CHAPTER 4

Setuptools: Harnessing Your Code

This chapter focuses on replicable builds—a small but vital part of continuous integration. If a build can't be replicated, then test harnesses lose their efficacy. If one build differs from another build of the same code, then it is possible for tests to succeed against one build while failing against another, and testing loses its meaning. In the worst case, if a build can't be replicated, then it can become well-nigh impossible to diagnose and fix bugs in a consistent manner.

Avoiding manual configuration is the key to replicable builds. This isn't a slight against developers. People are prone to errors, while computers are not. Every manual step is an opportunity for error and inconsistency, and every error and inconsistency is an opportunity for the build to subtly fail. Again and again, this point will drive the design of the harness that ties the disparate pieces of the build together.

The harness will be built using the package Setuptools. Setuptools supersedes Python's own Distutils library, but as of Python 2.5, it is still a third-party package. Obtaining and installing Setuptools with Python 2.5 and earlier is demonstrated in this chapter.

Setuptools uses distributable packages called *eggs*. Eggs are self-contained packages. They fulfill a similar role to RPMs in the Linux world, or GEMs in Ruby installations. I'll describe eggs and demonstrate how to build and install them, along with the steps involved in installing binaries. The mystery of version numbering will be explained, too.

When complete, the demonstration project can be built on any machine with no more than a stock Python installation. All dependent packages are bundled with it, including Setuptools itself. The harness produced here is generic and can be used in any project. This chapter's work will prepare you for the subsequent chapter on automated builds.

The Project: A Simple RSS Reader

For the next few chapters, we're going to be building a single project. It's a simple RSS reader. RSS stands for Really Simple Syndication. It is a protocol for publishing frequently updated content such as news stories, magazine articles, and podcasts. It will be a simple command line tool showing which articles have been recently updated.

This chapter and the next don't demand much functionality—just enough to verify building and installation—so the program isn't going to be very exciting. In fact, it won't be much more than Hello World, but it will run, and throughout the book it will grow. This way of doing things isn't just convenient for me. It also demonstrates the right way to go about developing a program.

Continuous integration demands that a program be built, installed, executed, and tested throughout development. This guarantees that it is deployable from the start. By moving deployment into the middle of the development process, continuous integration buffers the sudden shock that often arises when a product finally migrates to an operational environment.

Optimally, the build, installation, execution, and tests are performed after every commit. This catches errors as soon as they hit the source repository, and it isolates errors to a specific code revision. Since the changes are submitted at least daily, the amount of code to be debugged is kept to a minimum. This minimizes the cost of fixing each bug by finding it early and isolating it to small sets of changes.

This leads to a style of development in which programs evolve from the simplest implementation to a fully featured application. I'll start with the most embryonic of RSS readers, and I'll eventually come to something much more interesting and functional. This primordial RSS reader will be structured almost identically to the Hello World program in Chapter 3. The source code will reside in a directory called `src`, and `src` will reside in the top level of the Eclipse project.

Initially, we'll have two files: `src/rsreader/__init__.py` and `src/rsreader/app.py`. `__init__.py` is empty, and `app.py` reads as follows:

```
import sys

def main():
    print "OK" # give us some feedback
    return 0 # exit code

if __name__ == '__main__':
    sys.exit(main())
```

This project should be checked into your source repository as `svn:///usr/local/svn/repos/rsreader/trunk`.

Python Modules

Python bundles common code as packages. Python packages and modules map to directories and files. The presence of the file `__init__.py` within a directory denotes that the directory is a Python package. Each package contains child packages and modules, and every child package has its own `__init__.py` file.

Python supports multiple package trees. These are located through the Python path variable. Within Python, this variable is `sys.path`. It contains a `list` of directories. Each directory is the root of another tree of packages. You can specify additional packages when Python starts using the `PYTHONPATH` environment variable. On UNIX systems, `PYTHONPATH` is a colon-separated directory list. On Windows systems, the directories are separated by semicolons.

By default, the Python path includes two sets of directories: one contains the standard Python library or packages, and the other contains a directory called `site-packages`, in which nonstandard packages are installed. This begs the question of how those nonstandard packages are installed.

The Old Way

You've probably installed Python packages before. You locate a package somewhere on the Internet, and it is stored in an archived file of some sort. You expand the archive, change directories into the root of the unpacked package, and run the command `python setup.py install`. The results are something like this:

```
running install
running build
running build_py
running install_lib
creating /Users/jeff/Library/Python/2.5/site-packages/rsreader
copying build/lib/rsreader/__init__.py -> /Users/jeff/Library/➥
Python/2.5/site-packages/rsreader
copying build/lib/rsreader/app.py -> /Users/jeff/Library/➥
Python/2.5/site-packages/rsreader
byte-compiling /Users/jeff/Library/Python/2.5/site-packages/➥
rsreader/__init__.py to __init__.pyc
byte-compiling /Users/jeff/Library/Python/2.5/site-packages/➥
rsreader/app.py to app.pyc
running install_egg_info
Writing /Users/jeff/Library/Python/2.5/site-packages/➥
RSReader-0.1-py2.5.egg-info
```

`setup.py` invokes a standard package named Distutils, which provides methods to build and install packages. In the Python world, it fulfills many of the same roles that Make, Ant, and Rake do with other languages.

Note how the files are installed. They are copied directly into `site-packages`. This directory is created when Python is installed, and the packages installed here are available to all Python programs using the same interpreter.

This causes problems, though. If two packages install the same file, then the second installation will fail. If two packages have a module called `math.limits`, then their files will be intermingled.

You could create a second installation root and put that directory into the per-user `PYTHONPATH` environment variable, but you'd have to do that for all users. You have to manage the separate install directories and the `PYTHONPATH` entries. It quickly becomes error prone. It might seem like this condition is rare, but it happens frequently—whenever a different version of the same package is installed.

