruttati in contesto Python: una singola distribuzione di moduli, può contenere zero, uno o più package Python.)

distribuzione di moduli in puro Python una distribuzione di moduli che contiene solo moduli in puro Python e package. Qualche volta vengono chiamati anche "distribuzione pura".

distribuzione di moduli non in puro Python una distribuzione di moduli che contiene come minimo un modulo di estensione. Qualche volta viene anche chiamata "distribuzione non pura".

distribuzione root la directory principale del proprio albero di sorgenti (o distribuzione di sorgenti); la directory dove c'è 'setup.py'. Generalmente 'setup.py' viene lanciato da questa directory.

CAPITOLO

DUE

Scrivere uno script di setup

Lo script di setup è il centro di tutte le attività di compilazione, distribuzione ed installazione di moduli utilizzato da Distutils. Il principale scopo dello script di setup è descrivere il proprio modulo di distribuzione a Distutils, in modo che i diversi comandi che debbano operare sul modulo siano in grado di fare cosa giusta. Come abbiamo visto nella sezione 1.2 sopra, lo script di setup consiste principalmente nella chiamata di `setup()` e molte delle informazioni fornite a Distutils dallo sviluppatore del modulo vengono indicate attraverso gli argomenti chiave di `setup()`.

Di seguito un esempio un po' più complesso, che ci seguirà nella prossima parte della sezione: il proprio script di setup con le Distutils. (Da ricordare che le Distutils sono state incluse dalla versione 1.6 di Python, mentre nella versione 1.5.2 hanno un'esistenza indipendente, cosicché gli utenti possano utilizzarle per installare altre distribuzioni di moduli. Lo script di setup di Distutils, mostrato qui, viene utilizzato per installare il package in Python 1.5.2.):

```
#!/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'],
     )
```

Ci sono solo due differenza tra questo e la normale distribuzione di un singolo file presentata nella sezione 1.2: più metadata e la specifica di moduli Python puri per package, più che per modulo. Questo è importante in quanto Distutils è composto da un paio di dozzine di moduli suddivisi in due package; un'esplicita lista di ogni modulo sarebbe noiosa da generare e difficile da manutenere. Per maggiori informazioni sui metadata addizionali, vedete la sezione 2.6.

Si noti che ogni percorso con nome (file o directory) indicato nello script di setup dovrà essere scritto utilizzando la convenzione UNIX, ovvero separato da barre oblique. Le Distutils si prenderanno carico di convertire questa rappresentazione neutrale rispetto alla piattaforma in qualcosa che sia appropriato sulla piattaforma corrente, prima di utilizzare il percorso con nome. Questo assicura che lo script di setup sia portabile nei vari sistemi operativi e questo, per inciso, è uno dei punti di forza delle Distutils. In questo spirito, tutti i percorsi con nome di questo documento vengono separati da barre oblique. (I programmatori di Mac OS devono ricordare che l'*assenza* di un barra obliqua indica un percorso relativo, quando invece le convenzioni di Mac OS prevederebbero i due punti.)

Questo, solitamente, si applica solo ai percorsi con nome indicati alle funzioni di Distutils. Se, per esempio, si usano le funzioni standard di Python come `glob.glob()` o `os.listdir()` per specificare file, si dovrà fare attenzione a scrivere codice portabile invece di fare un uso pesante dei separatori di percorso.

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

2.1 Elencare tutti i package

L'opzione packages dice a Distutils di elaborare (compilare, distribuire, installare, etc.) tutti i moduli in puro Python trovati in ogni package elencato nella lista packages. Per fare questo naturalmente ci deve essere una corrispondenza tra ogni nome di package e le directory del filesystem. La corrispondenza predefinita è quella più ovvia, per esempio il package `distutils` si trova nella directory 'distutils' relativa all'apice della distribuzione. Comunque quando si dice `packages = ['foo']` nel proprio script di setup, si sta dicendo che le Distuitils troveranno un file 'foo/__init__.py' (che potrebbe essere pronunciato diversamente nel sistema in uso, ma non cambia la sostanza) relativo alla directory dove risiede lo script di setup. Se non si seguirà questa regola, le Distutils emetteranno un avviso ma tenteranno di elaborare comunque il package incompleto.

Se si sta utilizzando una differente convenzione per costruire la propria directory sorgente, non ci sono problemi: si dovrà semplicemente indicare l'opzione package_dir per informare le Distutils circa le proprie convenzioni. Per esempio, si terranno tutti i sorgenti Python in 'lib', in modo che tutti i moduli del "package principale" (non in tutti i package) siano in 'lib', i moduli in `foo` si troveranno in 'lib/foo', e così via. Si scriverà:

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

nel nostro script di setup. Le chiavi di questo dizionario sono nomi di package ed un nome di package vuoto rappresenta il package principale. In questo caso, quando si dice `packages = ['foo']`, si sta dichiarando che il file 'lib/foo/__init__.py' esiste.

Un'altra possibile convenzione è quella di mettere il package `foo` direttamente in 'lib', il package `foo.bar` in 'lib/bar', etc.. Questo verrebbe inserito nello script di setup come:

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

Una voce *package*: *dir* nel dizionario package_dir implicitamente viene applicata a tutti i packages e sotto *package*, per far si che il caso del modulo `foo.bar` sia gestito automaticamente. In questo esempio, `packages = ['foo','foo.bar]` ha informato le Distutils di cercare 'lib/__init__.py' e 'lib/bar/__init__.py'. (Da tenere a mente che anche se package_dir viene applicata ricorsivamente, si devono esplicitamente elencare tutti i package in packages: le Distutils *non* analizzeranno ricorsivamente l'albero sorgente alla ricerca di ogni directory con un file '__init__.py'.)

2.2 Elencare singoli moduli

Per una piccola distribuzione di moduli, si dovrebbe preferire elencare tutti in moduli diversamente da quanto avviene per i package—specialmente nel caso di un singolo modulo che va nel "root package" (per esempio se nessun package è presente). Questo semplicissimo caso viene mostrato nella sezione 1.2; di seguito un esempio lievemente più complesso:

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

Questo descrive due moduli, uno di loro nel package principale, l'altro nel package `pkg`. Nuovamente lo schema predefinito package/directory prevede che questi due moduli debbano essere trovati in 'mod1.py' e 'pkg/mod2.py' e che il file 'pkg/__init__.py' esista come descritto. Ancora, è possibile alterare le corrispondenze package/directory usando l'opzione package_dir.

2.3 Descrivere i moduli di estensione

Come la scrittura di moduli di estensione risulta un po' più complicata della scrittura di moduli in puro Python, anche la loro descrizione alle Distutils è lievemente più complicata. Diversamente dai moduli puri, non è sufficiente una semplice lista di moduli o package ed aspettarsi che le Distutils trovino autonomamente i giusti file; si dovranno specificare i nomi delle estensioni, i file sorgenti ed ogni richiesta di compilazione/link (incluse directory, librerie da linkare, etc. etc.).

Tutto questo viene fatto attraverso un altro argomento chiave di `setup()`, l'opzione extension. extension è semplicemente una lista di istanze `Extension`, ognuna delle quali descrive un singolo modulo di estensione. Supponiamo che la distribuzione includa una singola estensione, chiamata `foo` ed implementata da 'foo.c'. Se non sono necessarie ulteriori istruzioni per il compilatore/linker, descrivere quest'estensione è molto semplice:

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

La classe `Extension` può essere importata da `distutils.core` attraverso `setup()`. Pertanto, lo script di setup per la distribuzione di un modulo che contenga solamente un'estensione e niente altro, potrebbe essere:

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

La classe `Extension` (attualmente, quell'importante meccanismo di estensione/compilazione implementato dal comando `build_ext`) si comporta in maniera molto flessibile nei riguardi della descrizione delle estensioni Python e viene spiegata nella sezione seguente.

2.3.1 Nomi di estensioni e package

Il primo argomento del costruttore `Extension` è sempre il nome dell'estensione, incluso ogni nome di package. Per esempio:

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

descrive un'estensione che si trova nel package principale, mentre:

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

descrive la stessa estensione nel package `pkg`. Il file sorgente ed il risultante codice oggetto sono identici in entrambi i casi; l'unica differenza è dove nel filesystem (e pertanto all'interno della gerarchia dello spazio dei nomi di Python) le risultanti estensioni risiedono.

Se si hanno a disposizione un gran numero di estensioni, tutte all'interno dello stesso package (o tutte sotto lo stesso package base), si dovrà usare l'argomento chiave ext_package della funzione `setup()`. Per esempio:

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

compilerà 'foo.c' nell'estensione `pkg.foo`, e 'bar.c' in `pkg.subpkg.bar`.

2.3.2 Estensioni di file sorgenti

Il secondo argomento del costruttore `Extension` è una lista di file sorgenti. Siccome le Distutils supportano correntemente solo estensioni C, C++, e Objective-C, sono presenti normalmente solo file sorgenti C/C++/Objective-C. Si deve essere sicuri di usare le estensioni appropriate per distinguere il file sorgente C++: '.cc' e '.cpp' sembrano essere riconosciuti sia dai compilatori UNIX che Windows.

Comunque, potete anche includere nell'elenco file di interfaccia SWIG ('.i'); il comando `build_ext` sa come comportarsi con le estensioni SWIG: esegue SWIG sul file di interfaccia e compila il file risultante C/C++ nell'estensione scelta.

****Il supporto SWIG è ancora da rifinire e largamente non testato; specialmente il supporto SWIG per l'estensione C++! Vi saranno in questo documento spiegazioni più dettagliate quando l'interfaccia sarà completa.****

Su alcune piattaforme, si possono includere file non sorgenti che verranno elaborati dal compilatore ed inclusi nell'estensione. Attualmente, questo è valido solo per file di messaggi di testo Windows ('.mc') e file di definizione di risorse per Visual C++ ('.res'), che saranno linkati all'interno dell'eseguibile.

2.3.3 Opzioni del preprocessore

Tre argomenti facoltativi di `Extension` saranno di aiuto se si ha bisogno di directory include da cercare o macro per preprocessori per definire/non-definire: `include_dirs`, `define_macros` e `undef_macros`.

Per esempio, se l'estensione richiede file di intestazione nella directory 'include', sotto la directory principale della distribuzione, si usa l'opzione `include_dirs`:

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

Qui si può specificare la directory assoluta; se siamo a conoscenza che l'estensione verrà compilata solo su sistemi UNIX con X11R6 installato in '/usr', si può procedere con

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

Si dovrebbe evitare questa specie di uso non portabile se si stabilisce di distribuire il proprio codice: è sicuramente meglio scrivere codice C come:

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

Se si ha la necessità di includere file di intestazione di qualche altra estensione Python, ci si può avvantaggiare dal fatto che i file di intestazione vengono installati in modo coerente dal comando Distutils `install_header`. Per esempio, i file di intestazione Python per la rappresentazione numerica vengono installati (nelle installazioni standard Unix) in '/usr/local/include/python1.5/Numerical'. La posizione esatta può differire rispetto alla propria piattaforma ed al tipo di installazione Python. Finché la directory di inclusione di Python—in questo caso '/usr/local/include/python1.5' — viene sempre inclusa nel percorso di ricerca, quando si compilano le estensioni Python, il migliore approccio è scrivere codice C come:

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

Se si deve inserire la directory degli include 'Numerical' direttamente nel proprio percorso di ricerca delle intestazioni, comunque si può cercare questa directory usando il modulo Distutils `distutils.sysconfig`:

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

Si può considerare questo metodo portabile—lavorerà su ogni installazione Python, indipendentemente dalla piattaforma, è probabilmente più facile da scrivere così che direttamente in codice C.

Si possono definire o non-definire macro per il preprocessore con le opzioni `define_macros` e `undef_macros`. `define_macros` prende una lista di tuple `(name, value)`, dove `name` è il nome della macro da definire (una stringa) e `value` è il suo valore: può essere una stringa o `None`. (Definire una macro `#define F00` con `none` è l'equivalente di un semplce `#define F00` all'interno di codice sorgente C: con la maggior parte dei compilatori, questo imposta `F00` nella stringa `1`). `undef_macros` è semplicemente una lista di macro che non necessitano di essere definite.

Per esempio:

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

è equivalente ad avere questo codice al principio del proprio codice sorgente in C:

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

2.3.4 Opzioni per le librerie

Si possono anche specificare le librerie da linkare quando si compila la propria estensione e le directory dove cercare queste librerie. L'opzione `libraries` è una lista di librerie da linkare insieme, `library_dirs` è una lista di directory dove cercare le librerie al momento del linking e `runtime_library_dirs` è una lista di directory dove cercare per librerie condivise (caricate dinamicamente) durante l'esecuzione.

Per esempio, se si ha la necessità di linkare librerie che si trovano all'interno del percorso di ricerca della libreria standard sul sistema di destinazione:

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

Se si necessita di eseguire il link con la libreria, in posizioni non standard, se ne deve includere la posizione in `library_dirs`

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

Nuovamente, questa sorta di costrutto non portabile dovrebbe essere evitato se si desidera distribuire il proprio codice.

****Si dovrebbe menzionare la libreria clib qui o da qualche altra parte!****

2.3.5 Altre opzioni

Ci sono alcune altre opzioni che possono essere utilizzate per gestire casi particolari.

L'opzione extra_objects è una lista di oggetti file da passare al linker. Questi file non devono avere estensione, oppure dovranno avere l'estensione predefinita utilizzata dal compilatore.

Le opzioni extra_compile_args ed extra_link_args possono essere usate per specificare opzioni da riga di comando facoltative per le rispettive righe di comando del compilatore e del linker.

export_symbols è utile solo in ambiente Windows. Può contenere una lista di simboli (funzioni o variabili) che devono essere esportati. Questa opzione non è necessaria quando si preparano estensioni compilate: Distutils aggiunge automaticamente `initmodule` alla lista dei simboli esportati.

2.4 Installare script

Precedentemente si è discusso sulla gestione di moduli Python cosiddetti puri e non puri, che di norma non possono essere eseguiti se non importandoli attraverso degli script.

Gli script sono file che contengono codice sorgente Python, intesi per essere avviati da riga di comando. Gli script non richiedono Distutils per fare niente di complicato. L'unica vera particolarità è la prima riga dello script, che inizierà con `#!` e che conterrà la parola "python", le Distutils provvederanno ad aggiustare la prima riga per riferirsi alla posizione corrente dell'interprete.

L'opzione scripts è semplicemente una lista di file che devono essere gestiti in questa maniera. Dallo script di setup di PyXML:

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

2.5 Installare ulteriori file

L'opzione data_files può essere usata per specificare ulteriori file necessari alla distribuzione del modulo: file di configurazione, elenchi di messaggi, file di dati, qualsiasi altra cosa che non rientri nelle precedenti categorie.

data_files specifica una sequenza di coppie (*directory*, *files*) nel seguente modo:

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

Si noti che si possono specificare i nomi delle directory dove i file di dati verranno installati, ma non si possono rinominare i file di dati direttamente.