Distutils doesn't track the installed files either. It can't tell you which files are associated with which packages. If you want to remove a package, you'll have sort through the `site-packages` directories (or your own private installation directories), tracking down the necessary files.

Nor does Distutils manage dependencies. There is no automatic way to retrieve dependent packages. Users spend much of their time chasing down dependent packages and installing each dependency in turn. Frequently, the dependencies will have their own dependencies, and a recursive cycle of frustration sets in.

The New Way: Cooking with Eggs

Python eggs address these installation problems. In concept, they are very close to Java JAR files. All of the files in a package are packed together into a directory with a distinctive name, and they are bundled with a set of metadata. This includes data such as author, version, URL, and dependencies.

Package version, Python version, and platform information are part of an egg's name. The name is constructed in a standard way. The package PyMock version 1.5 for Python 2.5 on OS X 10.3 would be named `pymock-1.5-py2.5-macosx-10.3.egg`. Two eggs are the same only if they have the same name, so multiple eggs can be installed at the same time. Eggs can be installed as an expanded directory tree or as zipped packages. Both zipped and unzipped eggs can be intermingled in the same directories. Installing an egg is as simple as placing it into a directory in the `PYTHONPATH`. Removing one is as simple as removing the egg directory or ZIP file from the `PYTHONPATH`. You could install them yourself, but Setuptools provides a comprehensive system for managing them. In this way, it is similar to Perl's CPAN packages, Ruby's RubyGems, and Java's Maven.

The system includes retrieval from remote repositories. The standard Python repository is called the cheese shop. Setuptools makes heroic efforts to find the latest version of the requested package. It looks for closely matching names, and it iterates through every version it finds, looking for the most recent stable version. It searches the local filesystem and the Python repositories. Setuptools follows dependencies, too. It will search to the ends of the earth to find and install the dependent packages, thus eliminating one of the huge headaches of installing Distutils-based packages.

WHY THE CHEESE SHOP?

The cheese shop is a reference to a Monty Python sketch. In the sketch, a soon-to-be-frustrated customer enters a cheese shop and proceeds to ask for a staggering variety of cheeses, only to be told one by one that none of them are available. Even cheddar is missing.

Watching Setuptools and `easy_install` attempt to intuit the name of a package from an inaccurate specification without a version number quickly brings this sketch to mind. It helps to pass the time if you imagine Setuptools speaking with John Cleese's voice.

Setuptools includes commands to build, package, and install your code. It installs both libraries and executables. It also includes commands to run tests and to upload information about your code to the cheese shop.

Setuptools does have some deficiencies. It has a very narrow conception of what constitutes a build. It is not nearly as flexible as Make, Ant, or Rake. Those systems are configured using specialized Turing-complete programming languages. (Ant has even been used to make a simple video game.) Setuptools is configured with a Python dictionary. This makes it easy to use for simple cases, but leaves something to be desired when trying to achieve more ambitious goals.

Some Notes About Building Multiple Versions

One of the primary goals of continuous integration is a replicable build. When you build a given version of the software, you should produce the same end product every time the build is performed. And multiple builds will inevitably be performed. Developers will build the product on their local boxes. The continuous integration system will produce test builds on a build farm. A final production packaging system may produce a further build.

Each build version is tagged with a unique tag denoting a specific build of a software product. Each build is dependent upon specific versions of external packages. Building the same version of software on two different machines of the same architecture and OS should always produce the same result. If they do not, then it is possible to produce software that successfully builds and runs in one environment, but fails to build or run successfully in another. You might be able to produce a running version of your product in development, but the version built in the production environment might be broken, with the resulting defective software being shipped to customers. I have personally witnessed this.

Preventing this syndrome is a principal goal of continuous integration. It is avoided by means of replicable builds. These ensure that what reaches production is the same as what was produced in development, and thus that two developers working on the same code are working with the same set of bugs.

Most software products depend upon other packages. Different versions of different packages have different bugs. This is nearly obvious, but something else is slightly less obvious: the software you build has different bugs when run with different dependent packages. It is therefore necessary to tightly control the versions of dependent packages in your build environments. This is complicated if multiple packages are being built on the same machine. There are several solutions to the problem.

The *virtual Python* solution involves making a copy of the complete Python installation for each product and environment on your machine. The copy is made using symbolic links, so it doesn't consume much space. This works for some Python installations, but there are others, such as Apple's Mac OS X, that are far too good at figuring out where they should look for files. The links don't fool Python. Windows systems don't have well-supported symbolic links, so you're out of luck there, too.

The *path manipulation* solution is the granddaddy of them all, and it's been possible from the beginning. The `PYTHONPATH` environment variable is altered when you are working on your project. It points to a local directory containing the packages you've installed. It works everywhere, but it takes a bit of maintenance. You need to create a mechanism to switch the path, and more importantly, the installation path must be specified every time a package is added. It has the advantages that it can be made to work on any platform and it doesn't require access to the root Python installation.

I prefer the *location path* manipulation solution. It involves altering Python's search path to add local `site-packages` directories. This requires the creation of two files: the file `altinstall.pth` within the global `site-packages` directory, and the file `pydistutils.cfg` in your home directory. These files alter the Python package search paths.

On UNIX systems, the file `~/.pydistutils.cfg` is created in your home directory. If you're on Windows, then the situation is more complicated. The corresponding file is named `%HOME%/pydistutils.cfg`, but it is consulted only if the `HOME` environment variable is defined. This is not a standard Windows environment variable, so you'll probably have to define it yourself using the command `set HOME=%HOMEDRIVE%\%HOMEPATH%`.

This mechanism has the disadvantage that it requires a change to the shared site-packages directory. This is probably limited to root or an administrator, but it only needs to be done once. Once accomplished, anyone can add their own packages without affecting the larger site. The change eliminates an entire category of requests from users, so convincing IT to do it shouldn't be terribly difficult.

Python's site package mechanism is implemented by the standard site package. Once upon a time, accessing site-specific packages required manually importing the site package. These days, the import is handled automatically. A code fragment uses `site` to add a site package to add per-user site directories. The incantation to do this is as follows:

```
import os, site; ➥
 site.addsitedir(os.path.expanduser('~/lib/python2.5'))
```

You should add to the `altinstall.pth` file in the global `site-packages` directory. The site package uses `.pth` files to locate packages. These files normally contain one line per package added, and they are automatically executed when found in the search path. This handles locating the packages.

The second file is `~/.distutils.cfg` (`%HOME%\distutils.cfg` on Windows). It tells Distutils and Setuptools where to install packages. It is a Windows-style configuration file. This file should contain the following:

```
[install]
install_lib = ~/lib/python2.5
install_scripts = ~/bin
```

On the Mac using OS X, the first part of this procedure has already been done for you. OS X ships with the preconfigured per-user site directory `~/Library/python/$py_version_short/site-packages`, but it is necessary to tell Setuptools about it using the file `~/.pydistutils.cfg`. The file should contain this stanza:

```
[install]
install_lib = ~/Library/python/$py_version_short/site-packages
install_scripts = ~/bin
```

On any UNIX variant, you should ensure that `~/bin` is in your shell's search path.

Installing Setuptools

Setuptools is distributed as an egg. As of version 2.5, Python doesn't natively read eggs, so there is a "chicken-and-egg" problem. This can be circumvented with a bootstrap program named `ez_setup.py`, which is available at `http://peak.telecommunity.com/dist/ez_setup.py`. Once downloaded, it is run as follows:

```
$ python ez_setup.py
```

```
Downloading http://pypi.python.org/packages/2.5/s/setuptools/➥
setuptools-0.6c7-py2.5.egg
Processing setuptools-0.6c7-py2.5.egg
Copying setuptools-0.6c7-py2.5.egg to /Users/jeff/Library/Python/2.5/site-packages
```

```
Adding setuptools 0.6c7 to easy-install.pth file
Installing easy_install script to /Users/jeff/binInstalling easy_install-2.5➥
script to /Users/jeff/bin
Installed /Users/jeff/Library/Python/2.5/site-packages/➥
setuptools-0.6c7-py2.5.egg
Processing dependencies for setuptools==0.6c7
Finished processing dependencies for setuptools==0.6c7
```

ez_setup.py uses HTTP to locate and download the latest version of Setuptools. You can work around this if your access is blocked. ez_setup.py installs from a local egg file if one is found. You copy the appropriate egg from http://pypi.python.org/pypi/setuptools using your tools of choice, and you place it in the same directory as ez_setup.py. Then you run ez_setup.py as before.

Setuptools installs a program called ~/bin/easy_install (assuming you've created a local site-packages directory). From this point forward, all Setuptools-based packages can be installed with easy_install, including new versions of Setuptools. You'll see more of ez_setup.py later in this chapter when packaging is discussed.

Getting Started with Setuptools

Setuptools is driven by the program setup.py. This file is created by hand. There's nothing special about the file name—it is chosen by convention, but it's a very strong convention. If you've used Distutils, then you're already familiar with the process. Setuptools just adds a variety of new keywords. The minimal setup.py for this project looks like this:

```
from setuptools import setup, find_packages
setup(
    # basic package data
    name = "RSReader",
    version = "0.1",

    # package structure
    packages=find_packages('src'),
    package_dir={'':'src'},
)
```

A minimal setup.py must contain enough information to create an egg. This includes the name of the egg, the version of the egg, the packages that will be contained within the egg, and the directories containing those packages.

The name attribute should be unique and identify your project clearly. It shouldn't contain spaces. In this case, it is RSReader.

The version attribute labels the generated package. The version is not an opaque number. Setuptools goes to great lengths to interpret it, and it does a surprisingly good job, using it to distinguish between releases of the same package. When installing from remote repositories, it determines the most recent egg by using the version; and when installing dependencies, it uses the version number to locate compatible eggs. Code can even request importation of a specific package version.

In general, version numbers are broken into development and release. Both 5.6 and 0.1 are considered to be base versions. They are the earliest released build of a given version. Base versions are ordered with respect to each other, and they are ordered in the way that you'd expect. Version 5.6 is later than version 1.1.3, and version 1.1.3 is later than version 0.2.

Version 5.6a is a development version of 5.6, and it is earlier than the base version. 5.6p1 is a later release than 5.6. In general, a base version followed by a string between *a* and *e* inclusive is considered a development version. A base version followed by a string starting with *f* (for *final*) or higher is considered a release version later than the base version. The exception is a version like 5.6rc4, which is considered to be the same as 5.6c4.

There is another caveat: additional version numbers after a dash are considered to be development versions. That is, 5.6-r33 is considered to be earlier than 5.6. This scheme is typically used with version-controlled development. Setuptools's heuristics are quite good, and you have to go to great lengths to cook up a version that it doesn't interpret sensibly.

The `packages` directive lists the packages to be added. It names the packages, but it doesn't determine where they are located in the directory structure. Package paths can be specified explicitly, but the values need to be updated every time a different version is added, removed, or changed. Like all manual processes, this is error prone. The manual step is eliminated using the `find_packages` function.