Ogni coppia (*directory*, *files*) nella sequenza specificata, la directory d'installazione ed il file da installarvi dentro. Se *directory* è un percorso relativo, viene interpretato come relativo al prefisso di installazione (il `sys.prefix` di Python, per i package in puro python, `sys.exec_prefix` per i package che contengono moduli di estensione). Ogni nome di file in *files* viene interpretato relativamente allo script 'setup.py' all'inizio della distribuzione del sorgente del package. Nessuna informazione su directory, proveniente da *files*, viene usata per determinare l'ubicazione finale dei file installati; viene usato solo il nome del file.

È possibile specificare l'opzione data_files come una semplice sequenza di file, senza specificare una directory di destinazione, ma questo non è raccomandato ed il comando `install` stamperà un avvertimento in casi come questi. Per installare file di dati direttamente nella directory di destinazione, deve essere indicata una stringa vuota come directory.

2.6 Ulteriori meta-data

Lo script di setup può includere ulteriori meta-data, oltre al nome e la versione. Queste informazioni includono:

Meta-Data	Descrizione	Valore	Note
`name`	nome del package	breve stringa	(1)
`version`	versione di questa release	breve stringa	(1)(2)
`author`	nome dell'autore del package	breve stringa	(3)
`author_email`	indirizzo email dell'autore del package	indirizzo email	(3)
`maintainer`	il nome del manutentore del package	breve stringa	(3)
`maintainer_email`	indirizzo email del manutentore del package	indirizzo email	(3)
`url`	home page per il package	URL	(1)
`description`	breve, sommaria descrizione del package	breve stringa	
`long_description`	descrizione estesa del package	stringa lunga	
`download_url`	indirizzo da cui poter scaricare il package	URL	(4)
`classifiers`	una lista di classificatori Trove	lista di stringhe	(4)

Note:

(1) Questo campo viene richiesto.

(2) Si raccomanda che la versione sia nella forma *major.minor*[*.patch*[*.sub*]].

(3) Sia l'autore che il manutentore devono essere identificati.

(4) Questi campi non devono essere usati se il package deve essere compatibile con versioni di Python precedenti alla 2.2.3 o 2.3. L'elenco è disponibile presso il sito web PyPI.

'breve stringa' Una singola riga di testo, non più di 200 caratteri.

'stringa lunga' Righe multiple in puro testo in formato reStructuredText (vedete http://docutils.sf.net/).

'lista di stringhe' vedete più avanti.

Nessuno dei valori stringa deve essere in Unicode.

La codifica delle informazioni sulla versione è un'arte essa stessa. I package Python generalmente sono conformi al formato di versione *major.minor*[*.patch*][*sub*]. Il numero maggiore è 0 per la release iniziale e sperimentale del software. Viene incrementato per rilasci che rappresentano un punto importante di svolta del package. Il numero minore viene incrementato quando importanti, nuovi aggiornamenti vengono aggiunti al package. Il numero di patch aumenta quando vengono rilasciati dei bug-fix. Elementi aggiuntivi per informazioni varie sulle versioni vengono talvolta usati per rappresentare sottoversioni. Queste vengono solitamente denominate a1, a2,...,aN) (per versioni alpha, dove funzionalità e API possono cambiare), b1,b2,...,bN) (per versioni beta, che sono solamente dei bug-fix) e pr1,pr2,...,prN (per versioni di testing di pre-release della versione finale). Alcuni esempi:

0.1.0 La prima, versione sperimentale di un package

1.0.1a2 La seconda versione alpha della prima versione di patch della 1.0

I classifiers vengono specificati in una lista python:

```
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',
          ],
      )
```

Se si vuole includere dei classificatori nel proprio file 'setup.py' e si desidera anche restare compatibili all'indietro con le precedenti versioni di Python (prima della 2.2.3), si può includere il seguente frammento di codice nel proprio 'setup.py' prima della chiamata `setup()`:

```
# 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
```

2.7 Il debugging dello script di setup

Alcune volte le cose non vanno come dovrebbero e lo script di setup non sembra fare quello che lo sviluppatore vuole.

Le Distutils catturano ogni eccezione quando eseguono lo script di setup e stampano un semplice messaggio di errore prima che lo script termini. La motivazione di questo comportamento è non confondere l'amministratore che non conosce molto Python e sta tentando di installare un package. Se avesse un lungo traceback dal profondo di Distutils potrebbe pensare che il package o l'installazione di Python sia corrotta perché non viene letto tutto fino alla fine e vedrebbe che si tratta di un problema connesso ai permessi.

Al contrario, non aiuta lo sviluppatore a trovare le cause dell'errore. A questo scopo, la variabile d'ambiente DISTUTILS_DEBUG può essere impostata a qualsiasi valore tranne che una stringa vuota e le distutils stamperanno una serie dettagliata di informazioni circa quello che stanno facendo, stampando anche l'intera traceback nel caso in cui venga sollevata un'eccezione.

CAPITOLO

TRE

Scrivere il file di configurazione di setup

Spesso, non è possibile scrivere ogni cosa necessaria per costruire una distribuzione *a priori*: si potrebbe aver bisogno di ottenere qualche informazione dall'utente, o dal sistema dell'utente, per poter proseguire. Tanto più l'informazione è semplice—un elenco di directory dove cercare per file di intestazione C o le librerie, per esempio—che fornire un file di configurazione, 'setup.cfg', da editare da parte degli utenti, diviene un modo semplice ed economico per risolvere la questione. Il file di configurazione permette di fornire dei valori predefiniti per ogni opzione di comando, che l'installatore può sovrascrivere sia con la riga di comando che editando il file di configurazione.

Il file di configurazione di setup è una utile via di mezzo tra lo script di setup—che, idealmente, dovrebbe essere nascosto agli installatori [1]—e la riga di comando dello script di setup, che è fuori dal proprio controllo ed interamente sotto quello dell'installatore. Infatti, 'setup.cfg' (ed ogni altro file di configurazione di Distutils presente sul sistema di destinazione) viene elaborato dopo il contenuto dello script di setup, ma prima della riga di comando. Questo ha diverse utili conseguenze:

- Coloro che installeranno potranno sovrascrivere una parte di ciò che è stato inserito nel 'setup.py' editando il 'setup.cfg'
- Si possono fornire come non standard ma comunque predefinite, opzioni che non sono facilmente impostabili in 'setup.py'
- Coloro che installeranno potranno facilmente sovrascrivere qualsiasi cosa in 'setup.cfg' usando le opzioni da riga di comando di 'setup.py'

La sintassi di base del file di configurazione è semplice:

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

dove *command* è uno dei comandi di Distutils (come, ad esempio `build_py`, `install`) e *option* è una delle opzioni che il comando supporta. Qualsiasi numero di opzioni può essere fornito per ogni comando ed ogni numero di sezioni di comandi possono essere incluse nel file. Le righe vuote vengono ignorate, come lo sono i commenti che iniziano con un carattere '#' fino alla fine della riga. Valori di opzioni lunghe possono essere suddivise in righe multiple semplicemente indentando la riga di continuazione.

Si può verificare l'elenco delle opzioni supportate da un particolare comando con l'universale opzione **--help**, per esempio:

[1] Questo meccanismo probabilmente non verrà ottenuto finché l'auto-configurazione non verrà pienamente supportata dalle Distutils

```
> python setup.py --help build_ext
[...]
Opzioni per il comando 'build_ext':
  --build-lib (-b)     directory per i moduli di estensioni compilati
  --build-temp (-t)    directory per i file temporanei (prodotti dalla
                       compilazione)
  --inplace (-i)       ignora build-lib ed inserisce le estensioni compilate
                       nella directory dei sorgenti, accanto ai moduli Python
  --include-dirs (-I)  elenco di directory dove cercare i file di intestazione
  --define (-D)        macro del preprocessore C da definire
  --undef (-U)         macro del preprocessore C da non definire
[...]
```

Si noti che un'opzione indicata **--foo-bar** da riga di comando viene chiamata foo_bar nel file di configurazione.

Per esempio, si decide di volere la propria estensione compilata "sul posto"—che significa che si ha una estensione `pkg.ext` e si vuole che il file di estensione compilato ('ext.so' in UNIX, diciamo) sia inserito nella stessa directory sorgente, come i propri moduli Python `pkg.mod1` e `pkg.mode2`. Si può sempre utilizzare l'opzione **--inplace** da riga di comando per ottenere ciò:

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

Ma questo richiede che si specifichi sempre il comando `build_ext` esplicitamente, e di ricordarsi di indicare **--inplace**. Un modo semplice è di "impostare e dimenticare" questa opzione, codificandola in 'setup.cfg', ovvero il file di configurazione per questa distribuzione

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

Questo influenzerà tutte le compilazioni di questa distribuzione di moduli, indipendentemente se si specifica esplicitamente `build_ext`. Se si include 'setup.cfg' nella propria distribuzione di sorgenti, influenzerà anche le compilazioni finali dell'utente—che è probabilmente una cattiva idea per questa opzione, in quanto ogni compilazione di estensioni in loco interromperebbe l'installazione della distribuzione di moduli. In certi casi particolari, comunque, i moduli vengono compilati proprio nella loro directory di installazione, (Distribuire estensioni che si aspettano di essere compilate nella medesima directory d'installazione costituiscono comunque sempre un modo di procedere sbagliato.)

Un altro esempio: alcuni comandi dispongono di parecchie opzioni che non cambiano da esecuzione ad esecuzione; per esempio, `bdist_rpm` deve sapere tutto quanto possa servire per generare un file "spec" per creare una distribuzione RPM. Alcune di queste informazioni arrivano dallo script di setup, ed altre vengono automaticamente generate da Distutils (come la lista dei file installati). Ma alcune di queste devono essere indicate come opzione a `bdist_rpm` e potrebbe risultare molto noioso da farsi sulla riga di comando ad ogni esecuzione. Pertanto, ecco un piccolo estratto da un file 'setup.cfg' personalizzato di Distutils:

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

Si noti che l'opzione doc_files è semplicemente una stringa separata da spazi vuoti, suddivisa in righe multiple per una migliore leggibilità.

Vedete anche:

Installazione di moduli Python
(../inst/config-syntax.html)

Ulteriori informazioni sui file di configurazione sono disponibili nel manuale per gli amministratori di sistema.

CAPITOLO

QUATTRO

Creare una distribuzione di sorgenti

Come visto nella sezione 1.2, per creare una distribuzione sorgente si usa il comando `sdist`. Nel caso più semplice:

```
python setup.py sdist
```

(assumendo che non si sia stata specificata nessuna opzione `sdist` nello script di setup o nel file di configurazione), `sdist` crea l'archivio nel formato predefinito per la piattaforma corrente. Il formato predefinito è un file ('.tar.gz') in UNIX ed un file ZIP in Windows. ****Qui non c'è supporto per Mac OS****

Si possono specificare tutti i formati che si vuole usando l'opzione **--formats**, per esempio:

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

per creare un file tar.gz ed un file .zip. I formati disponibili sono:

Formato	Descrizione	Note
`zip`	file zip ('.zip')	(1),(3)
`gztar`	file tar e gzip ('.tar.gz')	(2),(4)
`bztar`	file tar e bzip2 ('.tar.bz2')	(4)
`ztar`	file tar compresso ('.tar.Z')	(4)
`tar`	file tar ('.tar')	(4)

Note:

(1) predefinito in Windows

(2) predefinito in UNIX

(3) richiede sia l'utility esterna **zip** o il modulo `zipfile` (parte della libreria standard Python dalla versione 1.6)

(4) richiede le utility esterne: **tar** e possibilmente uno tra **gzip**, **bzip2** o **compress**

4.1 Specifiche del file da distribuire

Se non si fornisce un'esplicita lista di file (o le istruzioni su come generarne una), il comando `sdist` ne inserisce un insieme predefinito minimo nel sorgente da distribuire:

- Tutti i file sorgenti Python indicati dalle opzioni py_modules e packages
- Tutti i file sorgenti in C menzionati nelle opzioni ext_modules o libraries (****la libreria corrente dei sorgenti in C è attualmente insoddisfacente—non c'è il metodo `get_source_files()` in 'build_clib.py'!****)

- script identificati dall'opzione scripts
- Qualsiasi cosa che somigli ad uno script di test: 'test/test*.py' (attualmente, Le Distutils non fanno nulla con gli script di test, eccetto che includerli nella distribuzione sorgente, ma in futuro verrà definito uno standard per testare le distribuzione di moduli Python).
- 'README.txt' (o 'README'), 'setup.py' (o qualsiasi altra cosa venga chiamata dallo script di setup) e 'setup.cfg'

Qualche volta questo è sufficiente, ma solitamente si vuole specificare ulteriori file da distribuire. La procedura tipica per fare questo è scrivere un *manifest template*, chiamato 'MANIFEST.in' predefinito. Lo schema del manifesto è semplicemente una lista di istruzioni su come generare il proprio file manifest, 'MANIFEST', che è l'esatto elenco dei file da includere nella propria distribuzione sorgente. Il comando `sdist` processa il template e genera il manifesto basato sulle sue istruzioni e con ciò che trova nel filesystem.

Se si preferisce creare autonomamente il proprio file di manifesto, il formato è semplice: un nome di file per riga, file regolari (o link simbolici riferiti a loro stessi). Se si fornisce il proprio 'MANIFEST', si deve specificare ogni cosa: l'insieme dei file predefiniti descritto sopra non si applica per questo caso.

Lo schema del manifesto ha un comando per riga, dove ogni comando specifica un insieme di file da includere o da escludere dalla distribuzione sorgente. Per un esempio, utilizzeremo ancora il template del manifesto proprio delle Distutils:

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

Il significato dovrebbe essere piuttosto chiaro: includere tutti i file nel livello principale della distribuzione che verificano '*.txt' o '*.py' ed escludere tutte le directory che verificano 'examples/sample?/build'. Tutto questo viene fatto *dopo* l'inclusione standard dell'insieme, così che si possano escludere file dall'insieme standard con istruzioni esplicite nello schema del manifesto. (O si può usare l'opzione **--no-defaults** per disabilitare interamente l'insieme standard). Ci sono diversi altri comandi disponibili nel mini-linguaggio dello schema del manifesto; si veda la sezione 8.2.

L'ordine dei comandi nello schema del manifesto è organizzato così: inizialmente abbiamo la lista dei file predefiniti come descritto sopra ed ogni comando nel template né aggiunge né rimuove dalla lista alcun file. Una volta che si è completamente processato lo schema del manifesto, si rimuoveranno i file che non verranno inclusi nella distribuzione sorgente:

- l'insieme dei file nell'albero "build" delle Distutils (il predefinito 'build/')
- tutti i file nelle directory chiamate 'RCS' o 'CVS'

Adesso abbiamo l'elenco completo dei file, che viene scritto nel manifesto per usi futuri e quindi usato per costruire l'archivio, o gli archivi, della distribuzione sorgente.