`find_packages` searches through a set of directories looking for packages. It identifies them by the `__init__.py` file in their root directories. By default, it searches for these in the top level of the project, but this is inappropriate for RSReader, as the packages reside in the `src` subdirectory. `find_packages` needs to know this, hence `find_packages('src')`. You can include as many package directories as you like in a project, but I try to keep these to an absolute minimum. I reserve the top level for build harness files—adding source directories clutters up that top level without much benefit.

The `find_packages` function also accepts a list of excluded files. This list is specified with the keyword argument `exclude`. It consists of a combination of specific names and regular expressions. Right now, nothing is excluded, but this feature will be used when setting up unit tests in Chapter 8.

The `package_dir` directive maps package names to directories. The mappings are specified with a dictionary. The keys are package names, and the values are directories specified relative to the project's top-level directory. The root of all Python packages is specified with an empty string (`""`); in this project, it is in the directory `src`.

Building the Project

The simple `setup.py` is enough to build the project. Building the project creates a working directory named `build` at the top level. The completed build artifacts are placed here.

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

```
running build
running build_py
creating build
creating build/lib
creating build/lib/rsreader
```

```
copying src/rsreader/__init__.py -> build/lib/rsreader
copying src/rsreader/app.py -> build/lib/rsreader
```

```
$ ls -lF
```

```
total 696
drwxr-xr-x   3 jeff  jeff     102 Nov  7 12:25 build/
-rw-r--r--   1 jeff  jeff    2238 Nov  7 12:14 setup.py
drwxr-xr-x   5 jeff  jeff     170 Nov  6 20:45 src/
```

Interpreting the build output is easier if you understand how Setuptools and Distutils are structured. The command `build` is implemented as a module within Setuptools. The setup function locates the command and then executes it. All commands can be run directly from `setup.py`, but many can be invoked by other Setuptools commands, and this happens here.

When Setuptools executes a command, it prints the message `running command_name`. The output shows the `build` command invoking `build_py`. `build_py` knows how to build pure Python packages. There is another build module, `build_ext`, that knows how to build Python extensions, but no extensions are built in this example, so `build_ext` isn't invoked.

The subsequent output comes from `build_py`. You can see that it creates the directories `build`, `build/lib`, and `build/lib/rsreader`. You can also see that it copies the files `__init__.py` and `app.py` to the appropriate destinations.

At this point, the project builds, but it is not available to the system at large. To install the package, you run `python setup.py install`. This installs `rsreader` into the local `site-packages` directory configured earlier in this chapter.

```
$ python setup.py install
```

```
running install
running bdist_egg
running egg_info
creating src/RSReader.egg-info
writing src/RSReader.egg-info/PKG-INFO
writing top-level names to src/RSReader.egg-info/top_level.txt
writing dependency_links to src/RSReader.egg-info/dependency_links.txt
writing manifest file 'src/RSReader.egg-info/SOURCES.txt'
writing manifest file 'src/RSReader.egg-info/SOURCES.txt'
installing library code to build/bdist.macosx-10.3-fat/egg
running install_lib
running build_py
creating build
creating build/lib
creating build/lib/rsreader
copying src/rsreader/__init__.py -> build/lib/rsreader
copying src/rsreader/app.py -> build/lib/rsreader
creating build/bdist.macosx-10.3-fat
creating build/bdist.macosx-10.3-fat/egg
creating build/bdist.macosx-10.3-fat/egg/rsreader
```

```
copying build/lib/rsreader/__init__.py -> build/bdist.macosx-10.3-fat/egg/rsreader
copying build/lib/rsreader/app.py -> build/bdist.macosx-10.3-fat/egg/rsreader
byte-compiling build/bdist.macosx-10.3-fat/egg/rsreader/__init__.py to __init__.pyc
byte-compiling build/bdist.macosx-10.3-fat/egg/rsreader/app.py to app.pyc
creating build/bdist.macosx-10.3-fat/egg/EGG-INFO
copying src/RSReader.egg-info/PKG-INFO -> build/bdist.macosx-10.3-fat/egg/➥
EGG-INFO
copying src/RSReader.egg-info/SOURCES.txt -> build/bdist.macosx-10.3-fat/egg/➥
EGG-INFO
copying src/RSReader.egg-info/dependency_links.txt -> build/bdist.macosx-10.3-fat/➥
egg/EGG-INFO
copying src/RSReader.egg-info/top_level.txt -> build/bdist.macosx-10.3-fat/egg/➥
EGG-INFO
zip_safe flag not set; analyzing archive contents...
creating dist
creating 'dist/RSReader-0.1-py2.5.egg' and adding 'build/bdist.macosx-10.3-fat/➥
egg' to it
removing 'build/bdist.macosx-10.3-fat/egg' (and everything under it)
Processing RSReader-0.1-py2.5.egg
Copying RSReader-0.1-py2.5.egg to /Users/jeff/Library/Python/2.5/site-packages
Adding RSReader 0.1 to easy-install.pth file

Installed /Users/jeff/Library/Python/2.5/site-packages/RSReader-0.1-py2.5.egg
Processing dependencies for RSReader==0.1
Finished processing dependencies for RSReader==0.1
```

You can see that install invokes four commands: bdist_egg, egg_info, install_lib, and build_py:

```
running install
running bdist_egg
running egg_info
creating src/RSReader.egg-info
...
installing library code to build/bdist.macosx-10.3-fat/egg
running install_lib
running build_py
creating build
...
```

install uses bdist_egg to produce a binary distribution for the package. bdist_egg calls egg_info and install_lib. The latter in turn calls build_py to produce a new build of the package to be bundled and installed.

egg_info produces a description of the egg. Among the files produced by egg_info are a list of dependencies and a manifest listing all the files in the egg. install_lib takes the products of build_py and copies them into an assembly area where they are finally packaged up by bdist_egg. In the very end, the egg is moved into place by install.