Si può disabilitare l'insieme predefinito dei file include con l'opzione **--no-defaults** e si può disabilitare l'esclusione standard con **--no-prune**.

Seguendo lo schema del manifesto proprio delle Distutils, si può seguire come il comando `sdist` costruisca la lista dei file da includere nella distribuzione sorgente delle Distutils:

1. include tutti i file sorgenti Python nelle sottodirectory 'distutils' e 'distutils/command' (in quanto i package corrispondenti a queste due directory erano menzionati nell'opzione packages nello script di setup—si veda la sezione 2)
2. include 'README.txt', 'setup.py' e 'setup.cfg' (file standard)
3. include 'test/test*.py' (file standard)
4. include '*.txt' nella distribuzione principale (questo cercherà 'README.txt' in un secondo tempo, ma alcuni duplicati verranno eliminati successivamente)

5. include qualsiasi cosa che verifichi ‘*.txt’ o ‘*.py’ nelle sottodirectory sotto ‘examples’
6. esclude tutti file nelle sottodirectory a partire dalla directory che verifica ‘examples/sample?/build’—questo potrebbe escludere file inclusi nei precedenti due passagi, è quindi importante che il comando `prune` nello schema del manifesto venga dopo il comando `recursive-include`
7. esclude l’intero albero ‘build’ ed ogni directory ‘RCS’ o ‘CVS’

Semplicemente, come nello script di setup, i nomi dei file e delle directory dello schema del manifesto devono sempre essere separati da slash; le Distutils si prenderanno carico di convertirle nella rappresentazione standard della propria piattaforma. In questo modo, lo schema del manifesto diventa portabile tra sistemi operativi diversi tra loro.

4.2 Opzioni relative al manifesto

Il normale corso delle operazioni per il comando `sdist` è il seguente:

- se il file manifesto, ‘MANIFEST’ non esiste, legge ‘MANIFEST.in’ e crea il manifesto
- se sia ‘MANIFEST’ che ‘MANIFEST.in’ non esistono, crea il manifesto con solamente l’insieme dei file predefiniti
- se sia ‘MANIFEST.in’ o lo script di setup (‘setup.py’) sono più recenti di ‘MANIFEST’, ricrea ‘MANIFEST’ leggendo da ‘MANIFEST.in’
- usa la lista dei file ora in ‘MANIFEST’ (che sia generata o letta) per creare l’archivio/archivi della distribuzione sorgente

C’é una coppia di opzioni che modificano questo comportamento. Primo, usare **--no-defaults** e **--no-prune** per disabilitare gli insiemi standard “include” ed “esclude”.

Secondo, si dovrebbe forzare il manifesto ad essere rigenerato—per esempio, se sono stati aggiunti o rimossi file o directory che verificano una struttura nel template del manifesto, si dovrebbe rigenerare il manifesto:

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

O, si vorrebbe generare/rigenerare il manifesto, ma senza creare la distribuzione sorgente:

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

--manifest-only richiede **--force-manifest**. **-o** è una una scorciatoia per **--manifest-only** e **-f** per **--force-manifest**.

CAPITOLO

CINQUE

Creare una distribuzione precompilata

Una “distribuzione precompilata” è quello che si intende pensando o a “pacchetti di binari” o ad un “installer” (dipendente dal background). Non è necessariamente binario, perché potrebbe contenere solo codice sorgente Python e/o byte-code; e non lo si chiama package, perché quella parola è già usata in Python. (E “installer” è un termine specifico nel mondo dei principali sistemi desktop).

Una distribuzione precompilata è il modo di rendere la vita facile agli installatori della distribuzione del vostro modulo: per utenti di sistemi Linux basati su RPM si tratta di un binario RPM; per gli utenti Windows, si tratta di un installer eseguibile; per gli utenti Linux basati su Debian, è un package Debian; e così via. Ovviamente, nessuna persona sarà in grado di creare distribuzioni per ogni piattaforma esistente, così le Distutils sono state progettate per permettere agli sviluppatori di moduli di concentrarsi sulla loro specialità—scrivere codice e creare la distribuzione sorgente—quando una figura intermedia chiamato *packager* si occupa di trasformare la distribuzione sorgente in distribuzioni compilate per tante piattaforme quanti sono i packagers.

Normalmente, lo sviluppatore di moduli può essere proprio il packagers; o il packager potrebbe essere un volontario “esterno” che ha accesso alla piattaforma a cui non non ha accesso lo sviluppatore; o potrebbe essere una persona che periodicamente recupera la nuova distribuzione sorgente e la trasforma nella distribuzione compilata per tutte le piattaforma a cui ha accesso. Indipendemente da chi sia, un packager usa lo script di setup e l’insieme dei comandi `bdist` per generare distribuzioni compilate.

Come semplice esempio, se si esegue il seguente comando nella directory principale dei sorgenti di Distutils:

```
python setup.py bdist
```

le distutils compilano la propria distribuzione di moduli (in questo caso le proprie Distutils), realizzando un’installazione “fasulla” (anche nella directory ‘build’, e creano il tipo predefinito della distribuzione compilata per la propria piattaforma. Il formato predefinito per le distribuzioni compilate è un stupido file tar su UNIX ed un semplice installer eseguibile su windows. (Quel file tar viene considerato “stupido” perché deve essere scompattato in una specifica locazione per fare in modo che lavori.)

Comunque, il citato comando in un sistema UNIX crea ‘Distutils-1.0.*plat*.tar.gz’; scompattando questo file tarball nella giusta posizione, installa le Distutils come se fosse stato codice scaricato e su cui viene eseguito `python setup.py install`. (Il “posto giusto” è o la root del filesystem o la directory Python *prefix*, a seconda dell’opzione data al comando `bdist_dumb`: quella predefinita crea la distribuzione stupida relativamente a *prefix*.)

Ovviamente, per le distribuzioni in puro Python, questo non è altro che l’esecuzione di `python setup.py install`—ma per le distribuzioni non pure, che includono estensioni che hanno la necessità di essere compilate, può significare la differenza tra coloro che sono in grado di usare le proprie estensioni e chi no. La creazione di distribuzioni compilate “smart”, come un package RPM o un installer eseguibile per Windows è spesso più conveniente per gli utenti, anche se la propria distribuzione non contiene alcuna estensione.

Il comando `bdist` ha un’opzione **--formats** simile al comando `sdist`, che si può usare per selezionare il tipo di distribuzione compilata da generare: per esempio,

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

può, quando eseguito su di un sistema UNIX, creare 'Distutils-1.0.*plat*.zip'—ancora, questo archivio dovrebbe venir scompattato nella directory principale per installare le Distutils.

I formati disponibili per distribuzioni già pronte sono:

Formato	Descrizione	Note
`gztar`	file tar gz ('.tar.gz')	(1),(3)
`ztar`	file tar compresso ('.tar.Z')	(3)
`tar`	file tar ('.tar')	(3)
`zip`	file zip ('.zip')	(4)
`pkgtool`	Solaris **pkgtool**	
`sdux`	HP-UX **swinstall**	
`rpm`	RPM	(5)
`wininst`	autoestrattore per file ZIP per Windows	(2),(4)

Note:

(1) predefinito su UNIX

(2) predefinito su ****to-do!****

(3) richiede utility esterne: **tar** e possibilmente uno tra **gzip**, **bzip2** o **compress**

(4) richiede o un'utility esterna come **zip** o il modulo `zipfile` (parte della libreria standard Python a partire da Python 1.6)

(5) richiede un'utility esterna **rpm**, versione 3.0.4 o maggiore (usare `rpm -version` per sapere quale versione si sta usando)

Non si deve necessariamente usare il comando `bdist` con l'opzione **--formats**; si può anche usare il comando che direttamente implementa il formato a cui si è interessati. Alcuni di questi "sottocommandi" `bdist` generano attualmente diversi formati simili: per inciso, il comando `bdist_dumb` genera tutti i formati archivio "stupido" (`tar`, `ztar`, `gztar` e `zip`) ed il sotto-comando `bdist_rpm` genera sia gli RPM binari che sorgenti. I sottocomandi `bdist` ed il formato generato da ciascuno sono:

Comando	Formato
`bdist_dumb`	tar, ztar, gztar, zip
`bdist_rpm`	rpm, srpm
`bdist_wininst`	wininst

La seguente sezione fornisce dettagli su ogni comando individuale `bdist_*`

5.1 Creare una stupida distribuzione compilata

****È necessario documentare assolutamente il propro package con il prefisso relativo, ma prima lo si deve implementare!****

5.2 Creare pacchetti RPM

Il formato RPM viene usato da molte popolari distribuzioni Linux, incluse Red Hat, SuSE, e Mandrake. Se una di queste (o una tra le altre distribuzioni Linux basate su RPM) distribuzioni è il vostro ambiente abituale, creare package RPM per altri utenti della propria distribuzione è semplice. In base alla complessità della propria distribuzione di moduli e le differenza tra le distribuzioni Linux, si potrebbe essere in grado di creare degli RPM che lavorano su differenti distribuzioni basate su RPM.

Il classico metodo per creare un RPM della propria distribuzione di moduli è di eseguire il comando `bdist_rpm`:

```
python setup.py bdist_rpm
```

oppure il comando `bdist` con l'opzione **--format**:

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

Il primo permette di specificare opzioni RPM specifiche: il secondo consente di specificare con facilità diversi formati in una singola esecuzione. Se si ha bisogno di entrambi, si possono esplicitamente specificare comandi multipli `bdist_*` e le loro opzioni:

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

La creazione di pacchetti RPM è guidata dal file '.spec', tanto quanto le distutils vengono controllate dallo script di setup. Per rendere la vita facile, il comando `bdist_rpm` crea normalmente un file '.spec' basato sulle informazioni fornite nello script di setup, da riga di comando ed in ogni file di configurazione delle Distutils. Diverse opzioni e sezioni nel file '.spec' sono derivate dalle opzioni presenti nello script di setup:

Opzioni del file '.spec' RPM o sezione	**Opzioni Distutils per lo script di setup**
Nome	nome
Sommario (nel preambolo)	description
Versione	version
Produttore	author e author_email o maintainer e maintainer_email
Copyright	licence
Url	url
%descrizione (sezione)	long_description

Inoltre, ci sono molte opzioni nei file '.spec' che non hanno una corrispondente opzione nello script di setup. Molte di queste vengono gestite attraverso le opzioni del comando `bdist_rpm` come segue:

Opzioni del file '.spec' RPM o sezione	**`bdist_rpm` opzione**	**default value**
Release	release	"1"
Group	group	"Development/Libraries"
Produttore	vendor	(vedete dopo)
Packager	packager	(none)
Fornisce	provides	(none)
Richieste	requires	(none)
Confligge	conflicts	(none)
Obsolete	obsoletes	(none)
Distribuzione	distribution_name	(none)
Opzioni richieste dalla compilazione	build_requires	(none)
Icone	icon	(none)

Ovviamente, fornire anche poche di queste opzioni da riga di comando sarebbe noioso e non esente da errori, così è preferibile inserirle nel file di configurazione di setup, 'setup.cfg'—vedete la sezione 3. Se si distribuiscono o si creano package con molte distribuzioni di moduli Python, si possono inserire le opzioni comuni da applicare nel proprio personale file di configurazione di Distutils (('~/.pydistutils.cfg').

Ci sono tre passi per costruire un package RPM binario e vengono gestiti automaticamente da Distutils:

1. creare un file '.spec', che descrive il package (analogamente allo script di setup delle distutils; infatti, molte delle informazioni nello script di setup corrispondono al file '.spec')

2. creare il sorgente RPM
3. creare il "binario" RPM (che potrebbe o non potrebbe contenere codice binario, a seconda di come sia composta la vostra distribuzione di moduli, se contiene o meno estensioni Python)

Normalmente, gli RPM raggruppano gli ultimi due passaggi, quando si usano le Distutils, tutti e tre i passaggi vengono tipicamente raggruppati assieme.

Invece, se lo si desidera, si possono separare questi tre passaggi. Si può usare l'opzione **--spec-only** per eseguire `bdist_rpm` solo per creare il file '.spec' ed uscire; in questo caso il file '.spec' verrà scritto nella "directory di distribuzione"—solitamente 'dist/', ma personalizzabile con l'opzione **--dist-dir**. (Normalmente il file '.spec' finisce in fondo all'"albero di compilazione" in una directory temporanea creata da `bdist_rpm`.)

5.3 Creare un installer Windows

Gli installer di eseguibili sono il formato naturale per le distribuzioni di binari in Windows. Mostrano una gradevole interfaccia utente e forniscono alcune informazioni circa la distribuzione del moduli che deve essere installata, prendendola dai metadata nello script di setup; lasciano all'utente la selezione di alcune opzioni e l'avvio e lo stop dell'installazione.

Siccome i metadata vengono presi dallo script di setup, creare installer per Windows è solitamente tanto facile quanto eseguire:

```
python setup.py bdist_wininst
```

o il comando `bdist` con l'opzione **--formats**:

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

Se si ha a disposizione una distribuzione di moduli puri (contenenti soltanto puri moduli e package Python), il risultante installer sarà una versione indipendente e avrà un nome simile a 'foo-1.0.win32.exe'. Questi installer possono essere creati sia su piattaforme UNIX che su quelle Mac OS.

Se si ha a disposizione una distribuzione non pura, le estensioni possono essere create solo su piattaforme Windows e saranno dipendenti dalla versione di Python. Il nome dell'installer riflette questo ed assume la forma di 'foo-1.0.win32-py2.0.exe'. Si deve creare un installer separato per ogni versione Python che si vuole supportare.

L'installer cercherà di compilare i moduli puri in bytecode dopo l'installazione sul sistema di destinazione in modalità normale ed ottimizzata. Se non si vuole che questo accada, si può eseguire il comando `bdist_wininst` con una od entrambe le opzioni **--no-target-compile** e **--no-target-optimize**.

In modo predefinito l'installer mostrerà, una volta avviato, il gradevole logo "Python Powered" ma si può anche fornire il progrio logo, che deve essere un file Windows '.bmp' con l'opzione **--bitmap**.

L'installer mostrerà anche un grande titolo sullo sfondo della finestra quando viene eseguito, che verrà costruito dal nome della propria distribuzione ed il numero di versione. Questo può essere sostituito con altro testo usando l'opzione **--title**.

Il file installer verrà scritto nella "directory di distribuzione" — normalmente 'dist/', ma è personalizzabile con l'opzione **--dist-dir**.

5.3.1 Lo script di post installazione

A partire da Python 2.3, uno script di post installazione può essere specificato con l'opzione **--install-script**. Il nome di base dello script deve essere specificato ed il nome dello script deve anche essere elencato nell'argomento degli script nella funzione di setup.

Questo script verrà eseguito durante la fase di installazione sul sistema di destinazione, dopo che tutti i file sono

stati copiati, con `argv[1]` impostato a **-install**, e nuovamente durante la fase di disinstallazione, prima che i file siano rimossi con `argv[1]` impostato a **-remove**.

Lo script di installazione viene eseguito internamente all'installatore di windows, ogni output (`sys.stdout`, `sys.stderr`) viene rediretto in un buffer e verrà mostrato nell'ambiente grafico, GUI, dopo che lo script è terminato.

Alcune funzioni particolarmente utili in questo contesto sono disponibili come funzioni integrate aggiuntive nello script di installazione.

`directory_created`(*path*)
`file_created`(*path*)

Queste funzioni dovrebbero essere chiamate quando una directory o file viene creata dallo script post-install durante la fase di installazione. Si deve registrare *path* con l'uninstaller, in modo da rimuovere i file quando la distribuzione verrà disinstallata. Per sicurezza, le directory vengono rimosse solo se sono vuote.

`get_special_folder_path`(*csidl_string*)

Questa funzione può essere usata per recuperare la posizione delle directory speciali di Windows come il Menu di Avvio o il Desktop. Restituisce il percorso completo della directory. *csidl_string* deve essere una delle seguenti stringhe:

```
"CSIDL_APPDATA"

"CSIDL_COMMON_STARTMENU"
"CSIDL_STARTMENU"

"CSIDL_COMMON_DESKTOPDIRECTORY"
"CSIDL_DESKTOPDIRECTORY"

"CSIDL_COMMON_STARTUP"
"CSIDL_STARTUP"

"CSIDL_COMMON_PROGRAMS"
"CSIDL_PROGRAMS"

"CSIDL_FONTS"
```

Se la directory non può essere recuperata, viene sollevata l'eccezione `OSError`.

Quale directory è disponibile dipende dalla versione esatta di Windows e probabilmente anche dalla configurazione. Per i dettagli si faccia riferimento alla documentazione Microsoft per la funzione `SHGetSpecialFolderPath()`.

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

Questa funzione crea uno shortcut. *target* è il percorso del programma che deve essere avviato dallo shortcut. *description* è la descrizione dello shortcut. *filename* è il titolo dello shortcut che l'utente vedrà. *arguments* specifica gli argomenti da riga di comando, se ce ne sono. *workdir* è la directory di lavoro del programma. *iconpath* è il file contenente l'icona per lo shortcut e *iconindex* è l'indice dell'icona nel file *iconpath*. Per ulteriori dettagli si consulti la documentazione Microsoft per l'interfaccia `IShellLink`.

CAPITOLO

SEI

Registrazione con l'Indice Python Package

L'Indice del Python Package (PyPI) mantiene i meta-data che descrivono le distribuzioni preparate con Distutils. Il comando Distutils `register` viene usato per indicare i meta-data della propria distribuzione all'indice. Viene invocato come qui di seguito:

```
python setup.py register
```

le Distutils risponderanno con il seguente prompt:

```
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]:
```

Nota: Se le proprie credenziali nome-utente e password vengono salvate localmente, questo menu non verra visualizzato.

Se non si è già registrati con PyPI, questo va fatto adesso. Si deve scegliere l'opzione 2 ed inserire i propri dati, come richiesto. Subito dopo questo passaggio, si riceverà una email che verrà usata per confermare la propria registrazione.

Una volta che si è registrati, si può scegliere l'opzione 1 dal menu. Verranno richiesti i propri nome-utente e password PyPI e `register` inserirà i propri meta-data nell'indice.

Si possono inserire ogni numero di versione della propria distribuzione nell'indice. Se si alterano i meta-data per una particolare versione, si devono inserire nuovamente e l'indice verrà aggiornato.

PyPI mantiene un record per ciascuna combinazione (nome, versione) inserita. Il primo utente che inserisce informazioni per un preciso nome, viene dichiarato come il proprietario (Owner) di quel nome. Si devono inserire i cambiamenti attraverso il comando `register` o attraverso l'interfaccia web. Tramite questi comandi si devono anche designare gli altri utenti come Proprietari o Mantainers. I Maintainers devono aggiornare le informazioni del package, ma non possono designare altri Proprietari o Maintainers.

In modo predefinito PyPI mostrerà tutte le versioni per il package indicato. Per nascondere alcune versioni, la proprietà Hidden dovrebbe essere impostata a Yes. Questo deve essere fatto attraverso l'interfaccia Web.

CAPITOLO

SETTE

Esempi

7.1 Distribuzione di Python puro (tramite moduli)

Se si sta semplicemente distribuendo una coppia di moduli, specialmente se non risiedono in un particolare package, si possono specificare individualmente usando l'opzione dello script di setup py_modules.

Nel caso più semplice, si avranno due file di cui preoccuparsi: uno script di setup ed il singolo modulo che si sta distribuendo, in questo esempio il file 'foo.py':

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

(In tutti i diagrammi in questa sezione, `<root>` si riferisce alla directory principale della distribuzione.) Uno script di setup minimale per descrivere questa situazione potrebbe essere:

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

Si noti che il nome della distribuzione viene specificato in modo indipendente attraverso l'opzione name e non ci sono regole che dicano che deve avere lo stesso nome dell'unico modulo della distribuzione (anche se potrebbe essere comunque una buona convenzione da seguire). Comunque, il nome della distribuzione viene usato per generare i nomi dei file, dovreste far uso di un buon numero di lettere, numeri, trattini bassi e trattini di congiunzione.

Visto che py_modules è una lista, si possono specificare moduli multipli, per esempio se si stanno distribuendo i moduli `foo` e `bar`, il setup dovrebbe somigliare a questo:

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

E lo script di setup potrebbe essere

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

Si possono mettere i file sorgenti dei moduli in un'altra directory, ma se si hanno abbastanza moduli per fare ciò, è probabilmente più facile specificare i moduli per package anziché specificarli individualmente.

7.2 Distribuzione di Python puro (tramite package)

Se si ha a disposizione più di una coppia di moduli da distribuire, specialmente se risiedono in package multipli, è probabilmente più facile specificare l'intero gruppo di package anziché ogni package individualmente. Questo funziona anche se il modulo non è in un package; si può semplicemente dire alle Distutils di elaborare i moduli dal package principale e tutto funziona nello stesso modo, come per ogni altro package (eccetto il caso in cui non si abbia un file '__init__.py').

Lo script di setup dell'ultimo esempio potrebbe anche essere scritto come:

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

(La stringa vuota identifica il package principale).

Se questi due file vengono spostati in una sottodirectory, ma restano nel package principale, per esempio:

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

si dovrebbe specificare il package principale, ma si deve dire anche alle Distutils dove i file sorgenti del package principale risiedono:

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

Solitamente, comunque, si vorranno distribuire molti moduli nello stesso package (o sottopackage). Per esempio, se i moduli `foo` e `bar` fanno parte del package `foobar`, un modo di dichiarare il proprio albero dei sorgenti è

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

Questo è infatti lo schema predefinito che le Distutils si attendono, ed è quello che richiede il minimo impegno per la descrizione nel proprio script di setup:

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

Se si vogliono inserire i moduli in directory non dichiarate per i loro package, si deve utilizzare nuovamente l'opzione package_dir. Per esempio, se la directory 'src' contiene i moduli del package `foobar`:

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

uno script di setup appropriato potrebbe essere

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

Altrimenti, se si volesse inserire i moduli del proprio package principale direttamente nella radice della distribuzione:

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

in questo caso lo script di setup sarebbe

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

(La stringa vuota indica anche la directory corrente.)

Se si hanno delle sottodirectory, devono essere esplicitamente elencate in packages, ma ogni voce in package_dir automaticamente si estende ai sotto package. (In altre parole, le Distutils *non* analizzano il proprio albero dei sorgenti, provando ad individuare quale directory corrisponda al package Python e cercando i file '__init__.py'. Comunque, se lo schema predefinito compone il sotto package:

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

il corrispondente script di setup potrebbe essere

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

(Nuovamente, la stringa vuota in package_dir è per la directory corrente).

7.3 Singoli moduli di estensione

I moduli di estensione vengono specificati utilizzando l'opzione ext_modules. package_dir non ha effetto su quale estensione di file sorgente venga trovata; interessa solo il sorgente per i moduli in puro Python. Il caso più semplice, un singolo modulo di estensione in un singolo file sorgente C, è:

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

Se l'estensione `foo` appartiene al package principale, il suo script di setup potrebbe essere

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

Con esattamente lo stesso schema d'albero del sorgente, questa estensione può essere inserita nel package `foopkg` semplicemente cambiando il nome dell'estensione:

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

Comandi di riferimento

8.1 Installare moduli: il familiare comando `install`

Il comando di installazione si assicura che il comando di compilazione sia stato eseguito e quindi esegue il sottocomando `install_lib`. `install_data` e `install_scripts`.

8.1.1 `install_data`

Questo comando installa tutti i file di dati forniti con la distribuzione.

8.1.2 `install_scripts`

Questo comando installa tutti gli script (Python) della distribuzione.

8.2 Creare una distribuzione sorgente: il comando `sdist`

I comandi per i modelli del template sono:

Commando	**Descrizione**
`include` *pat1 pat2* `...`	include tutti i file che trovano corrispondenza con l'elenco dei modelli
`exclude` *pat1 pat2* `...`	esclude tutti i file che trovano corrispondenza con l'elenco dei modelli
`recursive-include` *dir pat1 pat2* `...`	include tutti i file sotto *dir* che trovano corrispondenza con l'elenco dei modelli
`recursive-exclude` *dir pat1 pat2* `...`	esclude tutti i file sotto *dir* che trovano corrispondenza con l'elenco dei modelli
`global-include` *pat1 pat2* `...`	include tutti i file che che trovano corrispondenza nell'albero dei sorgenti nell'elenco dei modelli
`global-exclude` *pat1 pat2* `...`	esclude tutti i file che che trovano corrispondenza nell'albero dei sorgenti nell'elenco dei modelli
`prune` *dir*	esclude tutti i file sotto *dir*
`graft` *dir*	include tutti i file sotto *dir*

Qui, gli elementi sono del tipo "glob" in stile UNIX: `*` verifica ogni sequenza di caratteri nei nomi regolari dei file, `?` verifica ogni singolo carattere nei nomi regolari dei file e `[`*range*`]` verifica ognuno dei caratteri in *range* (per esempio, `a-z`, `a-zA-Z`, `a-f0-9_.`). La definizione di "carattere nei nomi regolari dei file" è specifico per la piattaforma: su UNIX è qualsiasi cosa eccetto lo slash; su windows ogni cosa eccetto backslash o due punti; su Mac OS qualsiasi cosa eccetto i due punti.

CAPITOLO

NOVE

L'API di riferimento

9.1 `distutils.core` — Funzionalità del core di Distutils

Il modulo `distutils.core` è il solo modulo che deve essere installato per usare le Distutils. Fornisce la funzione `setup()` (che viene richiamata dallo script di setup). Indirettamente fornisce le classi `distutils.dist.Distribution` e `distutils.cmd.Command`.

`setup`(*arguments*)

La funzione di base tuttofare che svolge quasi ogni cosa che gli si possa chiedere attraverso un metodo Distutils. Si veda XXXXX.

La funzione `setup()` richiede un gran numero di argomenti. Questi vengono mostrati nella seguente tavola.

nome argomento	valore	tipo
name	Il nome del package	come stringa
version	Il numero di versione del package	Si veda `distutils.versi`
description	Una singola riga che descrive il package	come stringa
long_description	Un'estesa descrizione del package	come stringa
author	Il nome dell'autore del package	come stringa
author_email	L'indirizzo email dell'autore del package	come stringa
maintainer	Il nome del manutentore corrente, se differisce da quello dell'autore	come stringa
maintainer_email	Il nome del manutentore corrente, se differisce da quello dell'autore	
url	La URL per il package (homepage)	come URL
download_url	L'URL da dove scaricare il package	come URL
packages	Un elenco di package Python che distutils può manipolare	come lista di stringhe
py_modules	Un elenco di moduli Python che distutils può manipolare	come lista di stringhe
scripts	Un elenco di script separati di file per compilare ed installare	come lista di stringhe
ext_modules	Un elenco di estensioni Python per compilare	Una lista di istanze di `distut`
classifiers	Un elenco di categorie Trove per il package	XXX link ad una migliore defi
distclass	La classe `Distribution` da usare	Una sottoclasse di `distutil`
script_name	Il nome dello script di setup setup.py - predefinito a `sys.argv[0]`	come stringa
script_args	Argomenti da fornire allo script di setup	come lista di stringhe
options	Opzioni predefinite per lo script di setup	come stringa
license	la licenza relativa al package	
keywords	Meta-data descrittivi. Si veda la PEP 314	
platforms		
cmdclass	Una mappa di nomi di comandi per sottoclassi `Command`	come un dizionario

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

Esegue uno script di setup in una specie di ambiente controllato e restituisce l'istanza `distutils.dist.Distribution` che gestisce le operazioni. Questo è utile se avete bisogno di estrarre i meta-data della distribuzione. (Passati come argomenti a parola chiave da *script* a `setup()`), o il contenuto dei file di configurazione o da riga di comando.

script_name è un file che verrà eseguito dalla funzione `execfile()`, *sys.argv[0]* verrà rimpiazzato con

script per la durata della chiamata. *script_args* è una lista di stringhe; se indicato, *sys.argv[1:]* verrà rimpiazzato da *script_args* per la durata della chiamata.

stop_after dice a `setup()` quando fermare il processo; valori possibili:

valore	descrizione
	Lo stop dopo l'istanza `Distribution` viene creata e popolata con argomenti a parola chiave per `setup()`
	Lo stop dopo i file di configurazione che devono essere analizzati (i dati verranno memorizzati nell'istanza `Distr`
	Lo stop dopo la riga di comando (`sys.argv[1:]` o *script_args*) verrà analizzata (i dati verranno memorizzati n
	Lo stop dopo che sono stati eseguiti tutti i comandi (come se fosse stato chiamato `setup()` nella solita maniera).

In aggiunta, il modulo `distutils.core` propone un certo numero di classi che si trovano altrove.