When the process is complete, you're left with a new dist directory at the top level. This contains the newly constructed egg file along with any previously constructed versions.

Each step can be invoked from the command line, and all can be configured independently. This is done through a file called setup.cfg. Later in this chapter, this file will be used to modify installation locations.

Installing Executables

The RSReader application has been installed into site-packages. It can be executed with Python using the -m option, as in the previous section. What you want is an executable. Executables are specified in setup.py with entry points, which can also specify rendezvous points for plug-ins.

The entry_points attribute describes the entry points. It is a dictionary of lists. The keys denote the kind of entry point, and the values name entry points and map each of them to a Python function. Executables are denoted with the console_scripts and gui_scripts keys. setup.py now looks like this:

```
from setuptools import setup, find_packages
setup(
    # basic package data
    name = "RSReader",
    version = "0.1",

    # package structure
    packages=find_packages('src'),
    package_dir={'':'src'},

    # install the rsreader executable
    entry_points = {
        'console_scripts': [
                'rsreader = rsreader.app:main'
                ]
    },
)
```

This entry_points stanza installs one executable. It will be named rsreader on UNIX systems. On Windows systems, it will be named rsreader.exe. Running this program will execute the function rsreader.app.main(). Note that the definition contains a colon between the package path and the function name.

The executable will be installed into the Python scripts directory ~/bin as configured in ~/.distutils.cfg. The location is reported in the output of python setup.py install:

```
$ python setup.py install
```

```
running install
running bdist_egg
...
Copying RSReader-0.1-py2.5.egg to /Users/jeff/Library/Python/2.5/site-packages
```

```
RSReader 0.1 is already the active version in easy-install.pth
Installing rsreader script to /Users/jeff/bin

Installed /Users/jeff/Library/Python/2.5/site-packages/RSReader-0.1-py2.5.egg
Processing dependencies for RSReader==0.1
Finished processing dependencies for RSReader==0.1
```

Dependencies

Setuptools manages dependencies. It locates appropriate versions of dependent packages, downloads them, and installs them. It searches and retrieves them from remote or local sources.

Dependencies are managed with the `external_requirements` attribute, which is a list of dependency expression strings. The simplest dependency expression is an unadorned package name. Setuptools then searches for the latest version of that package. The meaning of “latest” is determined using the rules described in the “Getting Started with Setuptools” section earlier in this chapter.

More complex dependency expressions have a package name on the left-hand side, a version on the right-hand side, and a comparison operator between them. The expression `docutils >= 3.4` means, “Get package Docutils version 3.4 or later.” Reproducibility is the primary goal, so this project will demand specific versions.

```
from setuptools import setup, find_packages
setup(
    # basic package data
    name = "RSReader",
    version = "0.1",

    # package structure
    packages=find_packages('src'),
    package_dir={'':'src'},

    # install the rsreader executable
    entry_points = {
        'console_scripts': [
                'rsreader = rsreader.app:main'
                ]
    },
    install_requires = [
                        'docutils == 0.4',
                        ],
)

$ python setup.py install
```

```
running install
running bdist_egg
...

Installed /Users/jeff/Library/Python/2.5/site-packages/RSRead➥
er-0.1-py2.5.egg
Processing dependencies for RSReader==0.1
Searching for docutils==0.4
Reading http://pypi.python.org/simple/docutils/
Reading http://docutils.sourceforge.net/
Best match: docutils 0.4
Downloading http://prdownloads.sourceforge.net/docutils/docu➥
tils-0.4.tar.gz?download
Processing docutils-0.4.tar.gz
Running docutils-0.4/setup.py -q bdist_egg --dist-dir /tmp/easy_install-ebwmnZ/➥
docutils-0.4/egg-dist-tmp-UqTwxP
"optparse" module already present; ignoring extras/optparse.py.
"textwrap" module already present; ignoring extras/textwrap.py.
zip_safe flag not set; analyzing archive contents...
docutils.parsers.rst.directives.misc: module references __file__
docutils.writers.html4css1.__init__: module references __file__
docutils.writers.newlatex2e.__init__: module references __file__
docutils.writers.pep_html.__init__: module references __file__
docutils.writers.s5_html.__init__: module references __file__
Adding docutils 0.4 to easy-install.pth file
Installing rst2html.py script to /Users/jeff/bin
Installing rst2latex.py script to /Users/jeff/bin
Installing rst2newlatex.py script to /Users/jeff/bin
Installing rst2pseudoxml.py script to /Users/jeff/bin
Installing rst2s5.py script to /Users/jeff/bin
Installing rst2xml.py script to /Users/jeff/bin

Installed /Users/jeff/Library/Python/2.5/site-packages/docu➥
tils-0.4-py2.5.egg
Finished processing dependencies for RSReader==0.1
```

The first line displayed in bold announces that Setuptools is processing the dependencies for your RSReader package. The next shows that it is searching for the dependency you specified. It searches for the package at `pypi.python.org`. `pypi.python.org` catalogs Python modules, but it doesn't store them. It has a reference to each module's download site.

Setuptools doesn't search other catalogs, indicating that it found the package's description at `pypi.python.org`. Instead, it follows the reference to `docutils.sourceforge.net`, and there it searches for a download link to the correct file version. It finds that link, and it downloads the file from `http://prdownloads.sourceforge.net`. Take note of the URL in the output; it will be important in the next section.

Once the file is downloaded, Setuptools announces the processing of the package. The name `docutils-0.4.tar.gz` indicates that the file is a TAR archive compressed with the gzip algorithm. The output shows that it is automatically uncompressed, unpacked, and installed using Docutils's own `setup.py`. The intermediate product is stored in a temporary directory that is removed at the end of the process.