- `Extension` da `distutils.extension`
- `Command` da `distutils.cmd`
- `Distribution` da `distutils.dist`

Segue una breve descrizione di ognuna di queste, ma si faccia riferimento ai moduli più importanti per un riferimento completo.

class `Extension`

Le classi di estensione descrivono un singolo modulo C o C++ in uno script di setup. Accetta i seguenti argomenti a parola chiave nel suo costruttore

nome dell'argomento	valore
name	Il nome completo dell'estensione, incluso ogni packages — ad esempio *non* un nome di un file od
sources	Un elenco di nomi di file sorgenti, relativi alla distribuzione principale (dove risiede il file script d
include_dirs	Un elenco di directory dove ricercare file d'intestazione per C/C++ (per portabilità nella forma UN
define_macros	Un elenco di macro da definire; ogni macro viene definita mediante una tupla composta da due ele
undef_macros	Un elenco di macro non definibili esplicitamente.
library_dirs	Un elenco di directory dove ricercare librerie C/C++ al momento del link.
libraries	un elenco di nomi di librerie (non nomi di file o percorsi) in previsione del link.
runtime_library_dirs	Un elenco di directory dove ricercare file o librerie C/C++ al momento dell'esecuzione (per estens
extra_objects	Un elenco di file extra di cui dovrà essere effettuato il link (come oggetti file non implicati in 'sorg
extra_compile_args	Ogni altra informazione extra, specifica per la piattaforma o per la compilazione, da usare nel file
extra_link_args	Ogni altra informazione extra, specifica per la piattaforma, da usare quando avviene il linkaggio d
export_symbols	Un elenco di simboli che dovranno essere esportati da estensioni condivise. Non sfruttabile su tutt
depends	Un elenco di file da cui dipendono le estensioni.
language	Estensioni del linguaggio (come `'c'`, `'c++'`, `'objc'`). Se non fornite, verranno rilevate dall'es

class `Command`

Una classe `Command` (o piuttosto un'istanza di una sua sottoclasse) implementa un singolo comando DistUtils.

class `Distribution`

Una `Distribution` descrive come costruire, installare ed impacchettare un package di software Python.

Si veda la funzione `setup()` per una lista di argomenti a parola chiave accettati dal costruttore di Distribution. `setup()` crea un'istanza di Distribution.

9.2 `distutils.ccompiler` — La classe base CCompiler

Questo modulo fornisce la classe di base astratta per la classe `CCompiler`. Un'istanza `CCompiler` può essere usata per tutti i passaggi di compilazione e link necessari per compilare un singolo progetto. I metodi vengono forniti per impostare le opzioni del compilatore — definizioni di macro, include delle directory, percorsi di link, librerie ed il resto.

Questo modulo fornisce le seguenti funzioni:

gen_lib_options(*compiler, library_dirs, runtime_library_dirs, libraries*)
Genera le opzioni di linker per la ricerca delle directory di libreria e di linking con le specifiche librerie. *libraries* e *library_dirs* sono, rispettivamente, una lista di nomi di libreria (non i nomi di file!) e le directory di ricerca. Restituisce una lista di opzioni da riga di comando adatta all'uso con alcuni compilatori (in funzione dei due formati di stringa passati).

gen_preprocess_options(*macros, include_dirs*)
Genera le opzioni del pre-processore C (-D, -U, -I) come usato dai due compilatori più comuni: il tipico compilatore UNIX e il Visual C++. *macros* è al solito, una lista di una o doppia tupla, dove *(name,)* significa non definire (-U), la macro *name* e *(name, value)* significa definire (-D), la macro *name* con *value*. *include_dirs* è semplicemente una lista di nomi di directory da aggiungere al percorso di ricerca dei file di intestazione -I. Restituisce una lista di opzioni da riga di comando adatta sia ai compilatori UNIX che Visual C++.

get_default_compiler(*osname, platform*)
Determina il compilatore predefinito da usare per la piattaforma indicata.

osname dovrebbe essere uno dei nomi standard di Python per i vari OS (per esempio uno fra quelli restituiti da *os.name* e *platform*, il valore comune restituito da *sys.platform* per la piattaforma in questione.

Il valore predefinito è `os.name` e `sys.platform` nel caso in cui i parametri non vengano indicati.

new_compiler(*plat=None, compiler=None, verbose=0, dry_run=0, force=0*)
Funzione integrata per generare un'istanza di sottoclasse di alcuni CCompiler per la combinazione fornita piattaforma/compilatore. *plat* assume in modo predefinito `os.name` (per esempio `'posix'`, `'nt'` e *compiler* predefiniti sono i predefiniti per quella piattaforma. Attualmente solo `'posix'` e `'nt'` sono supportati ed i compilatori predefiniti hanno "l'interfaccia tradizionale UNIX" (classe `UnixCCompiler`) e visual C++(classe `MSVCCompiler`). Si noti che è possibile chiedere che un compilatore UNIX generi oggetti sotto Windows e che un compilatore Microsoft compili oggetti sotto UNIX—se si fornisce un valore per *compiler*, *plat* viene ignorato.

show_compilers()
Stampa una lista di compilatori disponibili (usati dall'opzione **--help-compiler** di `build`, `build_ext`, `build_clib`).

class CCompiler([*verbose=0, dry_run=0, force=0*])
La classe astratta di base `CCompiler` definisce l'interfaccia che deve essere implementata da una classe compiler reale. La classe ha a disposizione anche alcuni metodi utili usati da diverse classi compiler.

L'idea di base dietro una classe di astrazione del compilatore è che ogni istanza può essere usata per tutti i passaggi di compilazione/link nella preparazione di un singolo progetto. Quindi, attributi comuni a tutti questi passaggi di compilazione e link — incluse directory, macro da definire, librerie da linkare assieme, etc. — sono attributi dell'istanza compiler. Per permettere la variabilità nel trattamento individuale di file, molti di questi attributi possono essere variati nella fase di precompilazione o pre-link.

Il costruttore per ogni sottoclasse crea un'istanza dell'oggetto Compiler. Ha a disposizione le seguenti opzioni *verbose* (mostra un output dettagliato), *dry_run* (non vengono eseguiti tutti i passaggi) e *force* (ricompila ogni cosa, indipendentemente dalle dipendenze). Tutte queste opzioni sono, in modo predefinito, impostate a `0` (off). Si noti che probabilmente non si vuole istanziare direttamente `CCompiler` o una delle sue sottoclassi – si usi piuttosto la funzione integrata `distutils.CCompiler.new_compiler()`.

I seguenti metodi permettono di alterare manualmente le opzioni del compilatore per l'istanza della classe Compiler.

add_include_dir(*dir*)
Aggiunge *dir* alla lista delle directory in cui verrà effettuata la ricerca per i file di intestazione. Al compilatore viene indicato di cercare nelle directory nell'ordine in cui vengono fornite dalle successive chiamate di `add_include_dir()`.

set_include_dirs(*dirs*)
Imposta la lista delle directory in cui verrà effettuata la ricerca in *dirs* (una lista di stringhe). Sovrascrive ogni precedente chiamata a `add_include_dir()`; chiamate successive di `add_include_dir()` si aggiungono alla lista passata a `set_include_dirs()`. Questo non influisce su nessuna lista delle directory di include standard in cui il compilatore, per definizione, effettua la ricerca.

add_library(*libname*)
Aggiunge *libname* alla lista delle librerie che verranno incluse in tutti i link guidati da questo oggetto compiler. Si noti che *libname* *non* dovrebbe essere il nome del file contenente la libreria, ma il nome della libreria stessa: il nome del file attuale verrà desunto dal linker, dal compilatore o dalla classe compiler (in funzione della piattaforma).

Il linker verrà istruito a linkare nuovamente le librerie nell'ordine in cui vengono fornite a `add_library()` e/o `set_libraries()`. È consentito duplicare i nomi di liberia; il linker verrà istruito a linkare nuovamente le librerie tante volte quante sono quelle menzionate.

set_libraries(*libnames*)
Imposta l'elenco delle librerie da includere in tutti i link guidati dall'oggetto compiler in *libnames* (una lista di stringhe). Questo non influenza nessuna libreria standard di sistema che il linker deve includere per definizione.

add_library_dir(*dir*)
Aggiunge *dir* alla lista delle directory in cui devono essere cercate le librerie specificate con `add_library()`. Il linker verrà istruito a cercare le librerie nell'ordine in cui vengono passate a `add_library_dir()` e/o `set_library_dirs()`.

set_library_dirs(*dirs*)
Imposta la lista delle directory di ricerca delle librerie a *dirs* (una lista di stringhe). Questo non influenza nessun percorso di ricerca di libreria standard che il linker deve cercare per definizione.

add_runtime_library_dir(*dir*)
Aggiunge *dir* alla lista di directory in cui si devono cercare le librerie condivise durante l'esecuzione.

set_runtime_library_dirs(*dirs*)
Imposta la lista delle directory di ricerca per le librerie condivise durante l'esecuzione a *dirs* (una lista di stringhe). Questo non influenza ogni percorso di ricerca standard che il linker durante l'esecuzione deve cercare per definizione.

define_macro(*name*[, *value=None*])
Definisce una macro di preprocessore per tutte le compilazioni guidate da questo oggetto compiler. Il parametro facoltativo *value* dovrebbe essere una stringa; se non viene indicato, la macro verrà definita senza un valore esplicito e l'esatto risultato dipende dal compilatore usato (XXX vero? ANSI dice qualcosa circa questo argomento?)

undefine_macro(*name*)
Non definisce una macro di preprocessore per tutte le compilazioni guidate dall'oggetto compiler. Se la stessa macro viene definita da `define_macro()` e non definita da `undefine_macro()` l'ultima chiamata assume la precedenza (include ridefinizioni multiple o non definizioni). Se la macro viene ridefinita/non-definita sulla base di precompilazioni (per esempio nella chiamata a `compile()`), poi ne assume la precedenza.

add_link_object(*object*)
Aggiunge *object* alla lista degli oggetti file (o analoghi, come un file di libreria espressamente indicato o il risultato di "compilatori di risorse") da includere in ogni link guidato da questo oggetto compiler.

set_link_objects(*objects*)
Imposta la lista degli oggetti file (o analoghi) che devono essere inclusi in ogni link ad *objects*. Questo non influisce sugli oggetti file standard che il linker deve includere per definizione (come le librerie di sistema).

I seguenti metodi implementano metodi per l'auto individuazione delle opzioni del compilatore, fornendo alcune funzionalità simili all'**autoconf** della GNU.

detect_language(*sources*)
Individua il linguaggio di un file indicato, o di una lista di file. Usa l'attributo di istanza `language_map` (un dizionario) e `language_order` (una lista) per fare il lavoro richiesto.

find_library_file(*dirs, lib*[, *debug=0*])
Effettua una ricerca nella specifica lista di directory per un file condiviso o statico *lib* e restituisce il percorso completo di quel file. Se *debug* è vera, cerca per una versione di debugging (se ha senso sulla piattaforma corrente). Restituisce `None` se *lib* non viene trovata in nessuna delle directory specificate.

has_function(*funcname* [, *includes=None*, *include_dirs=None*, *libraries=None*, *library_dirs=None*])
Restituisce un valore booleano che indica se *funcname* viene supportato sulla piattaforma corrente.

Gli argomenti facoltativi possono essere usati per implementare l'ambiente di compilazione fornendo file e directory di intestazioni addizionali e file o percorsi di libreria.

`library_dir_option`(*dir*)
Restituisce le opzioni del compilatore per aggiungere *dir* alla lista delle directory di ricerca delle librerie.

`library_option`(*lib*)
Restituisce le opzioni del compilatore per aggiungere *dir* alla lista delle librerie linkate in librerie condivise o nell'eseguibile.

`runtime_library_dir_option`(*dir*)
Restituisce l'opzione del compilatore per aggiungere *dir* alla lista delle directory dove vengono cercate le librerie di runtime.

`set_executables`(***args*)
Definisce gli eseguibili (e le loro opzioni) che verranno eseguiti nei diversi stadi della compilazione. L'esatto insieme di eseguibili che devono essere eseguiti possono essere qui specificati, in funzione pure della classe compiler (attraverso l'attributo di classe 'executables') ma comunque devono avere:

attributo	descrizione
compiler	Il compilatore C/C++
linker_so	linker usato per creare oggetti e librerie condivise
linker_exe	linker usato per creare binari eseguibili
archiver	creatore di libreria statico

Sulle piattaforme con una riga di comando (UNIX, DOS/Windows), ognuna di queste è una stringa che verrà suddivisa nel nome dell'eseguibile e (facoltativamente) una lista di argomenti. La suddivisione della stringa viene effettuata in modo similare a come la shell UNIX lavora: le parole vengono delimitate da spazi, ma virgolette e backslash possono sovrascriverle. Si veda `distutils.util.split_quoted()`.

I seguenti metodi invocano le diverse fasi del processo di compilazione.

`compile`(*sources*[, *output_dir*=`None`, *macros*=`None`, *include_dirs*=`None`, *debug*=`0`, *extra_preargs*=`None`, *extra_postargs*=`None`, *depends*=`None`])
Compila uno o più file sorgenti. Genera l'oggetto file (per esempio trasforma un file '.c' in un file '.o').

sources deve essere una lista di nomi di file, più esattamente file C/C++, ma in realtà ogni cosa che può essere gestita da un particolare compilatore e da una particolare classe Compiler (per esempio la `MSVCCompiler` può gestire file di risorse in *sources*). Restituisce una lista di nomi di oggetti file, uno per ogni nome di file sorgente presente in *sources*. A seconda dell'implementazione, non tutti i file sorgenti verranno necessariamente compilati, ma tutti i corrispondenti nomi di oggetti file verranno restituiti.

Se *output_dir* viene indicata, gli oggetti file vi saranno inseriti, mantenendo comunque il loro componente relativo al percorso originale. Quindi 'foo/bar.c' normalmente si compila in 'foo/bar.o' (per una implementazione UNIX); se *output_dir* è *build*, verrà quindi compilato in 'build/foo/bar.o'.

macros, se indicata, deve essere una lista di definizioni di macro. Una definizione di macro è sia una tupla-doppia *(name, value)* che singola *(name,)*. Il primo definisce una macro; se il valore è `None`, la macro viene definita senza un esplicito valore. Il caso della singola tupla non-definisce una macro. Successive definizioni/ridefinizioni/non-definizioni acquisiscono la precedenza.

include_dirs, se indicato, deve essere una lista di stringhe, le directory da aggiungere ai percorsi di ricerca standard per gli include, solamente in questa compilazione.

debug è booleano; se vero, il compilatore verrà informato di fornire in uscita i simboli di debug nell'oggetto/i file.

extra_preargs e *extra_postargs* sono implementazioni dipendenti. Sulle piattaforme che possiedono la notazione da riga di comando (per esempio UNIX, DOS/Windows), sono più semplicemente liste di stringhe: argomenti da riga di comando extra da aggiungere alla riga di comando del compilatore. Per le altre piattaforme, si consulti la documentazione dell'implementazione della classe. In ogni caso, vengono intese come una scappatoia per quelle occasioni in cui l'ambiente astratto del compilatore non rispetta le aspettative.

depends, se indicato, è una lista di nomi di file da cui tutti gli obiettivi dipendono. Se un file sorgente è più vecchio di ogni file nelle dipendenze, il file sorgente non verrà ricompilato. Questo supporta il tracciamento delle dipendenze, ma solo ad un livello di granularità grossolana.

Solleva l'eccezione `CompileError` in caso di fallimento.

`create_static_lib`(*objects, output_libname*[, *output_dir=None, debug=0, target_lang=None*])

Effettua il link dei diversi pezzi, "bunch of stuff", per creare un file di libreria statico. Il concetto di "bunch of stuff" si compone di una lista di oggetti file forniti come *objects*, oggetto file supplementare fornito ad `add_link_object()` e/o `set_link_objects()` e le librerie fornite da `add_library()` e/o `set_library()`, ed infine, se presenti, le librerie fornite come *libraries*.

output_libname dovrebbe essere un nome di libreria, non un nome di file; il nome del file verrà dedotto dal nome della libreria. *output_dir* è la directory dove il file di libreria verrà inserito.

debug è un tipo booleano; se vero, le informazioni di debugging verranno incluse nella libreria (si noti che su diverse piattaforme, è nel passaggio della compilazione che questo avviene: l'opzione *debug* qui viene inclusa solamente per coerenza).

target_lang è il linguaggio di destinazione per cui l'oggetto indicato viene compilato. Questo permette uno specifico trattamento durante la fase di linking per alcuni linguaggi.

Solleva l'eccezione `LibError` in caso di errore.

`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*])

Linka un "bunch of stuff" assieme per creare un eseguibile o una libreria condivisa.

Il concetto di "bunch of stuff" è composto da una lista di oggetti file forniti come *objects*. *output_filename* dovrebbe essere un nome di file. Se *output_dir* viene fornito, *output_filename* è a lui relativo (per esempio *output_filename* può fornire componenti di directory se necessario).

libraries è una lista di librerie da linkare assieme. Queste sono nomi di librerie, non nomi di file, dal momento che questi vengono tradotti in nomi di file nella modalità specifica della piattaforma (per esempio *foo* diventa 'libfoo.a' in UNIX e 'foo.lib' in DOS/Windows). Comunque, possono includere un componente di directory, che dice al linker di cercare in quella specifica directory anziché cercare in tutti i soliti posti.

library_dirs, se indicate, dovrebbero essere una lista di directory dove cercare le librerie che sono state specificate come nomi di libreria semplice (per esempio nessun componente di directory). Queste si trovano all'inizio del sistema predefinito e forniscono quanto richiesto a `add_library_dir()` e/o `set_library_dirs()`. *runtime_library_dirs* è una lista di directory che verranno inserite all'interno della libreria condivisa ed usate per cercare altre librerie condivise da cui *queste* dipendono durante l'esecuzione. (Questo è rilevante solo su UNIX).

export_symbols è una lista di simboli che le librerie condivise esporteranno. (Questo è rilevante solo su Windows.)

debug è come per `compile()` e `create_static_lib()`, con una piccola distinzione che attualmente riguarda la maggior parte delle piattaforme (al contrario di `create3_static_lib()`, che include un'opzione *debug* principalmente per la propria configurazione).

extra_preargs e *extra_postargs* sono come per `compile()` (eccetto per il fatto che abitualmente forniscono argomenti da riga di comando per il particolare linker da usare).

target_lang è il linguaggio di destinazione per cui l'oggetto indicato viene compilato. Questo permette un trattamento specifico durante il linking per particolari linguaggi.

Solleva l'eccezione `LinkError` in caso di errore.

`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*])

Effettua il linking nell'eseguibile. *output_progname* è il nome del file eseguibile, quando *objects* è una lista di nomi di oggetti file da linkare. Altri argomenti sono gli stessi del metodo `link`.

`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*])

Effettua il link di una libreria condivisa. *output_libname* è il nome della libreria risultante, quando *objects* è una lista di nomi di oggetti file da linkare. Altri argomenti sono gli stessi del metodo `link`.

`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*])

Linka un oggetto condiviso. *output_filename* è il nome dell'oggetto condiviso che verrà creato, mentre *objects* è una lista di nomi di oggetti file da linkare. Altri argomenti sono gli stessi del metodo `link`.

`preprocess`(*source*[, *output_file=None*, *macros=None*, *include_dirs=None*, *extra_preargs=None*, *extra_postargs=None*])
Preprocessa un singolo file sorgente C/C++, indicato in *source*. Il risultato verrà scritto nel file chiamato *output_file*, o *stdout* se *output_file* non viene fornito. *macros* è una lista di definizioni macro, come per `compile()`, che incrementerà l'insieme delle macro con `define_macro()` e `undefine_macro()`. *include_dirs* è una lista di nomi di directory che verranno aggiunte alla lista predefinita, alla stregua di `add_include_dir()`.
Solleva l'eccezione `PreprocessError` in caso di errore.

I seguenti comodi metodi vengono definiti dalla classe `CCompiler`, per essere usati da diverse concrete sottoclassi.

`executable_filename`(*basename*[, *strip_dir=0*, *output_dir=''*])
Restituisce il nome del file dell'eseguibile per il *basename* indicato. Tipicamente per piattaforme non-Windows il nome è il medesimo di *basename*, mentre in Windows si otterrà l'aggiunta di '.exe'.

`library_filename`(*libname*[, *lib_type='static'*, *strip_dir=0*, *output_dir=''*])
Restituisce il nome del file per il nome della libreria indicata sulla piattaforma corrente. In UNIX una libreria con *lib_type* di tipo `'static'` tipicamente sarà nella forma di 'liblibname.a', quando con *lib_type* di tipo `'dynamic'` sarà nella forma di 'liblibname.so'.

`object_filenames`(*source_filenames*[, *strip_dir=0*, *output_dir=''*])
Restituisce il nome degli oggetti file per i file sorgenti indicati. *source_filenames* dovrebbe essere una lista di nomi di file.

`shared_object_filename`(*basename*[, *strip_dir=0*, *output_dir=''*])
Restituisce il nome di un oggetto file condiviso per il nome di file indicato da *basename*.

`execute`(*func*, *args*[, *msg=None*, *level=1*])
Invoca `distutils.util.execute()`. Questo metodo invoca una funzione Python *func* con gli argomenti indicati *args*, dopo aver registrato ed inserito nell'account l'opzione *dry_run*. XXX si veda anche....

`spawn`(*cmd*)
Invoca `distutils.util.spawn()`. Questa invoca un processo esterno da eseguire per il comando indicato. XXX si veda anche...

`mkpath`(*name*[, *mode=511*])
Invoca `distutils.dir_util.mkpath()`. Questa crea una directory e qualsiasi directory collegata mancante. XXX si veda anche...

`move_file`(*src*, *dst*)
Invoca `distutils.file_util.move_file()`. Rinomina *src* in *dst*. XXX si veda anche...

`announce`(*msg*[, *level=1*])
Scrive un messaggio usando `distutils.log.debug()`. XXX si veda anche...

`warn`(*msg*)
Scrive un messaggio di avvertimento *msg* su standard_error.

`debug_print`(*msg*)
Se l'opzione *debug* viene impostata in questa istanza di `CCompiler`, stampa *msg* su standard output, altrimenti non fa nulla.

9.3 `distutils.unixccompiler` — Compilatore C Unix

Questo modulo fornisce la classe `UnixCCompiler`, una sottoclasse di `CCompiler` che gestisce la tipica riga di comando in stile UNIX del compilatore C:

- macro definite con **-D*name*[=value]**
- macro non definite con **-U*name***

- directory di ricerca di include specificate con **-I*dir***
- librerie specificate con **-l*lib***
- directory di ricerca di librerie specificate con **-L*dir***
- compilazione gestita da eseguibili **cc** (o simili) con l'opzione **-c**: compila '.c' in '.o'
- linka librerie statiche gestite con il comando **ar** (possibilmente con **ranlib**)
- linka librerie condivise gestite con **cc -shared**

9.4 `distutils.msvccompiler` — Compilatore Microsoft

Questo modulo fornisce `MSVCCompiler`, una implementazione della classe astratta `CCompiler` per Microsoft Visual Studio. Dovrebbe lavorare anche usando il compilatore disponibile liberamente come parte dell'SDK .Net. XXX link per il download.

9.5 `distutils.bcppcompiler` — Compilatore Borland

Questo modulo fornisce `BorlandCCompiler`, una sottoclasse della classe astratta `CCompiler` per il compilatore C++ di Borland.

9.6 `distutils.cygwincompiler` — Compilatore Cygwin

Questo modulo fornisce la classe `CygwinCCompiler`, una sottoclasse di `UnixCCompiler` che gestisce il port su windows a cura della Cygwin del compilatore GNU C. Contiene anche la classe Mingw32CCompiler che gestisce il port mingw32 del GCC (lo stesso di cygwin in modalità no-cygwin).

9.7 `distutils.emxccompiler` — Compilatore OS/2 EMX

Questo modulo fornisce la classe EMXCCompiler, una sottoclasse di `UnixCCompiler` che gestisce il port EMX a OS/2 del compilatore GNU C.

9.8 `distutils.mwerkscompiler` — Supporto al Metrowerks CodeWarrior

Contiene `MWeksCompiler`, un'implementazione della classe astratta `CCompiler` per il MetroWerks CodeWarrior su Macintosh. C'è bisogno di lavoro supplementare per supportare CW in Windows.

9.9 `distutils.archive_util` — Utilità per archiviare

Questo modulo fornisce alcune funzioni per creare file di archivio, come i tarball o i file zip.

`make_archive`(*base_name, format*[, *root_dir=`None`, base_dir=`None`, verbose=`0`, dry_run=`0`*])

Crea un file archivio (per esempio `zip` o `tar`). *base_name* è il nome del file da creare, meno ogni estensione specifica del formato; *format* è il formato dell'archivio: uno tra `zip`, `tar`, `ztar`, o `gztar`. *root_dir* è una directory che sarà la directory principale dell'archvio; per esempio tipicamente si lancia `chdir` in *root_dir* prima di creare l'archivio. *base_dir* è la directory da dove iniziare la creazione dell'archvio; per esempio

base_dir sarà il prefisso comune di tutti i file e le directory nell'archivio. Le *root_dir* e *base_dir* predefinite sono la directory corrente. Restituisce il nome del file archivio.

Avvertenze: Questo dovrebbe essere modificato per supportare i file in formato bz2

make_tarball(*base_name, base_dir*[, *compress='gzip', verbose=0, dry_run=0*])
Crea un archivio (facoltativamente compresso) come un file tar composto da tutti i file presenti in e sotto *base_dir*. *compress* deve essere `'gzip'` (predefinito), `'compress'`, `'bzip'`, o `None`. Sia `'tar'` che l'utility di compressione indicata con *'compress'* devono essere nei percorsi previsti come predefiniti dal programma, che è probabilmente UNIX-specific. Il file tar risultante verrà chiamato '*base_dir*.tar', possibilmente più l'appropriata estensione di compressione ('.gz', '.bz2' o '.Z'. Restituisce il nome del file risultante.

Avvertenze: Dovrebbe essere rimpiazzato con le chiamate al modulo `tarfile`.

make_zipfile(*base_name, base_dir*[, *verbose=0, dry_run=0*])
Crea un file zip da tutti i file in e sotto *base_dir*. Il file zip risultante verrà chiamato *base_dir* + '.zip'. Utilizza sia il modulo Python `zipfile`, se disponibile, o l'utility InfoZIP 'zip' (se installata e trovata nel percorso di ricerca predefinito). Se nessuno dei due tool è disponibile, solleva l'eccezione `DistutilsExecError`. Restituisce il nome del file zip risultante.

9.10 distutils.dep_util — Controllo delle dipendenze

Questo modulo fornisce le funzioni per eseguire semplici dipendenze basate sulla data di file o gruppi di file; inoltre, funzioni basate interamente su quelle analisi di dipendenza sulla data.

newer(*source, target*)
Restituisce vero se *source* esiste ed è stato modificato più di recente di *target*, o se *source* esiste e *target* no. Restituisce falso se entrambe esistono e *target* ha la stessa età o è più recente di *source*. Solleva l'eccezione `DistutilsFileError` se *source* non esiste.

newer_pairwise(*sources, targets*)
Esamina due liste di file in parallelo, controllando se ogni sorgente è più recente del corrispondente destinatario. Restituisce una coppia di liste (*source*, *targets*) dove la sorgente è più recente del corrispondente, in accordo alla semantica di `never()`.

newer_group(*sources, target*[, *missing='error'*])
Restituisce vero se *target* è scaduto rispetto ad ogni file elencato in *sources*. In altre parole, se *target* esiste ed è più recente di ogni file in *sources* restituisce falso; altrimenti restituisce vero. *missing* controlla cosa si deve fare quando un file sorgente è mancante; il predefinito `'error'` è intervenire con una eccezione `OSError` all'interno di `os.stat()`; se contiene `'ignore'`, in modo silenzioso salta ogni file sorgente mancante; se è `'never'`, ogni file sorgente mancante fa assumere che *target* sia obsoleto (questo è normale in modalità "dry-run": se è `'ignore'`, in modo silenzioso salta ogni file sorgente mancante; se è `'newer'`, ogni file sorgente mancante fa assumere che *target* sia scaduto (questo è utile in modalità "dry-run": pretende che sia gestito dal comando che non lavora perché il suo input è mancante, ma non si fa carico se non si esegue il comando necessario).

9.11 distutils.dir_util — Operazioni su alberi di directory

Questo modulo fornisce funzioni per operare sulle directory e su alberi di directory.

mkpath(*name*[, *mode=0777, verbose=0, dry_run=0*])
Crea una directory ed ogni directory intermedia mancante. Se la directory già esiste (o se *name* è una stringa vuota, che significa la directory corrente, che ovviamente esiste), non fa nulla. Solleva l'eccezione `DistutilsFileError` se non riesce a creare alcune directory (per esempio alcune sottodirectory esistono, ma c'è un file al posto di una directory). Se *verbose* è vero, stampa un sommario di una riga per ogni mkdir su stdout. Restituisce la lista delle directory attualmente create.

create_tree(*base_dir, files*[, *mode=0777, verbose=0, dry_run=0*])
Crea tutte le directory vuote sotto *base_dir*, necessarie per inserire *files*. *base_dir* è semplicemente il nome

di una directory che non necessariamente deve già esistere; *files* è una lista di nomi di file da interpretare relativamente a *base_dir*. *base_dir* + la porzione di directory di ogni file in *files* verrà creata se già non esiste. Le opzioni *mode*, *verbose* e *dry_run* sono come quanto già visto per `mkpath()`.