There are no required dependencies for this version of Docutils, but there are optional ones. The output indicates that these optional dependencies (`optparse` and `textwrap`) are already present. There are a few warning messages, and then a series of installation messages as a group of executables are installed. These executables convert from a text format called RST to other documentation formats.

■**Note** You've seen the message `zip_safe flag not set; analyzing archive contents...` several times now. It is an advisory warning. It indicates that Setuptools is not creating a zipped egg. Although Setuptools can do this instead of producing expanded directory trees, the feature can sometimes cause problems, and by default it is turned off.

Think Globally, Install Locally

Setuptools does a great job of finding packages on the Net. Sometimes I'm frightened at how good a job it does, but sometimes it fails. An author's download server may go offline. The package might get deleted. A version you depend on may no longer be available, and instead a broken version may take its place. An author might replace one version with another subtly different version with the same version number. More frequently, your Internet connection may go south.

All of these situations have the same result: the build can't be replicated. The project may not even be buildable. These aren't academic situations either—it has happened to me within the last two weeks.

The solution uses a local copy of the dependent package. That copy is checked into source control along with the project code. Setuptools is then directed to use that copy with the `--find-links` option.

I'll create a directory in the project called `thirdparty`. The file `docutils-0.4.tar.gz` is downloaded into `thirdparty` from the URL `http://prdownloads.sourceforge.net/docutils/docutils-0.4.tar.gz?download`. This location was gleaned from `python setup.py install`'s output.

```
$ mkdir thirdparty
$ cd thirdparty
$ curl -L -o docutils-0.4.tar.gz http://prdownloads.sourceforge.net/docutils➥
/docutils-0.4.tar.gz?download
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1208k  100 1208k    0     0  72013      0  0:00:17  0:00:17 --:--:-- 73992
```

```
f$ ls -lF
```

```
total 2424
-rw-r--r--   1 jeff  jeff  1237801 Nov  8 13:34 docutils-0.4.tar.gz
```

Removing an Existing Package: Undoing Your Hard Work

To demonstrate the repository, it is necessary to remove the Docutils module over and over again. Unfortunately, this is the hardest thing to do with Setuptools—the process is only partially supported. It has three steps: the entry specifying the default version must be removed from `site-packages/easy_install.pth`, the egg must be removed from `site-packages`, and the installed binaries must be removed.

The entry in `site-packages/easy_install.pth` is removed with the command `python setup.py easy_install`:

```
$ python setup.py easy_install -m 'docutils==0.4'
```

```
running easy_install
Searching for docutils==0.4
Best match: docutils 0.4
Processing docutils-0.4-py2.5.egg
Removing docutils 0.4 from easy-install.pth file
Installing rst2html.py script to /Users/jeff/bin
Installing rst2latex.py script to /Users/jeff/bin
Installing rst2newlatex.py script to /Users/jeff/bin
Installing rst2pseudoxml.py script to /Users/jeff/bin
Installing rst2s5.py script to /Users/jeff/bin
Installing rst2xml.py script to /Users/jeff/bin

Using /Users/jeff/Library/Python/2.5/site-packages/docutils-0.4-py2.5.egg

Because this distribution was installed --multi-version, before you can➥
import modules from this package in an application, you will need to 'import➥
pkg_resources' and then use a 'require()' call similar to one of these➥
examples, in order to select the desired version:

    pkg_resources.require("docutils")  # latest installed version
    pkg_resources.require("docutils==0.4")  # this exact version
    pkg_resources.require("docutils>=0.4")  # this version or higher

Processing dependencies for docutils
Finished processing dependencies for docutils
```

The `-m` option reinstalls the package in multi-version mode. In this mode, all programs must explicitly request the version of the package they require, but it has the desired side effect of removing this package's entry from `site-packages/easy_install.pth`. It also has the happy side effect of telling you exactly what you need to do in the next two steps. It tells you where the egg is, what scripts have been installed, and where they were installed to.

The egg is deleted:

```
$ rm -rf /Users/jeff/Library/python/2.5/site-packages/doc➥
utils-0.4-py2.5.egg
```

Then the binaries are deleted:

```
$ rm /Users/jeff/bin/rst2html.py
$ rm /Users/jeff/bin/rst2latex.py
$ rm /Users/jeff/bin/rst2newlatex.py
$ rm /Users/jeff/bin/rst2pseudoxml.py
$ rm /Users/jeff/bin/rst2s5.py
$ rm /Users/jeff/bin/rst2xml.py
```

Installing from the Local Copy

I'll demonstrate the process using the easy_install command. easy_install is the Setuptools component that installs eggs.

```
$ cd ..
$ python setup.py easy_install --find-links thirdparty 'docutils==0.4'
```

```
running easy_install
Searching for docutils==0.4
Best match: docutils 0.4
Processing docutils-0.4.tar.gz
Running docutils-0.4/setup.py -q bdist_egg --dist-dir➥
 /tmp/easy_install-LMZVog/docutils-0.4/egg-dist-tmp-Xkl2d6
"optparse" module already present; ignoring extras/optparse.py.
"textwrap" module already present; ignoring extras/textwrap.py.
zip_safe flag not set; analyzing archive contents...
docutils.parsers.rst.directives.misc: module references __file__
...

Installed /Users/jeff/Library/Python/2.5/site-packages/docutils-0.4-py2.5.egg
Processing dependencies for docutils==0.4
Finished processing dependencies for docutils==0.4
```

The output shows no external searching. Instead, the file was taken from the local repository, which is what you want. There is a problem, though: this argument only works with easy_install. There is no similar command-line option for the install command, but there is another mechanism that will work.