`copy_tree`(*src*, *dst*[*preserve_mode=1*, *preserve_times=1*, *preserve_symlinks=0*, *update=0*, *verbose=0*, *dry_run=0*])

Copia un intero albero di directory *src* in una nuova posizione *dst*. Sia *src* che *dst* devono essere nomi di directory. Se *src* non è una directory, viene sollevata l'eccezione `DistutilsFileError`. Se *src* non esiste, viene creata con *mkpath()*. Il risultato finale della copia è che ogni file in *src* viene ricorsivamente copiato in *dst*. Restituisce la lista dei file che sono stati copiati o che dovrebbero essere stati copiati, usando il loro nome di uscita. Il valore restituito non è affetto da *update* o *dry_run*: è semplicemente la lista di tutti i file sotto *src*, con il nome cambiato per andare sotto *dst*.

preserve_mode e *preserve_times* sono gli stessi di `copy_file` in `distutils.file_util`; si noti che si applica solo ai file regolari, non alle directory. Se *preserve_symlinks* è vero, i collegamenti vengono copiati come collegamenti (sulle piattaforme che li supportano!); altrimenti (il predefinito), verrà copiata la destinazione. *update* e *verbose* sono gli stessi di `copy_file()`.

`remove_tree`(*directory*[*verbose=0*, *dry_run=0*])

Ricorsivamente rimuove le *directory* e tutti i file e directory ivi contenuti. Ogni errore viene ignorato (a parte il fatto che siano riportati su `stdout` se *verbose* è vero).

****Alcune di queste dovrebbero essere sostituite con il modulo shutil?****

9.12 `distutils.file_util` — Operazioni su singoli file

Questo modulo contiene alcune funzioni di utility per operare su file individuali.

`copy_file`(*src*, *dst*[*preserve_mode=1*, *preserve_times=1*, *update=0*, *link=None*, *verbose=0*, *dry_run=0*])

Copia il file *src* in *dst*. Se *dst* è una directory, *src* verrà copiato al suo interno con lo stesso nome; altrimenti, deve essere un nome di file. (Se il file esiste, verrà semplicemente sovrascritto). Se *preserve_mode* è vero (il predefinito), il modo del file (tipo e maschera dei permessi, o qualsiasi cosa sia analogo nella piattaforma corrente) viene copiato. Se *preserve_times* è vero (il predefinito), la data di ultima modifica e di ultimo accesso vengono copiate. Se *update* è vero, *src* verrà copiata solamente se *dst* non esiste, o se *dst* esiste ma è più vecchia di *src*.

link permette di costruire degli hard-link (usando `os.link`) o link simbolici (usando `os.symlink`) al posto della copia: lo si imposti in `'hard'` o `'sym'`; se è impostato a `None` (il predefinito), i file vengono copiati. Non impostare *link* su sistemi che non lo supportano: `copy_file()` non verifica se il link hard o simbolico sono disponibili.

Sotto Mac OS 9, usa la funzione di copia di file nativa in `macostools`; su altri sistemi, usa *_copy_file_contents()* per copiare il contenuto dei file.

Restituisce una tupla '`(dest_name, copied)`': *dest_name* è il nome attuale del file risultante, e *copied* è vero se il file è stato copiato (o potrebbe essere stato copiato, se *dry_run* è vero).

`move_file`(*src*, *dst*[*verbose*, *dry_run*])

Sposta il file *src* in *dst*. Se *dst* è una directory, il file verrà spostato al suo interno con lo stesso nome; altrimenti, *src* viene semplicemente rinominato in *dst*. Restituisce il nuovo nome completo del file.

Avvertenze: La gestione degli spostamenti tra dispositivi in Unix usando `copy_file()`. Che accade con gli altri sistemi???

`write_file`(*filename*, *contents*)

Crea un file chiamato *filename* e scrive *contents* (una sequenza di stringhe senza terminatori di riga) al suo interno.

9.13 `distutils.util` — Altre funzionalità utili

Questo modulo contiene altri componenti e parti che non rientrano in nessun altro modulo di utility.

get_platform()
Restituisce una stringa che identifica la piattaforma corrente. Questa viene utilizzata principalmente per distinguere le directory di build specifiche per la piattaforma e la distribuzione compilata specifica per la piattaforma. Tipicamente include il nome e la versione del sistema operativo e l'architettura (come fornito da 'os.uname()'), sebbene l'informazione esatta dipenda dal sistema operativo; per esempio per IRIX l'architettura non è particolarmente importante (IRIX gira solo su hardware SGI), ma per Linux la versione del kernel è oggettivamente importante.

Esempio di valori restituiti:

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

Per piattaforme non POSIX, attualmente viene semplicemente restituito `sys.platform`.

convert_path(*pathname*)
Restituisce 'pathname' come un nome che è utilizzabile nel filesystem nativo, per esempio lo divide con '/' e restituisce tutto insieme usando il separatore di directory corrente. È necessario perché i nomi dei file nello script di setup vengono sempre indicati in stile Unix e devono essere convertiti nella convenzione locale prima di poterli utilizzare nel filesystem. Solleva l'eccezione `ValueError` su sistemi non UNIX se *pathname* inizia o finisce con una barra obliqua.

change_root(*new_root, pathname*)
Restituisce *pathname* con *new_root* attaccato. Se *pathname* è relativo, questo è equivalente a '`os.path.joinnew_root,pathname`'. Altrimenti, richiede che *pathname* diventi relativo e quindi unisca i due, cosa utile in DOS/Windows e Mac OS.

check_environ()
Assicura che 'os.environ' abbia tutte le variabili di ambiente che noi indichiamo all'utente come utilizzabili nei file di configurazione, opzioni da riga di comando, etc. Attualmente questo include:

- HOME - directory home dell'utente (solo UNIX)
- PLAT - descrizione della piattaforma corrente, incluso hardware e OS (si veda `get_platform()`)

subst_vars(*s, local_vars*)
Esegue la sostituzione delle variabili in stile shell/Perl su *s*. Ogni occorrenza di `$` seguita da un nome viene considerata una variabile e la variabile viene sostituita dal valore trovato nel dizionario *local_vars*, o in `os.environ` se il primo controllo/incrementato per garantire che contenga alcuni valori: si veda `check_environ()`. Solleva `ValueError` per ogni variabile non trovata sia in *local_vars* o in `os.environ`.

Si noti che questo non è una funzione di interpolazione di stringhe completamente fledged. Una valida `$variable` può consistere solo di caratteri maiuscoli e minuscoli, numeri e un underscore. Non è disponibile nessuno stile di quoting { } o .

grok_environment_error(*exc*[, *prefix='error: '*])
Genera un utile messaggio di errore dall'oggetto eccezione `EnvironmentError` (`IOError` o `OSError`). Gestisce gli stili di Python 1.5.1 e successivi e fa ciò che può per comunicare con gli oggetti eccezione che hanno un corrispondente file con nome (questo accade quando l'errore coinvolge due file in un'unica operazione, come nel caso di `rename()` o `link()`). Restituisce il messaggio di errore come una stringa con prefisso aggiunto da *prefix*.

split_quoted(*s*)
Suddivide una stringa in sintonia con le regole tipiche della shell Unix per virgolette e backslash. In breve: le parole vengono delimitate da uno spazio, quando questo non è anticipato da un backslash, o all'interno di una stringa delimitata da virgolette. Singole e doppie virgolette sono equivalenti ed i caratteri che compongono le virgolette possono essere anticipati da un backslash. Il backslash viene rimosso da ogni sequenza di caratteri preceduti proprio dal backslash, lasciando solamente il carattere. I caratteri che compongono le virgolette vengono eliminate da ogni stringa delimitata. Restituisce una lista di termini.

execute(*func, args*[, *msg=None, verbose=0, dry_run=0*])
Esegue alcune azioni che interessano il mondo esterno (per esempio, scrivere nel filesystem). Alcune azioni sono speciali perché vengono disabilitate dal flag *dry_run*. Questo metodo si prende carico di fare attenzione a tutti i particolari per l'utente; tutto quello che si deve fare è fornire la funzione da chiamare ed una tupla-argomento relativa (per incorporare l' "azione esterna" da eseguire) ed un messaggio facoltativo da stampare.

strtobool(*val*)
Converte una rappresentazione stringa di stato in vero (1) o falso(0).

I valori considerati vero sono `y`, `yes`, `t`, `true`, `on` e `1`; i valori considerati falso sono `n`, `no`, `f`, `false`, `off` e `0`. Solleva `ValueError` se *val* è qualsiasi altra cosa.

byte_compile(*py_files*[, *optimize=0, force=0, prefix=None, base_dir=None, verbose=1, dry_run=0, direct=None*])
Compila a livello di bytecode, una collezione di sorgenti Python in file '.pyc' o '.pyo' nella stessa directory. *py_files* è una lista di file da compilare; ogni file che non finisce in '.py' viene silenziosamente saltato. *optimize* deve essere uno dei seguenti:

- `0` - non ottimizzare (genera '.pyc')
- `1` - ottimizzazione normale (come '`python -O`')
- `2` - ottimizzazione extra (come '`python -OO`')

Se *force* è vera, tutti i file vengono ricompilati, indipendentemente dall'ora.

Il nome del file del sorgente codificato in ogni file bytecode predefinito, viene chiamato con i nomi dei file elencati in *py_files*; questi si possono modificare con *prefix* e *basedir*. *prefix* è una stringa che verrà rimossa da ogni nome di file sorgente e *base_dir* è un nome di directory che verrà aggiunto all'inizio del nome (dopo che *prefix* è stato rimosso). *prefix* e *base_dir* possono essere indicati singolarmente o entrambi (o nessuno).

Se *dry_run* viene impostato a vero, non esegue niente che potrebbe interessare il filesystem.

La compilazione in bytecode viene fatta direttamente in questo processo di interpretazione con il modulo standard `py_compile` o indirettamente predisponendo uno script temporaneo ed eseguendolo successivamente. Normalmente si può lasciare che `byte_compile()` si occupi di usare la compilazione diretta o meno (si veda il sorgente per i dettagli). L'opzione *direct* viene usata dallo script generato nel modo indiretto; finché non si sa cosa si sta facendo, è preferibile lasciarlo impostato a `None`.

rfc822_escape(*header*)
Restituisce una versione di *header* con i simboli di escape inseriti per la sua inclusione in un'intestazione RFC 822, assicurandosi che ci siano 8 spazi dopo ogni nuova riga. Si noti che non viene effettuata nessun'altra modifica della stringa.

9.14 `distutils.dist` — The Distribution class

Questo modulo fornisce la classe `Distribution`, che rappresenta la distribuzione del modulo che sta per essere compilato/installato/distribuito.

9.15 `distutils.extension` — The Extension class

Questo modulo fornisce la classe `Extension`, usata per descrivere i moduli di estensione C/C++ nello script di setup.

9.16 `distutils.debug` — Modalità debug per Distutils

Questo modulo fornisce l'opzione DEBUG.

9.17 distutils.errors — Eccezioni Distutils

Fornisce le eccezioni usate dai moduli Distutils. Si noti che i moduli Distutils possono sollevare eccezioni standard; in particolare systemExit viene usualmente sollevata per errori che sono ovviamente errori dell'utente finale (per esempio argomenti da riga di comando errati).

Questo modulo è usabile in sicurezza con la modalità '`from ... import *`'; esporta semplicemente i simboli i cui nomi iniziano con `Distutils` e finiscono con `Error`.

9.18 distutils.fancy_getopt — Wrapper around the standard getopt module

Questo modulo fornisce un wrapper per il modulo standard `getopt` che fornisce le seguenti novità aggiuntive:

- le opzioni corte e lunghe vengono tenute insieme
- le opzioni hanno una stringa di help, cosicché `fancy_getopt` può potenzialmente creare un completo sommario del suo utilizzo
- le opzioni impostano gli attributi degli oggetti passati
- le opzioni logiche possono avere "alias negativi" — per esempio **--quiet** è l'opzione negativa di **--verbose**, **--quiet** sulla riga di comando imposta *verbose* a falso

****Dovrebbe essere sostituito con `optik` (che è anche conosciuto come `optparse` in Python 2.3 e successivi).****

`fancy_getopt`(*options, negative_opt, object, args*)
Funzione Wrapper. *options* è una lista di tuple triple '`(long_option, short_option, help_string)`' come descritto nel costruttore per `FancyGetopt`. *negative_opt* dovrebbe essere un dizionario che mappa i nomi delle opzioni ai nomi delle opzioni, sia la chiave che il valore dovrebbero essere nella lista *options*. *object* è un oggetto che verrà usato per memorizzare valori (si veda il metodo `getopt()` della classe `FancyGetopt`). *args* è la lista degli argomenti. Userà `sys.argv[1:]` se si passa `None` come *args*.

`wrap_text`(*text, width*)
Imposta *text* a meno della larghezza *width*.

Avvertenze: Dovrebbe essere sostituita con `textwrap` (che è disponibile in Python 2.3 e successivi).

class `FancyGetopt`([*option_table=None*])
option_table è una lista di tuple triple: '`(long_option, short_option, help_string)`'

Se un'opzione prende un argomento, a *long_option* dovrebbe essere aggiunto `'='`; *short_option* dovrebbe essere solo un singolo carattere, in ogni caso, nessun `':'`. *short_option* dovrebbe essere `None` se *long_option* non ha una corrispondente *short_option*. Tutte le tuple delle opzioni devono avere un'opzione lunga.

La classe `FancyGetopt` fornisce i seguenti metodi:

`getopt`([*args=None, object=None*])
Analizza le opzioni da riga di comando in args. Le memorizza come attributi in *object*.

Se la variabile *args* è `None` o non viene indicata, usa `sys.argv[1:]`. Se la variabile *object* è `None` o non viene fornita, crea una nuova istanza `OptionDummy`, vi memorizza i valori dell'opzione e restituisce una tupla '`(args,object)`'. Se la variabile *object* viene indicata, viene modificata sul posto e `getopt()` restituisce solo *args*; in entrambi i casi, la variabile *args* restituita è una copia modificata della lista *args* passata inizialmente, che viene lasciata inalterata.

`get_option_order`()
Restituisce la lista delle tuple '`(option, value)`' processate dalla precedente esecuzione di `getopt()`. Solleva l'eccezione `RuntimeError` se `getopt()` non è stata già chiamata.

generate_help([*header=None*])
Genera un testo di aiuto (una lista di stringhe, una per riga di output suggerita) dalla tabella delle opzioni per questo oggetto `FancyGetopt`.

Se indicato, stampa l'intestazione *header* fornita, all'inizio dell'help.

9.19 distutils.filelist — La classe FileList

Questo modulo fornisce la classe `FileList`, usata per l'interazione con il file system e la costruzione delle liste di files.

9.20 distutils.log — Semplice logging PEP 282-style

Avvertenze: Dovrebbe essere rimpiazzato con il modulo standard `logging`.

9.21 distutils.spawn — Spawn a sub-process

Questo modulo fornisce la funzione `spawn()`, un front-end per le diverse funzioni specifiche di piattaforma per il lancio di altri programmi in un sotto processo. Fornisce anche `find_executable()` per la ricerca della posizione di un eseguibile indicato dal suo nome.

9.22 distutils.sysconfig — Informazioni circa la configurazione di sistema

Il modulo `distutils.sysconfig` consente l'accesso alle informazioni della configurazione di basso livello di Python. Le variabili specifiche di configurazione dipendono profondamente dalla piattaforma e dalla configurazione. Le variabili specifiche dipendono dal processo di compilazione per la particolare versione di Python in fase di esecuzione; le variabili sono quelli trovate in 'Makefile' e nelle intestazioni di configurazione che vengono installate con Python sui sistemi UNIX. L'intestazione di configurazione chiamata 'pyconfig.h' per le versioni di Python a partire dalla 2.2 e 'config.h' per le versioni precedenti.

Vengono fornite alcune ulteriori funzioni che svolgono alcune comode manipolazioni per altre parti del package `distutils`.

PREFIX
Il risultato di `os.path.normpath(sys.prefix)`.

EXEC_PREFIX
Il risultato di `os.path.normpath(sys.exec_prefix)`.

get_config_var(*name*)
Restituisce il valore di una singola variabile. Questo è equivalente a `get_config_vars().get(`*name*`)`.

get_config_vars(...)
Restituisce un insieme di definizioni di variabile. Se non ci sono argomenti, restituisce un dizionario di nomi di variabili di configurazione associati ai rispettivi valori. Se vengono forniti argomenti, devono essere stringhe ed il valore restituito sarà una sequenza che indicherà i valori associati. Se il nome indicato non ha un corrispondente valore, verrà incluso `None` per quella specifica variabile.

get_config_h_filename()
Restituisce il nome completo del percorso dell'intestazione di configurazione. Per UNIX, questa sarà l'intestazione generata dallo script **configure**; per altre piattaforme l'intestazione dovrà essere fornita direttamente dalla distribuzione del sorgente Python. Il file è un file di testo specifico per la piattaforma ospite.

get_makefile_filename()
Restituisce il nome completo del percorso relativo al 'Makefile' usato per compilare Python. Per UNIX, questo sarà un file generato dallo script **configure**; il risultato per le altre piattaforme sarà diverso. Il file, se esiste, è un file di testo specifico per la piattaforma ospite. Questa funzione è utile solo su piattaforme POSIX.

get_python_inc([*plat_specific*[, *prefix*]])
Restituisce la directory sia per il file generale che per gli include C dipendenti dalla piattaforma. Se *plat_specific* viene indicato, evidenzierà la directory che contiene gli include, in funzione della piattaforma in uso; se falso o omesso, verrà restituita una directory indipendente dalla piattaforma. Se viene fornito il parametro *prefix*, verrà usato sia come prefisso al posto di PREFIX che come il prefisso di esecuzione al posto di EXEC_PREFIX se *plat_specific* risulta avere valore vero.

get_python_lib([*plat_specific*[, *standard_lib*[, *prefix*]]])
Restituisce la directory della libreria generale o dipendente dalla piattaforma di installazione. Se *plat_specific* ha un valore vero, viene restituita la directory degli include dipendenti dalla piattaforma; se falso o omesso, viene restituita la directory indipendentemente dalla piattaforma. Se *prefix* viene indicata, viene usata sia come prefisso al posto di PREFIX che come prefisso di esecuzione al posto di EXEC_PREFIX se *plat_specific* ha come valore vero. Se *standard_lib* è vera, la directory per la libreria standard viene restituita al posto della directory per l'installazione di estensioni di terze parti.

Le seguenti funzioni sono intese solo per l'uso all'interno del package distutils.

customize_compiler(*compiler*)
Esegue ogni tipo di personalizzazione per la piattaforma specifica di un'istanza distutils.ccompiler.CCompiler.

Questa funzione è necessaria (per adesso) solo su sistemi UNIX, ma dovrebbe venir usata in modo consistente per supportare la compatibilità con il passato. Inserisce informazioni variabili che cambiano attraverso i diversi UNIX e vengono memorizzate nel 'Makefile' di Python. Queste informazioni comprendono il compilatore in uso, le opzioni del linker e le estensioni usate dal linker per gli oggetti condivisi.

Questa funzione è qualcosa di molto specifico e dovrebbe essere usata solo dalle procedure di compilazione interne a Python.

set_python_build()
Comunica al modulo distutils.sysconfig che è stato utilizzato come parte del processo di compilazione di Python. Questo modifica tutta una serie di indirizzi relativi ai file, permettendogli di essere posizionati nell'area di compilazione invece che all'interno di un'esistente installazione Python.

9.23 distutils.text_file — The TextFile class

Questo modulo fornisce la classe TextFile, che mette a disposizione un'interfaccia a file di testo che (facoltativamente) si occupa di eliminare i commenti, ignorare le righe vuote ed unire le righe con dei backslash.

class TextFile([*filename=None, file=None, **options*])
Questa classe fornisce un oggetto simil-file che si occupa di tutti i compiti che comunemente si vogliono fare quando si processa un file di testo che ha una sintassi riga per riga: elimina i commenti (quelli in cui # è il carattere che rappresenta il commento), salta le righe vuote, unisce righe adiacenti, inserendo l'escape al carattere di fine riga (per esempio backslash alla fine della riga), rimuove spazi vuoti all'inizio o alla fine. Tutto ciò è facoltativo e controllabile indipendentemente.

La classe fornisce un metodo warn() per la generazione di messaggi di avviso che riportano il numero di riga fisica, sia se la riga logica in questione spazia tra multiple righe fisiche. Fornisce anche unreadline() per l'implementazione di line-at-a-time lookahead.

L'istanza TextFile viene creata sia con *filename*, *file* o entrambe. L'eccezione RuntimeError viene sollevata se entrambi sono None. *filename* dovrebbe essere una stringa e *file* un oggetto file (o qualcosa che fornisce i metodi readline() e close()). Si raccomanda che venga fornito almeno *filename*, in modo che TextFile possa includerlo nel messaggio di avvertimento. Se *file* non viene fornito, TextFile crea il proprio, usando la funzione built-in *open()*.

Le opzioni sono tutte booleane e controllano i valori restituiti da *readline()*.

nome dell'opzione	descrizione
strip_comments	rimuove dal carattere '#' alla fine della riga, compreso ogni spazio vuoto, fino al carattere '#'—fatta
lstrip_ws	rimuove gli spazi vuoti restanti da ogni riga prima di restituirla
rstrip_ws	rimuove gli spazi vuoti prima e dopo (incluso il terminatore di riga!) da ogni riga prima di restituirla
skip_blanks	salta le righe vuote *dopo* la rimozione del commento e degli spazi vuoti. (Se sia lstrip_ws che rstrip
join_lines	se un backslash è l'ultimo carattere non-di-fine-riga su di una riga, dopo la rimozione dei commenti e
collapse_join	rimuove gli spazi restanti dalle righe che vengono unite a quella precedente; solo se '(join_lines

Si noti che come *rstrip_ws* può rimuovere il carattere di nuova riga, la semantica di `readline()`, deve differire da quella del metodo `readline()` integrato nell'oggetto file! In particolare, `readline()` restituisce `None` per la fine del file: una stringa vuota potrebbe essere semplicemente una riga vuota (o una riga di spazi vuoti), se *rstrip_ws* è vero ma *skip_blanks* no.

`open`(*filename*)
Apre un nuovo file *filename*. Questo si sovrappone a qualsiasi argomento del costruttore *file* o *filename*.

`close`()
Chiude il file corrente e rimuove qualsiasi informazione circa il file stesso (incluso il nome del file ed il numero della riga corrente).

`warn`(*msg*[,*line*=`None`])
Stampa (su stderr) un messaggio di avviso collegato alla riga logica corrente nel file corrente. Se la riga logica corrente nel file, riguarda righe fisiche multiple, l'avviso riguarda l'intero insieme, come ' `righe 3-5`'. Se *line* viene fornito, sovrascrive la numerazione di riga corrente; potrebbe essere una lista o una tupla, per indicare un insieme di righe fisiche, o un numero intero per una singola riga fisica.

`readline`()
Legge e restituisce una singola riga logica dal file corrente (o da un buffer interno se viene precedentemente considerata "unread" con `unreadline()`). Se l'opzione *join_lines* è vera, questo può indicare la lettura di righe fisiche multiple in una singola striga. Aggiorna il numero della riga corrente, così chiamando `warn()` dopo `readline()`, emette un avviso circa la riga/righe appena letta. Restituisce `None` sulla fine del file, fintantoché la stringa vuota può capitare se *rstrip_ws* è vera ma *strip_blanks* no.

`readlines`()
Legge e restituisce l'elenco di tutte le righe logiche restanti nel file corrente. Questo aggiorna il numero di riga corrente all'ultima riga nel file.

`unreadline`(*line*)
Inserisce *line* (una stringa) in un buffer interno che verrà verificato da future chiamate a `readline()`. Comodo per l'implementazione di un parser di righe che funzioni una riga per volta. Si noti che le righe considerate "unread" con `unreadline` non vengono successivamente ripulite (rimozione di spazi vuoti, o altro) quando lette con `readline`. Se chiamate multiple vengono fatte a `unreadline` prima di una chiamata a `readline`, le righe verranno restituite in ordine, a partire dalla più recente.

9.24 `distutils.version` — Classe rappresentativa del numero di versione

9.25 `distutils.cmd` — Classe astratta per comandi Distutils

Questo modulo fornisce la classe di base astratta `Command`.

class `Command`(*dist*)
Classe base astratta per la definizione delle classi di comandi, le "api operaie" delle Distutils. Una utile analogia per le classi di comando è pensare a loro come subroutine con variabili locali chiamate *options*. Le opzioni vengono dichiarate in `initialize_options()` e definite (assegnazione del loro valore finale) in `finalize_options()`, comprese quelle che devono essere definite da ogni classe di comando. La distinzione tra i due è necessaria perché i valori delle opzioni arrivano dal mondo esterno (riga di comando, file di configurazione, ...) ed ogni opzione dipendente da altre opzioni deve essere calcolata dopo che le influenze esterne sono state processate — da qui `finalize_options()`. Il corpo della subroutine,

dove viene svolto tutto il suo lavoro, basato sui valore delle sue opzioni, è il metodo `run()`, che deve essere anche implementato da ogni classe di comando.

Il costruttore di classe prende un argomento singolo *dist*, un'istanza di `Distribution`.

9.26 `distutils.command` — Comandi Distutils individuali

9.27 `distutils.command.bdist` — Realizza un installer binario

9.28 `distutils.command.bdist_packager` — Classe di base astratta per packagers

9.29 `distutils.command.bdist_dumb` — Realizza un installer “dumb”

9.30 `distutils.command.bdist_rpm` — Realizza una distribuzione binaria con un RPM o SRPM Redhat

9.31 `distutils.command.bdist_wininst` — Realizza un installer Windows

9.32 `distutils.command.sdist` — Realizza una distribuzione sorgente

9.33 `distutils.command.build` — Compila tutti i file di un package

9.34 `distutils.command.build_clib` — Compila ogni libreria C in un package

9.35 `distutils.command.build_ext` — Compila ogni estensione in un package

9.36 `distutils.command.build_py` — Compila i file .py/.pyc di un package

9.37 `distutils.command.build_scripts` — Compila gli script di un package

9.38 `distutils.command.clean` — Ripulisce l’area di compilazione di un package

9.39 `distutils.command.config` — Esegue la configurazione di un package

9.40 `distutils.command.install` — Installa un package

9.41 `distutils.command.install_data` — Installa i file di dati da un package

Il comando `register` registra il package con il Python Package Index. Questo è meglio descritto nella PEP 301.

9.46 Creare un nuovo comando Distutils

La sezione descrive i passaggi per creare un nuovo comando Distutils.

Un nuovo comando vive in un modulo nel package `distutils.command`. C'è un semplice template nella directory chiamata 'command_template'. Si copi questo file in un nuovo modulo con lo stesso nome del nuovo comando che si sta implementando. Questo modulo dovrebbe implementare una classe con lo stesso nome del modulo (e del comando). Così, per esempio, per creare il comando `peel_banana` (cosicché gli utenti possano eseguire '`setup.py peel_banana`'), si deve copiare 'command_template' in 'distutils/command/peel_banana.py', quindi lo si deve editare in modo da implementare la classe `peel_banana`, una sotto classe di `distutils.cmd.Command`.

Sottoclassi di `Command` definiranno i seguenti metodi.

`initialize_options()`(*I*)
: mposta i valori predefiniti per tutte le opzioni che questo comando supporta. Si noti che questi valori predefiniti potrebbero essere sovrascritti da altri comandi, dallo script di setup, dai file di configurazione o da riga di comando. Comunque, questo non è il posto dove piazzare le dipendenze di codice tra le opzioni; generalmente, l'implementazione `initialize_options()` è solo un insieme di assegnamenti '`self.foo = None` '.

`finalize_options`()
: Imposta il valore finale per tutte le opzioni che questo comando supporta. Questo viene chiamato sempre il più tardi possibile, del tipo dopo che ogni assegnamento di opzione da riga di comando da altri comandi è stata effettuata. Questo è comunque il posto dove codificare le dipendenze dell'opzione; se *foo* dipende da *bar*, è quindi sicuro impostare *foo* da *bar* sempre che a *foo* sia stato assegnato lo stesso valore in `initialize_options`.

`run`()
: Una 'ragion d'essere' di un comando: eseguire l'azione per il quale esiste, controllato da opzioni inizializzate da `initialize_options()`, personalizzato da altri comandi, dallo script di setup, dalla riga di comando, da file di configurazione e completato con `finalize_options()`. Tutti gli output di terminale e le interazioni con il filesystem dovrebbero essere fatte con `run()`.

sub_commands formalizza la notazione di una "famiglia" di comandi, per esempio `install` come padre con sotto comandi `install_lib`, `install_headers`, etc.. Il padre di una famiglia di comandi definisce *sub_commands* come un attributo di classe; è una lista di tuple doppie '`(nome comando, predicato)`', con *command_name* una stringa e *predicate* un metodo non legato, una stringa o None. *predicate* è un metodo di un comando di base che determina quale tra i corrispondenti comandi è applicabile nella situazione corrente (per esempio `install_headers` è applicabile solo se ci sono file di intestazioni C da installare). Se *predicate* ha il valore None, quel comando è sempre applicabile.

sub_commands è solitamente definito alla *fine* di una classe, perché i predicati possono essere metodi non legati, che devono essere già stati definiti. L'esempio canonico è il comando `install`.

INDICE DEI MODULI

D

INDICE

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W