Fixing Options with setup.cfg

Setuptools uses an optional configuration file named setup.cfg to configure values for options. The values set in this file work if a command is called directly by the user or if it is called indirectly through another command.

The file setup.cfg lives in the root directory of the project. The format is the standard Windows option file with stanzas that have bracketed section names. The section names are the same as the commands they configure.

In this case, the command name is easy_install, the option name is find-links, and the value is thirdparty. This can be added to the file by hand, or via Setuptools itself:

```
$ python setup.py setopt --command easy_install --option find_links➥
 --set-value thirdparty
```

```
running setopt
Writing setup.cfg
```

```
$ cat setup.cfg
```

```
[easy_install]
find_links = thirdparty
```

The file can be edited by hand, but using Setuptools has an advantage. Setuptools merges the property setting into the existing setup.cfg, so existing settings made by hand or script are left intact.

The repository should be used every time setup.py is invoked, so the file should be put under version control and checked in. From time to time, developers may need to make temporary customizations, but these shouldn't be checked in. If such modifications are checked in, the changes are usually caught by the continuous build system (the subject of the next chapter).

Bootstrapping Setuptools

You've put a great deal of effort into setting up a new development environment easily and reproducibly. However, there is one small problem: when the project is synched down to a completely virgin Python installation, it is necessary to install Setuptools using ez_setup.py before you can run your build script setup.py.

Luckily, Setuptools provides a way around this. Copy the ez_setup.py program (remember ez_setup.py from earlier in this chapter?) into the top-level directory of the project, and add the following lines to the very top of setup.py:

```
from ez_setup import use_setuptools
use_setuptools(version='0.6c7')
```

At this point, running `setup.py` will download and install Setuptools if it isn't already present. You are still dependent on a network connection, but you can solve this problem, too. If `ez_setup.py` finds a Setuptools egg in the current directory, then it will install from there. Copying the Setuptools egg from `site-packages` into the top-level project directory will suffice:

```
$ cp -rp ~/Library/Python/2.5/site-packages/setuptools-0.6c7-py2.5.egg
```

It's possible to test the whole process by deleting everything in `site-packages`, and then running `python setup.py install`. Indeed, this is what the build system will be doing in the next chapter. After the first run, the top-level directory should look something like this:

```
$ ls -lF
```

```
drwxr-xr-x   4 jeff  jeff     136 Nov  7 13:51 build/
drwxr-xr-x   3 jeff  jeff     102 Nov  7 13:51 dist/
-rw-r--r--   1 jeff  jeff    8960 Oct 27 23:20 ez_setup.py
-rw-r--r--   1 jeff  jeff    9189 Nov  7 13:50 ez_setup.pyc
-rw-r--r--   1 jeff  jeff      40 Nov  8 23:05 setup.cfg
-rw-r--r--   1 jeff  jeff    1201 Nov  8 22:56 setup.py
-rw-r--r--   1 jeff  jeff  322831 Oct 27 23:40 setuptools-0.6c7-➥
py2.5.egg
drwxr-xr-x   5 jeff  jeff     170 Nov  7 13:51 src/
drwxr-xr-x   4 jeff  jeff     136 Nov  8 13:34 thirdparty/
```

Subverting Subversion: What Shouldn't Be Versioned

The harness has created directories and files that should not be managed by Subversion. The `build` and `dist` directories contain ephemeral build artifacts generated by Setuptools. These can be regenerated at any time. Developers expect to customize `setup.cfg`, and their individual changes shouldn't be checked in. Required values in the file are generated by `setup.py` anyhow.

These files are reported by `svn status`. They could be ignored, but they clutter the output, and cluttered output is hard to understand. Compiled Python files are reported, too. `svn status` prefixes these unversioned files with ?.

```
$ svn status
```

```
?       build
?       dist
?       ez_setup.py
?       ez_setup.pyc
?       setuptools-0.6c7-py2.5.egg
?       setup.py
?       setup.cfg
?      thirdparty/docutils-0.4.tar.gz
?      configure.py
...
```

In CVS, these would be ignored via a `.cvsignore` file. Subversion doesn't use files to track this information. Its system is based on a more general mechanism called *properties*. Properties associate bits of metadata with files and directories. The metadata consists of key/value pairs. In this case, the metadata key is `svn:ignore` and the value is a list of patterns separated by newlines.

Newlines complicate setting the property. The clearest way to set multiple values for `svn:ignore` from the command line is using a temporary file. In this example, it is `/tmp/ignore.txt`, and it has the following three lines:

```
build
dist
*.pyc
```

The `svn:ignore` property is then set with `svn propset`, and the file is fed into `svn propset` via the `-F` option. Finally, the temporary file `/tmp/ignore.txt` is deleted.

```
$ svn proplist .
$ svn propset svn:ignore -F /tmp/ignore.txt .
$ rm /tmp/ignore.txt
$ svn status
```

```
M      .
?      ez_setup.py
?      setuptools-0.6c7-py2.5.egg
?      setup.py
?      thirdparty/docutils-0.4.tar.gz
...
```

```
$ svn status --no-ignore
```

```
I      build
I      dist
I      ez_setup.pyc
```

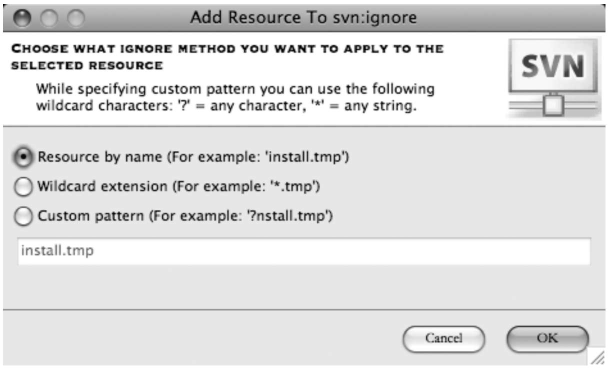
The `svn:ignore` list is attached to the top-level project directory. Property changes must be committed just like any other file modification. Since the changes are in source control, they affect all other developers. As they update their working copies with `svn update`, the `build`, `dist`, and `*.pyc` files will disappear from Subversion's reports:

```
$ svn commit -m "added build, dist, and *.pyc to svn:ignore" .
```

```
Sending        .
Committed revision 16.
```

The Easy Way with Eclipse

The same changes can be made using Eclipse and Subversive. The file or directory to be ignored is highlighted in the Package Explorer, and the context menu is brought up. Team ➤ “Add to svn:ignore” is selected, and it brings up the window shown in Figure 4-1.


Add Resource To svn:ignore

CHOOSE WHAT IGNORE METHOD YOU WANT TO APPLY TO THE SELECTED RESOURCE

While specifying custom pattern you can use the following wildcard characters: '?' = any character, '*' = any string.

[✓ Resource by name (For example: 'install.tmp')]
Wildcard extension (For example: '*.tmp')
Custom pattern (For example: '?nstall.tmp')

install.tmp

Figure 4-1. *Adding a file to svn:ignore*

The selected radio button, “Resource(s) by name,” is the desired choice, as it chooses the full name of the file. Then you can click OK to add the selected file to `svn:ignore`. If you'd selected the file `ez_setup.pyc` then the "Wildcard extension" option would add the pattern `*.pyc to svn:ignore`, and the “Custom pattern” option would add the pattern specified in the text box below.

Subversive combines each new setting with the previous ones, eliminating one of the major headaches of managing `svn:ignore` from the command line. The drawback is that Subversive has no direct support for removing files from `svn:ignore`. You'll have to go back to the command line for that.

Checking in Changes: Not Losing It

At this point, quite a few changes have been made, and they need to be checked in. The files in question are `setup.py`, `ez_setup.py`, `thirdparty` (and all its contents), `configure.py`, and `setuptools-0.6c7-py2.5.egg`. These are to be placed under Subversion’s control.

Working in Development Mode

RSReader was installed earlier in this chapter. Its contents are available to every other Python module in our environment, but we are actively developing it. Each change made should be available to other packages. It is possible to install the package after every change. This is feasible for small packages, but it begins to slow down development with larger packages. Besides, it’s a manual step, and at some point it will be forgotten, with frustrating results.

Setuptools provides a way around this called *development mode*. Putting a package in development creates a link file in `site-packages` and a special entry in the `easy-install.pth` file that lists the packages managed by `easy_install`. These redirect imports to your project directory instead of `site-packages`.

```
$ python -m rsreader.app
```

```
OK
```

I'll change the message printed by `src/rsreader/app.py`. The main entry point becomes

```
...
def main():
    print "SPAM!" # give us some feedback
    return 0 # exit code
...
```

The module is run again to demonstrate that the change has had no effect:

```
$ python -m rsreader.app
```

```
OK
```

```
$ python setup.py develop
```

```
running develop
running egg_info
writing src/RSReader.egg-info/PKG-INFO
writing top-level names to src/RSReader.egg-info/top_level.txt
writing dependency_links to src/RSReader.egg-info/dependency_links.txt
writing manifest file 'src/RSReader.egg-info/SOURCES.txt'
running build_ext
Creating /Users/jeff/Library/Python/2.5/site-packages/RSReader.egg-link (link to➥
src)
Removing RSReader 0.1 from easy-install.pth file
Adding RSReader 0.1 to easy-install.pth file

Installed /private/tmp/rsreader/src
Processing dependencies for RSReader==0.1
Finished processing dependencies for RSReader==0.1
```

The new link entry now points to the project directory. If you run the verification command, you can see that the change is picked up:

```
$ python -m rsreader.app
```

```
SPAM!
```

Finally, the print statement is changed from SPAM! back to OK, and you can verify that the change has taken effect:

```
$ python -m rsreader.app
```

```
OK
```

Summary

This chapter began with a project that will be developed throughout this book. It is a simple command-line RSS reader, implemented in only the most minimal fashion—the focus is on the harness surrounding it. This harness is constructed using Setuptools rather than the Python standard library Distutils, but the project builds from source on a machine with a stock Python installation. No work is required other than running `python setup.py install`.

Distutils and Setuptools are closely related. Distutils is the stock package for managing Python packages. It is limited in its capabilities. Setuptools is an evolution of Distutils that uses a new package format called eggs. The combination of Setuptools and eggs eliminates many shortcomings of Distutils.

Development and build environments need to be as free from interference as possible. Packages installed by other users or software on a system may cause unnoticed conflicts. This possibility is eliminated through the use of several techniques such as virtual Python installations or per-user `site-packages` and `bin` directories.

Setuptools is not a standard package, so it must be obtained and installed. Setuptools-based packages can be configured to bootstrap Setuptools, and the harness does this.

Setuptools manages external dependencies. It makes best-effort attempts at locating dependent packages. By default, it searches many sites on the Internet, but it can be configured to check local resources preferentially. Depending on third parties with potentially flaky resources is anathema to replicable builds, so the harness uses locally stored eggs. The search locations are overridden using a `setup.cfg` file. This file is checked into source control along with the rest of the project.

Sometimes it is necessary to remove a package, and doing so is fairly straightforward.

Several build and artifact directories are generated while building and installing the project. These files need to be excluded from revision control.

Setuptools's development mode links the development environment directly into `site-packages` so that changes there are directly reflected in the Python installation.

Now that I've shown you how to build and install your code using Setuptools, in the next chapter I'll show you how this can be done automatically using Buildbot.